Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Estratégias para Controlar o Tamanho da Suíte de Teste Gerada a partir de Abordagens MBT

Emanuela Gadelha Cartaxo

Tese submetida a Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para a obtenção do grau de doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
(Orientadora)
Antonia Bertolino
(Co-Orientadora)

Campina Grande, Paraíba, Brazil

Resumo

Teste é a técnica mais comumente utilizada para avaliar a qualidade do software como parte do processo de validação e verificação. Entretanto é normalmente uma atividade cara. Prometendo reduzir os custos e também promover efetividade, abordagens de Teste Baseado em Modelos (*Model-based Testing - MBT*) têm sido propostas, onde os casos de teste podem ser obtidos a partir de especificações. Em MBT, os algorítmos usados para obter casos de teste são normalmente baseados em "busca" em um modelo comportamental e, na maioria das vezes, o critério de parada é baseado em um critério de cobertura estrutural que é exaustivamente aplicado. Portanto, neste contexto, o número de casos de teste tende a ser muito grande. Por outro lado, nem sempre há recursos suficientes (tempo e dinheiro) para executar todos eles. Também, alguns casos de teste podem exercitar sequências comuns de funcionalidades. Neste sentido, redundância é um conceito importante que pode ser considerado para obter uma suíte de teste menor, uma vez que partes redundantes podem não incrementar a cobertura de funcionalidades ou cobertura de faltas.

Algumas estratégias para controlar o tamanho da suíte de teste têm sido propostas: seleção de casos de teste e redução de suítes de teste. A primeira normalmente considera um propósito de teste (para reduzir o espaço de busca) e/ou fixa um número de casos de teste desejado sem levar em consideração o conceito de redundância. Por outro lado, algumas estratégias para redução de suítes de teste são propostas e experimentadas considerando a redundância estrutural no contexto de teste caixa branca.

Obviamente, é necessário buscar estratégias para controlar o tamanho das suítes de teste geradas a partir de abordagens MBT que considerem o conceito de redundância. Diferentes estratégias para controlar o tamanho das suítes de teste foram propostas nesta tese focando em seleção e redução. Os resultados mostram que estratégias para seleção e redução baseadas em Similaridades são boas para detectar faltas e prover um adequada cobertura. As estratégias propostas podem ser aplicadas a diferentes níveis de teste, porém o foco é teste de sistema.

Por fim, um novo modo de avaliar estratégias para redução de suítes de teste - considerando a taxa de detecção de faltas - é proposta. A taxa de detecção de faltas é uma

métrica largamente utilizada para comparar estratégias de priorização de suítes de teste, entretanto até agora não tinha sido considerada para avaliar estratégias de redução de suítes de teste.

Abstract

Testing is the most commonly applied technique to evaluate the quality of software as part of verification & validation processes. However, it is usually an expensive activity. Promising to reduce costs as well as promoting effectiveness, Model-based Testing (MBT) approaches have been proposed, where test cases can be obtained from specifications. In MBT, the algorithms used to obtain test cases are usually based on a "search" in a behavioral model and, in most of the times, the stop decision is based on structural coverage criteria that are exhaustively applied. Therefore, in this context, the number of applicable test cases tends to be very high. On the other hand, usually, there are not sufficient resources (time and money) to execute all of them. Also, some test cases may exercise common sequences of functionalities. In this sense, redundancy is an important concept that can be considered to obtain a smaller test suite, once that redundant parts may not increase functionality coverage or fault detection.

Some strategies for controlling the size of the test suites have been proposed: test case selection and test suite reduction. The former usually considers a test purpose (to reduce a space search) and/or fix a number of test cases that are desired without taking into account the redundancy concept. On the other hand, some strategies for test suite reduction are proposed and experimented considering structural redundancy for white-box testing.

Obviously, it is necessary to seek strategies for controlling the size of the test suites generated from MBT approaches that consider the redundancy concept. Different strategies for controlling the size of test suites are proposed in this thesis focusing on selection and reduction. Results show that strategies for selection and reduction based in Similarities are good to detect faults and provide a adequate coverage. Even though the strategies proposed can be applied to different testing levels, the focus is on system testing.

Finally, a new way to evaluate test suite reduction strategies - by considering the rate of fault detection - is proposed. Even though, the rate of fault detection is a metric widely used to compare test suite prioritization strategies, it has not yet been considered to evaluate test suite reduction strategies.

Agradecimentos

Inicialmente, eu gostaria de agradecer a Deus por tudo.

Obrigada aos meus pais e irmãs pelo amor, compreensão e suporte dado em todas as dificuldades encontradas durante estes anos. Provavelmente eu nunca teria chegado até aqui sem isso.

Obrigada a José Lima Júnior por sua paciência, compreensão e amor durante estes anos.

Eu gostaria de agradecer a minha orientadora Patrícia Machado pelo seu apoio profissional. Ela foi não somete uma boa professora, mas também amiga. Seu apoio e paciência foram valiosos durante esses anos.

Eu gostaria de agradecer a minha co-orientadora Antonia Bertolino pela sua colaboração durante este trabalho e por ter me adotado em seu grupo. Todas as discussões foram valiosas para este trabalho.

Eu gostaria de agradecer a Eda Marchetti pelo seu apoio profissional e pessoal (incluindo os sorrisos e "chiacchieri") enquanto eu estava em Pisa.

Eu gostaria de agradecer a Francisco de Oliveira Neto pelo seu apoio técnico, discussões valiosas durante a implementação das estratégias e, claro, por estar sempre aberto a escutar meus desabafos.

Eu gostaria de agradecer a João Felipe Ouriques e Priscila Vieira pelo apoio técnico.

Eu gostaria de agradecer a equipe do projeto de pesquisa Motorola Brazil Test Center (Motorola BTC-RD) por todas as discussões.

Por fim, obrigada aos meus amigos: Ana Emília, Ana Esther, Andréia Karla, Danilo, Laísa, Neto, Rafael, Rafaelly, Ramon, Raniere, Spachson e Verlaynne pelo apoio e por ter tolerado minhas mudanças de humor.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Teste é uma atividade para avaliar a qualidade das aplicações e é normalmente usado na prática como uma atividade do processo de verificação e validação. Esta atividade frequentemente consome uma quantidade significante de recursos em projetos de desenvolvimento [46], dessa forma pesquisas têm sido direcionadas para desenvolver abordagens que possam contribuir para diminuir os custos (tempo e dinheiro [33; 8]) que são demandados por esta atividade. Uma dessas abordagens é o Teste baseado em Modelos (*Model-based Testing - MBT*).

MBT tem se tornado popular devido a necessidade de garantia de qualidade e também devido ao paradigma de desenvolvimento emergente centrado no modelo e metodologias de desenvolvimento centradas em teste [48]. MBT promete controlar a qualidade do software e reduzir os custos inerentes do processo de teste, uma vez que os casos de teste podem ser gerados a partir da especificação do software. Assim, casos de teste podem ser obtidos antes ou durante o processo de desenvolvimento e portanto, quando o código da aplicação está disponível, os casos de teste podem ser executados. Resumindo, MBT tem sido apontada com o uma abordagem para aumentar confiabilidade, efetividade e produtividade no processo de software, desde que ela promete controlar a qualidade do software e reduzir os custos [48].

Normalmente, abordagens de teste baseado em modelos geram um grande número de casos de teste (suíte de teste grande) [41]. Dado que tempo e dinheiro para executar todos os casos de teste são restritos [25] (particularmente para teste manual), é necessário diminuir a quantidade de casos de teste, ou seja, precisamos obter um subconjunto de casos de teste. Este subcojunto deve conter os melhores casos de teste, isto é, aqueles que são capazes de

revelar faltas e prover uma boa cobertura (por exemplo: cobertura de transições ou requisitos, entre outros). Esta é uma tarefa difícil, uma vez que nós necessitamos observar muitas variáveis tais como cobertura de funcionalidade ou restrições de recursos. Na prática, a tarefa de reduzir o tamanho da suíte de teste é um processo manual sujeito a erros e sem garantia que os sistema será efetivamente testado, uma vez que critério de cobertura ou detecção de faltas não são usados como parâmetros para a execução da estratégia.

Adicionalmenente, suítes de teste geradas a partir de abordagens MBT apresentam um considerável grau de redundância entre os casos de teste, isto é, dois casos de teste podem ser muito similares. É provável que eles não adicionem valor a suíte, uma vez que podem não garantir uma melhor cobertura de um dado critério ou por não terem capacidade de revelar defeitos ainda não revelados. Neste caso, nós nomeamos de casos de teste redundantes.

Pesquisadores têm investigado duas abordagens que são direcionadas a resolver o problema do tamanho da suíte de teste [65]:

- Seleção de Casos de Teste - Algorítmos (tais como os propostos por Rothermel e Harrold [51; 52] e Jard e Jeron [40]) para seleção de casos de teste, selecionam um subconjunto de casos de teste da suíte original que pode ou não prover a mesma cobertura da suíe de teste original;

- Redução de Suíte de Teste - Algorítmos (tais como os propostos por Wong et al.[64], Zhong et al. [67], Harrold et al. [35], Ma et al. [65] e Chen e Lau [18]) para redução de suíte de teste seleciona um subcojunto representativo que satisfaz os requisitos de teste definido (critério de cobertura), portanto seleciona um subconjunto da suíte original que provê a mesma cobertura (de acordo com o requisito de teste definido) que a suíte original.

Observe que ambas as abordagens lidam como o problema do tamanho da suíte de teste, tentando reduzir o número de casos de teste. A diferença entre elas é que para reduzir o tamanho da suíte de teste, a abordagem de redução de suíte de teste considera um específico conjunto de requisitos de teste (critério de cobertura) e a suíte de teste reduzida satisfaz aquele conjunto como a suíte de teste original.

As próximas seções deste capítulo apresentam: uma visão geral da tese e suas principais contribuições (Seção 1.1); a metodologia adotada (Seção 1.2); e por fim, a estrutura desta

tese é apresentada (Seção 1.3).

1.1 Visão Geral da Tese

Nesta tese, nós buscamos uma solução para o problema do tamanho da suíte de teste causado pela redundância no contexto MBT aplicado ao teste de sistema. Nossas questões de pesquisa são:

- É possível reduzir (seleção de casos de teste/ redução de suíte de teste) o tamanho da suíte de teste por eliminar casos de teste redudantes baseado em uma função de similaridade e ainda guardar uma cobertura razoável de um dado critério de teste?
- É possível definir uma estratégia que produza uma suíte de teste menor baseada naquela função para maximizar cobertura de transições da suíte de teste resultante?
- É possível combinar aquela estratégia com outras estratégias tais como baseada em valores que são aplicadas para focar em cenários de uso e maximizar a capacidade de detecção de faltas?

Como respostas a estas questões, nós propomos algumas estratégias que lidam com o problema do tamanho da suíte de teste. As principais contribuições do nosso trabalho são:

- **Função de Similaridade** - Esta função calcula a distância entre casos de teste de uma suíte de teste. Esta distância representa o grau de redundância entre cada par de casos de teste;
- **Seleção baseada em Similaridade** - Uma nova estratégia para *seleção* de casos de teste é proposta baseada na função de similaridade. Esta estratégia é comparada a estratégia de seleção aleatória;
- **Similaridade Balanceada (WSA)** - Uma nova estratégia para *seleção* de casos de teste é proposta baseada na função de Similaridade e pesos. Esta estratégia é comparada a outras estratégias bem conhecidas na literatura;
- **Dissimilaridade** - Uma nova estratégia para *redução* é proposta baseada na função de similaridade. Esta estratégia é comparada a 4 estratégias bem conhecidas na literatura;

- **Novo Modo de Avaliar Suítes de Teste Reduzidas** - O principal critério usado para comparar estratégias para redução de suítes de teste é o tamanho da suíte de teste reduzida, onde a menor suíte é a melhor. Nós propomos usar a ordem de seleção para comparar as estratégias de redução de suíte de teste, uma vez que as faltas são importantes e nem sempre é possível executar todos os casos de teste (é necessário parar a execução de uma suíte reduzida).

Como dito antes, estas estratégias lidam com o problema da redundância e elas são aplicadas a abordagens de testes baseado em modelos visando melhorar os resultados dos algorítmos de geração de casos de teste por eliminar casos de teste redundantes da suíte gerada. As estratégias são baseadas no uso de uma função de similaridade que revela os casos de teste mais diferentes. Neste sentido, as estratégias podem ser efetivas sob as seguintes pressupostos:

- Casos de teste similares são redudantes no sentido que eles cobrem um conjunto comum de funcionalidades e têm a capacidade similar de revelar faltas. Dessa forma, alguns deles podem ser eliminados para satisfazer as restrições de recursos de um projeto;

- Provavelmente não existe ganho adicional em mantê-los (os casos de teste redundantes) na suíte de teste já que eles não afetam significativamente a cobertura de funcionalidades ou faltas.

1.2 Metodologia

O objetivo das estratégias para controlar o tamanho das suítes de teste é diminuir o tamanho da suíte de teste. Nesta tese, essas estratégias são propostas de acordo com uma função de similaridade visando selecionar os mais diferentes casos de teste, obtendo uma melhor cobertura de funcionalidades.

O processo genérico para selecionar casos de teste/ reduzir suítes de teste a ser seguido é mostrado na Figura 1.1. Inicialmente as suítes de teste são obtidas a partir de modelos LTS (*Labeled Transtion Systems* - Sistemas de Transições Rotuladas). As estratégias para seleção de casos de teste/ redução de suítes de teste são aplicadas nessas suítes de teste. Para aplicar

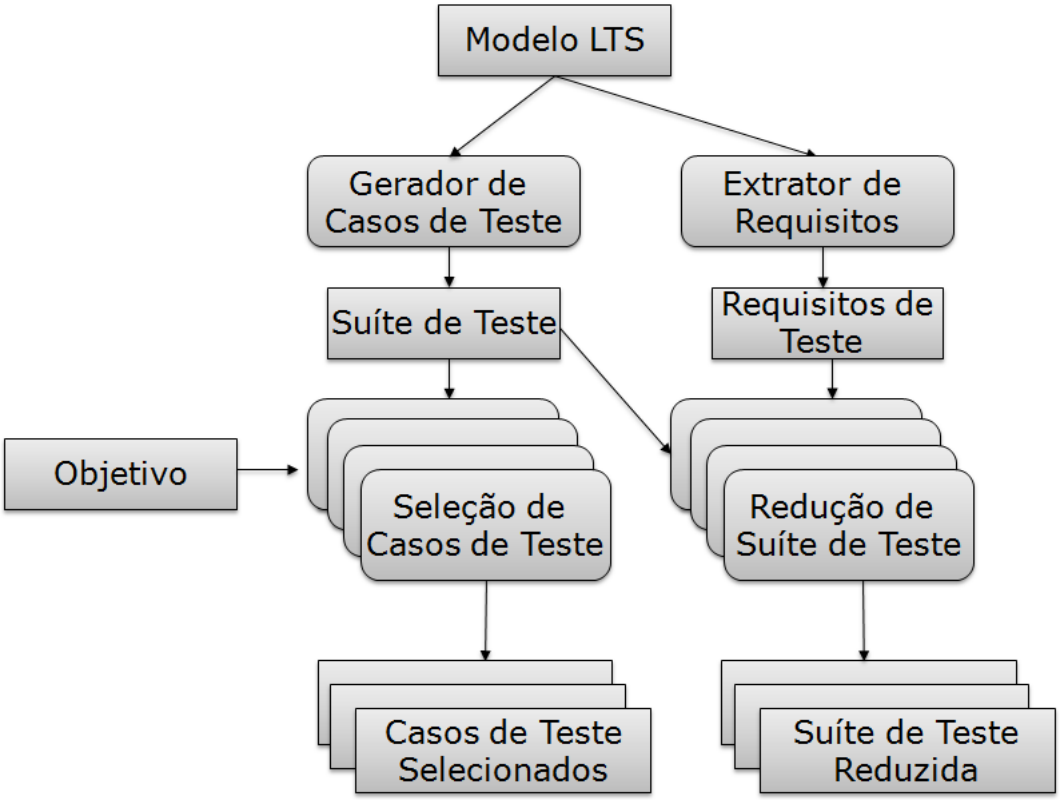


Figura 1.1: Visão geral do processo de seleção de casos de teste/ redução de suítes de teste

as estratégias para seleção de casos de teste, é necessário definir um objetivo, tais como um propósito de teste, o tamanho desejado da suíte, entre outros. Por outro lado, para aplicar as estratégias para redução de suítes de teste, requisitos de teste podem ser automaticamente obtidos do modelo.

Após aplicar as estratégias para seleção de casos de teste ou redução de suítes de teste, nós obtemos um subconjunto da suíte de teste. Para avaliar estas estratégias, alguns estudos de caos e experimentos foram executados. Cobertura de transições e detecção de faltas são usados como critério para comparar e avaliar essas estratégias.

1.3 Estrutura da Tese

Os seguintes capítulos desta tese são a fundamentação teórica, as estratégias propostas e suas respectivas avaliações, trabalhos relacionados e algumas conclusões (incluindo trabalhos futuros). Parte do conteúdo em alguns capítulos já foram publicados ([6; 14; 15; 17; 7]). Mais especificamente, os tópicos desta tese são organizados em capítulos como segue:

Capítulo 2

Visando tornar esta tese auto-contida, este capítulo apresenta termos e conceitos usados em Teste de Software, Teste Baseado em Modelos, alguns Critérios de Cobertura, Seleção de Casos de Teste, Redução de Suítes de Teste, Priorização de Casos de Teste, Abordagens Baseada em Valor, Experimentação em Engenharia de Software e Análise Estatística.

Capítulo 3

Neste capítulo nós apresentamos o nosso conceito de redundância e ilustramos o problema em uma aplicação real. Depois disso, apresentamos a nossa *Função de Similaridade*. Esta função calcula o grau de similaridade entre um par de casos de teste, considerando o número de transições idênticas entre dois casos de teste e seus respectivos tamanhos. Para calcular todos os graus de similaridade entre todos os casos de teste do conjunto de testes, uma matriz de similaridade é construída.

Capítulo 4

Nossa estratégia *Seleção baseada em Similaridade* é apresentada neste capítulo. O algorítmo da estratégia e um exemplo são apresentados a fim de demonstrar a aplicação da estratégia. A avaliação desta estratégia é realizada através de um estudo de caso (analisar a cobertura de transições e faltas) e de um experimento (análise da cobertura de transição). Os resultados são comparados com estratégia de seleção aleatória.

Capítulo 5

Neste capítulo, propomos outra estratégia - chamada emph Similaridade Balanceada (WSA) - para seleção dos casos de teste baseado em similaridades com pesos associados a casos de teste. O algorítmo e um exemplo são apresentados. Dois estudos de caso são executados para comparar WSA a seleção aleatória, Similaridade (Capítulo 4) e a Aleatória Guiada, considerando cobertura de transições e faltas.

Capítulo 6

Este capítulo apresenta a estratégia para a redução de suíte de testes baseada na similaridade. Essa estratégia, chamada *Dissimilaridade*, é capaz de reduzir a suíte de teste utilizando

como requisito de teste, o critério de cobertura de transições. Para explicar a estratégia, apresentamos o algorítmo e a execução em um exemplo. Para avaliar e comparar a nossa estratégia a 4 estratégias conhecidas na literatura para a redução de suíte de testes , um caso estudo e um experimento foram realizados.

Capítulo 7

Neste capítulo, nós apresentados uma nova fora de analisar as suítes de teste reduzidas baseadas na ordem dos casos de teste. Este trabalho é inspirado em uma métria utilizada para avaliar estratégias de priorização de casos de teste. Um exemplo é apresentado para ilustrar nossa motivação. Dois estudos de caso foram realizados usando a métrica.

Capítulo 8

Este capítulo apresenta algumas pesquisas relacionadas as estratégias de seleção de casos de teste e redução suíte de testes. O foco é em soluções que podem ser automatizados, uma vez que nosso escopo é MBT e testes de sistema.

Capítulo 9

Este capítulo apresenta as considerações finais e trabalhos futuros relacionados as nossas contribuições.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta alguns conceitos básicos sobre Teste de Software, Teste baseado em Modelos, Critérios de Cobertura, Seleção de Casos de Teste, Redução de Suítes de Teste, Priorização de Casos de Teste, Abordagens baseada em Valor, Experimentação em Engenharia de Software e Análise Estatística, deixando claros a terminologia e os conceitos que serão utilizados nesta Tese.

2.1 Software Testing

Software testing is a very important activity that is part of the software development process. Since this activity can spend more than 50% of the resources of the software development [5], that is, in general, not executed properly or even skipped due to resource (cost and time) constraints [25].

There are two main reasons to execute software testing [5]: to assess the quality of the application; and to reveal problems in application under testing. Since testing is concerned with "error", "fault", and "failure", it is important to clarify these terms before presenting the other concepts about testing. According to Jorgensen [41]:

- An *error* occurs because of an incorrect or missing code;
- A *fault* or *defect* is the result of an error;
- A *failure* occurs when the fault executes, then the application does not perform the

functionality as required. This is noted when the output is wrong, an abnormal termination occurs or time restrictions are violated.

Software testing is the activity of designing tests and exercising the software with them, in order to investigate on quality attributes and find defects. We can classify the test process according to its goal [56]:

- **Defect testing:** Where the goal is to reveal faults in the software;
- **Validation testing:** Where the goal is to demonstrate to the developer and the system customer that the software meets its requirements.

Our strategies are independent of the goal of the testing process, however the case studies are focused on validation testing.

The process of software testing can be divided into [41]: test planning, test case development, running test cases, and evaluating test results. Our focus is on test case development. In the next subsections, we present a definition of test case, testing methods and level of testing.

2.1.1 Test Case

The essential task of software testing is to determine a set of test cases for testing the specific system. Each test case is associated with a system behavior, and is composed by [41]:

- **An identity:** An identifier can be associated to the test case, for testing management and requirements tracing for example;
- A set of inputs, where an input can be:
  - **Pre-condition:** The system state that must hold before test case execution;
  - **Actual inputs:** Actions that should be executed;
- Set of expected outputs, where an expected output can be:
  - **Post conditions:** The system state that must hold after test case execution;
  - **Actual outputs:** An output of the system.

In order to execute one test case, the system must hold the specific state (pre-condition). Then, the system is exercised with the inputs, collecting the outputs until the post condition is reached or a failure is detected. Finally, the obtained results are compared with the expected ones to check if the test has passed or not, i.e., if the software behaved as expected.

This is a general format of a test case. Depending on the kind or level of testing, this format is tailored. For example, a unit test is a method call, where the inputs are values that instantiate parameters. On the other hand, at a system test level is an execution scenario of the application. Then, usually, inputs are a sequence of actions executed by a user, and the respective system outputs are observed.

Figure 2.1 presents a System Test Case. This test case executes the system to validate the scenario "add phonebook contact with success". To execute this test case, the system must be in Idle, the Phonebook application must be installed in the phone and must have enough memory to add a new contact. For each input (user action) of this test case, an expected output (system state) is presented. This expected output is then compared to the real system state. If they are the same, then we "pass" the test case.

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

Input (User Action)	Output (System Response)
Press Menu Center Key	List of features is showed.
Scroll until Phonebook icon	Phonebook icon is highlighted.
Start Phonebook application.	The contact list is displayed.
Select the New Contact option.	The New Contact form is displayed.
Type the contact name and the phone number.	The new contact form is filled.
Confirm the contact creation.	A new contact is created in My Phonebook application.

Figura 2.1: System Test Case

2.1.2 Testing Methods

Testing methods are used to identify test cases. A testing method may follow a functional testing approach or a structural testing approach [5].

Functional Testing

Functional testing is based on the view that any application can be considered as a "*black box*", where only inputs and outputs are taken into consideration, and the implementation is not known. Since the implementation is not considered, only the specification is used to obtain the test cases.

For functional test cases, there are two advantages: the test cases can be obtained in parallel with the implementation, and, if there are any changes in the source code (except changes of the functionalities), the test cases do not change. In general, functional test cases may present redundancies among themselves [41], which may increase the costs of software testing.

Structural Testing

In contrast to the functional testing, the structural testing approach considers the implementation of the system to obtain the test cases. This approach is also called "*white box*", where it is necessary to observe inside the box in order to identify the test cases. In other words, to obtain the test cases, the implementation needs to be available. This approach lends to the definition and use of test coverage metrics [5]. Such coverage metrics define which part of software will be tested.

2.1.3 Level of Testing

Beizer [5] and Jorgensen [41] show three levels of testing: unit, integration, and system testing. Each level has a different goal, and thus different methods are applied to perform the test.

Unit Testing

A unit is the smallest testable piece of an application, that is usually the work of one programmer. Unit testing is performed to guarantee that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [5].

A component can be considered an unit. This way, each component/subsystem is tested separately.

Integration Testing

Integration is a process by which components are put together to create a larger component. The goal of integration testing is to reveal faults that arise when the components are put together. This testing considers that each component has already been tested and are individually satisfactory, as demonstrated by a successful passage of component tests.

The hard task in integration testing is to locate the "faults". Usually, to make that easier we should use an incremental approach to system integration and testing [56].

System Testing

A system can be consider a big component. Then, a system testing is performed when all components are already together(i.e., after the integration occurs). The goal of system testing is to reveal issues and behaviors that can only be exposed by testing the entire integrated system.

This testing focus on capabilities and characteristics that are presented only with the entire system. System scope can be classified by the kind of conformance [10]:

- **Functional:** The goal is to find errors in the functionality of the system, in other words, this testing assess if for given inputs, the right outputs are generated;
- **Performance:** The goal is to observe the behavior of the system under heavy load;
- **Stress or load:** The goal is to find failures in the system under unexpected inputs, unavailability of dependent applications, and hardware or network failures.

Our strategies are independent of the level of testing, however the case studies are focused in functional system testing.

2.2 Model-Based Testing

Model-Based Testing (MBT) is a functional approach and consists in the automatic generation of tests using models extracted from the system specification [60]. For its application, it is necessary that the software requirements are precisely defined, in order to characterize with exactness the system behavior [5].

This section presents concepts about models (Subsection 2.2.1) and activities of Model based Testing (Subsection 2.2.2).

2.2.1 Models

System Models are an abstract view of a system. Since a model is an abstraction, it is a simplification of the reality that highlights the most important characteristics [37].

Models may represent a system from different perspectives [56]:

- **External:** The environment of the system is represented by the model;
- **Behavioral:** The behavior of the system is represented by the model;
- **Structural:** The architecture of the system is represented by the model.

Since model-based testing is a black-box approach, the behavioral perspective of the system is adopted. For behavioral perspective, the following models can be highlighted: Decision Tables, Finite State Machines (FSM), Markov Chains, Statecharts, UML diagrams, Labelled Transition System (LTS), among others. A model is chosen according to the characteristics of the system. In this work, we consider the specific type of LTS model - ALTS (Annotated LTS).

Labeled Transition System - LTS

LTS is a directed graph in which vertices are named states, and edges are named transitions. These models are largely used as the semantic formalism of several specification notations [40] and so they can be easily obtained from functional specifications by using translation tools, such as UMLAUT [38]. Several tools use LTSs as the model for obtaining test cases. Among these tools are: SPACES [3], TGV [40], LTS-BT [16] and TaRGeT [47].

Formally, an LTS can be defined as a 4-tuple $S = (Q, A, T, q0)$, where [26]:

- 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;
- q_0 is the initial state.

Usually, LTS take into account internal and external actions [40]. Since our focus is on functional testing, we show two different LTS that can be used for modeling the functional behavior of the applications: Input-Output LTS and Annotated LTS.

Annotated LTS - ALTS.

Annotated LTS (ALTS) is an LTS that has transitions actions and also, annotations. These annotations are insert in the LTS with a specific goal. In our case, this goal is to generate functional test cases, therefore, such annotations are related to this activity. Figure 2.2 shows an example of an Annotated LTS model that represents the behavior of an application where the user wants to save one phone number that is embedded in a message.

Observe that each label has an annotation for the action: *steps*, *conditions* or *expectedResults*. These correspond to, respectively, a user action, a pre-condition and a system response, and are inserted in the LTS to facilitate the test case generation.

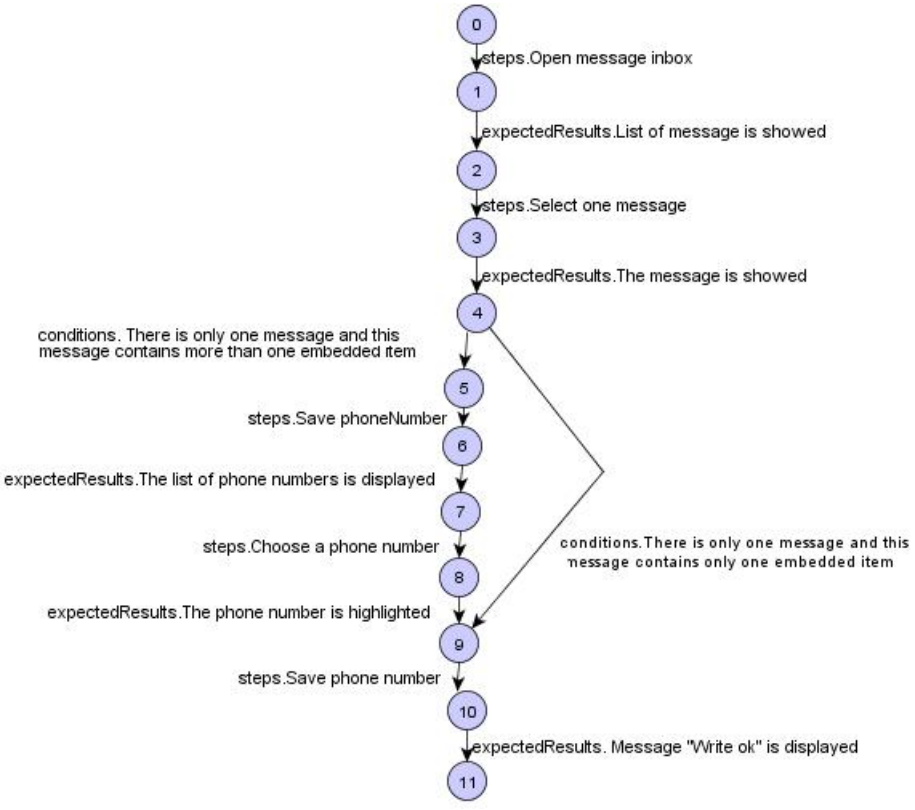


Figura 2.2: Annotated LTS model

2.2.2 Activities of MBT

The activities related to MBT can be described as follows [27]:

1. **Build the model**: The formal model is built from the software specification. This model needs to be formal, i.e., precise, consistent and unambiguous.

2. **Generate expected inputs and outputs**: The test inputs and outputs are generated from the formal model. In order to exercise the system, we need to generate the sequence of the inputs, while the expected outputs represent the expected system responses.

3. **Run tests**: The system is executed with the generated inputs, generating outputs;

4. **Compare outputs with expected outputs**: The generated outputs are compared to the expected outputs.

For example, consider we have the following requirement: *The user must be able to save a phone number that is embedded in a message*. From this requirement, we build the model (activity 1) and obtain the ALTS shown in Figure 2.2.

Using the ALTS showed in Figure 2.2, we can transverse this model by using Depth Search First (DFS) and generate 2 test cases (path coverage criteria) and its respective inputs and outputs (activity 2). The test cases to be generated from a model depends on the adopted coverage criteria (more details in Section 2.3). Figure 2.3 shows one of them.

Pre-Condition: There is only one message and this message contains only one embedded item

Input (User Action)	Output (System Response)
Open Message Inbox	List of messages is showed.
Select one message	The message is showed.
Save Phone Number	Message "Write ok" is displayed

Figura 2.3: Test Case

Figure 2.4 shows the flow of the MBT activities. As can be seen, the models are obtained from the requirements. This activity is usually done manually, and it requires a specialist

in the notation used to construct the formal model. However, there are already attempts, in practice, to automatically generated models from requirements specification written in a natural controlled language [47].

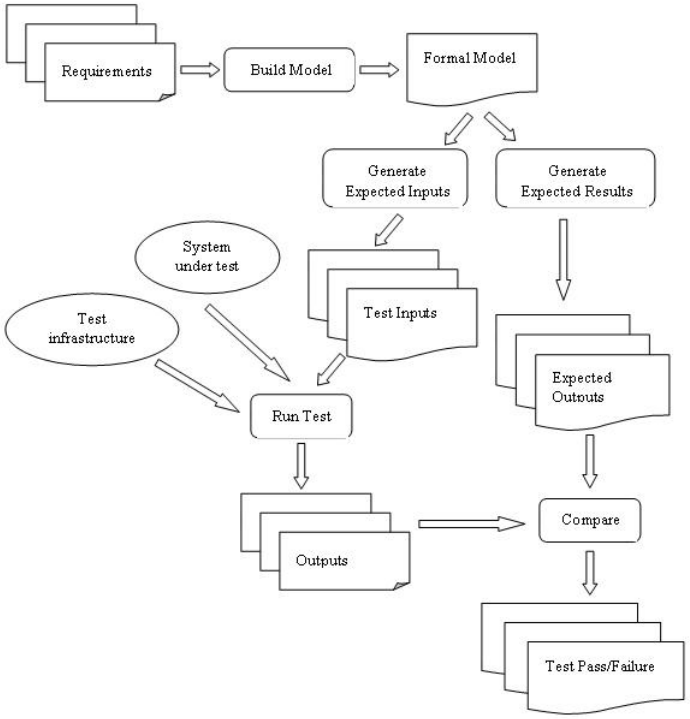


Figura 2.4: Model Based Testing

Inputs to execute the system and expected outputs are extracted from that model. Then, the system is executed with the tests (activity 3). When the system is executed, the outputs are produced. Finally, these outputs are compared to the expected outputs and the test cases are defined as Pass or Fail.

2.3 Coverage criteria

A coverage criterion is a set of rules that imposes test requirements on a test suite [2]. Coverage criteria specify the items of the system that must be exercised during testing. There are two purposes [60]:

- **Measuring the adequacy of a test suite:** The coverage level of a specific criterion is an indicator of the quality of the test suite;

- **Deciding when to stop testing:** The tests are run until reaching a coverage level of a specific criterion.

There are consolidated coverage criteria for code coverage (white-box coverage criteria), and many of these coverage criteria are used for black-box coverage [60]. Since our focus is on MBT and the model is LTS, we will present the main coverage criteria used for Transition-Based Coverage Criteria: *all-states, all-configurations, all-transitions, all-transition pairs, all-loop-free-paths, all-one-loop-paths, all-round-trips and all-paths* [60]. Here, we do not consider *all-configurations* because it is not applied to our context, since it is mostly used for *statecharts*.

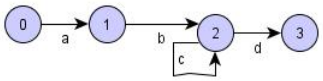


Figura 2.5: LTS model

- **All-states coverage**: Every state must be visited at least once. Considering the LTS of Figure 2.5, to reach this coverage, only one test case is required: abd;

- **All-transitions coverage**: Every transition must be visited at least once. Observing the LTS of Figure 2.5, this coverage can be reached using only one test case: $abcd$;

- **All-transition-pairs coverage**: Every pair of adjacent transition in the model must be traversed at least once. In the LTS of Figure 2.5, to reach this coverage, we need to have test cases that traverse ab, bc and bd at least once. In this case, this coverage is reached with the test cases: abd and abc;

- **All-loop-free-paths coverage**: Every loop-free path must be traversed at least once. A path is loop free when it does not have repetitions. For the LTS in Figure 2.5, only one test case is required to reach this coverage criterion: abd;

- **All-one-loop-paths coverage**: Each path is traversed at most once the loop. Therefore, we have one test for each loop. In Figure 2.5, we need two test cases to reach this coverage criterion: abd and $abcd$.

- **All-round-trips coverage**: This coverage criterion is similar to *all-one-loop-paths* since it requires that all loops are tested in the model, however it is a weaker criterion, since it only requires one path for testing one loop. For the LTS in Figure 2.5, to reach this coverage criterion, the following test cases are needed: abd and abc.

- **All paths coverage**: Every path must be traversed at least once. This corresponds to an exhaustive testing in LTS models, and the generation algorithm should have an heuristic to avoid the state space explosion, enabling the proper use of this coverage.

2.4 Test Case Selection

Test case selection is an activity to select a subset of the test suite according to a specific criterion, for example transitions or requirements coverage. The selected subset may not provide the same coverage as the original test suite [65], however, it may lower the costs of the test process. We can enumerate some strategies for test case selection:

1. **Deterministic:** Where we have a manual choice. For example, one specialist can use his or hers know-how to select test cases;

2. **Random:** The subset of test cases is randomly chosen;

3. **Statistical:** The weights are assigned to guide the choice [4]. One of them is the **Guided Random**, where the choice of the test cases is guided by probability values [49; 3]. For each decision node in an LTS, a probability is assigned. Then, this strategy tries to select the most important transitions (i.e., the transitions with the highest probability values) using a Depth-first search. The goal is to define an unbiased test suite that can be more effective for fault detection, and also to make reliability estimation possible;

4. **Test Purpose:** A test purpose denotes a scenario of a functionality of the system under testing [37]. Jard and Jéron present TGV tool [40], where the test selection is performed when the model is delimitated by the test purpose.

2.5 Test Suite Reduction

Test suite reduction is a technique that produces a representative subset of the original test suite. This subset has equivalent coverage in relation to the original test suite, concerning a specific criterion [65]. This problem can be stated as follows [35]:

Given: Test Suite TS, a Set of Test Requirements $req_1, req_2, ..., req_n$ that has to be covered to provide the desired test coverage of the program, and subsets of TS ($TS_1, TS_2, ..., TS_n$), where each test case from TS_i can be used to test req_i;
Problem: Find a representative set of test cases from TS that satisfies all of the $Req's$.

A test requirement can be a statement, a block, a decision, a requirement and so on. A representative set of test cases must have at least one test case for each Req. Therefore, these test cases satisfy all of the $Req's$. The maximum reduction occurs when the smallest representative subset is found. This is a *NP-complete* problem [21; 35].

The main advantage of test suite reduction is the reduction of the size of the test suite. However, since the test suite is reduced, we risk to decrease the capability of faults detection. Wong et al. [64] and Rothermel et al. [53] propose experimental researches to investigate this risk.

Finally, test suite reduction techniques deal with structural redundancy. The classic definition for redundant test case is: A test case is redundant if other test cases in the test suite provide the same coverage of the program [35]. For example, considering branch coverage as a test requirement, a test suite reduction strategy must find a subset that reaches 100% branch coverage.

Some heuristics for test suite reduction were proposed in the literature. These heuristics are detailed below, since they are compared to our proposed strategy (Chapter 6). For applying the proposed heuristics, it is necessary to have the satisfiability relation between the Test Suite (TS) and the Test Requirements ($Req = req_1, req_2, ..., req_n$). First we illustrate the satisfiability relation required to perform the test suite reduction with the heuristics.

Satisfiability Relation

For each req_n, there is a subset of TS ($T_1, T_2, ..., T_n$), such that all the test cases belonging to T_n can be used to test req_n. Table 2.1 presents a sample of a Satisfiability Relation.

Tabela 2.1: TS consists of Test Cases $t_1, ..., t_7$, Test Requirement req_n, and Associated Testing Sets are T_n - Example 1

n	Req_n	TS_n
1	req_1	$\{t_2\}$
2	req_2	$\{t_6, t_7\}$
3	req_3	$\{t_1, t_5, t_7\}$
4	req_4	$\{t_1, t_6\}$
5	req_5	$\{t_3, t_4, t_7\}$
6	req_6	$\{t_1, t_2\}$
7	req_7	$\{t_3, t_7\}$

The goal of test suite reduction, as said before, is to meet a subset (Reduced Set - $RS \subseteq TS$) that provides 100% coverage of test requirement, in other words, to satisfy *all* test requirements.

Greedy Heuristic

The Greedy heuristic [22; 24] repeatedly selects the test case t that satisfies the maximum number of unsatisfied test requirements, if there is a tie situation, a random choice is made. The selected test case is added to the Reduced Set (RS) and all test requirements that can be satisfied by that test case are marked as an already satisfied test requirement. This algorithm stops when all test requirements are satisfied. Applying this algorithm to the example showed in Table 2.1, we have:

t_7 satisfies the maximum number of unsatisfied test requirements, then $RS = \{t_7\}$, the requirements req_2, req_3, req_5 and req_7 are marked as satisfied. Now, there is a tie situation: t_1 and t_2 satisfy the maximum number of unsatisfied test requirements. This way, a random choice is made. Considering that t_1 is chosen, then $RS = \{t_1, t_7\}$, the requirements req_4 and req_6 are marked as satisfied.

Finally, the unique requirement that has not yet been marked is req_1, since this requirement is satisfied by t_2, therefore $RS = \{t_1, t_2, t_7\}$, which provides 100% coverage of the test requirements. Rehman et al. proposed TestFilter, a technique to reduce test suites based on

statement coverage [59]. This technique uses the Greedy Heuristic for test suite reduction. The authors used statement coverage as test requirement.

Heuristic Greedy - Essential (GE)

This heuristic (defined by Chen and Lau) is based on [19]:

- **Essential strategy:** Responsible for selecting all essential test cases. A test case is essential when only that specific test case covers one specific requirement;
- **Greedy heuristic:** Responsible for selecting a test case that satisfies the maximum number of not yet satisfied requirements.

Initially, all essential test cases are observed, and their respective requirements are marked. Then, the greedy heuristic is applied. The focus is on solutions that can automated since our scope of study is on MBT. Using the example from Table 2.1, the strategy execute as following.

First, t_2 is chosen, since it is an essential test case. Therefore $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now that we do not have essential test cases remaining, we must apply the greedy heuristic. Since t_7 satisfies the maximum number of unsatisfied test requirements, we add it to the reduced set. Thus, $RS = \{t_2, t_7\}$, and the requirements req_2, req_3, req_5 and req_7 are marked as satisfied.

Finally, the unique requirement that has not been marked yet is req_4. Therefore we apply a random choice between t_1 and t_6. Considering that t_1 is chosen, we obtain the reduced set $RS = \{t_1, t_2, t_7\}$ and req_4 is marked as satisfied. Now that all requirements are satisfied, the algorithm stops.

Heuristic Greedy - $1 - to - 1$ - Redundancy Essential (GRE)

This heuristic (defined by Chen and Lau) is based on [18]:

- The **Greedy** and **Essential** strategies, both presented above;
- **1-to-1 redundancy strategy:** A test case $t_{1-1} \in TS$ is said 1-to-1 redundant, if [20]:

$$\exists t \mid t \neq t_{1-1} \;\&\; t \in TS \;\&\; req(t_{1-1}) \subseteq req(t).$$

In other words, when all requirements satisfied by t_{1-1} are also satisfied by t.

The essential and 1-to-1 strategies are applied alternatively, until no essential or 1-to-1 redundant test cases can be found. That means that the greedy strategy is only applied if neither the essential or 1-to-1 redundancy can be applied. Considering the example from Table 2.1 the algorithm is applied as following.

First, t_2 is chosen, since it is an essential test case. Therefore, $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now, we do not have any other essential test cases. Thus, we have to search for 1-to-1 redundant test cases.

During this search we identify t_4, t_3 and t_5 as 1-to-1 redundant test cases, since: $req(t_4) \subseteq req(t_7)$, $req(t_3) \subseteq req(t_7)$ and $req(t_5) \subseteq req(t_7)$. The test cases t_3, t_4 and t_5 are not considered, since those are redundant in relation to t_7.

Now, at this point, t_7 becomes an essential test case, and needs to be placed in the reduced set. Thus, $RS = \{t_2, t_7\}$, and the requirements req_2, req_3, req_5 and req_6 are marked as satisfied. Finally, we have only req_4 as not satisfied. A random choice is performed, between t_1 and t_6. Considering that t_1 is chosen, the resulting subset is $RS = \{t_1, t_2, t_7\}$ and req_4 is marked as satisfied. Since all the requirements are satisfied, the algorithm stops.

Heuristic H

Harrold et al. [35] present a test suite reduction technique, refered as Heuristic H. Each test requirement has a cardinality, that is the number of test cases that covers that specific requirement. When a test case is added to the reduced set, all requirements covered by that test case are marked. The first step is to identify the test requirement(s) with the lowest cardinality, since they represent the most essential test cases.

Among the unmarked test requirements with the lowest cardinality, the algorithm selects the most frequently occurring test case, i.e., the test case that covers most requirements. If there is a tie, the algorithm chooses the test case that occurs most frequently at the next higher cardinality and so on (if there is another tie where the cardinality is maximum, then a random choice is applied). This algorithm stops when the reduced set has test cases that cover all test requirements.

Summarizing, the main idea is to select test cases according to their essentialness, i.e.,

Tabela 2.2: Cardinality

Cardinality	Req
1	req_1
2	$req_2, req_4, req_6, req_7$
3	req_3, req_5

keeping in the reduced set the test cases in the order from the most essential to the least essential. Below, we apply this algorithm to the example showed in Table 2.1.

First, we need to calculate the cardinality of each test requirement. The results can be seen in Table 2.2.

For the lowest cardinality (in this case is one), there is only one test case (t_2). Thus, $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now, the lowest cardinality is two, tied between req_2, req_4 and req_7. Also there is a tie between the most frequent test cases within req_2, req_4 and req_7. These test cases are t_6 and t_7.

Then we must see the next higher cardinality (in this case is 3 - req_3 and req_5) to determine which one of them occur most frequently. Observing req_3 and req_5, we identify t_7 as the most frequent test case. Therefore t_7, is chosen and then $RS = \{t_2, t_7\}$, also req_2, req_3, req_5 and req_7 are marked as satisfied.

Finally, the unique requirement that has not been yet marked is req_4. Again, we have a tie situation, however we do not have a next higher requirement cardinality, therefore we apply a random choice between t_1 and t_6. Considering that t_6 is chosen, the reduced subset is $RS = \{t_2, t_6, t_7\}$, and req_4 is marked as satisfied. Since all requirements are marked, and thus satisfied, the algorithm stops.

2.6 Test Case Prioritization

Test case prioritization is a technique that orders test cases in an attempt to maximize an objective function. The problem is defined by Elbaum et al. as follows [29]:

Given: A test suite TS Test Suite; PTS, a set of permutations of TS; and, f, a function that maps PTS to real numbers ($f : PTS \rightarrow \mathbb{R}$).

Problem: Find a $TS\prime \in PTS \mid \forall\, TS\prime\prime\, (TS\prime\prime \in PTS)\, (TS\prime\prime \neq TS\prime) \cdot f(TS\prime) \geq f(TS\prime\prime)$

The objective function is defined according to the goal of the prioritization. The manager may need to quickly increase the rate of fault detection or the coverage of the source code. Then, a set of permutations PTS is obtained and the $PTS\prime$ that has the highest value of $f(TS\prime)$ is chosen.

Note that the key point is the goal, and the success of the prioritization is measured by this goal. However, it is necessary to have some data (according to the defined goal) to calculate the function for each permutation. Then, for each test case, a priority is assigned and test cases with the highest priority are scheduled to execute first. When the goal is to increase fault detection, there is a metric largely used in the literature, named Average Percentage of Fault Detection – APFD. The highest the APFD value is, the faster and better the fault detection rates are [29].

2.7 Value Based approach

Value based software engineering has been introduced in 1981 with Boehm's Software Engineering Economics book [12] and has inspired the value based management movement in the early 1990 [9]. Its philosophy is that "*quality should not be a goal in itself in the absence of favorable economics*".

Since then, the consideration for software-related value has expanded its scope and values have been incorporated deeply into successful developments process. The common purpose has been promoting the different considerations/measures/knowledges to the foreground so that the software engineering decisions could be guided and optimized by these values.

Saying exactly once for all what "*value*" represents in this discipline is not possible. The referred numbers can be related to various topics. They can represent the economical and financial aspect, they can be percentage or probability, they can represent abstract concepts as quality, availability, usability and so on.

In software development, generally the various system functionalities do not have the same "importance" for overall system performance or dependability, and the testing effort should be planned and scheduled accordingly. Different criteria can be adopted in order to define what "importance" means for test purposes, e.g., component complexity, or usage

frequencies (such as in reliability testing [45]).

Often, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers. Several criteria for assigning the importance factors could be adopted. Obviously this aspect in the proposed approach remains highly subjective, more in the realm of expert judgment than mechanizable methods.

The basic idea is that the test managers must explicit these criteria. The main task is to express, for each functionality, a value belonging to the [0,1] interval, representing its relative "importance" with respect to the other functionalities. This value, called the *weight*, must be assigned in such a manner that the sum of the weights associated to all children of one level is equal to 1; the more critical a functionality is, the greater its weight.

It is worth noticing that the process of functionalities annotation implies a beneficial side-effect: for assigning the appropriate values, the managers are forced to reflect on the relative complexity of each functionality with respect to the context in which it is inserted. Consequently, they focus on the parts where problems could be more critical and become more aware of the importance of each node for the system development.

2.8 Experimentation in Software Engineering

In this section, some basic concepts about Scientific Methods and Experimentation in Software Engineering are presented.

2.8.1 Scientific Methods in Software Engineering

There are four scientific methods that are used for doing research in software engineering. Those methods are [32; 61]:

- **Scientific:** A model is built by observing the world;
- **Engineering:** New solutions are proposed, and evaluated, from changes of a current solution;
- **Empirical:** A model is proposed and evaluated through empirical studies;
- **Analytical:** A formal theory is proposed and compared with empirical observations.

Here, we will address the empirical method. An empirical study can be conduct through:

- **Survey:** The goal is to obtain descriptive and explanatory conclusions [61] from a sample. One sample in a representative part of a population. The data used in the analysis are gathered, usually, through interviews or questionnaires. It is not possible to manipulate variables;
- **Case Study:** Data is collected for a specific purpose. Normally, a case study is used for monitoring projects or activities. From the obtained results, the statistical analysis can be applied;
- **Experiment:** It is a rigorous, formal and controlled investigation. Normally, it is executed in a laboratory environment. It is possible to manipulate variables.

The next subsection shows in details the flow of an experiment. We show elements from a process that defines each step required to perform the experimental study.

2.8.2 Experiment

In this work, we use a process for experimental studies in software engineering defined by Wohlin et al. [61]. This process is composed of the following activities: definition, planning, operation, analysis and interpretation, presentation and package. Each one of these activities are detailed below.

Definition

In this phase, the experiment is defined in terms of a problem, an objective and a goals. It is required to specify a general hypothesis relating the goal of the experiment, with the problem being addressed. In order to define the goal, the key questions proposed by Wohlin *et al.* should be answered[61]:

1. **What is studied?** Object of study: this is the entity that is studied in the experiment;
2. **What is the intention?** Purpose: intention of the experiment;
3. **Which effect is studied?** Quality focus: primary effect under study;

4. **Whose view?** Perspective: viewpoint from which the results are interpreted;

5. **Where is the study is conducted?** Context: environment which the experiment is run.

From these answers, a goal definition template is filled. This template helps organizing the main elements of the experiment, and provides a general overview of the goal and purpose of the experiment. The template is structured as follows:

Analyze *object of study*
for the purpose of *purpose*
with relation to *quality focus*
from the point of view of the *perspective*
in the context of *context*.

The environment defines the personnel involved in the experiment (subjects) and the software artifacts used in the experiment (objects). It is necessary to define the quantity, priority, know-how for each subject and quantity, size, complexity and application domain for the objects.

Planning

Once the experiment is defined, the experiment design is specified. In order to properly plan the experimental study, it is required to specify several elements, such as [61]: context selection, variable, hypothesis, design, instrumentation and threats.

Context Selection

Aiming to have more general and real results, it is necessary that the experiment is executed by professional staff in large and real software projects [61]. However, this scenario is costly. In order to reduce the costs, the project can be run off-line, being performed by students and using toys (e.g. simple models, or software application) in a specific context. Thus, the context of the experiment can be classified in four dimensions [61]:

- On-line vs. Off-line
- Student vs. Professional
- Toy vs. Real Problems

- Specific vs. General

Variables Selection

One of the main elements of the experiments are the variables. These variables comprise the elements that are modified, observed, analyzed and executed during the experimental study. The variables that will compose the experiment are defined as following:

- **Dependent:** Variables that will be observed in the experiment;
- **Independent:** Variables that will be controlled in the experiment.

Hypothesis Formulation

The experiment definition is formalized into hypotheses, that will be tested during the analysis of the experiment. The hypotheses testing is the basis for the statistical analysis of an experiment. The experiment definition is formalized as following:

- A null hypothesis, H_0: This is the hypothesis that the experimenter wishes to reject under a specific significance level;
- An alternative hypothesis, H_1: This is the hypothesis that the experimenter wishes to accept.

Selection of Subjects

The subjects of an experiment are the people involved in it. This is a very important step, since depending on the selection of the subjects, the experiment can be generalized. The larger the sample (of subjects), the lower the error becomes when generalizing the results.

Experiment Design

The Experiment design is defined from the characteristics of the experiment, such as: amount of object, subjects, factors and levels. [61; 39]. The design types are suitable for experiments with:

- **One factor with two treatments:** To compare two treatments;
- **One factor with more than two treatments:** The comparisons between more than two treatments;

- **Two factors with two treatments:** It is necessary to compare the treatments in each factor with the others (2*2 factorial design);

- **More than two factors with more than two (k) treatments:** It is necessary to compare the treatments in each one of the factors with those from the others (2^k factorial design);

From the Experiment design, the statistical resources and the number of replication are defined. The number of necessary replications (n) can be calculated using the following formula [39]:

$$n = \left(\frac{100 \cdot Z \cdot s}{r \cdot \overline{x}}\right)^2 \tag{2.1}$$

Where Z, for a 95% confidence level, is 1.96 (a standard value from the normal distribution table); s is the standard deviation from the sample; r is the desired accuracy; and $\overline{x}$ is the mean of the sample.

Instrumentation

The instrumentation comprises the elements used to automate and execute the experimental study. These elements are called instruments. During this step, three types of instruments are specified [61]:

- **Objects:** The artifacts used to execute the experiment (e.g., specification models, implementation, among others);

- **Guidelines:** The guidelines are required to properly guide the subjects in the experiment;

- **Measurements:** The method in which the data will be collected.

Threats to the Validity

Usually, threats to validity are identified during the planning phase. This is an important step because if the data are not valid, the obtained conclusions of the experiment can not be trusted. The sample needs to have an adequate validity for the population, therefore, any threat to validity need to be considered.

There are different types of validity, each one related to a specific aspect (theory, implementation, observation, among others) of the experiment. The validity can be classified in [23]:

- **Internal:** Related to the relationship between the treatments;
- **External:** Related to the ability to generalize the results;
- **Construct:** Related to the experiment setting;
- **Conclusion:** Related to draw correct conclusion about an experiment.

Each type of validity must be addressed by the experimenter, and it is necessary to identify each elements that threats the validity of the experiment. A validity threat that is not properly handled by the experiment also threats the experimental study itself. Therefore, specifying a proper validity evaluation method is one of the main elements to evaluate and validate the experimental study.

Operation

In this phase, the experiment is executed, and the measurements are collected. This is divided:

- **Preparation:** To prepare the subjects/material to collected data;
- **Execution:** The execution of the experiment is performed;
- **Data Validation:** To make sure the collected data are valid.

Analysis and Interpretation

The collected measurements are analyzed by using descriptive statistic. The interpretation is done by determining, from the analysis, if the null or alternative hypothesis are accepted or rejected. The statistical resources must be properly use, in order to avoid validity threat concerning the conclusion of the results. Therefore, it is necessary to analyze each data, and each sample obtained during the execution before using a specific statistical test (specially the parametric ones, such as Analysis of Variance, and t Test).

Presentation and Package

After analyzing the results of the experiment, the conclusions and artifacts of the experiment must be organized and be presented available, so they can be properly presented to other researches. Therefore, during this step, the entire process is organized in reports, and the data, statistical resources, should be organized in graphics and other visual resources, to ease the understanding of the performed experiment.

2.9 Statistical Analysis

In this section, some concepts about descriptive statistic, graphical visualization and hypothesis testing are presented. These resources are presented in order to provide a better understanding of the analysis performed in this work.

2.9.1 Descriptive Statistic

Descriptive Statistics is used to describe and show - graphically - characteristics of the data set. The goal is to know the data distribution (to identify abnormal data points). Usually, this is done before performing the hypothesis testing. These statistic can be measures of a central tendency, or dispersion.

The measures of central tendency provide an overview to estimate an stochastic variable. There are three measurements often used to indicate the central tendency of a data set(x) [61]:

- **Mean $\overline{x}$:** It is the sum of the values divided by the number of values;
- **Median:** It is the numeric value separating the higher half of a sample (considering the ordered data set);
- **Mode:** It is the value that occurs most frequently in a data set.

In turn, the measures of dispersion show how much variation there is from the mean. There are two measurements often used to indicate the dispersion of a data set [61]: **Variance** and **Standard Deviation** (s). A low value of variance or standard deviation indicates that

the data points tend to be too close to the mean, whereas a high value indicates that the data is spread out. The difference between **Variance** and **Standard Deviation** is that the latter is expressed in the same $unit$ as the data, whereas the variance is expressed in $(unit)^2$.

2.9.2 Graphical Visualization

By representing some measures graphically, we are, usually, able to draw conclusion about the data. A box plot shows the dispersion of a sample and the central value. An example of a box plot is illustrated is presented in Figure 2.6. The picture shows the first and third quartiles (respectively, the upper and lower edges of the box), and the mean value that is represented by the central line in each box. The whiskers extending from the quartiles represent the farthest observation lying within 1.5 times the interquartile range. The outliers (unfilled dots) represent the individual values beyond the whiskers.

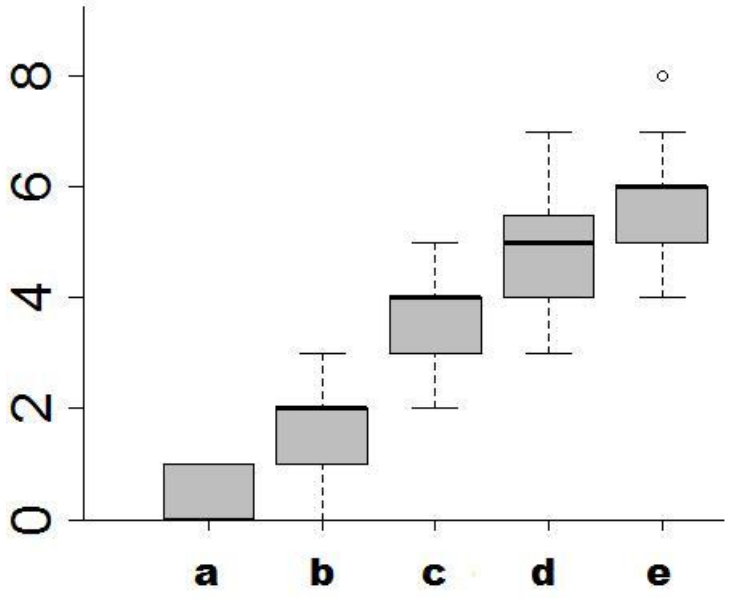


Figura 2.6: Sample of Box Plot

A confidence interval (CI) provides an estimated range of values which is likely to include an unknown population parameter. The estimated interval is calculated from a given set of sample data with a chosen confidence level. Thus, for different set of data different CI are calculated and plotted (see Figure 2.7).

We are able to use the CI from two or more samples to determine if these samples come

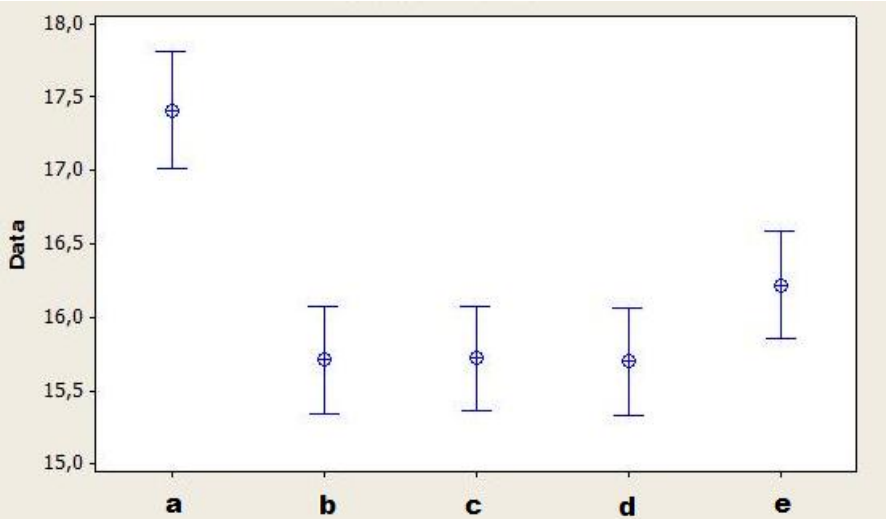


Figura 2.7: Confidence Intervals - a, b, c, d and e

from the same population. If there is no overlap among the CIs, we can conclude that the population are different and, if a hypothesis tests is performed, we are able to reject the null hypothesis (see next subsection), according to the specified confidence level.

Observing the intervals in Figure 2.7, we can state that “a” is different from “b”, “c”, “d” and “e”. However, we can not state anything about “b”, “c”, “d” and “e”, therefore further statistical investigation needs to be done.

2.9.3 Hypothesis Testing

The goal of Hypothesis Testing is to check if it is possible to reject a null hypothesis, H_0, based on a sample from some statistical distribution. Since from the graphical visualization we are not able to reject the null hypothesis, then further statistical investigation needs to be done (applying more statistical tests), aiming to obtain more significant conclusions.

First, the data distribution of the sample is checked by applying tests that investigates distributions, such as Anderson-Darling or Kolmogorov-Smirnov [39].Depending on the data distribution, the Hypothesis Testing can be classified as [61]:

- **Parametric:** The data set presents a known distribution (e.g., normal distribution). Thus, mean and standard deviation of the sample is used to calculate the results;
- **Non-parametric:** The data set does not present a known distribution.

The choice of an adequate test is done by observing the data distribution and the experimental design. Different statistical tests are presented in Table 2.3.

Tabela 2.3: Statistical tests for different Experimental Designs and data distribution

.

Experimental Design	**Parametric**	**Non-parametric**
One factor with two treatments	$t-test$	Mann-Whitney
One factor with two treatments (paired comparison)	Paired $t-test$	Wilcoxon
One factor with more than two treatments	ANOVA	Kruskal-Wallis
More than one factor	ANOVA	

For interpreting the statistical tests results, it is necessary to observe the resulting $p-value$ of the applied test. This value is compared to the significance level (α) in order to decide if it is possible reject the null hypothesis with the specified confidence level.

2.10 Concluding Remarks

In this chapter some important concepts were presented. According to these concepts, our testing method is functional, comprising MBT approaches that use LTS as a model are considered. The next chapters present the proposed strategies and their respective evaluation and analysis using case studies and experiments. The main variable analyzed in our case studies and experiments is transition coverage, since this variable provides an adequate overview of functionality coverage. The statistical analysis is performed through hypothesis testing.

Capítulo 3

Similaridade

Este Capítulo apresenta o problema da redundância. Para ilustrar o problema, nós apresentamos uma parte de um modelo de uma aplicação real (*Phonebook*). Depois disso, nós apresentamos a nossa função de Similaridade que é responsável por calcular a distância entre dois casos de teste (Seção 3.2), seguida da matriz de similaridade (Seção 3.3) e algumas conclusões (Seção 3.4).

A partir da matriz proposta (que calcula a similaridade entre casos de teste), nós propomos 3 estratégias (2 para seleção de casos de teste e 1 para redução de suítes de teste) que serão apresentadas nos próximos capítulos (4, 5 e 6). Maiores detalhes são apresentados nas próximas seções.

3.1 Redundancy

Model-based testing is an approach that has become popular [48], however the test suites generated from MBT approaches, usually, contain a considerable degree of redundancy among test cases. Our redundancy concept considers that two test cases are redundant if they cover the same set of functionalities and present the same fault capability. Therefore, one of them can be discarded without significantly impacting the coverage and fault detection. Thus, a test case is considered redundant, if it can be discarded of the test suite without significantly affecting the fault detection and coverage of functionalities.

Additionally, if there are not redundant test cases and it is still necessary to eliminate some test cases to meet the constraints (money and time), the degree of redundancy among

test cases can be observed. The higher is the degree of redundancy among test cases, probably the similar the coverage of functionalities and fault detection capability are.

To better understand, observe the LTS model presented in Figure 3.1. This LTS presents part of the behavior of a real phonebook application. This part is about "adding" a new contact and the LTS illustrate the different flows of execution that can be considered. 3.1.

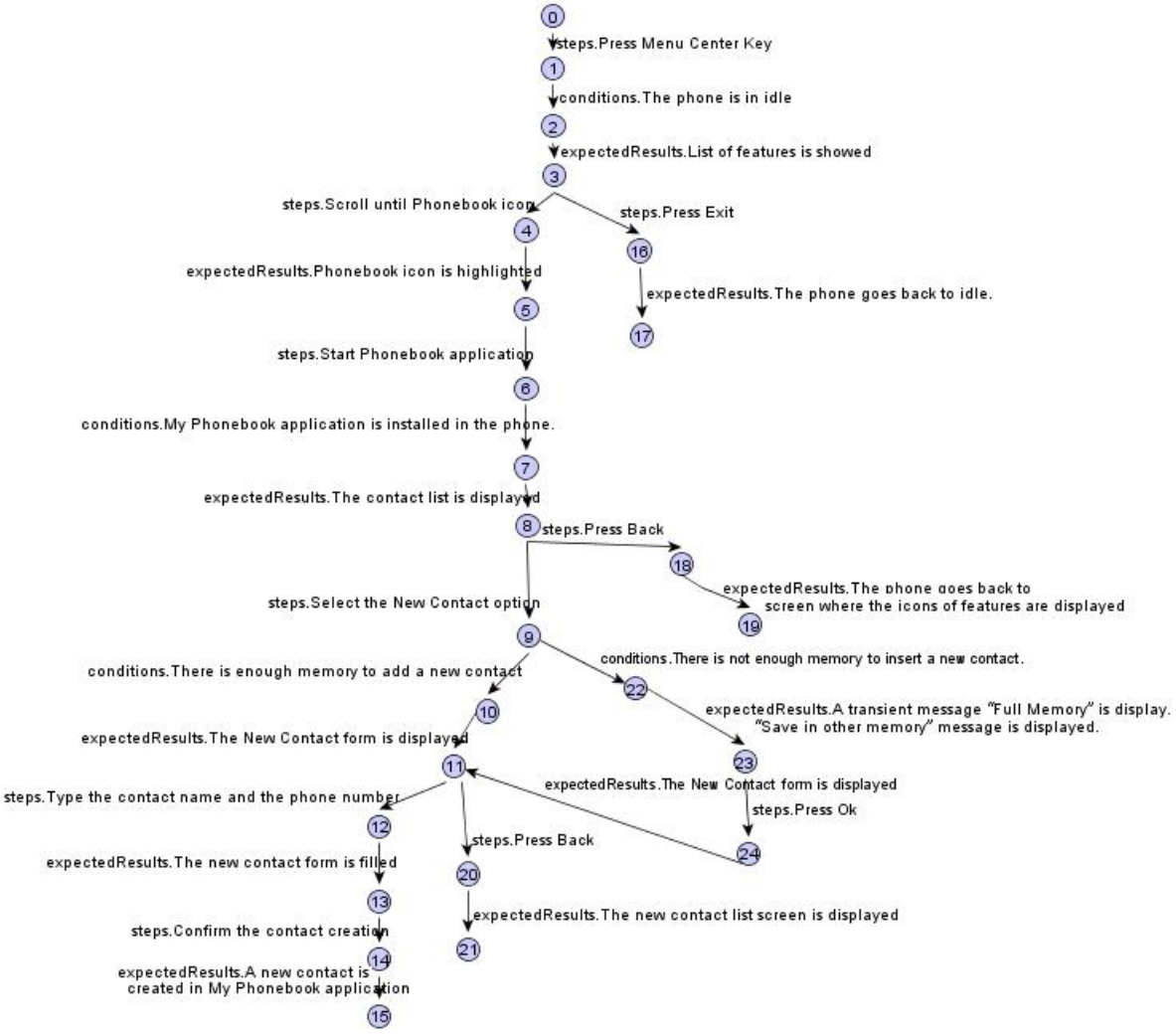


Figura 3.1: LTS Behaviour Model - Phonebook

By following this sequence (using *Depth First Search* algorithm), we obtain 6 test cases. The Table 3.1 shows these test cases generated from LTS model presented in Figure 3.1 (for the sake of simplicity, for each test case, we show the sequence of states that are covered, meaning that the correspond action/response between two states have been executed/produced).

Tabela 3.1: Test Cases generated from LTS model presented in Figure 3.1 and their respective lengths

Test Case Id	Test Case	Length
TC1	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	15
TC2	0 1 2 3 4 5 6 7 8 9 10 11 20 21	13
TC3	0 1 2 3 4 5 6 7 8 9 22 23 24 11 12 13 14 15	17
TC4	0 1 2 3 4 5 6 7 8 9 22 23 24 11 20 21	15
TC5	0 1 2 3 4 5 6 7 8 18 19	10
TC6	0 1 2 3 16 17	5

From the main flow, there are alternative flows that characterize the behavior of the feature. Therefore, the test cases will differ mostly by a step of input and output. In this sample, the main flow is represented by the path 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15. This path represents the addition of a contact.

In some points of the main flow, the user of the application can chose to proceed to an alternative flow. For example, when the contact list is displayed (state 8), the user of the application can choose to "Press back" (alternative flow) and goes back to the previous screen (8 18 19) or choose to "Select a new contact option (state 9).

Observe that all test cases present the same initial transitions, by following the sequence *0 1 2 3*. Then, there is a certain degree of redundancy among all test cases, since 3 transitions of both sequences are the same. In this case, the redundancy is one step composed by the three elements (a user action, a condition and the system response).

We are able to observe that among TC1, TC2, TC3, TC4 and TC5, the degree of redundancy is higher (*0 1 2 3 4 5 6 7 8*), see that those test cases have 8 identical transitions. More specifically, among TC1, TC2, TC3 and TC4, the number of identical transitions grows to 9 (*0 1 2 3 4 5 6 7 8 9*). Between TC1 and TC2 there are 11 identical transitions (*0 1 2 3 4 5 6 7 8 9 10 11*), and between TC3 and TC4 there are 14 (*0 1 2 3 4 5 6 7 8 9 22 23 24 11 13*).

To clarify, observe Figure 3.2. Analyzing this figure, we can conclude that the same parts of the system are being executed several times with a little bit of difference. Observe also, that the difference between TC1 and TC2 is only one step, i.e., the test cases are identical

until the last step, therefore, TC1 and TC2 are so similar that if we remove TC2 from the test suite, we will loose only one step. In the next section, we will present our proposal of a Similarity Function to calculate the similarity degree between a pair of test cases.

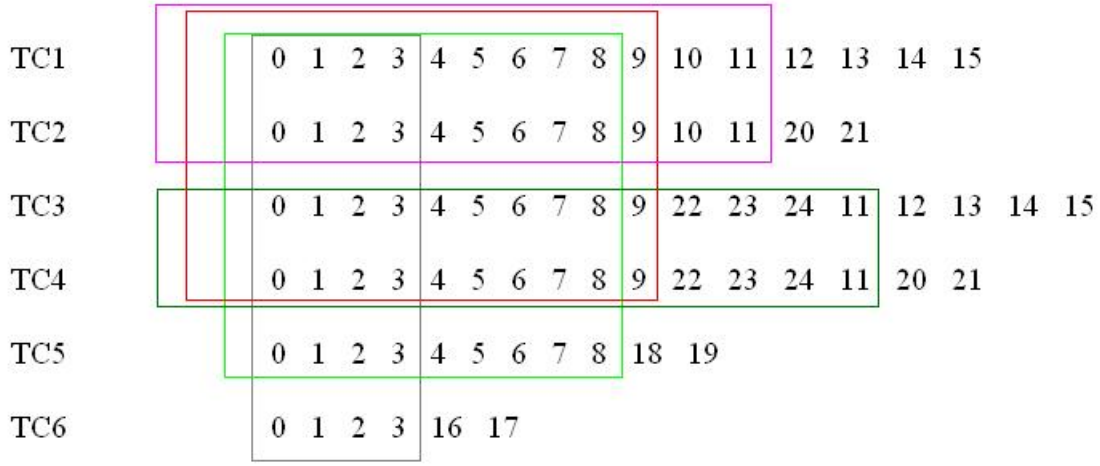


Figura 3.2: Test Cases - Redundancy

3.2 Similarity Function

By observing this real example, in order to measure the similarity degree between two test cases (two paths in an LTS), we need to count the number of identical transitions between them. Two transitions are identical (*it*) if they have exactly the same source and target states (q) and the same label (α). More formally:

$$\forall q \xrightarrow{\alpha} q', q'' \xrightarrow{\alpha'} q''' \in T \cdot it(q \xrightarrow{\alpha} q', q'' \xrightarrow{\alpha'} q''') \Longleftrightarrow q = q'' \wedge q' = q''' \wedge \alpha = \alpha'$$

Therefore, in order to obtain the similarity degree of the test cases, we need to calculate the number of identical transitions between each pair of test cases. Considering the example presented before, the number of identical transitions for each pair of test cases can be seen in Table :

The number of identical transitions (nit) that is used to calculate the similarity between two test cases discloses how much a test case is similar to another one, according to the function. Since, it is necessary to calculate the redundancy of one test case with the other

Tabela 3.2: Pair of Test Case and Number of Identical Transitions

Pair of Test Case	nit
TC1/TC2	11
TC1/TC3	13
TC1/TC4	9
TC1/TC5	8
TC1/TC6	3
TC2/TC3	9
TC2/TC4	11
TC2/TC5	8
TC2/TC6	3
TC3/TC4	13
TC3/TC5	8
TC3/TC6	3
TC4/TC5	8
TC4/TC6	3
TC5/TC6	3

ones, this number (nit) is then divided by the average between the paths length in order to balance the similarities. With this, we avoid representing a low similarity between two small test cases due to the length of the test cases and a high similarity between two very big test cases that are not so similar.

$$SimilarityFunction(i,j) = \frac{nit(i,j)}{avg(|i|,|j|)} \tag{3.1}$$

Next, we show the use of the similarity function for the example presented above. For example, we calculate the similarity degree between TC1 and TC2 presented in Table 3.1:

1. **Number of identical transitions (**$nit(TC1,TC2)$**):** 11;
2. **Average between Paths' Length (**$avg(|i|,|j|)$**):** 14;
3. **SimilarityFunction(TC1,TC2):** 11 / 14 = 0.78.

3.3 Similarity Matrix

Note that, to disclose the similarity among all test cases, it is necessary to apply the similarity function for each pair of test cases. Thus, we can represent similarity in a matrix, named as *Similarity Matrix*, that is defined as follows:

- $n \times n$ **(square matrix)**, where n is the number of paths and each n represents one path, that is called a test case;
- **Each element of the matrix** a_{ij} is defined by computing the similarity between test cases i and j. This function is calculated by observing the number of identical transitions ($nit(i, j)$), i.e., whether states "from" and "to", and labeled transition are the same (see definition in Section 2), and the average between paths length ($avg(|i|, |j|)$).

We are able to observe that the similarity matrix is symmetric, since $a_{ij} = a_{ji}$. As a result, $a_{ij} = SimilarityFunction(i, j)$ where $i = j$, is not considered, since it is not to our interest to calculate the similarity of a test case in relation to itself. For the example presented in Figure 3.1, we obtain the Matrix 3.2. The computational complexity to build the matrix is $O(n^2)$, where n is the number of test cases in the test suite.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0.78 & 0.81 & 0.60 & 0.69 & 0.30 \\ TC2 & & & 0.60 & 0.78 & 0.69 & 0.33 \\ TC3 & & & & 0.81 & 0.59 & 0.27 \\ TC4 & & & & & 0.64 & 0.30 \\ TC5 & & & & & & 0.40 \\ TC6 & & & & & & \end{pmatrix} \quad (3.2)$$

The matrix provides an overview of the similarity degree of each pair. The next step is to observe the values in the matrix, in order to draw conclusions concerning the redundancy. Observing the Matrix 3.2, we can conclude that:

- **The highest value (0.81):** This means that the test cases TC1/TC3 and TC3/TC4 are the most similar ones (they present more redundant parts). Observe that both the pairs TC1/TC3 and TC3/TC4 have 13 identical transitions (see Table 3.2).

- **The lowest value (0.30):** This means that the test cases TC1/TC6 and TC4/TC6 are the most different ones in the test suite. Note that both the pairs TC1/TC6 and TC4/TC6 have only 3 identical transitions (see Table 3.2).

3.4 Concluding Remarks

Here, a way of measuring redundancy among test cases of one test suite was presented. By observing the Similarity Matrix, some conclusions can be drawn. For the pair i and j, if the value is:

- **Zero (0):** This means that there is no similarity between the test cases i and j;
- **One (1):** This means that the test cases i and j are equal. When two test cases are equal, it is necessary to execute only one of them;
- **The highest value:** This means that the test cases i and j are the most similar ones of the test suite, i.e., the difference between the test cases is very small;
- **The lowest value:** This means that the test cases i and j are the least similar ones of the test suite, i.e., the difference between the test cases is very big.

In the next Chapters (from Chapter 4 to 6) we present three strategies that use the Similarity matrix presented here. They are:

- **Similarity Strategy (Chapter 4):** The goal is to select the most different test cases from a test suite to be executed (the number of test cases is defined by the test manager). In this case, we consider that if two test cases are very similar (the highest value of the matrix), one of them can be discarded (aiming to meet the resources constraints while having the adequate coverage of functionalities);
- **Weighted-Similarity Approach (Chapter 5):** The goal is to select the most different and important test cases from a test suite to be executed. The importance of each test case and the number that will be executed are defined by the test manager. In this case, we need to calculate a weighed-similarity matrix;

- **Dissimilarity Strategy (Chapter 6):** The goal is to reduce a test suite according to a test requirement, in this case the transition coverage is considered. Since the intention is to cover all transitions faster, the best thing is to find the lowest value of the similarity matrix (that means that the most different test cases) and place both in the reduced set.

Capítulo 4

Seleção baseada em Similaridade

Neste Capítulo apresentamos nossa proposta para Seleção de casos de teste baseada em Similaridade (conceito apresentado no Capítulo 3). Esta estratégia considera as restrições de recursos (um certo número de casos de teste pode ser executado). Assim são selecionados os mais diferentes casos de teste visando ter uma melhor cobertura de funcionalidades e de faltas.

Na Seção 4.1, nós apresentamos a nossa estratégia para seleção de casos de teste; um exemplo é usado para ilustrar a nossa estratégia na Seção 4.2; na Seção 4.3 é mostrado um estudo de caso real que foi executado comparando nossa estratégia de seleção aleatória (analisando cobertura de transições e de faltas); na Seção 4.4 é apresentado o experimento que nós executamos (analisando cobertura de transições).

Os resultados mostram que:

Estudo de Caso

- **Cobertura de Transições** - A estratégia de similaridade pode ser mais efetiva que a estratégia aleatória. Há vantagens consideráveis em utilizar Similaridade quando a quantidade de casos de teste desejado é maior ou igual a 20%;

- **Cobertura de Faltas** - Para todos os estudos de caso realizados, temos que Similaridade apresentada uma melhor performance relacionada a cobertura de faltas.

Experimento:

Considerando a cobertura desejada é 50% dos casos de teste, Similaridade é melhor que a estratégia de seleção aleatória. Em outras palavras, aplicando a estratégia de similaridade,

nós obtemos melhores resultados comparados a quando aplicamos a estratégia aleatória.

Maiores detalhes são apresentados nas próximas seções.

4.1 Definition

The idea is to keep in the test suite the least similar test cases according to a goal that is defined in terms of the intended size of the test suite. The least similar test cases provide the chance of having a better coverage of both requirements and faults, once that it covers the most different transitions.

This strategy uses the similarity function to build the similarity matrix (as shown in Chapter 3). The inputs are:

- **Percentage:** The desired percentage of test cases, defined, for example, according to the resources constraints;
- **Test Suite:** The set of test cases;
- **Similarity Matrix:** The matrix that contains the information regarding the similarity among all test cases of the test suite.

The Algorithm 1 presents the steps of this strategy. The first step is to calculate the desired number of test cases (line 1) according to the percentage, i.e., the number representing the total of test cases that have to be selected. Since the idea is to keep in the Similarity Matrix, the most different test cases, the highest value of the matrix is found. The two test cases correspondent to that value are analyzed and one of them is removed from the matrix (lines 2 - 14). This procedure is repeated until the number of test cases in the matrix is equal to the desired value (line 2).

At this point, the maximum values of the matrix are found. When a tie among maximum values (more than one maximum value) is found, in the similarity matrix, the idea is to randomly choose one of them (lines 3 - 4). From the maximum value we are able to discover the correspondent test cases (lines 5 - 6) from the most similar pair.

Now, the size of the test cases are compared, and the idea is to keep in the matrix the longest test case (lines 7 - 10), i.e. the test cases with more transitions, since it can represent

the highest functionality coverage. If the size between the two test cases is the same, a random choice is applied (lines 12 - 13).

Regarding the complexity analysis of Algorithm 1 we are able to observe a repeating structure (`while` command in line 2) where, within each iteration, the method `getAllMaxValue` ($O(n^2)$) is used to search the matrix for the highest similarity values. Therefore Algorithm 1 has a complexity of $O(n^3)$, where n is the number of test cases in the test suite.

It is possible to think that selecting long test cases would provide a test suite that requires a lot of time to execute. However, to our knowledge, no empirical evidence that relates the number of transitions of a test cases and the time required to execute the test suite, has been performed.

```
   input  : percentage, testSuite, similarityMatrix
   output: selectedTestCases
1  numberOfRequiredTestCases = calculateNumberOfDesiredTestCases(percentage, testSuite);
2  while (selectedTestCases.size() < numberOfRequiredTestCases) do
3      maxValues = getAllMaxValue(similarityMatrix);
4      chosenPair = pairs.shuffle.get(0);
5      testCase1 = chosenPair.getTestCase1();
6      testCase2 = chosenPair.getTestCase2();
7      if (testCase1.size() > testCase2.size()) then
8          similarityMatrix.remove(testCase2);
9      else if (testCase1.size() < testCase2.size()) then
10         similarityMatrix.remove(testCase1);
11         else
12             chosenTestCase = randomChoice(testCase1,testCase2);
13             similarityMatrix.remove(chosenTestCase);
14     selectedTestCases = similarityMatrix.getTestCases();
```

Algorithm 1: Similarity based Selection - Algorithm

4.2 Example - Similarity Selection

In order to illustrate the strategy, an example is presented below. An LTS model is presented in Figure 4.1. From this LTS model, 6 test cases are obtained. Both the test cases, and their respective sizes, can be seen in Table 4.1. In turn, the similarity matrix is presented in Matrix 4.1.

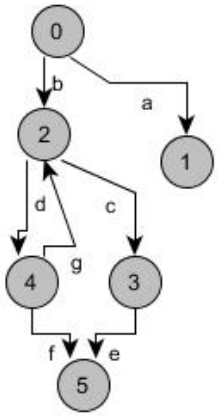


Figura 4.1: Example - LTS model

Tabela 4.1: Test Cases and Size of test cases

TC id	Path	Test Size
1	a	1
2	b c e	3
3	b d f	3
4	b d g	3
5	b d g d f	5
6	b d g c e	6

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 & 0 \\ TC2 & & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & & 0.75 & 0.75 \\ TC5 & & & & & & 0.6 \\ TC6 & & & & & & \end{pmatrix} \tag{4.1}$$

Considering that, the desired percentage is 50%, the desired number of test cases is 3 ($50\% \cdot 6$). Then, the first step is to find the highest value in the matrix. In this matrix, there is a tie among TC2/TC6, TC3/TC5, TC4/TC5 and TC4/TC6. Therefore, a random choice is done. Considering that the pair TC2/TC6 is chosen, TC2 is removed from the matrix, since $|TC6| > |TC2|$. The line and column of the removed test cases are also removed from the matrix, and the new matrix can be seen in Matrix 4.2.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 \\ TC3 & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & 0.75 & 0.75 \\ TC5 & & & & & 0.6 \\ TC6 & & & & & \end{pmatrix} \tag{4.2}$$

Following the algorithm of this strategy, again, it is necessary to find the highest value of similarity. As we can see, there is a tie among the highest values: TC3/TC5, TC4/TC5 and TC4/TC6. Thus, a random choice is performed again. Considering that the pair TC3/TC5 is chosen, since $|TC5| > |TC3|$, TC3 is removed from the matrix.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 \\ TC4 & & & 0.75 & 0.75 \\ TC5 & & & & 0.6 \\ TC6 & & & & \end{pmatrix} \tag{4.3}$$

So far, 2 test cases were excluded from the matrix (TC2 and TC3), thus we need to exclude one more. Searching for the highest value, we identify another tie between the TC4/TC5 and TC4/TC6. Randomly, TC4/TC5 is chosen and, since $|TC5| > |TC4|$, TC4 is excluded. Finally, the similarity matrix has 3 test cases (see Matrix 4.4) and thus, our set of selected test cases is composed by TC1, TC5 and TC6.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC5 & TC6 \\ TC1 & & 0 & 0 \\ TC5 & & & 0.6 \\ TC6 & & & \end{pmatrix} \tag{4.4}$$

4.3 Case Study

In order to evaluate the use of the similarity strategy, we conducted a case study. The goal of this case study is to compare Similarity and Random selection by considering fault and transition coverage. The Similarity and random strategies were applied having the percentage of the test suite (path coverage) goals ranging from 5% to 95% (increased by 5).

4.3.1 Application

The application used for this is a desktop tool named TaRGeT. This tool automatically generates test cases [47]. LTS-BT tool [16] was used for executing this case study. The input is a use case template [47], written by Motorola experts. All test cases, generated for this case study, were manually executed by Motorola employees. The collected metrics are:

- **Transitions Coverage:** We are able to measure the coverage of transitions of the model by counting the **number of excluded transitions**. The total number of transitions that are excluded by considering all of the discarded test cases of a given test suite represents the idea of measuring whether the strategies keep a reasonable coverage of functionalities even though discarding some test cases.

- **Faults coverage:** The total number of faults that are uncovered by the test suite during test execution. For this, we considered real faults. The idea is to measure whether the strategies preserve the fault detection capability of the original test suite.

These metrics are widely used in works of the literature and they are able to provide an overview of how much the strategy is adequate to select test cases considering the specified budget constraint. We considered these metrics adequate since our interest is to cover more functionalities (this is measured through transition coverage) and faults (measured through fault coverage).

4.3.2 Case Study - Preparation

Since, Similarity and of course, Random selection present a random choice in their algorithms, then each strategy was executed one hundred times (for each percentage) and the metrics were collected.

4.3.3 Results of the Case Study

This subsection shows the results of the case study. The TaRGeT application, used in this case study has 168 transitions in its LTS model. Also, LTS-BT tool was able to generated a total of 84 test cases, and these cases were manually executed, where a total amount of 13 failures were revealed. Each failure, in this case study, corresponds to a fault, in the application (i.e., 13 failures correspond to 13 faults).

Number of Excluded Transitions

The results can be seen in Figure 4.2. In this graph, the x-axis (or abscissa) represents the intended test cases percentage and in the y-axis (or ordinate) the average of excluded transitions obtained with 100 replications. The most effective strategy regarding this criterion is the one that presents the lower curve. The results show that most of the times, the Similarity strategy discards less transitions.

Below 20% of test cases a use of the random selection is more adequate. However, in the best case the Random selection (5% of test cases) excludes only 3.02% less transitions than Similarity. And the best case is when the percentage of desired test cases is 50%, where the Random strategy excludes 41.83% more transitions than Similarity.

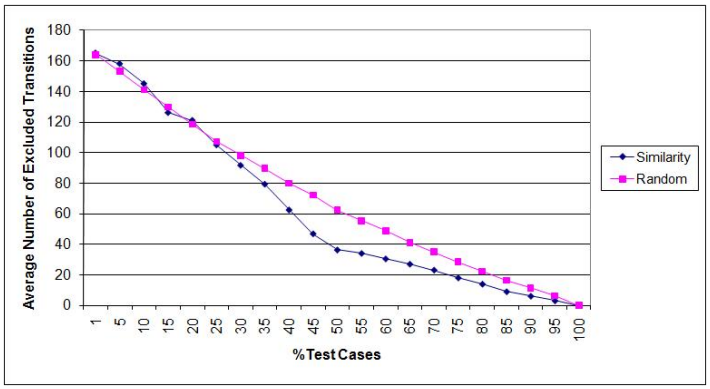


Figura 4.2: Average Number of excluded transitions by running each test selection strategy 100 times for each test selection goal

Faults Coverage

For fault coverage, the results are presented in Figure 4.3. We represented in the x-axis the intended test cases percentage and in the y-axis the average of covered faults obtained with 100 replications. The most effective strategy regarding this criterion is the one that presents the highest curve. As can be seen, the Similarity strategy reveals more faults for all percentages and the best case is when the percentage of test case is 55%. In this case, Similarity is able to reveal 41.33% more faults than Random.

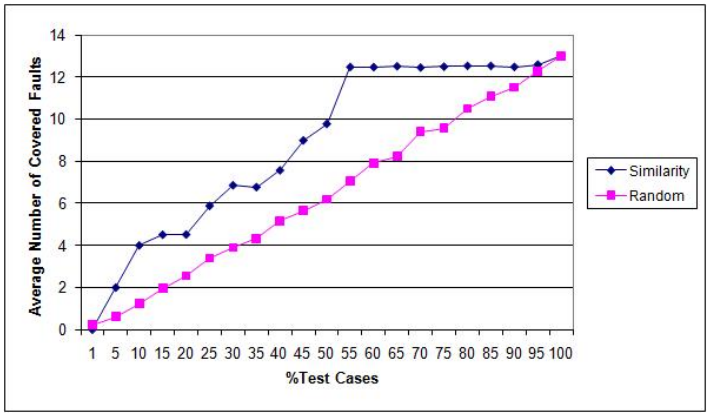


Figura 4.3: Average Number of covered faults by running each test selection strategy 100 times for each test selection goal

4.3.4 Concluding Remarks - Case Study

The case study performed in the evaluation suggests that the similarity approach can be more effective than random choice, usually by considering path coverage (test suite percentage) of more than 20%. The main threat to validity is the use of only one model, whereas the structural elements of the LTS may affect the performance of the analyzed strategies. Aside from that, it would be more appropriate to compare Similarity with other selection strategies, however our main objective with this case study was to obtain an overall perspective concerning the Similarity strategy, and not a comparative overview.

More case studies have been executed and can be seen in Appendix A. One of the executed case studies is the same presented here, however with another version of the use case document and a different level of abstraction, were used. Then, the metrics are different than the ones presented here.

4.4 Experiment - Selection

This Section presents the experiment and the obtained results of the execution. The used framework - proposed by Wohlin *et al.* - was presented in Chapter 2. The definition, the planning, the operation and the Analysis and Interpretation results of the experiment are showed in the next subsections.

Our general hypothesis is that Similarity presents the best performance in relation to the number of excluded transitions, considering 50% of the test cases. This percentage was considered because the highest difference between Similarity and Random strategy in case study is 50%, as presented before (Section 4.3).

4.4.1 Definition

The first step is to define the experiment. Therefore, the key questions proposed by Wohlin *et al.* were answered:

1. **What is studied?** Selection strategies;
2. **What is the intention?** To investigate;

3. **Which effect is studied?** Number of excluded transitions;

4. **Whose view?** The tester;

5. **Where is the study is conducted?** Model-Based Testing (MBT).

From these answers, the goal definition template is filled. In summary, the goal of this experiment is:

Analyze *selection strategies*
for the purpose of *investigating*
with relation to *number of excluded transitions*
from the point of view of the *tester*
in the context of *MBT*.

For this experiment, the (input) objects are LTS models. Since the strategies execute automatically, there is no need for subjects to be involved in this experiment.

4.4.2 Planning

Once the elements of the experiment are properly defined, the following steps of the study must be planned. Following the chosen framework, the context selection, the variables (dependent and independent), hypothesis, design and instrumentation were defined.

Context Selection

The context of this experiment can be characterized as a "toy vs. real" problem. In this case the objects are LTS models, randomly generated from a configuration. This configuration is characterized by a specific number for the depth of the LTS, the number of loops, branches and joins (these elements are detailed in Appendix B.

Variables Selection

In order to characterize the experiment, the variables must be defined. The variable chosen to observe (dependent variables) and to control (independent variables) are:

- **Dependent:** The Number of Excluded Transitions (NET).

- **Independent:** The test cases percentage; the configuration chosen the depth and the amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 2 levels: Similarity (Sim) and Random.

Hypothesis Formulation

Once the variables are defined, we are able to structure our null and alternative hypothesis. Their definition is formalized as following:

- **A null hypothesis** (H_0): $NET_{Sim} = NET_{Random}$ - The two strategies exclude the same number of transitions, in another words, the strategies present the *same behavior*;
- **An alternative hypothesis,** (H_1): $NET_{Sim} \neq NET_{Random}$ - The two strategies exclude a different number of transitions, i.e., the strategies present *different behavior*.

Experiment Design

As seen before, there is one factor (test case selection strategy) with 2 levels (or treatments). Thus, there is one factor and 2 treatments, where, for each object, the two treatments are applied. The chosen confidence level is 95% (significance level is $\alpha = 0.05$), as suggested by statistical literature [39].

Aiming to define the number of replications, necessary to guarantee statistical significance for the specified level of confidence (95%), 40 replications were performed, collecting the number of excluded transitions. These data are presented in Table 4.2.

Tabela 4.2: Mean, Standard Deviation and number of necessary replications for each technique.

Technique	Similarity	Random
Mean ($\overline{x}$)	17.9	20.0
Standard Deviation (s)	6.99	4.61
Number of Necessary Replications (n)	235	82

Observing the Table 4.2, we are able to see that 235 and 82 replications for Similarity and Random selection, respectively, provide a statistical significance for the obtained data. Therefore, this experiment design will consider 300 replications for each strategy.

Instrumentation

The next step of the planning is to specify the instruments of the experiment. In this step, there are three types of instruments [61]:

- **Objects:** The objects are LTS models randomly generated from a configuration (depth, number of loops, forks and joins).
- **Guidelines:** This experiment uses no guidelines, since the strategies do not require subjects to configure them.
- **Measurements:** The NET will be collected for each treatment. The tool LTS-BT provides support for both executing the experiments and collecting the data.

Validity Evaluation

The objects used in this experiment can be considered the main threat to validity. These objects are automatically generated, and therefore, they can not represent a real behavior. Besides, since they are randomly generated from a specific configuration, both the traceability and controllability of the elements of the model (transitions and states) are reduced.

On the other hand, we are able to obtain an overview of the execution of the strategies in several models, since they are randomly generated. Thus, we avoid being presented with a conclusion that is specific to only one LTS (if we would have used the same LTS in every execution). A proper scenario would be to have several real applications to execute the strategies. However, most real applications and their respective specification are not available to the open public.

4.4.3 Operation

To execute this experiment, it was necessary to implement the two strategies and the LTS generator (see Appendix B). Both the LTS generator and the strategies are implemented in the Java programming language[1].

The objective, in using this LTS generator, is to automatically generate different models. Therefore, a specific configuration for the depth and structure of the LTS is specified and

[1] http://www.sun.com/java/

the generator is able to place these structures (loops, forks and joins), in different ways. A deeper LTS provides more option to place the structures (branches, loops and joins).

Through executions of the generator, we were able to observe that an LTS, generated with a smaller depth and several structures, generates a lot of test cases with the same size. This fact does not represent real applications, since the test cases of real applications vary the size (they have different number of flows). Besides, when there are several test cases with the same size, the Similarity strategy applies a random selection between each pair of test cases (most similar), whereas random selection can pick any test case from the test suite. That scenario would not provide a fair comparison of the strategies. Therefore we chose the following configuration:

- **Depth:** 15;
- **Number of loops:** 2;
- **Number of branches:** 3;
- **Number of joins:** 3.

This configurations provides a wide range of possible LTS. The generated LTS begins with 16 states, where the generator can choose 15 states, out of the 16, to place the structures according to the constraints described in Appendix B. Aiming to have different size of test cases, we decide to have a higher depth in relation to the number of structures.

There is only one experimental design (with only one factor - test suite reduction strategy) with a null and an alternative hypothesis, where the intention is to reject the null hypothesis. Each strategy was executed 300 times, using a machine with the following configurations:

- Intel Core 2 quad 2.33 GHz;
- 4GB RAM;
- 1TB for Hard Disk Memory.

4.4.4 Analysis and Interpretation

The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[2]. The results can be seen in Figures 4.4 and 4.5. In this graph, the red dots, should overlap the blue line, in order to indicate that the data fit a normal distribution.

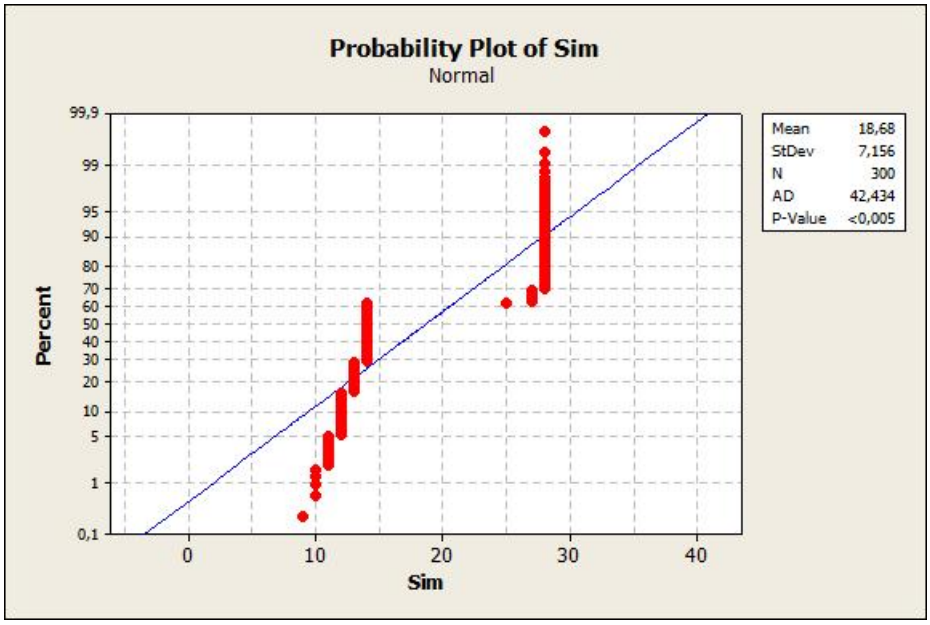


Figura 4.4: Anderson-Darling normality test - Similarity

As can be seen, the data do not fit a normal distribution, once the *p-values* are less than 0.05. Then, it is necessary to apply a non-parametric test. Since we have only one factor and two treatments, we can apply a Mann-Whitney testing to check the null hypothesis. The results are presented in Table 4.3.

Since $p - value = 0.0004$, and this is less than 0.05 (α), H_0 (the null hypothesis) can be rejected. Therefore, the data support the hypothesis that there is a difference between the population medians ($ETA1 - ETA2$). The difference between the two population medians is greater than or equal to -2.000 and less than or equal to -1.001. As we can see, the difference between the population medians of Similarity and Random is negative, therefore, the NET from Similarity is less than the NET from Random selection.

[2]http://www.minitab.com/

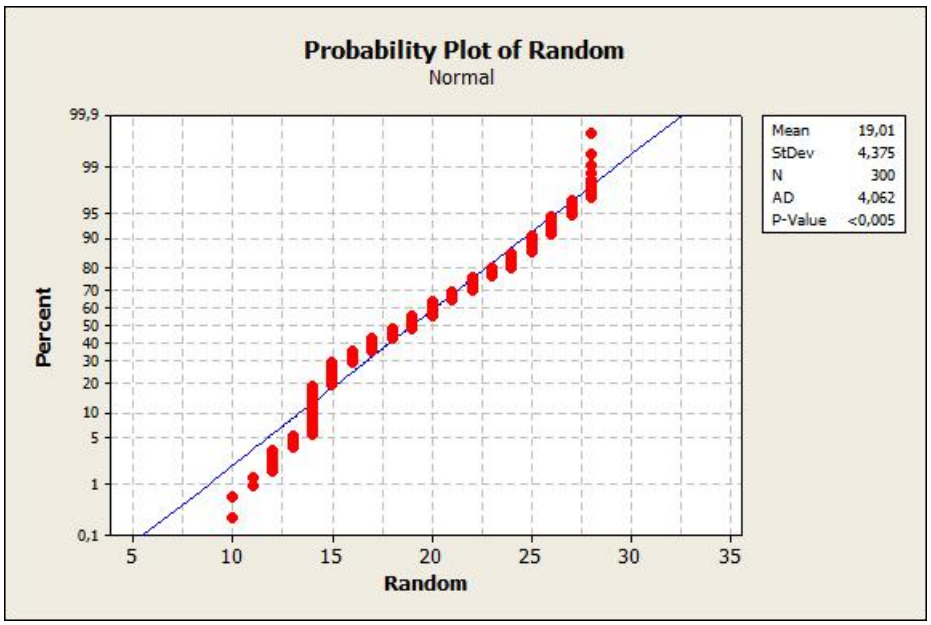


Figura 4.5: Anderson-Darling normality test - Random

Tabela 4.3: Mann-Whitney Test - Sim and Random

Technique	N	Median
Sim	300	14.000
Random	300	19.000
Point estimate for ETA1-ETA2 is -1.000		
95.0 Percent CI for ETA1-ETA2 is (-2.000;-1.001)		
The test is significant at 0.0004		

4.4.5 Concluding Remarks - Experiment

With 95% of confidence level, we are able to reject our null hypothesis. Therefore, our general hypothesis is confirmed, since the behavior of the Similarity - considering 50% of the number of the test cases - is better than the one observed with Random. In another words, by applying the Similarity strategy we obtain better results than by applying Random.

4.5 Concluding Remarks

In this Chapter, we presented one of our strategies for test case selection. This strategy tries to meet the resources constraints and then, keeps in a test suite, only the test cases that will

be executed. The focus is to increase functionality and fault coverage as high as possible. A Case study and an experiment were performed, and the results show evidence that:

Case studies:

- **Concerning transition-based coverage:** For the conducted case study, the similarity strategy can be more effective than the random strategy. There are considerable advantages when the desired coverage is higher or equal to 20% of the test case;
- **Concerning fault coverage:** For the conducted case studies, the similarity strategy has also a superior performance. For all case studies (the ones presented in this chapter, and the ones presented in Appendix A, the highest coverage is only achieved by the similarity approach.

Experiment:

Considering a desired coverage of 50% of the number of the test cases, Similarity is better than the Random strategy. In another words, by applying the Similarity strategy we obtain better results than by applying Random.

Note that the obtained results in cases studies and in the experiment are not contradictory. It is important to say that when the model presents test cases with the same length, the behavior of similarity strategy can be the same or worse than the one observed in the Random strategy. That situation presents itself, in the Similarity strategy, when the test cases to be discarded are chosen by the length, and this length is the same for both test cases.

The drawback of this strategy is that important test cases could be eliminated. For instance, a test case may be focused on either a frequent used functionality, or a very important functionality with respect to the user needs (or on critical path of the application). Since this strategy does not distinguish the importance of test cases, crucial ones may be discarded. Therefore, we integrate the similarity method with a value-based test strategy, called the Weighted-Similarity Approach, that will be presented in next Chapter (Chapter 5).

Capítulo 5

Similaridade Balanceada (WSA)

Neste Capítulo apresentamos nossa proposta para Seleção de casos de teste baseado em Similaridade (Capítulo 3) que também considera pesos (atribuídos aos casos de teste). Nossa estratégia é definida na Seção 5.1 e um exemplo para ilustrar é apresentado na Seção 5.2. Na Seção 5.3 são mostrados dois estudos de caso que nós executamos para comparar WSA com outras estratégias (seleção aleatória, Similaridade e a Aleatória Guiada) considerando cobertura de transições e faltas.

Os resultados dos estudos de caso mostram evidências que:

- **Cobertur de Faltas:** WSA é a estratégia mais efetiva, mas é necessário conhecer ou acertar o ponto das faltas;
- **Cobertura de Transições:** Similaridade - na maioria dos casos - apresenta uma melhor performance.

Maiores detalhes são apresentados nas próximas seções.

5.1 Definition

In Chapter 4, the Similarity strategy was presented. This strategy considers the resource constraints and thus, selects the most different test cases aiming to have a better functionality and fault coverage. The results of both the case study and the experiment showed that Similarity is an efficient strategy, in relation to transition and fault coverage. However, when applying

this strategy, important test cases (e.g., test cases that discover faults) could be eliminated, since no information about this "importance" is considered by that strategy.

However, if this information is available, this can be used to get most important and different test cases. For this, we integrate similarity based selection strategy with a value-based test strategy, named Weighted-Similarity Approach (WSA). The main goal of this approach is to exploit the software engineering knowledge and experience, in order to minimize the size of a test suite by keeping in it only the test cases that can be feasibly executed according to the user behavior and resources available to the testing process. To accomplish this goal, we use the concept of *similarity* and show how it can be used for test cases selection. WSA foresees the test cases selection by prioritizing accordingly with the probabilities set by the test manager. The idea is that when choosing between two similar test cases to be discarded, the one that has a greater probability is kept. In the original strategy (Similarity-based selection), a random choice is performed when they have the same size. Therefore, we aim at testing suites that have the most different test cases and yet, these are also the most important (according to the probability) ones.

Using an LTS model, the user can set the desired path coverage (i.e. amount of desired test cases) and provide the weights to be associated to each possible flow of the LTS. We can consider that each weight indicates the expected frequency of use as a concept of "importance" for a given flow. Taking in consideration that the sum of weights for every flow of a branch is 1 (the total flow - before the branch - is 1, so the sum should be 1).

Given the LTS model with weights assigned, the test cases similarity and *final weight* for each flow is computed. The similarity between two test cases is computed as the number of common steps in the two test cases. The *final weights* of a test case is obtained by multiplying the attributed weight in their branches. As result, a weighted similarity matrix is built with test cases as columns and rows, where each element is defined as the similarity between two test cases divided by the weight of the test case of the respective row. Note that, the most similar test cases must be eliminated and the most important must be kept. For this, we balance the similarity with the weight of each test case.

With the LTS behavior model and probabilities (weights), we can calculate the similarity and test case weights (this is obtained by multiplying the attributed probabilities in their branches), build the weighted-similarity matrix, where each $a_{ij} =$

$WeightedSimilarityFunction(i, j)$. The $WeightedSimilarityFunction(i, j)$ is defined as follows:

$$WeightedSimilarityFunction(i, j) = \frac{SimilarityFunction(i, j)}{W(i)} \tag{5.1}$$

As can be seen, the equation 5.1 uses the equation 3.1 as its numerator. Similarly to the Similarity Selection, we apply this equation to each pair of test cases, in order to obtain the weighted similarity matrix. Therefore, the weighted-similarity matrix is:

- $n \times n$ (square matrix), where n is the number of paths and each n represents one path, that is called a test case;
- Each element of the matrix $a_{ij} = WeightedSimilarityFunction(i, j)$.

The weighted-similarity matrix is not symmetric, since the similarity between i and j ($SimilarityFunction(i, j)$) is balanced by the weight of the i ($W(i)$). The highest value represents the test case that is most similar to the other ones, and least important. Thus, in order to choose a test case to be removed, we search for the highest value in the matrix. This corresponds to a test case that is very similar to other test case, and has a lower weight. If there is a tie, then the smallest test case must be eliminated. If there is a tie again, random choice is applied.

This strategy uses the weighted-similarity function to build the weighted-similarity matrix. The inputs are:

- **Percentage:** The desired percentage of test cases. This percentage is defined according resources constraints;
- **Test Suite:** The set of test cases;
- **Weights:** The weights of the test cases;
- **Weighted-Similarity Matrix:** The matrix that contains the information about similarity and importance among all test cases of the test suite.

The Algorithm 2 presents the steps of this approach. The first step is to calculate the desired number of test cases (line 1) according to the percentage, that number represents

the total of test cases that have to be selected. Since the idea is to keep in the Weighted-Similarity matrix, the most different and most important test cases. Then, the highest value of the matrix is found and discarded of the matrix (lines 2 - 13). This procedure is repeated until the number of test cases in *wsMatrix* is equal to the desired value (line 2).

In line 3 of the algorithm, the maximum values are found. When a tie among maximum values is found (more than one maximum value), the idea is to randomly choose the least important (line 4), then the least important is discarded of the matrix (lines 5 - 6). If there is also a tie among the weights (importance) of test cases (lines 7 - 12), the idea is to discard the smallest test case (lines 8 - 10) to guarantee the highest coverage of functionalities. Finally, if there is a tie between the smallest test cases, then the random choice is applied (lines 11 - 12).

```
   input  : percentage, testSuite, weights, wsMatrix
   output: selectedTestCases
1  numberOfRequiredTestCases = calculateNumberOfDesiredTestCases(percentage, testSuite);
2  while (selectedTestCases.size() < numberOfRequiredTestCases) do
3      maxValues = getAllMaxValue(wsMatrix);
4      choosedTestCases = getLessImportant(maxValues, weights);
5      if (choosedTestCases.size()=1) then
6          wsMatrix.remove(choosedTestCases.get(0));
7      else
8          choosedSmalls=getSmallTestCase();
9          if (choosedSmalls.size()=1) then
10             wsMatrix.remove(choosedSmalls.get(0));
11         else
12             wsMatrix.remove(randomChoice(choosedSmalls));
13     selectedTestCases = wsMatrix.getTestCases();
```

Algorithm 2: WSA - Algorithm

The complexity analysis of Algorithm 2 is similar to the one presented for the Similarity-based Selection algorithm (Algorithm 1 in Section 4.1, Chapter 4). Performing the analysis we are able to obtain a computational complexity $O(n^3)$, where n is the number of test cases in the test suite, for Algorithm 2.

5.2 Example - WSA

In this Section, we will show one simple example to illustrate the proposed strategy. For this example, the use case (main and alternative flows) and the respective LTS model are presented (this approach is not strictly related to a specific model-based approach - use case). Then, the approach is applied.

5.2.1 Example - Description

The use case describes the creation of a new contact in the Contact list (the use case model presented here was presented by Nogueira and his colleagues [47]). In Figure 5.1 the main flow of this use case is presented. This flow represents an user that successfully adds a contact to a phone book.

Main Flow

flow Description: description Create a new contact

From Step: fromSteps START

To Step: toSteps END

Step Id	User Action	System State	System Response
step id 1M id	action Start My Phonebook application. action	condition My Phonebook application is installed in the phone. condition	response My Phonebook application menu is displayed. response step
step id 2M id	action Select the New Contact option. action	condition condition	response The New Contact form is displayed. response step
step id 3M id	action Type the contact name and the phone number. action	condition condition	response The new contact form is filled. response step
step id 4M id	action Confirm the contact creation. action	condition There is enough phone memory to insert a new contact. condition	response A new contact is created in My Phonebook application. response step

flow

Figura 5.1: Creating a New Contact - Main Flow

In Figure 5.2 two alternative flows are presented. The first alternative flow describes the scenario where the user is able to cancel the creation of the new contact. This can happen in two cases: the form is opened (Step ID 2M) and is filled (Step ID 3M). The second

alternative flow describes the scenario where the new contact is not created because there is not available phone memory, this can happen when the user tries to add the contact (after filling the form in Step ID 3M).

Alternative Flows

«flow» Description: «description» Cancel the new contact creation

From Step: «fromSteps» 2M, 3M

To Step: «toSteps» END

Step Id	User Action	System State	System Response
«step» «id» 1B «id»	«action» Press Cancel softkey. «action»	«condition» «condition»	«response» The phone goes back to My Phonebook application menu. «response» «step»

«flow»

«flow» Description: «description» A new contact is not created because there is not available phone memory

From Step: «fromSteps» 3M

To Step: «toSteps» END

Step Id	User Action	System State	System Response
«step» «id» 1C «id»	«action» Confirm the contact creation. «action»	«condition» There is not enough phone memory to insert a new contact. «condition»	«response» A message: "There is not enough phone memory" is displayed. «response» «step»

«flow»

Figura 5.2: Creating a New Contact - Alternative Flows

By following the Step IDs, we can obtain the Labeled Transition System (LTS) Behavior model (this can be seen in Figure 5.3). Observing this figure, we can see that, after the user executes the Step 2M, the user can execute the Steps 3M or 1B. The same way for Step 3M, the user can execute the Steps 4M, 1C or 1B.

We can use a test case generation algorithm, to traverse the model, where each path in the LTS is a test case. In this case, we used LTS-BT tool, to generate the test cases from the LTS in Figure 5.3. Thus, we obtain 4 test cases, as seen in Table 5.1.

For applying the weighted-similarity approach, it is necessary to define the meaning of "importance". Then, we define weights for each branch that is originated by alternative flows after step 2M and 3M (Figure 5.4). In this case, the weight is related to the expected frequency of execution (by the end user) of that specific step.

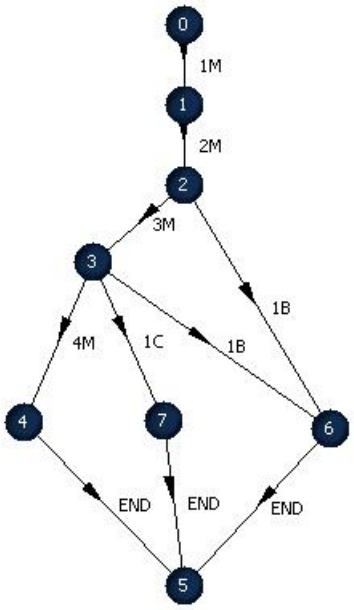


Figura 5.3: Labeled Transition System (LTS) Behavior model

Tabela 5.1: Test Cases generated from LTS model presented in Figure 5.3 and their respective lengths

Test Case Id	Test Case	Length
TC1	1M 2M 3M 4M	4
TC2	1M 2M 3M 1C	4
TC3	1M 2M 3M 1B	4
TC4	1M 2M 1B	3

BRANCHES		Probabilities
2M	3M	0,7
	1B	0,3
3M	4M	0,6
	1C	0,2
	1B	0,2

Figura 5.4: Probabilities

Tabela 5.2: Weights of the test cases obtained from LTS Model 5.3 and assigned probabilities 5.4

Test Case Id	Weight
TC1	$0.7 \times 0.6 = 0.42$
TC2	$0.7 \times 0.2 = 0.14$
TC3	$0.7 \times 0.2 = 0.14$
TC4	0.3

For calculating the weighted-similarity matrix, it is necessary to calculate the weighted-similarity function for all a_{ij}. Then, for example, the $WeightedSimilarityFunction(TC1, TC2)$ is given by:

1. **Number of identical transitions** (nit): 3;
2. **Average between Paths' Length** ($avg(|TC1|, |TC2|)$): 4;
3. **SimilarityFunction(TC1,TC2)**: 3 / 4 = 0.75.
4. **WeightedSimilarityFunction(TC1,TC2)**: 0.75 / 0.42 = 1.78.

Calculating every a_{ij}, we are able to obtain the weighted-similarity matrix, and thus, the information regarding the similarity and importance of each test case. The complete matrix for this example is presented in Matrix 5.2.

$$WeightedSimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 \\ TC1 & & 1.78 & 1.78 & 1.36 \\ TC2 & 5.35 & & 5.35 & 4.08 \\ TC3 & 5.35 & 5.35 & & 4.08 \\ TC4 & 1.90 & 1.90 & 1.90 & \end{pmatrix} \quad (5.2)$$

Considering that there are enough resources to execute only 50% of the test cases, we need to discard two test cases. The highest values of the Matrix is between rows TC2 and TC3. Since we need to choose one of them, and the importance and length of both test cases

are the same, a random choice is applied. Supposing that TC2 is discarded, the new Matrix is presented in Matrix 5.3.

$$WeightedSimilarityMatrix = \begin{pmatrix} & TC1 & TC3 & TC4 \\ TC1 & & 1.78 & 1.36 \\ TC3 & 5.35 & & 4.08 \\ TC4 & 1.90 & 1.90 & \end{pmatrix} \tag{5.3}$$

Observing the new matrix, the highest value is in the row of TC3. Therefore, TC3 is discarded. The test cases the are kept to execute are TC1 and TC4. Observe that TC2 and TC3 are so similar to TC1, but TC1 is kept because it is more important than the other ones. And TC4 is also kept because it is the most different and has the highest weight.

5.3 Case Study

In order to evaluate the use of WSA, we conducted two case studies. The goal of these case studies is to compare WSA, Random Selection, Guided Random (Random choice guided by the same transition probabilities applied with the WSA approach) and Similarity by considering fault and transition coverage. All strategies were applied having percentage of test suite (path coverage) goals ranging from 10% to 90% (increased by 10).

5.3.1 Applications

Two real applications provided by Motorola Software Engineers have been selected for the case studies. They are briefly described as follows:

- **Application 1:** TaRGeT is a desktop application that supports the model-based testing process, where it automatically generates test cases from use case documents.
- **Application 2:** Direct License Acquisition (DLA) Support is a feature for mobile phone applications that handles acquisition of the WMDRM License (Windows Media Digital Rights Management) for the Windows Media platform. This License provides secure delivery of audio and/or video content.

Tabela 5.3: Number of Test Cases and Faults

	# Test Cases	# Faults
Application 1	84	13
Application 2	28	2

For each one of these applications, Motorola Software Engineers elaborated the use case documents [47]. From these documents, LTS-BT generated the test suites that have the metrics showed in Table 5.3. The test cases were manually executed and the captured failures were associated with faults that can be detected by the suite.

5.3.2 Metrics

Once the applications used to apply the strategies have been explained, our next step, is to present the metrics considered in our analysis. Aiming to do a comparison among WSA, Random Selection, Guided Random, Similarity, and a manual selection, the same metrics used in the case study performed for the Similarity-based Selection (Section 4.3, Chapter 4) will be observed. These metrics are the **Transitions Coverage**, and the **Fault Coverage**.

5.3.3 Case Study - Preparation

Since, all strategies, besides the manual selection, present a random choice in their algorithms, then each one of them was executed one hundred times and the metrics were collected. The results are presented in the next subsection. For each case study, the following activities were performed:

- Reading and understanding the related documents, use cases (Target templates), and LTS model (generated from use cases);
- Assigning probabilities to branches in the LTS model;
- Manually selecting test cases to produce a minimized test suite from 10% to 90% of selection goal, starting from 10% (performed by the test designer only).

Assigning Probabilities - Subjects

Case studies have been conducted by one WSA designer and by one test designer. In both cases, the goal is to reduce the test suite, maximizing the ability for fault detection and also providing an adequate coverage of functionalities.

- **The test designer:**
  - Has experience on test selection;
  - Has experience with all the approaches considered in this study;
  - Has no previous knowledge of the defects presented in the case studies.
- **The WSA designer:**
  - Proposed the WSA approach;
  - Has experience with all the approaches considered in this study;
  - Knows all defects presented in the case studies.

Our assumption is that by using experience on test selection and also knowledge of the application domain, the test designer can assign probabilities to guide the selection of the most effective test cases. By knowing the defects, the WSA designer can maximize the results obtained by WSA. Therefore, we should be able to assess the limits of the approach in the best case. From this, we may identify strategies for assigning probabilities as well as uncover limitation of practical use.

Assigning Probabilities - How to specify the values

Each subject (one test designer and one WSA designer) received the use case document and the respective LTS of each application. For each branch of the LTS, the subject assigned the probabilities. The assigned value considers the probability to discover faults. The subjects were instructed to assign weights (probabilities) between 0 and 1, whereas the closer to zero, the lower the chance of reveal faults.

5.3.4 Results of the Case Study

This subsection presents the results of the case study. As a reminder, the variables considered in the study are the size of the reduced test suite (RTS) and the fault detection.

Faults Coverage

Application 1

The results obtained - for Application 1 - by applying WSA, Random Selection, Guided Random, Similarity and Manual selection (done by the test designer), considering the probabilities assigned by WSA and the test designer can be seen, respectively, in the Figures 5.5 and 5.6.

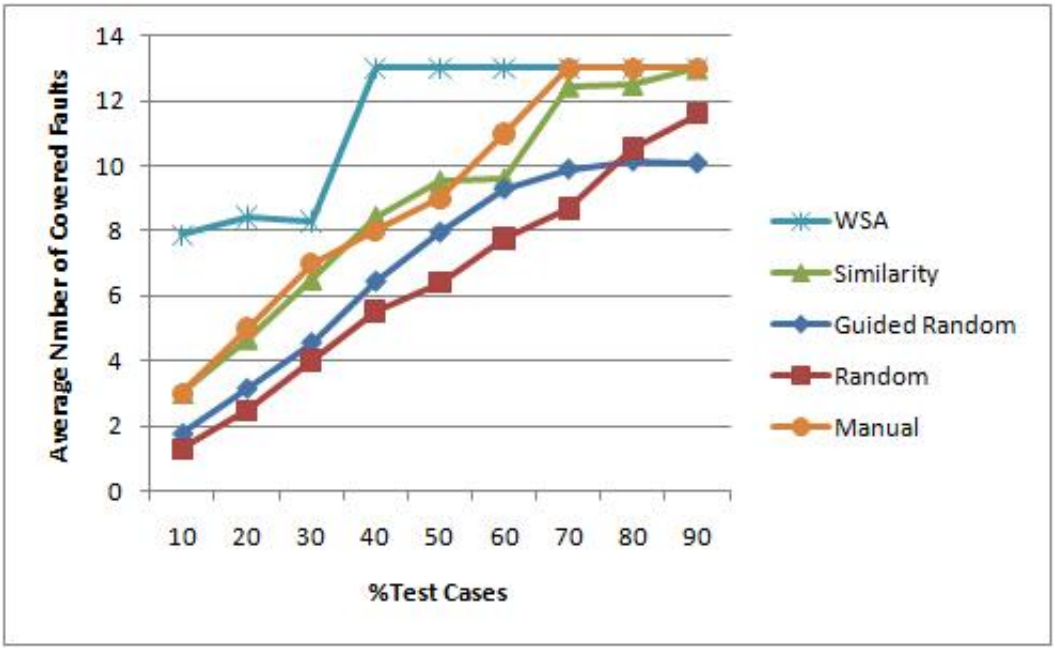


Figura 5.5: Average number of covered faults by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 1.

Observing the results, we can conclude that when the probabilities are well assigned, it's better to use WSA for any desired percentage of test cases. In another words, effective probability distribution can contribute to better results. Observe that, when the probabilities are not well distributed, the manual selection can perform better, and the Similarity, that uses

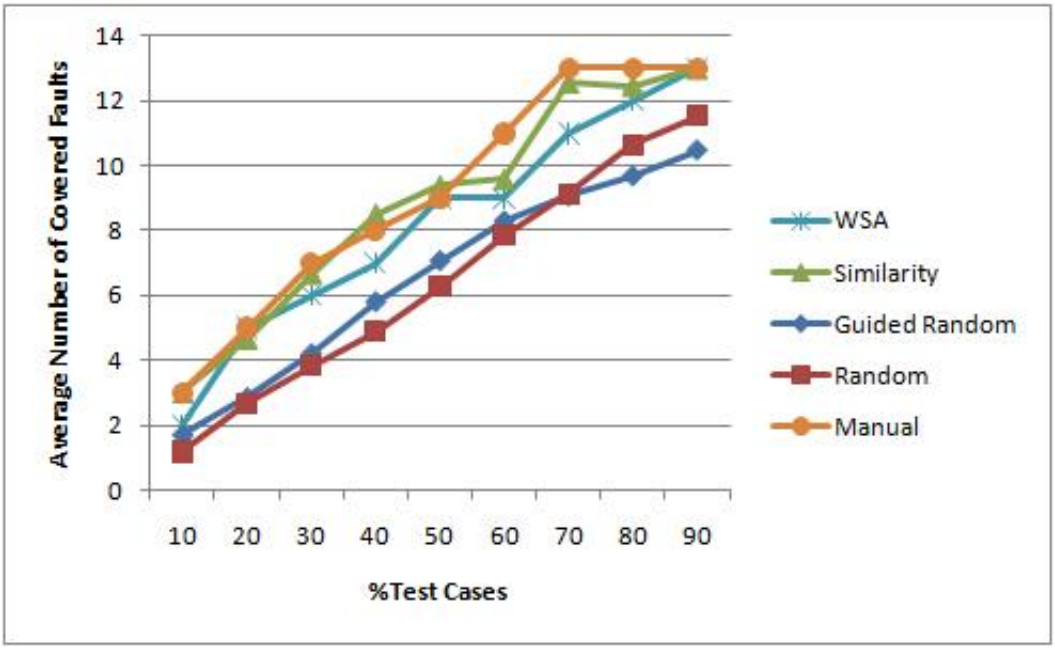


Figura 5.6: Average number of covered faults by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 1.

no probabilities values, presents a better behavior, but it is necessary to consider the overhead to apply it.

Application 2

The results obtained - for Application 2 - by applying WSA, Random Selection, Guided Random, Similarity and Manual selection (done by test designer), considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.7 and 5.8.

Again, WSA brings better results depending on probabilities assignment and also on the ability of the tester to pinpoint faults. Note that the manual selection for Application 2 was not as successful as for Application 1. Moveover, for faults distributed among the longest test cases selected from several branches, the guided random presents a better behavior since the selection is always based by the best local choices. On the other hand, WSA computes the product of probabilities to choose the test case to select, favoring the choice of shorter test cases.

Fault Coverage - Concluding Remarks

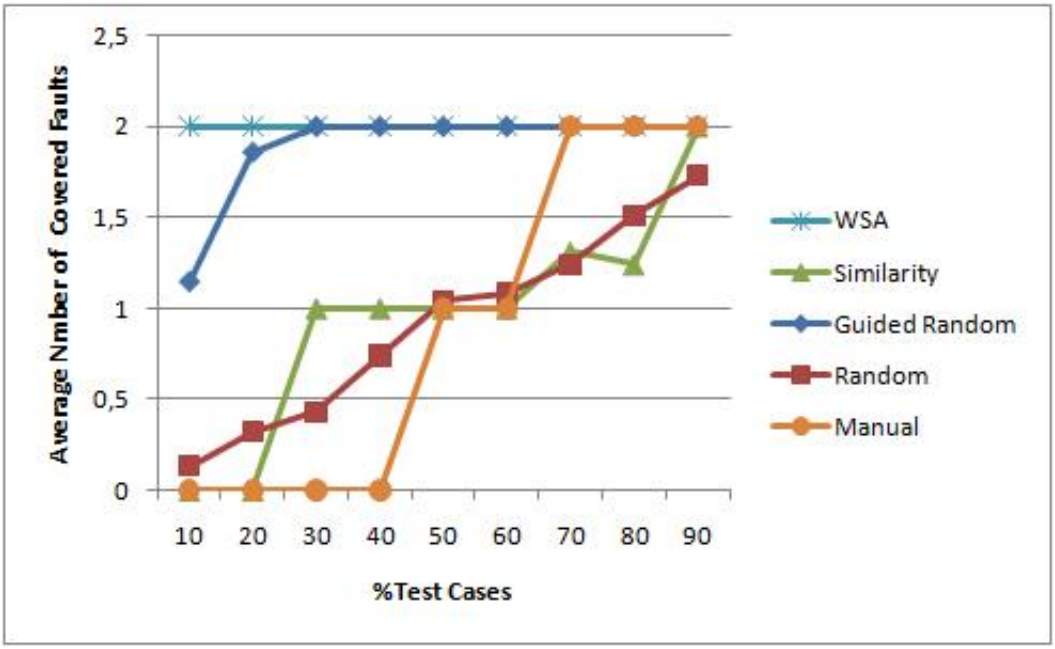


Figura 5.7: Average number of covered faults by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 2.

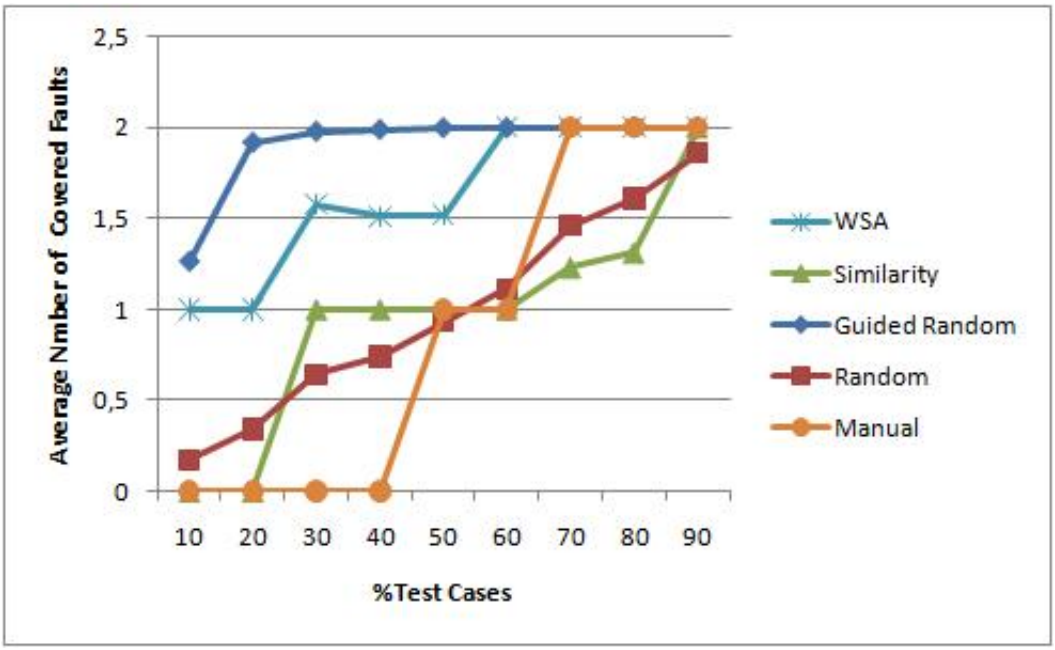


Figura 5.8: Average number of covered faults by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 2.

For both applications, we can see through the graphics showed in Figures 5.5, 5.6, 5.7 and 5.8 that, in order to obtain good results it is necessary to know the points of possible faults. The results obtained when WSA designer assigned the probabilities show that WSA is an effective technique.

Transition Coverage

Application 1

The results obtained - for Application 1 - by applying WSA, Random Selection, Guided Random, and Similarity, considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.9 and 5.10.

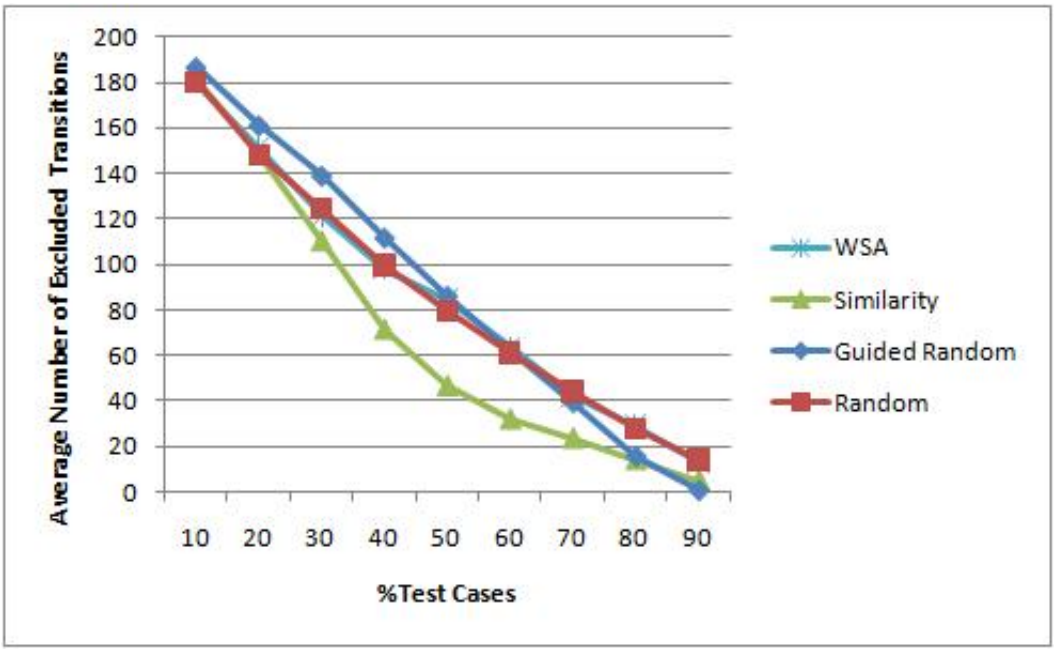


Figura 5.9: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 1.

By observing the results, we can conclude that the similarity presents better performance in relation to the others, since it excludes less transitions, and therefore, it presents the best coverage. This fact can be explained because similarity does not consider the importance of the test cases. Particularly, similarity keeps in the test suite the biggest test cases.

Application 2

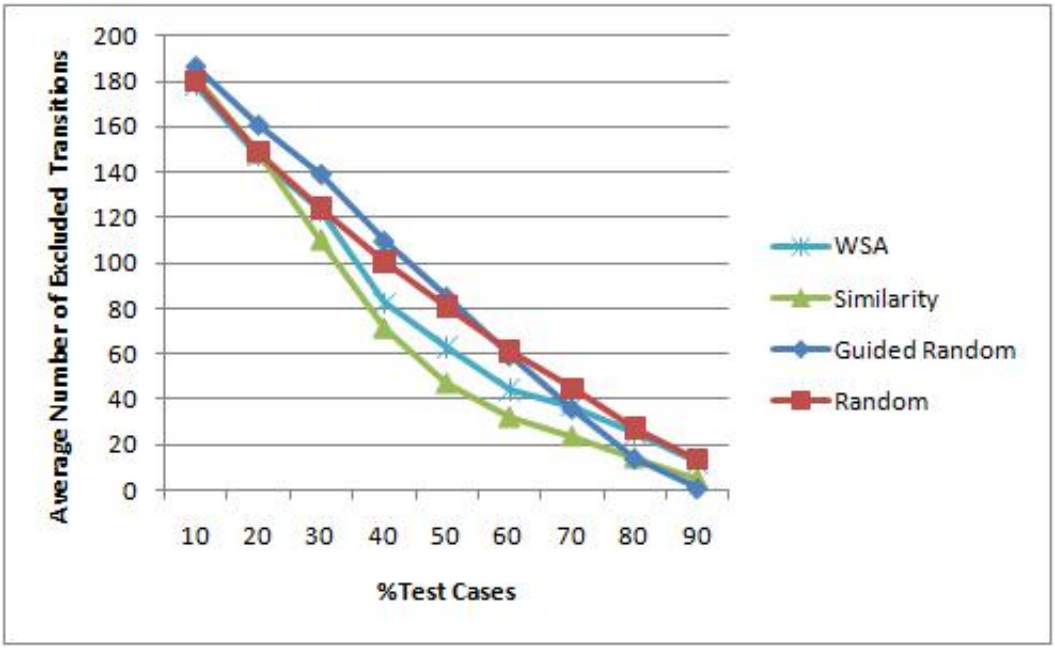


Figura 5.10: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 1.

The results obtained - for Application 2 - by applying WSA, Random Selection, Guided Random, and Similarity, considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.11 and 5.12.

By observing these graphics, we can conclude that the similarity presents better performance in relation to the others in most of the cases. However, WSA can be better when considering a path coverage bellow 30%. This can be an evidence that - bellow 30% - the importance and length of the test cases are the same and random choice is applied.

Transition Coverage - Concluding Remarks

For both applications, we can see through the graphics showed in Figures 5.9, 5.10, 5.11 and 5.12 that Similarity presents a better performance - in most of the cases. However, when considering the probabilities based approaches (WSA and Guided Random), WSA presents generally a better performance since it inherits the similarity principles.

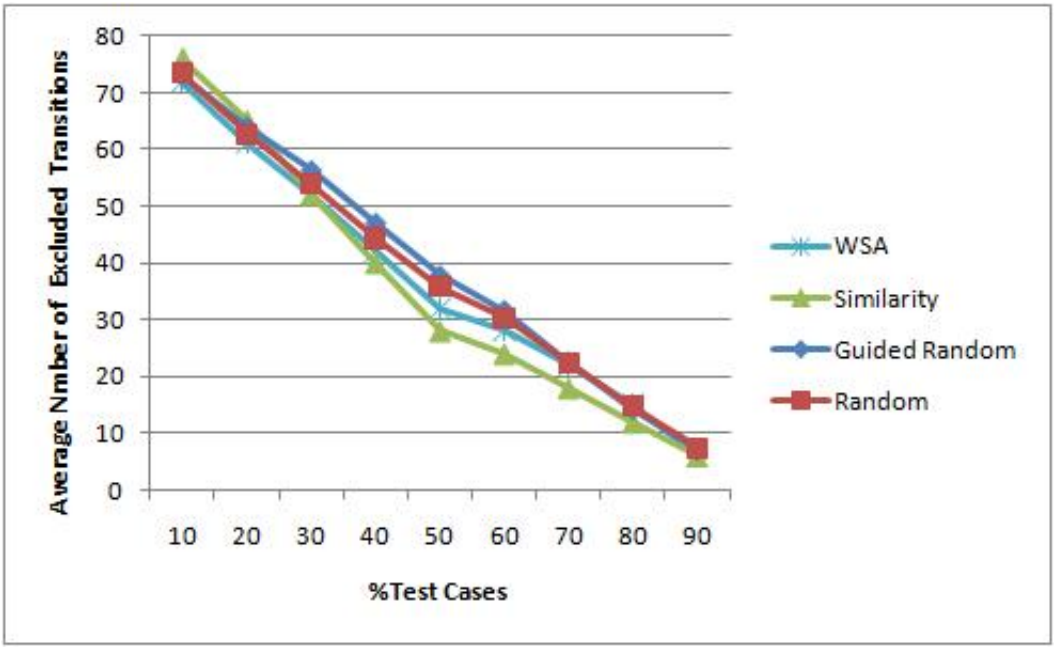


Figura 5.11: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 2.

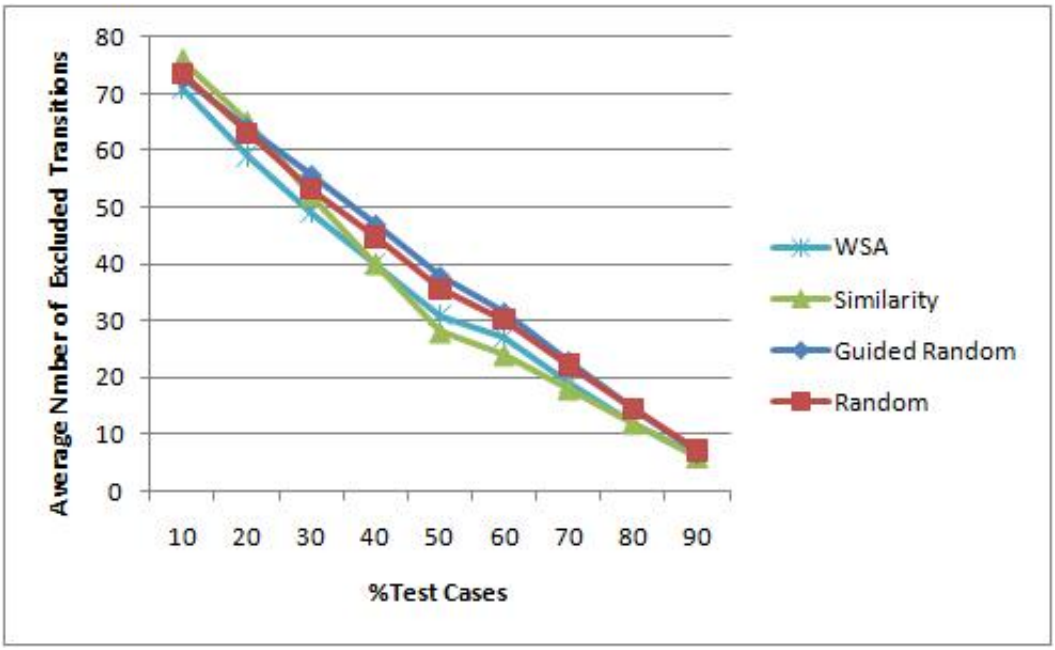


Figura 5.12: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 2.

5.4 Concluding Remarks

In this Chapter, we presented our proposal for test case selection based on Similarity (Chapter 3) that also considers weights that are assigned to test cases (WSA). The goal of WSA is to keep in test suite the most important and different test cases. Case studies were performed, and the results show evidence that:

- **Fault Coverage:** WSA is an effective strategy, but it is necessary to know and/or guess the point of the faults;
- **Transition Coverage:** Similarity - in most of the cases - presents a better performance.

The main threats to validity of the performed case study are related to the weights assigned. In the case study the weights were assigned by only two subjects. One of them (WSA designer) had knowledge about the faults and the strategies. The use of more subjects to assign different weights would provide a more general overview of the analyzed strategies. Also no guidelines were used to assign the probability values, which may affect the performance of some strategies (e.g. WSA and Guided Random). Aside from those threats the use of only two models affect the generalization of the obtained results.

Capítulo 6

Redução baseada em Dissimilaridade

Neste capítulo apresentamos nossa proposta para Redução de Suítes de Teste baseada em Similadidade (Capítulo 3). O objetivo da redução de suítes de teste, como dito em capítulos anteriores, é manter a cobertura de um requisito de teste. Entretanto, a cobertura de faltas, normalmente é afetada. Uma vez que nós obtivemos resultados satisfatórios (considerando cobertura de faltas) aplicando estratégias de Similaridade anteriormente, nosso objetivo é aplicar a função de similaridade para escolher os mais diferentes casos de teste, que possivelmente cobrirão mais requisitos de teste enquanto guardam uma boa cobertura de faltas.

Na Seção 6.1, nós apresentamos a idéia e na Seção 6.2, um exemplo para ilustrar a estratégia. Um estudo de caso real foi executado para comparar nossa estratégia a 4 heurísticas bem conhecidas na literatura, analisando cobertura de transições e faltas (Seção 6.3). Na Seção 6.4 é mostrado o experimento que executamos analisando a cobertura de transições.

Os resultados do estudo de caso mostram evidências que Dissimilaridade apresenta o pior percentual de redução comparada as Heurísticas, porém apresenta a melhor cobertura de faltas. Analisando os dados de Dissimilaridade, é necessário - em média - executar 3.06% mais casos de teste comparada a melhor estratégia. Por outro lado, o percentual de faltas é incrementado em 9.26%. Para o estudo de caso apresentado, isso significa executar 2.57 mais casos de teste, porém ser capaz de revelar 1.2 mais faltas.

Os resultados do experimento confirmam o resultado obtido com o estudo de caso: Dissimilaridade apresenta o pior percentual de redução. Neste caso considerando apenas 75% de cobertura de transições como requisito de teste.

Maiores detalhes são apresentados nas próximas seções.

6.1 Definition

The idea is to keep in the reduced test suite the most different test cases while providing 100% coverage of one defined test requirement (in our case, transitions coverage). Then, the test cases are chosen according to the degree of similarity and are placed in a reduced set named as the Reduced Test Suite (RTS).

Overall, this strategy uses the similarity function to build the similarity matrix (as showed in Chapter 3). The inputs are:

- **Test Suite:** the set of test cases that should be reduced;
- **Test Requirements:** the set of transitions that should be covered;
- **Similarity Matrix:** the matrix that contains the information about similarity among all test cases of the test suite.

Since the proposal of any test suite reduction technique is to cover 100% test requirements, then it is important to identify all essential test cases, since an essential test case is the only one that covers a specific requirement. Therefore, all essential test cases should be in the RTS for reaching 100% of test requirements.

The algorithm of this strategy is presented in Algorithm 3. The first step of this strategy is to remove all essential test cases from the similarity matrix, and add all of them to the RTS (lines 1 and 2). Then, all test requirements satisfied by those test cases are marked (line 3). After that, the idea is to find, in the matrix, the minimum value that represents the most different test cases and try to keep them in the RTS, always verifying if all test requirements have already been covered (lines 4 - 30).

When a tie among minimum values (more than one minimum value) is found, in the similarity matrix, the idea is to choose the pair that covers the maximum number of not yet covered requirements, however if there is a new tie, a random choice is applied (lines 5 - 7).

The next step (lines 10 - 17) is to verify if the test cases (of the chosen pair) are 1-to-1 redundant. This occurs when the set of covered requirements of one of them is contained within the other one. In this case, the idea is to place in the RTS, the test case that covers more not yet covered requirements so far. Since the two test cases were already analyzed,

they are removed from the similarity matrix, and all new covered requirements are placed in the marked requirements set.

If the pair is not 1-to-1 redundant, then it is necessary to check if the requirements covered by each one of them are already covered (lines 19 - 20 and 25 - 26), because if the requirements have already been covered, these test cases are redundant. Otherwise, the test cases are added to the RTS and the new satisfied requirements are added to marked requirements set (lines 21 - 24 and 27 - 30).

Regarding the complexity analysis of Algorithm 3 we are able to observe a repeating structure (`while` command in line 4) that repeats *m* times, where *m* is the number of the test requirements. The method `getAllMinValue` searches the matrix for the lowest values of similarity, having, thus, a complexity $O(n^2)$, where n is the number of test cases in the test suite. Considering that the method `getAllMinValue` is executed m times, we obtain a complexity $O(m \times n^2)$ for Algorithm 3.

6.2 Example - Dissimilarity

In order to illustrate the strategy, an example is presented below. An LTS model is presented in Figure 6.1. From this LTS model, 6 test cases are obtained as can be seen in Table 6.1.

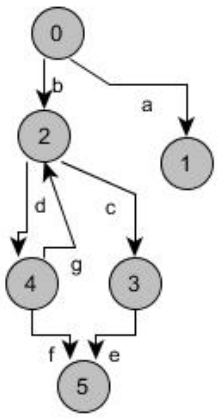


Figura 6.1: Example - LTS model

Considering that, the test requirement (coverage criteria) is transition coverage, the RTS should cover all transitions. By applying the presented idea, the matrix is showed in Matrix

```
input  : testSuite, testRequirement, similarityMatrix
output: reducedTestSuite
similarityMatrix.removeEssentialTestCases();
reducedTestSuite.add(essentialTestCases);
markedRequirements.add (satisfiedRequirements(essentialTestCases));
while (!(satisfyAllRequirements(testRequirements,markedRequirements)) do
    minValues = getAllMinValue(similarityMatrix);
    pairs = PairsCoverMaxNumOfNotCoveredRequirements(markedRequirements,minValues);
    choosedPair = pairs.shuffle.get(0);
    testCase1 = choosedPair.getTestCase1();
    testCase2 = choosedPair.getTestCase2();
    if (containsRequirements(testCase1,testCase2)) then
        similarityMatrix.remove(testCase1,testCase2);
        markedRequirements.add(satisfiedRequirements(testCase1));
        reducedTestSuite.add(testCase1);
    else if (containsRequirements(testCase2,testCase1)) then
        similarityMatrix.remove(testCase1,testCase2);
        markedRequirements.add(satisfiedRequirements(testCase2));
        reducedTestSuite.add(testCase2);
        else
            if (containsRequirements(markedRequirements,testCase1)) then
                similarityMatrix.remove(testCase1);
            else
                similarityMatrix.remove(testCase1);
                markedRequirements.add(satisfiedRequirements(testCase1));
                reducedTestSuite.add(testCase1);
            if (containsRequirements(markedRequirements,testCase2)) then
                similarityMatrix.remove(testCase2);
            else
                similarityMatrix.remove(testCase2);
                markedRequirements.add(satisfiedRequirements(testCase2));
                reducedTestSuite.add(testCase2);
```

Algorithm 3: Dissimilarity-based Reduction - Algorithm

Tabela 6.1: Test Cases and Size of test cases

TC id	Path	Test Size
1	a	1
2	b c e	3
3	b d f	3
4	b d g	3
5	b d g d f	5
6	b d g c e	6

6.1. The size of test cases are also presented in Table 6.1.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 & 0 \\ TC2 & & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & & 0.75 & 0.75 \\ TC5 & & & & & & 0.6 \\ TC6 & & & & & & \end{pmatrix} \quad (6.1)$$

Since the test requirements are transitions, then $testRequirements = \{a, b, c, d, e, f, g\}$. The first step is to find essential test cases. For this example, TC1 is essential. Therefore, the variables $markedTestRequirements$ and $reducedTestSuite$ are updated: $markedTestRequirements = \{a\}$; $reducedTestSuite = \{TC1\}$. The new matrix is presented in Matrix 6.2.

$$SimilarityMatrix = \begin{pmatrix} & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC2 & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & 0.75 & 0.75 \\ TC5 & & & & & 0.6 \\ TC6 & & & & & \end{pmatrix} \quad (6.2)$$

There is only one minimum value (0.25) found between TC2 and TC5. They are not

1-to-1 redundant test cases, and the satisfied requirements (covered transitions) by each one of them have not been covered yet. Thus, both test cases are added to RTS, and, the variables are updated again: $markedTestRequirements = \{a, b, c, d, e, f, g\}$; $reducedTestSuite = \{TC1, TC2, TC5\}$.

Finally, all requirements are satisfied by the RTS composed by TC1, TC2, TC5. In other words, $markedTestRequirements = testRequirements$.

6.3 Case Study

A case study was executed. The goal of this case study is to compare the fault detection capability and the RTS size for the Heuristics - G, GE, GRE and H - and Dissimilarity. The defined test requirement is 75% of transitions coverage. This number was defined because the stop criteria is the amount of covered transitions, and in this case if all transitions are covered, all faults will be revealed (the faults are linked to transitions). Therefore, decreasing the test requirements from 100% to 75%, provides a more realistic and fair comparison.

6.3.1 Application

The application used for this is a desktop tool named TaRGeT, that automatically generates and selects test cases [47]. In order to execute the case study the tool LTS-BT [16] was used. For this case study, the input is a use case template [47], written by Motorola experts. This template contains information regarding the use cases and the scenarios, in the application, that can be executed by a user. All test cases, generated for this case study, were manually executed by Motorola employees.

The objective of this case study is to analyze the behavior of the strategy concerning aspects of the reduced test suite. Therefore we need to observe how much of the test suite the strategy is able to reduce, and also how many faults were able to be detected by executing the reduced test suite. These information can be obtained from the following metrics:

- **Test Suite Size:** The size of a test suite is measured by the number of test cases that it contains. In this study, the test suite size is 84 test cases;

- **Fault Coverage:** As said before, all test cases are executed manually. From the 84 test cases, 13 revealed failures. And each one of these failures were caused by a different fault. Summarizing, the complete test suite revealed 13 different faults.

6.3.2 Case Study - Preparation

As said before, the test requirement is 75% of the transitions of the model. Thus, for each execution, 75% of transitions are randomly chosen, from the model, in order to establish the test requirement set. Since the set of test requirements is different, the results can change. Therefore, 3 sets of test requirements, i.e. three different sets that cover 75% of the transitions, were applied to analyze each strategy.

In other hand, the Heuristics and the Dissimilarity strategy present a random choice in their algorithms. Thus, each one of them was executed one hundred times, for each set of test requirement. The results are presented in the next subsection.

6.3.3 Results of the Case Study

This subsection presents the results of the case study. As a reminder, the variables considered in the study are the size of the RTS and the fault coverage.

Reduced Test Suite Size

By applying Dissimilarity and the heuristics to reduce the test suite, the different RTS size can be observed. The Table 6.2 presents the obtained results (for one hundred executions) for each one of the 3 different, randomly defined, set of test requirements.

Tabela 6.2: Average of RTS size (100 executions) for all 3 sets of test requirements

	Dissimilarity	GRE	GE	G	H
1	31.78	29	29	29	29.52
2	32.08	30	30	30	31.89
3	29.83	27	27	27	29.36
Average	31.23	28.66	28.66	28.66	30.25

The results show that GRE, GE and G present better results, followed by H. In this case, Dissimilarity presents the worst results. In average, Dissimilarity reduces the test suite 62.82 %; G, GE and GRE, 65.88%; and H, 63.99%. The Figure 6.2 presents the graph that shows the obtained average for the 3 sets of test requirements, in each one of the 100 executions.

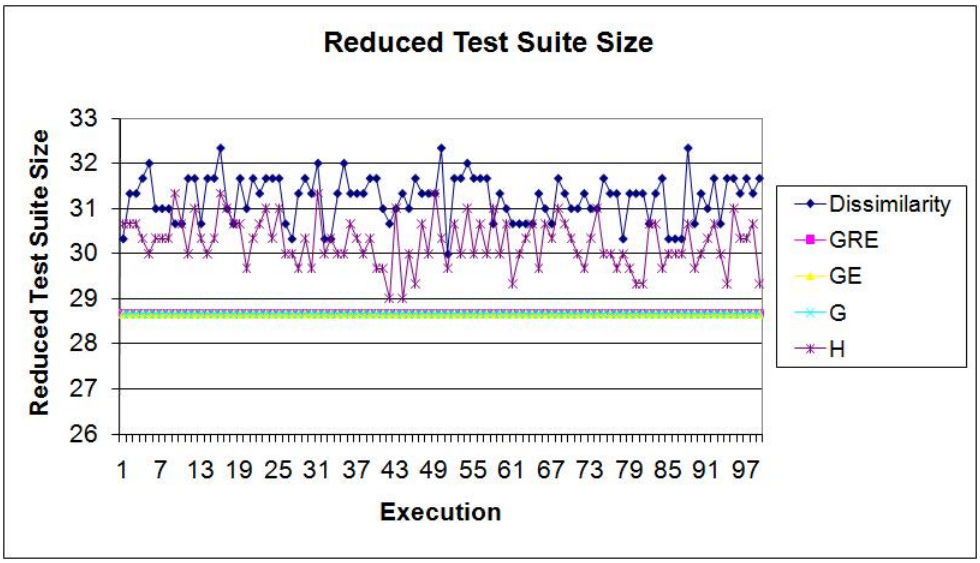


Figura 6.2: TaRGeT - Reduced Test Suite Size

Fault Coverage

The obtained results for each strategy, considering the 3 different sets of test requirements, are presented in Table 6.3.

Tabela 6.3: Average of test suite reduced size (100 executions) for all 3 sets of test requirements

	Dissimilarity	GRE	GE	G	H
1	8.92	7	7	7	7
2	4.51	4	3.58	3.48	3.99
3	5.2	4.68	5.17	5.26	5.26
Average	6.21	5.22	5.25	5.24	5.41

The results show that most of the times, the Dissimilarity strategy reveals more faults. In average, Dissimilarity reveals 51.28% of the total amount of faults; H reveals 43.58%; GRE and GE, 42.02%; and G, 38.46%.

The Figure 6.3 presents the graph that shows the obtained average by considering the 3 sets of test requirements in each one of 100 executions. The conclusion is that, by considering each execution (i.e., the average among the 3 sets of test requirements), Dissimilarity reveals more faults.

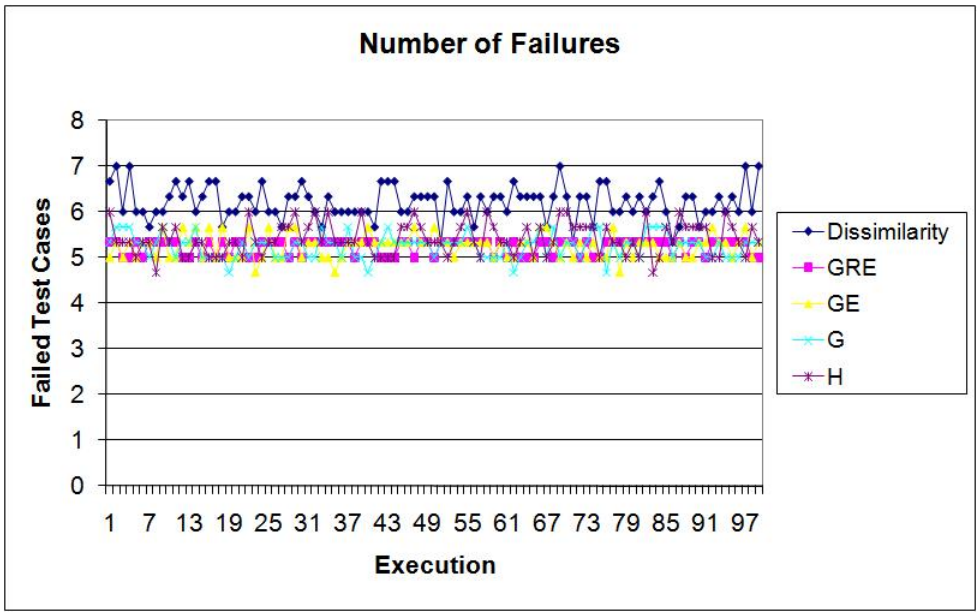


Figura 6.3: TaRGeT - Failures

Concluding Remarks - Case Study

The Table 6.4 presents the summary of this case study, in terms of the percentage of reduction and fault coverage for Dissimilarity, G, GE, GRE and H.

Tabela 6.4: Summary - Percentage of Reduction and Fault Coverage

	Dissimilarity	GRE	GE	G	H
Percentage of Reduction (%)	62.82	65.88	65.88	65.88	63.99
Percentage of Fault Coverage (%)	51.28	42.02	42.02	38.46	43.58

Summarizing the data of the table: Dissimilarity presents the worst percentage of test suite reduction (the best is reached by applying GRE ,GE or G), however it presents the best percentage of the revealed faults (the worst is obtained by applying G). Analyzing the data from Dissimilarity, it is necessary - in average - to execute 3.06% more test cases, than the

best strategy. In other hand, the percentage of the faults is increased by 9.26%. For this case study, this means that executing 2.57 more test cases, we are able to reveal 1.2 more faults.

The use of only one model can be considered a threat to validity of this case study. Since we measure structural elements from the LTS, mainly the transitions, using more models would provide a more general overview of the strategies. Another threat to validity is the chosen test requirement. In order to generalize the results, it would be necessary to use different test requirements (e.g. fault coverage, requirements coverage, among others) and observe the behavior of each strategy.

6.4 Experiment - Reduction

This Section presents the executed experiment and results. The used framework was presented in Chapter 2. The process was followed and the results are presented in the next subsections.

Our general hypothesis is that Dissimilarity presents the best performance in relation to the rate of reduction, considering, as test requirement, 75% of transitions.

6.4.1 Definition

To define the goal, the key questions proposed by Wohlin *et al.* were answered[61]:

1. **What is studied?** reduction stragegies;
2. **What is the intention?** to investigate;
3. **Which effect is studied?** reduced test suite size;
4. **Whose view?** the tester;
5. **Where is the study is conducted?** Model-Based Testing (MBT).

From these answers, the goal definition template is filled. In summary, the goal of this experiment is:

Analyze *reduction strategies*
for the purpose of *investigating*

with relation to *reduced test suite size*
from the point of view of the *tester*
in the context of *MBT*.

The (input) objects are LTS models. In this case, this experiment does not present subjects.

6.4.2 Planning

For this phase, it is necessary to define [61]: context selection, variable, hypothesis, design and instrumentation.

Context Selection

The context of this experiment can be characterized as a "toy vs. real" problem [61]. In this case the objects are LTS models, randomly generated from a configuration. This configuration is characterized by a specific number for the depth of the LTS, the number of loops, branches and joins (these elements are detailed in Appendix B.

Variables Selection

The variable chosen to observe (dependent variables) and to control (independent variables) are:

- **Dependent:** The Reduced Test Suite Size - RTSS .
- **Independent:** The test requirement percentage; the configuration chosen for the depth and amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 5 levels: G, GE, GRE, H and Dissimilarity (DSim).

Hypothesis Formulation

The experiment definition is formalized as following:

- **A null hypothesis (H_0):** $RTSS_G = RTSS_{GE} = RTSS_{GRE} = RTSS_H = RTSS_{DSim}$: All strategies have a similar behavior in relation to the reduced test suite size;

- **An alternative hypothesis, H_1:** $RTSS_G \neq RTSS_{GE} \neq RTSS_{GRE} \neq RTSS_H \neq RTSS_{DSim}$: All strategies have a different behavior in relation to the reduced test suite size.

Experiment Design

As seen before, there is one factor (test suite reduction strategy) with 5 levels (or treatments). Thus, there is one factor and 5 treatments, where, for each object, all five treatments are applied. The chosen confidence level is 95% (significance level is $\alpha = 0,05$).

Aiming to define the number of replications, necessary to guarantee statistical significance for the specified level of confidence (95%), 40 replications were executed and the data are presented in Table 6.5.

Tabela 6.5: Mean, Standard Deviation and number of necessary replications for each technique.

Technique	G	GE	GRE	H	DSim
Mean ($\overline{x}$)	4.3	3.75	3.75	3.75	3.85
Standard Deviation (s)	1.26	1.05	1.05	1.05	1.02
Number of Necessary Replications (n)	133	122	122	122	110

Observing the Table 6.5, we are able to see that 133 replications provide a statistical significance for the obtained data. Therefore, this experiment design will consider 200 replications for each technique, in order to better explain the results (since they are expressed using percentages).

Instrumentation

In this step, there are three types of instruments [61]:

- **Objects:** The objects are LTS models randomly generated from a configuration (depth, number of loops, branches and joins).

- **Guidelines:** This experiment uses no guidelines, since the strategies do not require subjects to configure them.

- **Measurements:** The RTS size will be collect for each treatment. The tool LTS-BT provides support for executing the experiments and collecting the data.

Validity Evaluation

The objects used in this experiment, can be considered the main threat to validity. They can not represent a real behavior since these are automatically generated. From a specific configuration, the objects are randomly generated from a specific configuration, both the traceability and controllability of the elements of the model (transitions and states) are reduced.

On the other hand, we are able to obtain an overview of the execution of the strategies in several models that has a same configurations, since they are randomly generated. Then, we avoid presenting a conclusion that is specific to only one LTS of a configuration. A proper scenario would be to have several real applications to execute the strategies. However, most real applications and their respective specification are not available to the public.

6.4.3 Operation

To execute this experiment, it was necessary to implement the 5 strategies and the LTS generator (see Appendix B). Both the LTS generator and the strategies are implemented in Java programming language[1].

The objective is to automatically generate different models using the LTS generator. In order to depict a real application in these generated LTS, the configuration used for the generator was chosen from observing a real application (used in the case study - Section 6.3). Then the chosen configuration is:

- **Depth:** 15;

- **Number of loops:** 2;

- **Number of branches:** 3;

[1] http://www.sun.com/java/

- **Number of joins:** 3.

There is only one experimental design (with only one factor - test suite reduction strategy) with a null and an alternative hypothesis, where the intention is to reject the null hypothesis. Each strategy was executed 100 times, using a machine with the following configurations:

- Intel Core 2 quad 2.33 GHz;
- 4GB RAM;
- 1TB for Hard Disk Memory.

6.4.4 Analysis and Interpretation

The confidence intervals are plotted for each strategy. The graph can be seen in Figure 6.4. Since the confidence intervals of G, GE, GRE and H overlap, a statistical test is required [39] to test the hypothesis. In this case, since the confidence interval for G, GE, GRE and H is overlapped, we will do first a comparison among them. In order to show the comparison among the four strategies, the statistical tests are applied and the results are presented in Appendix C. In this Appendix, by applying the statistical test, we concluded that the obtained results can not be considered different, for the heuristics (G, GE, GRE and H).

Since the behavior of the heuristics can not be considered different, the Dissimilarity strategy needs to be compared with only one of them. Once that Dissimilarity includes some concepts from GRE, the comparison will be between GRE and Dissimilarity. The experiment definition is formalized as following:

- **A null hypothesis (H_0):** $RTSS_{GRE} = RTSS_{DSim}$: All techniques have a similar behavior in relation to the reduced test suite size;
- **An alternative hypothesis (H_1):** $RTSS_{GRE} \neq RTSS_{DSim}$: All techniques have a different behavior in relation to the reduced test suite size.

The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[2]. The results can be seen in Figures 6.5 and 6.6. In this graph, the red dots, should overlap the blue line, in order to indicate that the data fit a normal distribution.

[2]http://www.minitab.com/

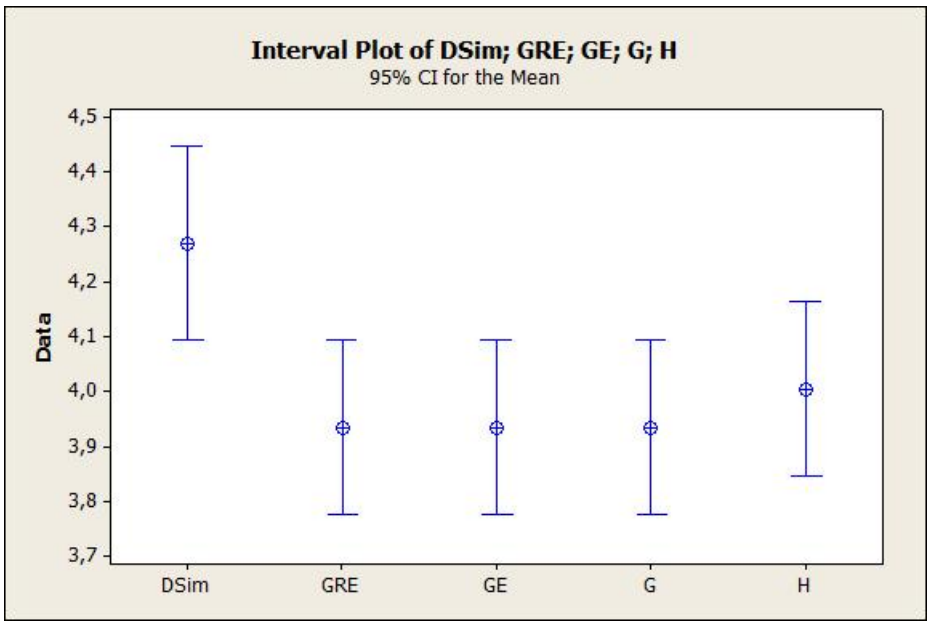


Figura 6.4: Interval Plot

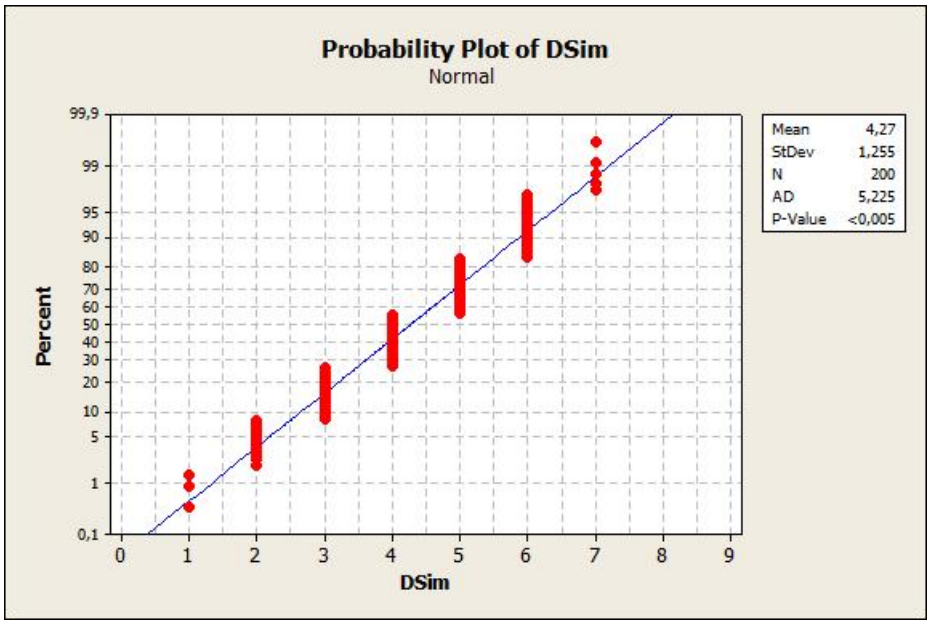


Figura 6.5: Anderson-Darling normality test - Dissimilarity

The $p - Values$ are smaller than the significance value ($\alpha = 0.05$), thus the data do not show a normal distribution. In this case, it is necessary to use a non-parametric test. Since this experiment has a factor with two treatments (GRE and Dissimilarity), the Mann-Whitney is applied to check the null hypothesis. The results for the test can be seen in the Table 6.6.

Since $p - Value = 0.0074$, and the $p - value$ is lower than 0.05 (α), H_0 (the null

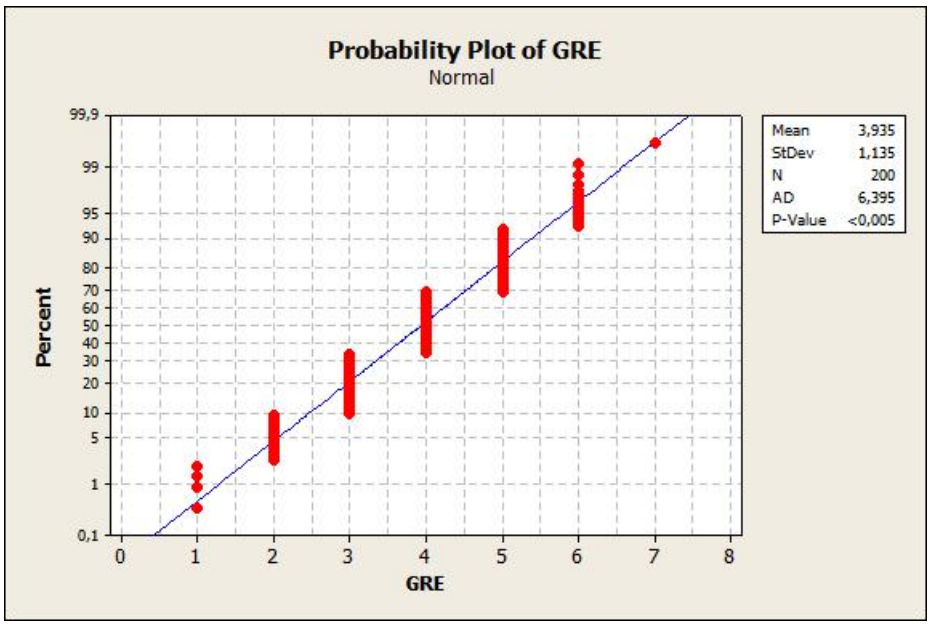


Figura 6.6: Anderson-Darling normality test - GRE

Tabela 6.6: Mann-Whitney Test - GRE and DSim

Technique	N	Median
GRE	200	4.000
DSim	200	4.000
The test is significant at 0.0074		

hypothesis) can be rejected. The box plot that presents the difference of the RTSS between DSim and GRE can be seen in Figure 6.7. By this figure, we can conclude that in the most cases, the difference is 0 or 1, but in some cases (not often) can be 2.

6.4.5 Concluding Remarks - Experiment

Summarizing the data of the experiment: Dissimilarity presents the worst percentage of test suite reduction. The best is reached by applying GRE ,GE or G. However, we can conclude that the difference is 0 or 1 or 2.

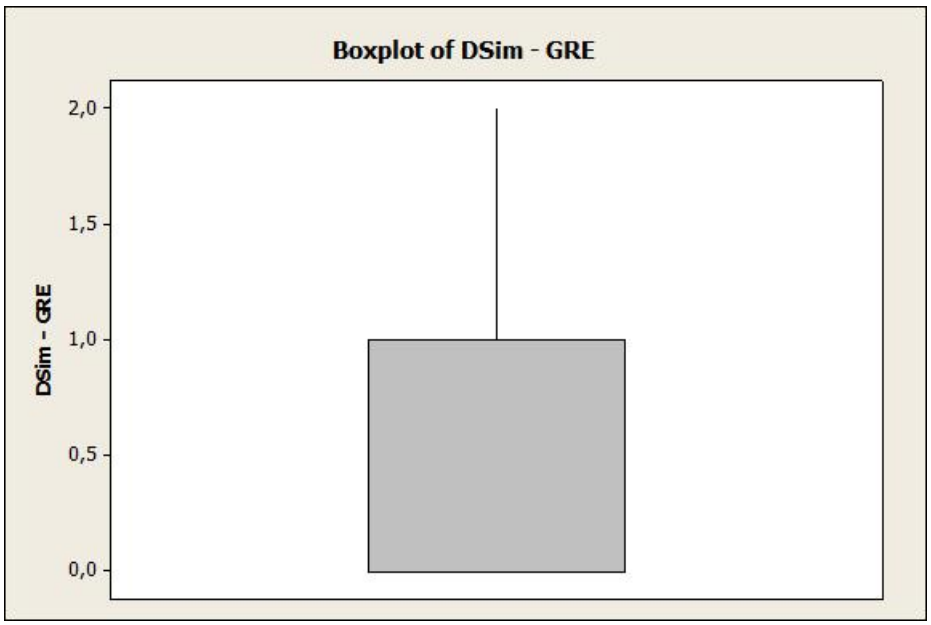


Figura 6.7: Box Plot RTSS of DSim - GRE

6.5 Concluding Remarks

In this Chapter, we presented our proposal for test suite reduction based on Similarity (Chapter 3) named as Dissimilarity. A case study and an experiment are performed. The results of the case study show evidence that Dissimilarity presents the worst percentage of test suite reduction (the best is reached by applying GRE ,GE or G), however it presents the best percentage of the revealed faults (the worst is obtained by applying G). Analyzing the data from Dissimilarity, it is necessary - in average - to execute 3.06% more test cases, than the best strategy. In other hand, the percentage of the faults is increased by 9.26%. For this case study, this means that executing 2.57 more test cases, we are able to reveal 1.2 more faults.

The experiment confirms one of the obtained results of the case study: Dissimilarity presents the worst percentage of test suite reduction. In this case, considering 75% of transition coverage, as the test requirement.

Capítulo 7

Analisando Redução baseada na Ordem de Seleção

Neste Capítulo introduzimos uma nova perspectiva para avaliar estratégias de redução de suítes de teste baseada na taxa de detecção de faltas. Este critério é padrão para avaliar estratégias de priorização de casos de teste, mas até então não foi utilizado para redução. Nossa proposta considera que o teste pode ser parado antes da execução completa da suíte de teste e que a ordem com que os casos de teste são colocados na suíte de teste reduzida é também importante.

Diferentes estudos na literatura têm sido desenvolvidos para comparar estratégias de priorização e redução considerando um único ponto de vista: eficácia em diminuir o tamanho das suítes de teste e o impacto na efetividade da detecção de faltas. Uma prática comum é comparar estratégias considerando que todos os casos de teste serão executados. Entretanto, sob restrições de orçamento, deve-se executar um menor número de casos de teste do que aqueles que realmente deveriam ser executados.

Nós comparamos quatro heurísticas de redução considerando a taxa de detecção de faltas e mostramos que a estratégia de redução a ser escolhida quando o tempo está em questão pode ser diferente daquela que apresentava a melhor performance quando executava a suíte completa.

Nossos resultados sugerem que - provavelmente - um novo modo de medir a performance das heurísticas é necessária, levando em consideração a variabilidade de número de casos de teste a serem executados e as faltas detectadas até agora.

Maiores detalhes são apresentados nas próximas seções.

7.1 Motivation

Different test reduction heuristics have been proposed. The common practice to compare them is by considering that the whole reduced test suite will be executed [35; 19]. Unfortunately during development managers might be forced for many reasons to stop the testing earlier than planned, and thus a lower number of test cases is run than those in the reduced test-suite. The ideal solution would be to maximize the number of failures detected while selecting a subset of non-redundant test cases that covers all requirements.

To motivate such an approach, let us consider the same toy example presented in [29], where a program is supposed to contain 10 faults and a test suite of 5 test cases, called for simplicity (A,B,C,D,E), is available. Table 7.1 shows the fault detection capability of each test case. When all test cases are executed, independently of their order, the percentage of fault detection is always 100%. To select the best prioritization technique the Average Percentage of Fault Detection (APFD) measure reached by the associated combination of test cases has been proposed. For instance, as described in [29], if three different prioritization techniques are applied that produce the sets (A,B,C,D,E), (E,D,C,B,A), (C,E,B,A,D), they yield respectively the following APFD: 50%, 64 % and 84%. Hence the third one gives the best ordering.

Now, let us suppose to apply three different reduction techniques on the same test suite,

Tabela 7.1: Test Suite and Faults exposed

Test	faults									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Tabela 7.2: APFD of the considered reduction heuristic

#TC	TS1	TS2	TS3
1	40	20	40
2	35.5	22.5	30
3	47.85	34.65	37.95
4	47.85	47.5	46.25
5	47.85	47.5	62

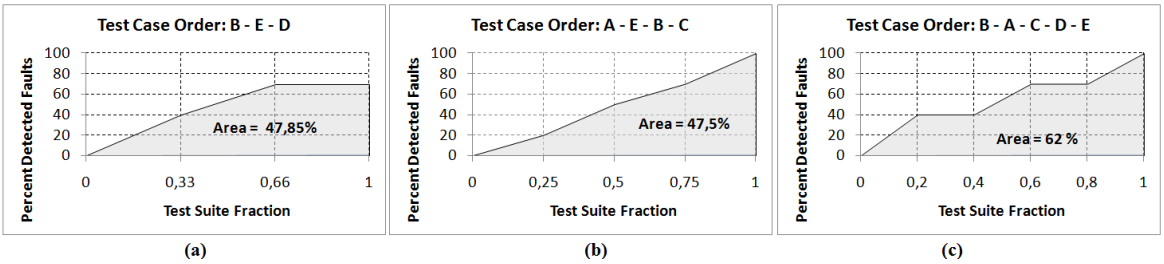


Figura 7.1: Test Case Order

obtaining the following reduced sets:

- TS1 = (B,E,D)
- TS2 = (A,E,B,C)
- TS3 = (B,A,C,D,E)

We are assuming that all of the three sets reach 100% requirements coverage, hence TS1 performs the best in terms of test size reduction. However, it should be considered that with only 3 test cases TS1 discovers the 70% of the faults, while TS2 and TS3 detect the 100%. We believe that in test reduction a practical compromise that takes into consideration both the number of test cases executed and the rate of fault detection should be defined. For instance if we use the APFD measure for comparing the respective rates of fault detection of the three test suites, we get the results reported in Figure 7.1: TS1 has 47.85% of APFD, TS2 has 47.5% and TS3 has 62%. Thus the best technique in terms of quickly detecting faults would be TS3, but it is the one having the worst performance in terms of test suite reduction.

On the other hand, if for any reasons the test phase needs to be stopped after only two test case are executed, which is the relative performance of the three test suites? If the APFD of TS1, TS2, TS3 are again evaluated considering only their first two tests, the following results can be observed:

- TS1_r = (B,E) detects 7 faults and reaches the 35.5% of APFD
- TS2_r = (A,E) detects 5 faults and reaches the 22.5% of APFD
- TS3_r = (B,A) detects 4 faults and reaches the 30% of APFD

This changes the previous measures. The best APFD when only two test cases have been executed belongs to TS1 and not anymore to TS3. Thus if testing is stopped before all test cases in the reduced test-suite are run, the ordering in the reduced test-suite is important for selecting the most effective test strategy. This is why we propose to mix the concepts of reduction and prioritization.

It is important to notice that the kind of data analyzed in the above example would only be available a posteriori. This example is used to evidence the existence of different points of view in the evaluation of a test reduction strategies when taking in consideration also realistic problems. However, the fault detection of each test case is available only after the test case is executed and not before, and constraints forcing the manager to stop testing in advance cannot be foreseen.

From a practical point of view the knowledge about the rate of fault detection of reduction heuristics could be derived by the application of fault seeding or from the history of test execution for similar products.

7.2 General definition

From the example discussed in the previous section, we think that test reduction and test prioritization can be seen as two aspects of a more general problem that is to select an optimal set of test cases under the existing constraints that minimizes redundancy (via reduction) and maximizes fault detection (via prioritization). In this section we formalize the procedure followed for deriving Table 7.2 and consequently provide a general definition of a criterion to select the reduction heuristics to be applied.

In particular, considering the definition of APFD of [57], given a program having a number of faults equal to m, a number of requirements equal to q and an ordered test suite $TS = (T_1, ...T_n)$ of cardinality n, the following functions can be defined:

- $TF(i)$ for $1 \leq i \leq m$ represents the position of the first test case in TS that exposes the fault i
- For $1 \leq h \leq n$ $req(T_h) = \{r_p | 1 \leq p \leq q$ and r_p is a requirement covered by $T_h\}$ represents the set of requirements covered by T_h

- For $1 \leq h \leq n$ $r(T_h) = |req(T_h)|$ represents the number of requirements covered by T_h

Consequently if j, $1 \leq j \leq n$, represents cumulative number of test cases executed during a testing phase till a certain point in time, and f, $1 \leq f \leq m$ is the cumulative number of faults in $T_1, ..T_j$, the following cumulative functions can be defined:

- $APFD(j) = 1 - \frac{TF(1)+...+TF(f)}{jf} + \frac{1}{2j}$ represents the incremental APFD after the execution of j test cases and the detection of f faults
- $REQ(j) = \bigcup_{i=1}^{j} req(T_i)$ represents the set of requirements covered by the execution of the subset $T_1, ..T_j$
- $R(j) = |REQ(j)|$ represents the cumulative number of requirements discovered by the execution of the first j test cases

In this case if q represents the cumulative amount of requirements to be covered for a specific system, and m is the cumulative amount of faults, then when $j = n$ the previous formulas become:

- $APFD(n) = 1 - \frac{TF(1)+...+TF(m)}{nm} + \frac{1}{2n}$ represents the standard formula of APFD
- $REQ(n) = \bigcup_{i=1}^{n} req(T_i) \leq q$ represents the set of requirements covered by the reduced test suite $T_1, ..T_n$
- $R(n) = |REQ(n)|$ represents the cumulative number of requirements discovered by the execution of the reduced test suite

In general, given q and m as above, let us assume that k different heuristics, $H_1, ...H_k$ are available for test reduction, such that the reduced test suites have cardinality $h1, ..., hk$ respectively and are represented by $(T_{h1_1}, ..., T_{h1_{h1}})$, ..., $(T_{hk_1}, ..., T_{hk_{hk}})$.

To take into account the number of executed test case, we denote by $S \in (H_1, .., H_k)$ an index representing the heuristic having the best fault detections effectiveness, then:

for every j such that $1 \leq j \leq (max(h1, .., hk)$

- $\bigvee 1 \leq i \leq k$ Calculate the $APFD(j)_{Hi}$. If $j > hi$ than $APFD(j)_{Hi} = APFD(hi)_{Hi}$

- Determine $MAX(j) = max(APFD(j)_{H1}, ...APFD(j)_{Hk})$
- Define $S = Hs$ such that $APFD(j)_{Hs} = MAX(j)$.
- If there are two (or more) heuristics, Hp and Hq, such that $MAX(j) = APFD(j)_{Hp} = APFD(j)_{Hq}$ then

  if $R(j)_{Hp} \geq R(j)_{Hq}$ then $S = Hp$

  otherwise $S = Hq$

The last rule simply says that in case the two heuristics have the same fault detection effectiveness the heuristic yielding the highest requirements coverage is selected.

The procedure above construct the referring table of the different APFD and provide a guideline for manager to select the heuristic having the best performance depending on the number of test cases to be executed.

7.3 Case Studies

In two real-world case studies we compared four well-known test suite reduction heuristics – G, GE, GRE and H – from the new viewpoint. The idea is to measure the rate of fault detection in order to show that the techniques based on these heuristics may present a different performance when considering coverage in different intervals of the ordering of selection up to 100% coverage of the test requirements. As mentioned before, the reason is that the heuristics may pick test cases in a different order. Depending on the order, the rate of fault detection can be maximized or not with the first few test cases selected.

7.3.1 Case Studies Design

The heuristics were implemented in Java programming language using the LTS-BT tool [16] as execution environment. These heuristics receive a test suite TS and a satisfiability relation. After processing the suite, the output is the reduced test suite.

Figure 7.2 illustrates the whole process for obtaining a test suite and reducing it using a heuristic. In this figure, the round-edge rectangles represent components of LTS-BT and oval forms represent the artifacts produced by the components. These are used as input for

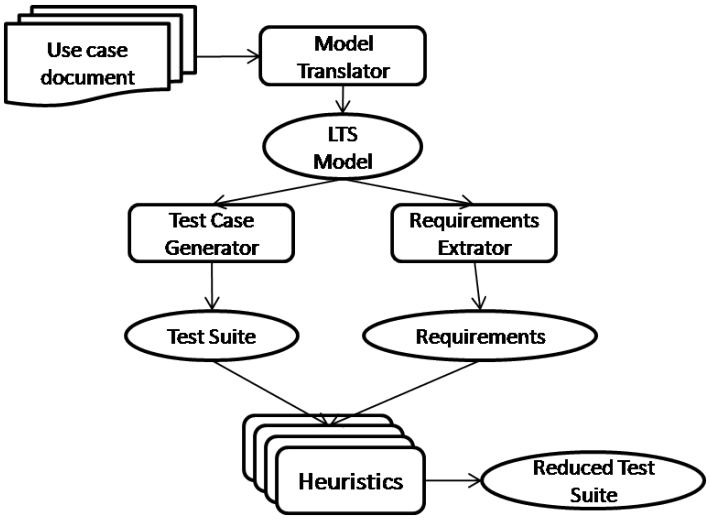


Figura 7.2: Overview of a test suite reduction process

the next component. LTS-BT is a model-based testing tool that automatically generates test cases from use case documents. Initially the use case document is translated into an LTS from which the test suite TS is derived and the requirements mapping $ReqM$ is defined. Then, for the reduction process, the heuristics receive TS and $ReqM$ as input and after processing the output, the reduced test suite is produced.

For the case studies, we considered transition coverage as requirement. This means that each transition in the LTS represents a test requirement and the test cases that contain a given transition satisfy the corresponding test requirement.

Applications under testing

Two real-world applications provided by Motorola Software Engineers have been selected for the case studies. They are briefly described as follows.

Application 1: TaRGeT is desktop application that supports the model-based testing process, where it automatically generates test cases from use case documents.

Application 2: Direct License Acquisition (DLA) Support is a feature for mobile phone applications that handles acquisition of the WMDRM License (Windows Media Digital Rights Management) for the Windows Media platform. This License provides secure delivery of audio and/or video content.

For each one of the applications, Motorola Software Engineers elaborated the use case

documents [47]. From these documents, LTS-BT generated the test suites that have the following particularities.

Application 1: 84 test cases that present redundancy, taking into account the transitions as test requirements, i.e., there are some test cases that cover the same requirement. Therefore the heuristics are able to reduce this test suite.

Application 2: 28 test cases that present redundancy, but each test case has at least one transition that is only covered by it. In this case, the heuristics are not able to reduce the test suite since 100% coverage of the test requirements is needed and this can only be achieved if all test cases are considered.

The test cases were manually executed and the failures captured were associated with faults that can be detected by the suite. For Application 1, 13 faults have been defined, whereas for Application 2, 2 faults have been defined.

Evaluation Metrics

As said before, in general, to evaluate a prioritized test suite taking into account the fault detection, the APFD metric is calculated. However, as discussed in Section 7.1 the APFD metric applied to the complete reduced suite is not suitable here: we have 4 reduced test suites (one for each heuristic) and we need to compare them and analyze which of them presents the best ordering. Therefore, we measure the number of faults detected by the test cases selected up to a given position in the ordering. For the case studies considered, there is exactly one test case associated with each fault. Then, we need to identify whether the test case has been included in the selection in order to count the fault.

For presenting the results, we consider groups of 5 test cases from the first one to be selected up to the last one and count the faults that can be detected up to the group.

Implementation

Output data in the form of the reduced test suites are collected only at the end of the heuristic processing. This means that the instrumentation does not influence the heuristic performance. The code instrumentation was done by inserting Java code to store the information in a data structure that access entries in constant time, because this is indexed by the heuristic and metric names.

Tabela 7.3: Application 1: Reduced Test Suite Size and Number of Faults.

Heuristic	**Reduced Test Suite Size (Average)**	**Number of Faults (Average)**
Greedy	74	10.4
GE	74	10.4
GRE	74	10.5
H	74.45	10.75

The experiment consists in executing the four heuristics for each application. The inputs are the test suite TS, generated from the use case document, and the requirement mapping $ReqM$ to produce the reduced test suite, with the additional information on the position of each test case that is associated with a fault.

This process is repeated 20 times for each heuristic since these heuristics may have a random choice.

7.3.2 Results

Table 7.3 presents the obtained metrics for Application 1: the average of reduced test suite size and the number of faults obtained with 20 executions for each heuristic.

For Application 2, as said before, the heuristics did not reduce the test suite, since each of the 28 test cases has at least 1 transition that is covered only by it. So, all heuristics kept on the reduced test suite the same number of test cases of original test suite and, consequently, the rate of fault detection is not decreased.

To evaluate fault detection effectiveness, we construct box plots to show the distribution of faults in 20 executions. In the following figures, the x-axis represents the ordered test cases that are grouped from 5 to 5 and the y-axis represents the number of detected faults. The edges of the box mark the first and third quartiles. The mean value is represented by the central line in each box. The whiskers extend from the quartiles represent the farthest observation lying within 1.5 times the interquartile range. The outliers (unfilled dots) represent the individual values beyond the whiskers. Figures 7.3, 7.4, 7.5 and 7.6 show the box plots obtained for Application 1 respectively for heuristics GE, GRE, Greedy and H. Figures 7.7,

7.8, 7.9 and 7.10 show the box plots obtained for Application 2 respectively for heuristics GE, GRE, Greedy and H.

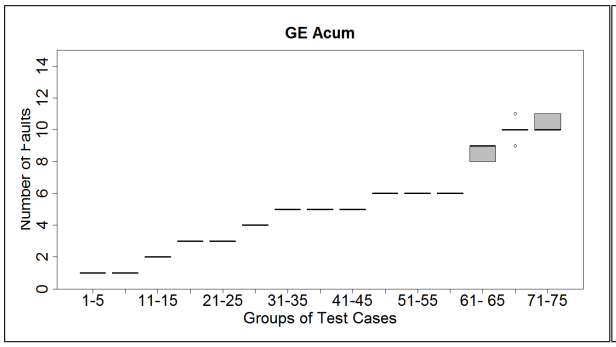


Figura 7.3: Application 1 - GE

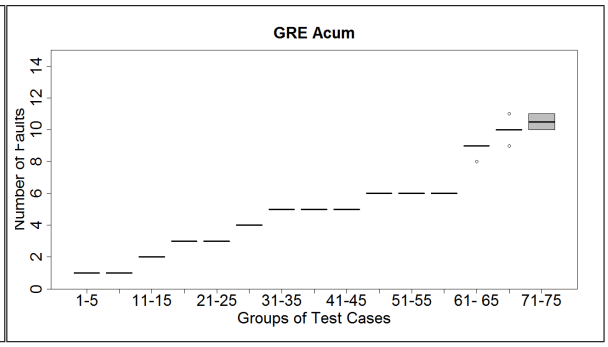


Figura 7.4: Application 1 - GRE

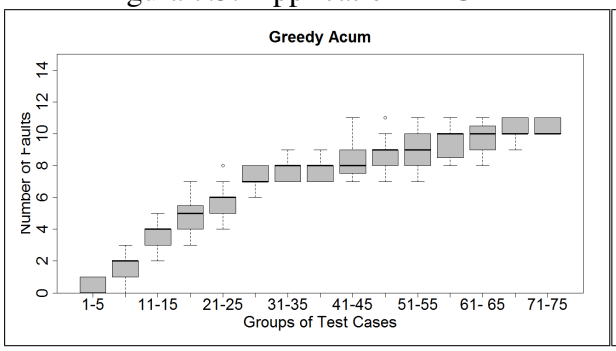


Figura 7.5: Application 1 - Greedy

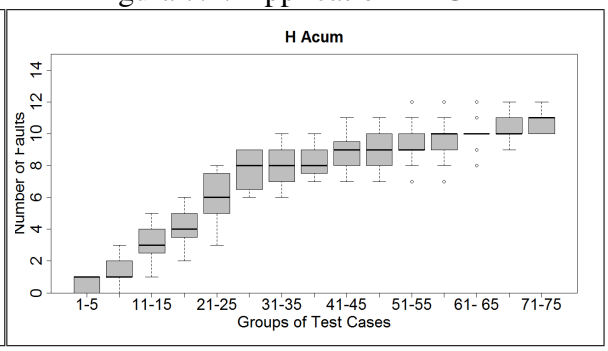


Figura 7.6: Application 1 - H

7.3.3 Threats to validity

To prevent threats to internal validity we have replicated the reported case studies 20 times. So we believe that the resulted reported in Figures 7.3 – 7.10 are reliable as concerns the two applications TaRGeT and DLA. We see however important threats to external validity, i.e., the results observed cannot of course be generalised to other applications different from the two case studies considered here. The main problem, as observed in Section 7.1, is that the comparison of the fault detection capabilities of the test cases involves the knowledge can only be carried out a posteriori. The only conclusion we can safely draw is that the ordering of test cases in the reduced test suite is important, but we cannot deduce which heuristic is superior.

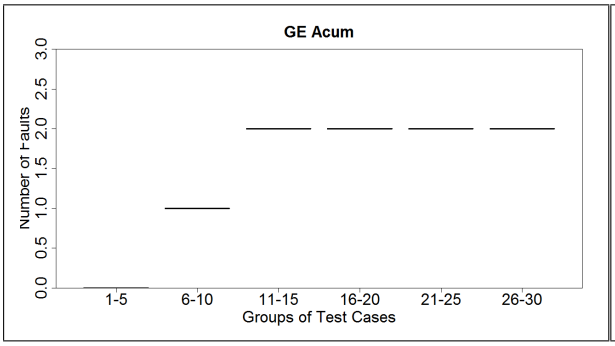


Figura 7.7: Application 2 - GE

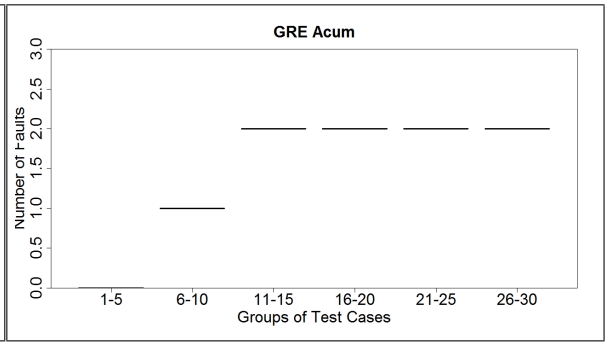


Figura 7.8: Application 2 - GRE

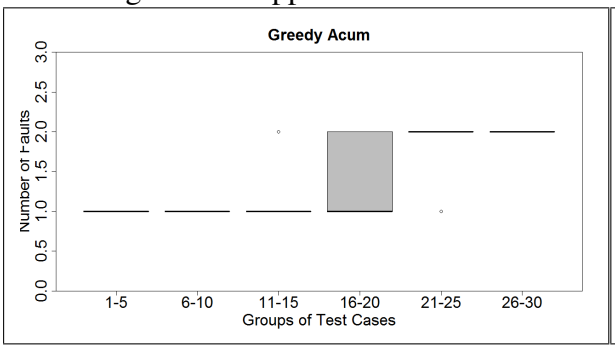


Figura 7.9: Application 2 - Greedy

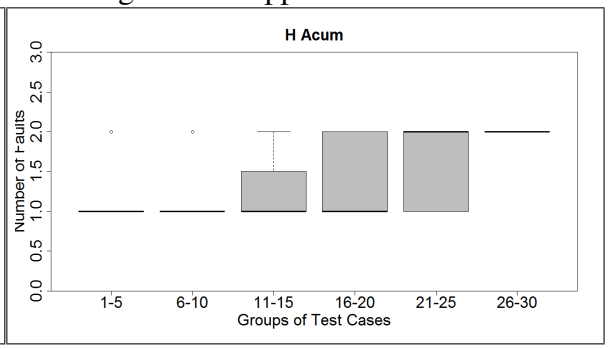


Figura 7.10: Application 2 - H

7.4 Discussion

For Application 1, it can be noticed, from Figures 7.3, 7.4, 7.5 and 7.6, that:

- 1 - 5 test cases – For this group, GE and GRE present the best fault detection effectiveness (always 1); H and Greedy present a variation between 0 and 1, but the order obtained by H can be better than the one obtained by Greedy because its median value is 1;

- 6 - 20 test cases – For this group, Greedy presents the best fault detection effectiveness (the minimum number of faults is 3 and the maximum is 7, whereas the mean value is 5); H presents the second best fault detection effectiveness (the minimum number of faults is 2 and the maximum is 6, whereas the mean value is 4). The fault detection effectiveness for GE and GRE are the same (always 3);

- More than 20 test cases – For these groups, H presents the best fault detection effectiveness.

Therefore, for this application and the faults considered, if the tester is able to execute only 1-5 test cases, then GE and GRE is the best choice. From 6-20 test cases, Greedy is the best choice, whereas for more than 20 test cases, H is the best choice. Note that GE and GRE present the same behavior until 60 test cases.

For Application 2, it can be noticed, from Figures 7.7, 7.8, 7.9 and 7.10, that:

- 1 - 5 test cases – For this group, H presents the best fault detection effectiveness where the number of detected faults is – most of the time – 1, but it can occur that the 2 faults considered are detected; Greedy presents the second best fault detection effectiveness as it always detects 1 fault. Finally, the behaviour of GE and GRE is the same, none of them detect faults;

- 6 - 10 test cases – For this group, H presents the best fault detection effectiveness where the number of detected faults is – most of the time – 1, but it can occur that the 2 faults considered are detected. Greedy, GE and GRE presents the same behavior, always detecting 1 fault.

- More than 10 test cases – For these groups, GE and GRE presents the best fault detection effectiveness (always 2). About Greedy and H:

  - 11 - 15 test cases – For this group, H presents the highest chance of detecting 2 faults;
  - 16 - 20 test cases – For this group, H and Greedy present the same behaviour;
  - 21 - 25 test cases – For this group, H and Greedy present a similar behaviour (they can detect 1 or 2), but Greedy detects 2 faults at most of the executions.

Therefore, for this application and the faults considered, if the tester is able to execute only 1-10 test cases, then H is the best choice. For more than 10 test cases, GE and GRE are the best choice. It is important to highlight that GE and GRE heuristics present the same behaviour for all positions. From Figures 7.7, 7.8, the first failure is detected after executing 6-10 test cases.

From these case studies, it can be noticed that by analysing the rate of fault detection, it is possible to observe which technique can be more effective, depending on the goals of

the tester. Some techniques are more effective when only the first selected test cases can be handled, whereas others improve their performance as more test cases are considered.

Note that the differences on the best technique at different points between the two applications may have been caused by the fact that we are not controlling important factors such as fault distribution and redundancy level. Nevertheless, the purpose of this study is to illustrate the importance and the information that can be gained if reduction techniques are also evaluated from this new viewpoint rather than by the size of the reduced test suite only. This may lead to more effectiveness on selection strategies. As mentioned before, drawing a general conclusion of which heuristic is the best in each circumstance is out of the scope of this study (and probably cannot be ever stated).

7.5 Concluding Remarks

In literature different studies have been developed for comparing prioritization and reduction techniques considering a unique point of view: for instance the efficacy in decreasing test-suite size or the impact on fault detection effectiveness. A common practice is to compare and then select the methodologies for test generation considering that all the test cases will be executed during the testing phase. However, if under budget constraints a lower number of test cases than scheduled have to be run, the test methodology chosen for deriving the test cases during the planning could not be the best choice anymore.

We presented a criterion that generalizes the APFD (Average Percentage of Fault Detection) metric for evaluating the performance of test suite reduction heuristics in subsequent moments of the test activity. The purpose is to analyze how the fault detection effectiveness of the reduction heuristics could change when testers are forced to drastically reduce the number of test cases scheduled for a certain software. Thus we considered four well-known test suite reduction heuristics – G, GE, GRE and H – and measured their rate of fault detection using two real-world applications, namely TaRGeT and DLA, by varying the number of test cases executed and by picking one-by-one the test cases in the generation order of the four heuristics.

The analysis of the case studies evidences how the performance of the four heuristics can be really influenced by the number of the test cases executed: it is possible that the heuristic

having the best performance after the execution of few test cases is not the best when the all the reduced test suite is executed. Of course the studies performed so far cannot be used for general conclusions and further investigations are necessary. In particular, since we considered real applications we could not have the complete controlling of important factors: fault distribution and redundancy level. Probably using techniques for seeding faults on the model or comparing applications that have similar level of redundancy could have provided more effective results. However the task of this work was not to conclude which technique is the best in every situation. As shown in our case studies the heuristics having the best performance in a case study have not the same behavior in the other one. Thus no general conclusion can be derived. Our goal was to show that the metrics adopted so far for assessing the relative effectiveness of various reduction approaches probably do not completely match a reality in which testing phase can be shorten depending on time and cost constraints.

Our results suggested that probably a new way of measuring the performance of various heuristics could be necessary, which takes into consideration the variability of the number of test case to be executed and the faults detected so far. Giving such a new assessment approach is part of our future work as well as to execute more case studies and experiments.

Capítulo 8

Revisão de Trabalhos em Seleção de Casos de Teste e Redução de Suítes de Teste

Este capítulo apresenta alguns trabalhos relacionados a estratégias de seleção de casos de teste e redução de suítes de teste. O foco é em soluções que podem ser automatizadas desde que nosso escopo é Teste Baseado em Modelos e Teste de Sistema.

Nós concluimos que, em geral, as estratégias para seleção de casos de teste são guiadas por algum propósito e não consideram o conceito de redundância. Entretanto as estratégias de redução de suítes de teste levam em conta o conceito, mas, por reduzir a suíte de teste, a capacidade de detecção de faltas pode ser reduzida.

Maiores detalhes são apresentados nas próximas seções.

8.1 Review of Work on Test Case Selection

Related works are presented in this section according to the general kind of the strategy followed to select test cases. For the sake of simplicity, the focus is on works that are more closely related to ours, particularly in the model-based testing area.

Combinatory Selection.

Grindal et al. [33] present an evaluation of strategies for test case selection (the All Combination Strategy, the Each Choice Strategy, the Base Choice Strategy, Automatic Efficient Test Generator and Orthogonal Arrays). These strategies are mostly devoted to testing activities where a number of combinations, between parameters and values, need to be considered.

The combination strategies were evaluated by considering the number of test cases, the number of revealed faults, failure size, and the number of decisions covered. Our strategies, for test case selection (Similarity and WSA) can be applied to refine the test suite produced by those other strategies, whenever applicable, as some of the selected combinations may still be redundant.

Test Purpose Selection.

Based on a test purpose, that may denote a functionality, or scenario of a functionality to be tested, test case generation algorithms can reduce their search space by considering only sequences that are related to the test purpose. This is a strategy used by the TGV tool [40]. The inputs of this tool are a specification in an Input/Output LTS (IOLTS) model and a test purpose. The outputs are test cases that cover the functionality that was modelled in a test purpose.

Therefore, a selection of part of the model that meets the test purpose is performed. Even though, this can greatly minimize the search space, the redundancy problem is not addressed. LTS-BT tool [16] also implements this idea. The inputs are annotated LTS and the outputs are test cases that cover the test purpose.

Statistical Testing

The Cleanroom software certification process [49] is based on statistical usage testing that consists in selecting a sample of test cases from a Markov chain model whose probabilities are defined to reflect a usage profile. The goal is to define an unbiased test suite that can be more effective for fault detection and also to make reliability estimation possible.

Following a similar idea, the Cow Suite tools focus on specifications in Unified Modeling Language (UML), such as UML sequence and use case diagrams [4] for integration testing. The test case selection is done by considering a weight function, i.e., for each diagram, it is attributed a weight regarding its functional importance.

Also, for testing from UML models, the SPACES tool [3] uses UML diagrams. This tool is used for functional component testing. For each model's transition, a weight is attributed. According to the weights, the most important set of test cases are selected. The main disadvantage of these strategies is the need for attributing weights or probabilities.

These presented strategies do not cope directly with redundancy. Our strategy can be integrated into both tools (Cow Suite and SPACES) to handle redundancy between test cases as a test case selection strategy.

Remarks - Test Case Selection

Here, we present some remarks on the related works concerning test case selection. The Table 8.1 presents a comparison between the test case selection strategies of the literature and our test selection strategies (Similarity selection and WSA).

In general, test selection strategies are guided by some purpose or number of test cases and they do not consider a redundancy concept. As advantages, all of the selection techniques addressed by these works can be fully automated and are based on sound theory. However, MBT presents several limitations in practice that cannot be completely addressed by them.

Test purposes alone can reduce the scope of search, but the TGV tool usually produces a huge number of test cases even for a simple test purpose. Statistical testing (Cleanroom, CowSuites and Spaces) take into account the cost restriction (size - we can define the number of test cases that we wish) and can lead to unbiased choices. However, the redundancy concept is not taken in consideration. Similarity and WSA strategies are adequate for MBT approaches and deal with the redundancy concept. The inputs are LTSs that can be obtained from some specifications such as UML [38].

8.2 Review of Work on Test Suite Reduction

There is a growing interest among the testing community in strategies for test suite reduction. As said before, this is an *NP-complete* problem. Therefore, algorithms based on clusters and some heuristics have been proposed. A number of experimental studies have been conducted to compare different strategies proposed in the literature.

Tabela 8.1: Kinds of strategies for selecting test cases compared to the Similarity strategy and WSA strategy

	Goal	**Coverage Criterion**	**Scope of Application and Experimentation**	**Input Required**	**Redundancy Concept**
Combinatory Selection	Select a minimal number of combinations of different factors according to a pattern defined as criterion	All combinations according to a combinatory pattern	Data selection at Integration Testing Level	Table where columns are factors and lines are values associated with the factors	NO
Test Purpose	Select a minimal part of the model that covers the test purpose	All paths in the model that are related to the test purpose	Model-based testing	Labelled Transition Systems: a specification and a test purpose	No
Statistical Testing	Select a given number of test cases randomly guided by a usage profile that weights the importance of certain execution scenarios	A number of test cases to be selected	Model-based testing	Markov Chain	No
Similarity Strategy	Select the most different test cases according to a percentage of test selection goal	A percentage of all-one-loop-paths coverage	Model-based testing	Paths generated from a Labelled Transition System	Yes
WSA	Select the most different and important test cases according to a percentage of test selection goal	A percentage of all-one-loop-paths coverage	Model-based testing	Paths generated from a Labelled Transition System and probabilities	Yes

Clustering Test Cases.

Simão et al. [55] present a technique to reduce the test suites for regression testing. This technique uses ART-2A self-organizing neural network architecture to classify test cases (feature vector). These test cases are classified into clusters. When the new source code is available, the modified arcs are evaluated and the most important clusters are selected.

Also to address regression testing, Ma et al. [44] investigate the use of genetic algorithms for defining a minimum test suite for regression testing. The algorithm builds the initial population based on test history, calculates the fitness value using coverage and cost information, and selectively produces the successive generations using genetic operations until found a minimized test suite.

Clustering based strategies are complementary to ours: if we do not have enough resources to execute all test cases, since the test cases in clusters are very similar, we are able to apply our strategy, within the cluster, and choose only the most different test cases.

Heuristics.

The heuristics are based on the notion of defining the minimal test suites that covers 100% of testing requirements. A set of test requirements is defined for satisfying a given testing objective/criterion. Some heuristics were presented in Chapter 2:

- Greedy Heuristic (G);
- Heuristic H;
- Heuristic GE;
- Heuristic GRE.

As these heuristics focus on coverage of a specific testing objective, they may be too strict and discard test cases that are important for other criteria. Our strategy is more flexible in this sense, since it relies on the general similarity of the test cases that may favor one or more criteria. However, these heuristics can be used to extend our proposed strategy as a criterion for discarding similar test cases.

Experimental works.

A number of experimental studies have been conducted to compare different reduction strategies proposed in the literature. Chen and Lau [19] present the results of a simulation study of four heuristics - H, G, GE and GRE - applied to compute a representative test suite that covers a given testing requirement. The results can be used as a guideline for choosing the most appropriate one for test suite reduction. For using this guideline, it is necessary to know the satisfiability relation and the overlapping ratio (the average number of test cases that satisfy one requirement). Since these results were obtained from simulation data, they may not reflect the real situation [67].

Zhong et al. [67] present an experimental comparison of test suite reduction techniques - H, GRE, genetic algorithm-based approach and ILP-based Approach [11]. The conclusion is that the four techniques can dramatically reduce the test suite size. However H, GRE and ILP have a better behavior since they can reduce more and the test suite sizes are almost the same. The smallest test suite is obtained from ILP-approach. This is applied for test regression, because it requires that error detection information are available.

Wong et al. in [63; 64] present empirical studies conducted to evaluate the effect of reducing the size of the test suite, keeping the block and all-uses criteria coverage. The idea is to evaluate the effect on fault detection of reducing the size of a test suite, while keeping coverage constant. The conclusion is that representative sets have almost the same capability to reveal defects as the original test suite.

Rothermel et al. [54] present empirical studies of test suite reduction for heuristic H [35]. The test suite size and fault detection capability for the reduced test suite are analyzed and reveal that the test suite reduction can compromise fault-detection capability.

Heimdahl and George [36] present an experiment where they investigate the effects of test suite reduction in test suites generated from model based testing. The algorithm used to reduce the test suite randomly and retrieve a test case in a test suite. This test case is added to the reduced set if the coverage criterion is improved. The used coverage criteria were: Variable Domain, Transition, Decision, Decision Usage, MCDC, MCDC usage. They concluded that there is an unacceptable loss in terms of test suite quality.

Zhang et al. [66] present an empirical evaluation of test suite reduction (heuristics G', H', GE' and GRE') for boolean specification-based testing. They show a guideline to choose among these strategies.

Remarks - Test Suite Reduction

To sum up, we can see that test suite reduction has been extensively experimented, but results are not conclusive and also divergent. Results depend on the techniques and choice of requirements and also on the context of application. Further research is needed to identify the most appropriate ones for MBT. Fraser [31; 30] proposed an algorithm to optimize the total costs of a test suite with respect to two factors: the test suite size and the test suite length, using concepts from model checkers. In the resulting test suite, individual test cases can be longer than in the original test suite.

A related topic is that of test case prioritization. In contrast to test suite reduction strategies which attempt to discard test cases from the test suite, the test case prioritization techniques (such as the ones presented by Elbaum *et al.* [28], Wong *et al.* [62], Korel *et al.* [43], Kim and Porter [42]) that only re-order the execution of test cases within a suite with the goal of maximizing some objective function [50].

The Table 8.2 presents a comparison among the test suite reduction strategies of the literature and Dissimilarity - our test reduction strategy.

For test suite reduction, related works focus on code-based criteria. Test suite reduction strategies are guided by a test requirement defined in terms of some coverage criteria. They reduce the size of a test suite by fixing some coverage criterion, in this case a chosen criterion is favored, however other important test cases (to reveal faults) can be excluded by being considered redundant in relation to the chosen criteria.

8.3 Concluding Remarks

In general, test selection strategies are guided by some purpose or number of test cases and they do not consider a redundancy concept. However, test suite reduction take in account that concept, but, by reducing the test suite, the fault detection capability can be decreased. The choice of a test requirement can favor or not the fault detection.

Tabela 8.2: Kinds of strategies for reduction test suites compared to the Dissimilarity strategy

	Goal	**Coverage Criterion**	**Scope of Application and Experimentation**	**Input Required**	**Redundancy Concept**
Heuristics for Test Suite Reduction	Select a minimal suite that keeps 100% of requirements coverage	All testing requirements determined by a testing objective/criterion	White-box testing	Traceability Matrix (test requirements $\times$ test cases)	Yes
Clustering Test Cases	Grouping of similar test cases in a cluster	All test cases that can be grouped in the cluster based on a fitness function	White-box testing Unit testing	Test cases and code excerpt that is targeted	No
Dissimilarity Strategy	Select the most different test cases	All test requirements	Model-based testing	Paths generated from a Labelled Transition System	Yes

Capítulo 9

Conclusões e Trabalhos Futuros

Este é o capítulo de conclusão desta tese. Aqui, algumas conclusões são mostradas (Seção 9.1) e alguns possíveis trabalhos futuros são apresentados (Seção 9.2).

9.1 Conclusions

In this thesis, we proposed a similarity function to measure redundancy among test cases of a test suite in the context of model-based testing, focusing on LTSs and test case generation algorithms guided by structural coverage criteria such as the *all-one-loop-paths* coverage. From that function, strategies for test case selection and test suite reduction are proposed. The idea is to decrease the test suite size by observing redundant parts. We have contribution in two angles, as follows:

Test Case Selection Strategies

Two strategies for test case selection are proposed and evaluated. These strategies are:

- **Similarity-based Selection:**

  The goal is to apply a similarity function to help on the selection of the most different test cases among the set of automatically generated ones according to a target number of test cases (represented by a percentage of the full test suite). To evaluate this strategy, two kinds of coverage criteria (transition and fault coverage) were considered in order to compare it with a random selection strategy that is largely used in practice.

The results show that the similarity strategy is usually more effective to eliminate redundant test cases according to these criteria. In particular, for the conducted case studies (system testing of reactive mobile phone and desktop applications), for test case selection goal equal or higher than 20%, the similarity strategy is clearly more appropriate for selection considering the investigated criteria. When the percentage is too small, the use of a random selection can be more appropriate. An experiment considering a specific configuration of LTS and 50% of the number of test cases shows that by applying the Similarity strategy we are able to obtain better results than by applying Random, considering transition coverage.

- **Weighted-Similarity Approach:**

  The main goal of this approach is to exploit the knowledge (expertise) and the similarity concept to minimize the size of a test suite. Those concepts are combined and a desired number of test cases are selected. To evaluate this strategy two case studies were executed, considering fault and transition coverage.

  The results show evidences that WSA can be an effective strategy for faults detection, however the results depend on probabilities assignment and also on the ability of the tester to pinpoint faults. In other hand, for transition coverage Similarity-based selection presents a better performance - in most of the cases. Considering only the probabilities based approaches, WSA presents generally a better performance since it inherits the similarity principles.

Since these strategies are defined for LTSs, they can be largely applied in practice and adapted to be implemented in different tools. Both strategies are currently implemented in the LTS-BT Tool [16]. TaRGeT tool [47] implements a version of Similarity-based Selection.

Test Suite Reduction

One strategy for test suite reduction is proposed in this work. This strategy, named Dissimilarity, is detailed below.

- **Dissimilarity:** The main goal of this approach is to reach transition coverage as fast as possible, considering the redundancy concept. As how less similar are the test cases, more different they are. A case study and an experiment were performed.

The results of the case study and experiment showed evidences that the Dissimilarity strategy presents the worst percentage of test suite reduction. However, the case study show that Dissimilarity presents the best percentage of the revealed faults. The experiment confirms the results of the case study: Dissimilarity presents the worst percentage of test suite reduction. In this case, considering 75% of transition coverage, as the test requirement.

A new perspective in assessing test suite reduction

We propose a new perspective in assessing test suite reduction techniques based on their rate of fault detection. Our proposal takes in account that, under pressure, testing could be stopped before all tests in the reduced test suite are run, and in such cases the ordering in the reduced test suite is also important, since the APFD of an entire test suite is different from the APFD of a part of the same test suite. Two case studies were performed and they show that, by considering the rate of fault detection, the reduction strategy to be chosen, when time is an issue, might be different from the one presenting the best performance, when testing is completed.

9.2 Future works

Many other problems need to be solved and improvements in our strategies are possible. In the future, we wish to evaluate the strategies more exhaustively by executing more industrial case studies and experiments, by considering different LTSs configurations and fault models.

- Test Case Selection
  - Similarity-based Selection

    A random choice is applied when there is a tie between values of the similarity matrix. This point can be improved by considering, for example, transition coverage or other criteria, rather than a simple random choice. The other point that can be improved is when discarding the test cases. In our proposal, the biggest test cases are kept. We performed a parallel work (presented in Cartaxo et al.

[14]) that kept the smallest. Further investigation addressing this concern can be conducted in future works.

  - Weighted-Similarity Approach

    A guideline to assign the probabilities needs to be proposed, since this is essential to the performance of the WSA strategy. Other point that needs more investigation is how to combine the probabilities that are assigned by the test manager. The current strategy considers the multiplication of the values. When the considered path (test cases) passes by several branches (where the probabilities are assigned), the final weight of the test case can be very small due to the multiplication.

- Test Suite Reduction

  - Dissimilarity

    The fact of placing two test cases at a time in the reduced test suite has hampered the performance, considering the percentage of the test suite reduction. It is probably that, by analyzing the placement of the test cases, one by one, this aspect might be improved.

- A new perspective in assessing test suite reduction

  A new perspective in assessing test suite reduction was presented and a new way to choose a test suite reduction strategy is showed. It is necessary to perform further investigations of this new way of choosing a reduction strategy, since it is still necessary to execute all strategies in order to pinpoint the best one.

Finally, our scope is MBT approaches and system testing. It is probable that these strategies are good also for regression test. Then more experimentation can be done in this direction. Besides, we provide, through LTS-BT, a tool support to execute the proposed strategies presented in this work. Therefore, other strategies can also be implemented in the tool, so they can be executed, compared and analyzed in regards to our strategies.

Bibliografia

[1] Bernhard K. Aichernig. Mutation testing in the refinement calculus. *Formal Aspects of Computing*, 15(2-5):280–295, 2004.

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

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

[4] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow suite approach to planning and deriving test suites in uml projects. In *UML'02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 383–397, London, UK, 2002. Springer-Verlag.

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

[6] A. Bertolino, E. Cartaxo, P. Machado, and E. Marchetti. Weighting influence of user behavior in software validation. In *19th International Conference on Database and Expert Systems Application - DEXA 2008 Workshops*, pages 495–500. IEEE Computer Society, 2008.

[7] A. Bertolino, E. Cartaxo, P. Machado, E. Marchetti, and Jo ao Ouriques. Test suite reduction in good order: Comparing heuristics from a new viewpoint. In *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*, pages 13–18. CRIM, 2010.

[8] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[9] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher. *Value-Based Software Engineering*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2005.

[10] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[11] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society.

[12] B.W. Boehm. *Software Engineering Economics*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[13] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, 2008.

[14] E. G. Cartaxo, P. D. L. Machado, F. G. O. Neto, and J. F. S. Ouriques. Usando funções de similaridade para redução de conjuntos de casos de teste em estratégias de teste baseado em modelos. In *Simposio Brasileiro de Engenharia de Software 08 (SBES 2008)*, Campinas, Sao Paulo, October 2008. SBC.

[15] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Automated test case selection based on a similarity function. In *Model-based Testing 07 (Motes'07)*, Bremen, Germany, September 2007. Lecture Notes in Informatics.

[16] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. 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, 2008. ACM.

[17] Emanuela Gadelha Cartaxo, Patricia Duarte Lima Machado, and Francisco Gomes Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *STVR Journal of Software Testing, Verification, and Reliability*, 2009.

[18] T. Y. Chen and M. F. Lau. A new heuristic for test suite reduction. *Information & Software Technology*, 40(5-6):347–354, 1998.

[19] T. Y. Chen and M. F. Lau. A simulation study on some heuristics for test suite reduction. *Information & Software Technology*, 40(13):777–787, 1998.

[20] T. Y. Chen and M. F. Lau. On the completeness of a test suite reduction strategy. *COMPJ: The Computer Journal*, 42, 1999.

[21] T. Y. Chen and M. F. Lau. On the divide-and-conquer approach towards test suite reduction. *Inf. Sci.*, 152(1):89–119, 2003.

[22] V. Chvatal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[23] T D Cook and D T Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[25] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[26] Rene G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. In *Proceedings of Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, pages 115–128, 1998.

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

[28] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28:159–182, 2002.

[29] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *In Proc. of the Int. Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[30] Gordon Fraser. *Automated Software Testing with Model Checkers*. PhD thesis, Graz University of Technology, October 2007.

[31] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 291–305, Berlin, Heidelberg, 2007. Springer-Verlag.

[32] Robert L. Glass. The software-research crisis. *IEEE Software*.

[33] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[34] Dick Hamlet. When only random testing will do. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 1–9, New York, NY, USA, 2006. ACM.

[35] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[36] Mats P. E. Heimdahl and Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Anders Hessel. Model-based test case generation for real-time systems, 2007.

[38] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: An extendible uml transformation framework. In *ASE '99: Proceedings of the*

14th IEEE international conference on Automated software engineering, Washington, DC, USA, 1999. IEEE Computer Society.

[39] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[40] Claude Jard and Thierry Jeron. 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, 2005.

[41] Paul Jorgensen. *Software Testing: A Craftman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 2001.

[42] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA, 2002. ACM.

[43] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, pages 34–43, New York, NY, USA, 2007. ACM.

[44] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. Test-suite reduction using genetic algorithm. In *Advanced Parallel Processing Technologies*, volume 3756 of *Lecture Notes in Computer Science*, pages 253–262, 2005.

[45] John D. Musa. Software-reliability-engineered testing. *Computer*, 29(11):61–68, 1996.

[46] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[47] Sidney Nogueira, Emanuela Cartaxo, Dante Torres, Eduardo Aranha, and Rafael Marques. Model based test generation: An industrial experience. In *1st Brazilian Workshop on Systematic and Automated Software Testing - SBBD/SBES 2007*, Joao Pessoa, PB, Brazil, 2007.

[48] Alexander Pretschner. Model-based testing. In *Proceedings of International Conference on Software Engineering - ICSE*, pages 722–723, 2005.

[49] Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[50] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. *(ICSM '99) Proceedings. IEEE International Conference on Software Maintenance*, pages 179–188, 1999.

[51] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–184, New York, NY, USA, 1994. ACM.

[52] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[53] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *In Proceedings of the International Conference on Software Maintenance*, pages 34–43, 1998.

[54] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability*, 12:219–249, 2002.

[55] Adenilso da Silva Simao, Rodrigo Fernandes de Mello, and Luciano Jose Senger. A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02*, pages 93–96, Washington, DC, USA, 2006. IEEE Computer Society.

[56] I. Sommerville. *Software Enginnering*. Van Nostrand Reinhold, eighth edition, 2007.

[57] Praveen Ranjan Srivastava. Test case prioritization. *Journal of Theoretical and Applied Information Technology*, pages 178–181, 2008.

[58] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[59] Saif ur Rehman Khan, A. Nadeem, and A. Awais. Testfilter: A statement-coverage based test case reduction technique. In *Multitopic Conference. INMIC '06. IEEE*, pages 275–280. IEEE, 2006.

[60] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[61] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[62] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. *Software Reliability Engineering, International Symposium on*, 0:264, 1997.

[63] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[64] W. Eric Wong, Joseph Robert Horgan, Aditya P. Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software*, 48(2):79–89, 1999.

[65] Xue ying MA, Zhen feng He, Bin kui Sheng, and Cheng qing Ye. A genetic algorithm for test-suite reduction. In *IEEE International Conference on System, Man and Cybernetics*, pages 133–139, 2005.

[66] Xiaofang Zhang, Baowen Xu, Zhenyu Chen, Changhai Nie, and Leifang Li. An empirical evaluation of test suite reduction for boolean specification-based testing (short paper). In Hong Zhu, editor, *QSIC*, pages 270–275. IEEE Computer Society, 2008.

[67] Hao Zhong, Lu Zhang, and Hong Mei. An experimental comparison of four test suite reduction techniques. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 636–640, New York, NY, USA, 2006. ACM.

Apêndice A

Similarity based Selection - Case Studies

The content of this chapter have been published in the paper Cartaxo et al. [17].

A.1 Introduction

In order to evaluate the use of the similarity strategy, three different case studies were conducted, where test cases were selected by the similarity strategy and also by a plain random selection strategy. The evaluation focused on assessing whether the similarity strategy (when compared to the random strategy) is suitable for test case selection with the goal of producing test suites of a given smaller size that still keep effective coverage. The random choice strategy basically consists in randomly selecting a test case at a time to be removed from the test suite according to a random function with an even probability distribution. The similarity strategy has been applied by using the LTS-BT tool [16].

The random selection strategy was chosen because, when coverage and diversified choices are of concern, random choice has been accepted to be more effective than deterministic choice in the model-based testing area [49]. Therefore, this strategy has compatible expectations when compared to the similarity ones. Also random testing methods have proven to be more effective in practice in situations where information is lacking to make systematic approaches applicable [34]. When selecting test cases from plain labelled transition systems (the context of this work), which is different from selection from high-level specifications, this situation arises. For example, at system level (the scope of the case studies presented here), systematic approaches will often require assumptions or information such as opera-

tional profile and domain partition. Moreover, random selection is usually applied by other selection, reduction and prioritisation strategies when they reach an undecided situation and get blocked due to a tie among test cases to be selected. This makes random selection a very important and representative selection strategy in practice. Furthermore, random selection is usually considered as fundamental basis of comparison in most of the empirical studies in the area.

For the sake of simplicity when explaining the case studies, the term “percentage of test cases selected"is used with the same meaning as “percentage of *all-one-loop-paths* coverage". This comes from the fact that the selected paths are indeed the test cases.

Next sections present an overview of the case studies applications (Section A.2), how the case studies were defined and conducted (Section A.3), and the results obtained during case studies execution (Sections A.4 and A.5).

A.2 Overview of Case Study Applications

In this section, the case studies chosen are briefly described. They are named, for further reference, as Case Study 1, Case Study 2 and Case Study 3. All of them are reactive applications i.e., applications that react to stimuli of their environment [40]. Particularly, Case Studies 1 and 2 are mobile phone applications, whereas Case Study 3 is a desktop application. The focus is on system testing and test suites are for manual execution. Therefore, the LTS models represent use scenarios of the applications. In summary, the case studies are described as follows:

- Case study 1 is an application for adding contacts in a mobile phone’s contact list;
- Case study 2 is a message application that deals with embedded items. An embedded item can be an URL, phone number or e-mail. For each embedded item, it is possible to execute certain tasks (See Table A.1);
- Case study 3 is an application that generates test cases automatically from use case scenarios - the TaRGeT tool [47].

It is important to remark that, for all case studies, the same basic process has been followed to obtain the LTS model. Basically the process has the following activities:

Tabela A.1: Embedded item and available Tasks

Embedded item	**Tasks**
URL	Store, Go to
Phone number	Send message, Send voice message, Store, Call
E-mail	Send message, Store

- Writing use cases from natural language requirement documents according to the format defined by Cabral and Sampaio [13];
- Use cases inspection and review for consistency, completeness and conformance;
- LTS model generation that combines the behaviours of all use cases [47]. Basically, each LTS transition represents a use case step. The use case flows may be branched, resulting in different paths in the LTS.

All case studies have also been conducted by the same team. Therefore, this evaluation assumed that the level of details and consistency of the LTS models obtained is similar for all of them. Also, the LTS models cover 100% of the known requirements for each application.

Table A.2 shows some metrics on the case studies in order to illustrate their complexity such as the number of use cases, the number of branch nodes, the maximum level of nested sub-branches, the number of loops and the number of transitions. At system use case level specification, loops are quite rare, unless repetitive interactions are needed in a use. Actually, since all-one-loop-paths coverage has been considered, loops are not significant here. Table A.2 also presents the number of test cases, transitions and faults for each case study considering 100% *all-one-loop-paths* coverage. Faults are defined according to a fault model (Subsection A.3.3). Each fault model makes reference to faults that can be included in an implementation in case programmers do erroneously interpret requirements and, as result, the implementation produces responses that are not in conformance with the LTS model. The reason for this is that it is important to obtain a similar level of fault distribution in all case studies. For Case Study 3, actual faults detected by test execution and debugging were also considered.

Tabela A.2: Case Studies - Metrics

	Case Study 1	**Case Study 2**	**Case Study 3**
Number of Use Cases	1	33	25
Number of Branches	7	34	21
Maximum level of Sub-Branches	5	1	3
Number of loops	1	0	0
Number of Transitions	121	826	631
Number of Test Cases	24	66	130
Smallest Test Case (Number of Transitions)	6	11	6
Biggest Test Case (Number of Transitions)	29	27	38
Most Common Test Case Size	24	19	14
Number of Most Effective Test Cases (Associated with more faults)	3	33	4
Number of Faults	23	99	127

Tabela A.3: Faults per Number of Transitions and Test Cases and Test Cases per Transitions (Similarity Rate)

	Case Study 1	**Case Study 2**	**Case Study 3**
Faults/Number of Transitions	0,190	0,120	0,201
Faults/Test Cases	0,958	1,500	0,977
Test Cases/Transitions	0,198	0,080	0,206

Table A.3 presents the rates of faults per number of transitions and test cases as well as the rate of test cases per transition that may characterize the similarity degree of paths in the application and, consequently, the similarity degree of test cases.

For the sake of confidentiality and also for the sake of simplicity, the LTS models of these case studies are not presented in this paper. However, it is important to comment that Case Studies 1 and 3 have more redundant test cases than Case Study 2, since the latter has 3 disjoint groups of functionalities that are handled in isolation. This can also be observed by the rate of test cases per transitions for Case Study 2 (see Table A.3), that is, less test cases for more transitions. The most effective test cases of Case Study 1 are relatively more redundant than the ones of Case Study 3 since the former is about a single and cohesive use case.

A.3 Overview of Case Studies Definition

This section presents the evaluation criteria, path selection strategy and fault model structure defined for conducting and evaluating the case studies.

A.3.1 Evaluation Criteria

In the general research area on test case selection which is the main focus of the case studies, the main criteria used to evaluate the resulting test suite are:

(i) Structural coverage;

(ii) Fault-coverage;

(iii) The number of faults detected by the most effective test contained in it;

Regarding (i), for the case studies presented in this paper that are of system level testing (abstracting from code), transition-based coverage criteria are the most appropriate ones (actually the only ones that make sense), since the focus of this work is on plain labelled transition systems without either guards or datatypes or parallel composition. In this case, the only observable behaviours are transitions that represent outputs. Therefore, transition coverage is a very important metric, since the number of observable behaviours that are

going to be evaluated at testing time can be measured. The most popular transition-based coverage criteria that have been applied to model-based testing are: *all-states* (every state must be visited at least once), *all-configurations* (every configuration of a statechart is visited at least once), *all-transitions* (every transition must be visited at least once), *all-transition-pairs* (every pair of adjacent transition in the model must be traversed at least once), *all-loop-free-paths* (every loop-free path must be traverse at least once) , *all-one-loop-paths* (all the loop-free paths through the model must be visited at least once, plus all the paths that loop once), *all-round-trips* (requires a test for each loop in the model, but do not require that all the paths the precede or follow a loop to be tested) and *all-paths* (every path must be traversed at least once) [60]. For them, it is valid to say that:

- *all-paths* subsumes all of them, but this is not applicable to the case studies since the test generation algorithm only guarantees *all-one-loop-paths* coverage in order to avoid the state space explosion problem;
- *all-transitions* subsumes *all-states*;
- *all-transition-pairs* subsumes *all-transitions*;
- *all-configurations* is not observable in the context of this work;
- *all-one-loop-paths* subsumes *all-round-trips* and *all-loop-free-paths*;
- *all-round-trips* is based on breadth search that selects the shortest test cases guided by this search. Even though, the similarity strategy is independent of whether the generation algorithm is based on depth or breadth search, we used a depth search one for these case studies. Therefore, all-round-trips is not applicable here.

Regarding (ii) and (iii), faults were abstracted by possible observable failures during test execution. Faults are associated with test cases that are capable of exhibiting the corresponding failure behaviour. Finally, for (iii), instead of counting the number of faults revealed by one most effective test case, the most effective test cases were defined and counted (how many of them are included in the selected suite) - a stronger criterion.

In summary, the following criteria were considered to evaluate the test suites obtained from each strategy:

- **Transition-based coverage** - The total number of transitions and pairs of transitions that are covered by considering all of the selected test cases of a given test suite. The idea is to measure whether the strategies keep a reasonable coverage of functionalities.

- **Fault-based coverage** - The total number of faults that are uncovered by the test suite during test execution. For this, versions of the case studies that include faults were considered and also fault models were defined. The idea is to measure whether the strategies preserve the fault detection capability of the original test suite. It was also measured whether the most effective test cases are kept in the minimised suites.

The reason for choosing these criteria is to make it possible to investigate, in the context of the case studies, questions such as: (i) Is test case selection based on the similarity strategy more effective than random selection regarding a given criterion? (ii) What are the limitations of the similarity strategy? (iii) In which circumstances is it more advisable to apply each strategy?

A.3.2 Test Case Selection Goals

For each case study, the similarity and random selection strategies were applied having test selection goals ranging from 5% to 95% (increased by 5) of the test cases. The purpose is to identify which strategy (similarity or random) assures the best selection according to the criteria mentioned above. This is reflected in the final transition coverage and observable failures. Also, due to the random choice that is presented in both strategies, for each path coverage goal, the selection algorithm has been executed 100 times for each strategy. In this case, the average of the values obtained for each metric is considered.

A.3.3 Fault Model

For each case study, test cases were associated with the faults that they are capable of revealing. Then, the number of faults covered by a test suite is computed by the total number of different faults covered by its test cases.

As mentioned before, all case studies are reactive applications. Moreover, their LTS models represent system level scenarios that are derived from use case specifications. Branches

in the LTS represent either different expected inputs to the system or different outputs that can be produced by the system. A choice of input is made by the environment (and this is usually controlled/pre-defined for each test case), whereas the actual output produced during a test case execution is defined by internal behaviour of the system that may or not depending on conditions to be met. Therefore, outputs are the central information to be observed for deciding on the success of a test execution [58].

In this context, a fault model aimed at generic system scope may consider faults that lead to: 1) undesirable feature interactions, 2) incorrect output, 3) abnormal termination, 4) inadequate response time [10]. Since the models considered in the studies do not express requirements on feature interaction and timing, only faults of type 2 and 3 were considered to build the fault models.

At system testing specification level, a fault can only be viewed through the failure that expose it, i.e., an output produced by the system during test execution that is different from the expected ones. Therefore, the first step is to identify possible points of failure and them to assume that one or more faults in the code cause them (assuming also that test cases are sound, i.e., they would not produce false positives). For example, consider the excerpt of the LTS model of Case Study 1 presented in Figure A.1. Transitions leaving Node 1 are input actions (input action have “?"as prefix). If “`?go to main menu`"action occurs, then the expected output is “`!main menu is displayed`"(output actions have “!"as prefix). When a different output is produced, this indicates a failure that is caused by one or more faults at different points in the code. For the sake of simplicity, the procedure for constructing the fault model described below assumes that only one fault is associated with each failure.

One possible way of identifying points of failure, is to consider the possible mutations that could be made upon output transitions at specification level aiming at anticipating design errors that could lead to failures [1]. That is, possible ways of mutating the specification in order to investigate on possible non-conformant implementations that could be produced. Based on this, the general procedure applied to construct the fault model is as follows. The main idea is to mark, in the LTS model, all failure-prone occurrences of output transitions, i.e., the ones where failures are more likely to occur (the expected output is not the one produced). All marks represent a different fault that can be uncovered. Then, test cases

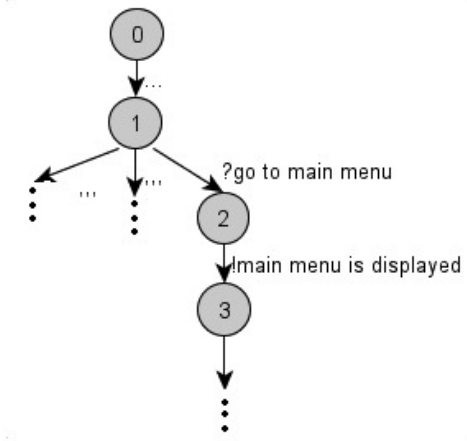


Figura A.1: An excerpt of the LTS model for Case Study 1.

that include the transition are marked as the ones that can reveal the corresponding fault. In summary, the main steps are as follows:

1. Transitions that are part of a branch of outputs are marked. The reason is that these outputs are likely to be defined by a combination of conditions. Since whether a condition is completely faulty cannot be decided (fails for any combination of values in the input domain), the fact that all outputs can be erroneously produced must be considered. Therefore, there is a chance of failure at each one.

2. Transitions that represent a single expected output (only one output transition out of a node) are marked if: 1) there is a chance of abnormal termination; 2) the output is produced as a result of a risk operation; 3) a failure is very likely to occur.

3. A matrix of faults and test cases is constructed where the intersection of fault i and test case t is marked if and only if the output transition represented by i is included in test case t.

The output transition in Figure A.1 is marked in Step 2, because at this point it is likely that by failure a different menu is displayed. This is often caused by pointer manipulation faults.

Even though, for the purposes of the case studies, it is important to define as many faults as possible in order to favour a more coherent evaluation of the results, not all possible occurrences of outputs were considered since this is not realistic for real systems: test cases

are usually designed for revealing specific faults and stable functionality is rarely faulty (Step 2). An example of a fault model defined according to these steps is presented in Table A.4.

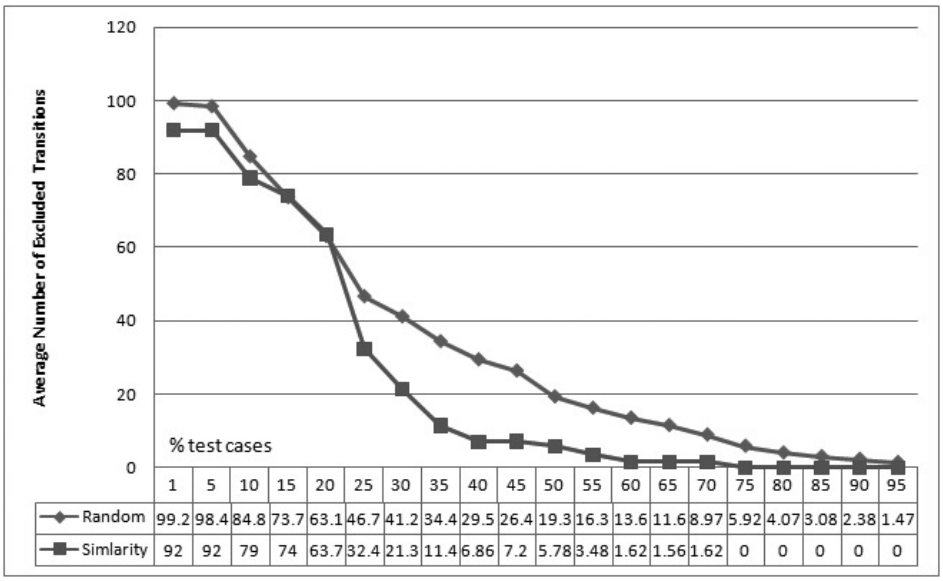
A.4 Case Studies Results

This section presents and discusses the results obtained by considering transition-based and fault-based coverage respectively. For each criterion, results are analysed considering the questions posed at the end of Subsection A.3.1.

A.4.1 Transition Based Coverage

On the LTS model of the applications, the functionalities are represented as labelled transitions. Then, transition coverage when we applied the similarity and random strategies was observed. As transition-based criteria, *all-transitions* and *all-transition-pairs* coverage were considered (see Subsection A.3.1).

Transition Coverage. Figure A.2, Figure A.3 and Figure A.4 illustrate the obtained results for the similarity strategy and the random choice strategy for Case Studies 1, 2 and 3, respectively.

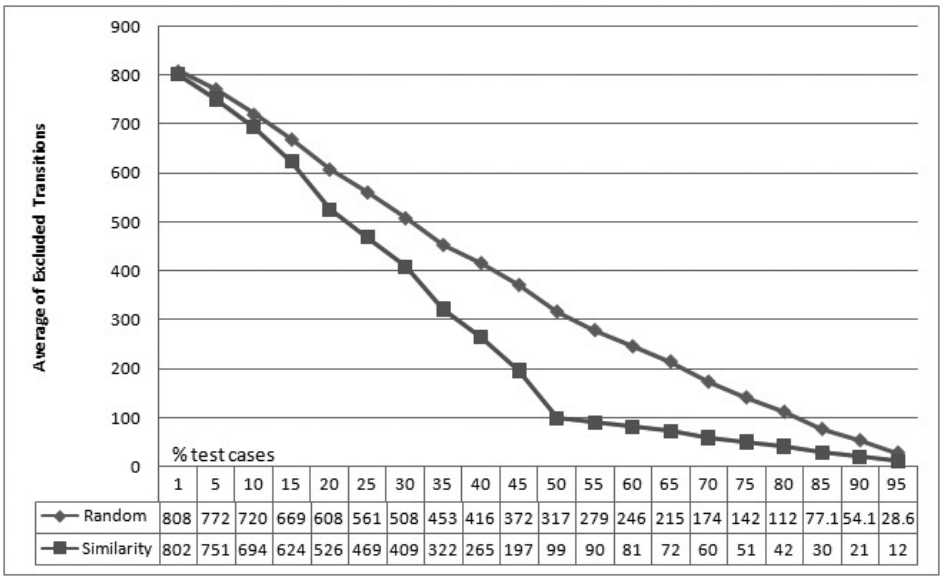


% test cases	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	99.2	98.4	84.8	73.7	63.1	46.7	41.2	34.4	29.5	26.4	19.3	16.3	13.6	11.6	8.97	5.92	4.07	3.08	2.38	1.47
Simlarity	92	92	79	74	63.7	32.4	21.3	11.4	6.86	7.2	5.78	3.48	1.62	1.56	1.62	0	0	0	0	0

Figura A.2: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 1.

Tabela A.4: Fault Model - Case Study 1. Test cases 04, 12, 18 are the most effective test cases w.r.t. the number of faults covered

Faults/ Test Cases	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
F001	-	-	-	-	-	-	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-
F002	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-
F003	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-
F004	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X
F005	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
F006	-	-	-	-	-	-	-	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-
F007	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	-	-	-
F008	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
F009	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-
F010	-	-	X	X	-	-	-	-	-	-	X	X	-	-	-	-	X	X	-	-	-	-	-	-
F011	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-
F012	-	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-
F013	X	X	X	X	X	X	-	-	-	-	-	-	-	-	X	X	X	X	X	X	-	-	-	-
F014	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-
F015	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X
F016	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X
F017	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-
F018	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
F019	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-
F020	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
F021	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-
F022	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-
F023	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
# Faults	4	5	5	6	5	4	4	3	4	5	5	6	5	4	4	5	5	6	5	4	4	1	2	3



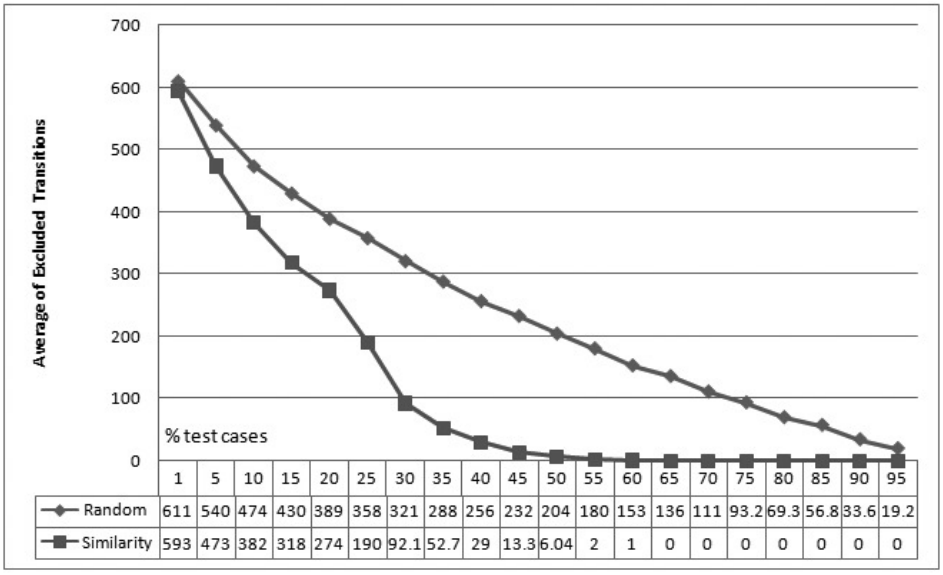
% test cases	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	808	772	720	669	608	561	508	453	416	372	317	279	246	215	174	142	112	77.1	54.1	28.6
Similarity	802	751	694	624	526	469	409	322	265	197	99	90	81	72	60	51	42	30	21	12

Figura A.3: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 2.

For each of the figures, the x-axis (or abscissa) represents the intended percentage of test cases to be selected and the y-axis (or ordinate) represents the average number of excluded transitions obtained with 100 experiments. The higher the value in the y-axis the worse is the coverage obtained. Therefore, the most effective strategy regarding this criterion is the one that present the lower curve.

For Case Study 1 (Figure A.2), the similarity approach is clearly more effective when 25% to 75% of the test cases are selected, with the best case for similarity achieved at 35%: 91% of transitions are preserved in the selected test suite, whereas only 82% are preserved by the random strategy. From percentages 5 to 20, the performance of the similarity strategy was similar to the random one. In the worst case, at 5% goal, the similarity strategy kept 24% of the transitions, whereas the random strategy kept only 19%. Particularly, in this case study, a number of similar paths have the same size. Therefore, when such strict selection percentages are applied, the choice for discarding one test case is mostly a random one. From 75% of test case selection goal, *all-transitions* coverage is achieved by the similarity strategy.

For Case Studies 2 (Figure A.3) and 3 (Figure A.4), the similarity approach is clearly more effective. The best case for similarity in Case Study 2 is achieved when 50% of the test cases are selected: 88% of transitions are preserved in the selected test suite, whereas



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	611	540	474	430	389	358	321	288	256	232	204	180	153	136	111	93.2	69.3	56.8	33.6	19.2
Similarity	593	473	382	318	274	190	92.1	52.7	29	13.3	6.04	2	1	0	0	0	0	0	0	0

Figura A.4: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 3.

only 62% are preserved by the random strategy. The best case for similarity in Case Study 3 is achieved when 35% of the test cases are selected: 92% of transitions are preserved in the selected test suite, whereas only 54% are preserved by the random strategy. Also, for this case study, all-transitions coverage is achieved from a selection of 65% of the test cases.

Regarding the questions put in Section A.3.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? From Figure A.5, the average of the percentage of excluded transitions in all case studies is lower for the similarity strategy. As mentioned above, Case Studies 1 and 3 present more redundant test cases. Therefore, the percentage of excluded transitions is lower for these two case studies when compared to Case Study 2. From 75% of test cases selected, all-transitions coverage is achieved. This indeed shows that the similarity approach is more effective than random choice for these case studies. The reason is that the similarity strategy is more systematic and more precisely pinpoints the similarity, keeping the most different test cases and therefore the best transition coverage, even for Case Study 2 with less redundant test cases than the other ones.

(ii) What are the limitations of the similarity strategy? The similarity strategy performs

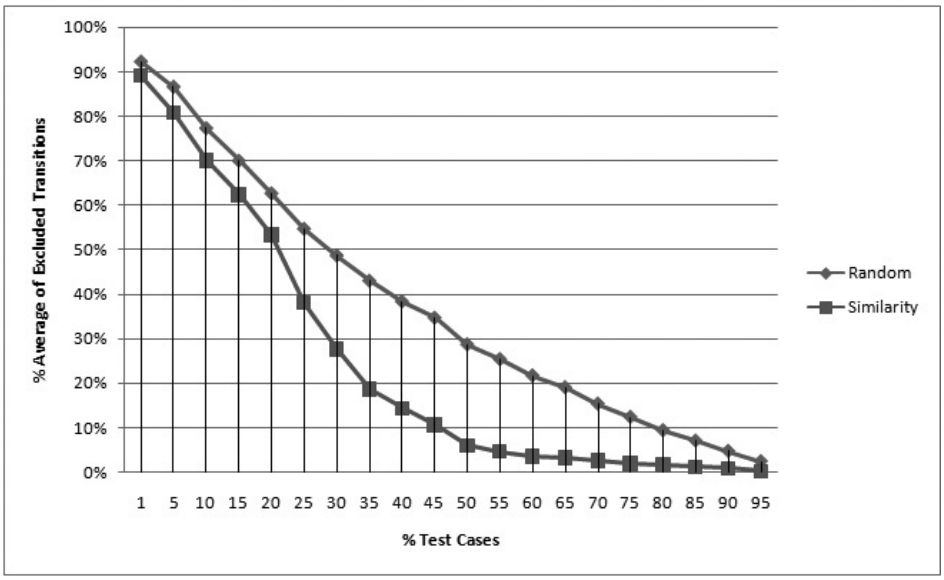


Figura A.5: Percentage of the average number of excluded transitions in all case studies for each test case selection goal.

as well as random selection whenever the criterion for choosing one test case to be discarded cannot be decided. In this case, random selection is applied. This happened in Case Study 1, where some similar paths have the same size and the criterion is based on keeping the one with the biggest size.

(iii) In which circumstances is it more advisable to apply each strategy? By generally comparing the results obtained, the figures suggest that, usually for test case selection goal from 20%, it can more adequate to use the similarity strategy than the random, even in the case where the application presents a considerable number of redundant test cases as in Case Studies 1 and 3, where redundancy may favour the random choice strategy performance.

Transition-Pairs Coverage In order to apply this strategy, all pairs of transitions at each node of the LTS model are computed. The aim is to check whether the strategies preserve combinations of transitions in the test cases. The evaluation was conducted in the same way as for transition coverage and the results are very similar with the same advantages and limitations for each case study. Therefore, for the sake of simplicity, only a summary is presented in Figure A.6. In the best case, the similarity approach preserves 23% more

pairs of transitions than the random approach. For transition coverage, in the best case, the similarity approach preserves 24% more transitions than the random approach.

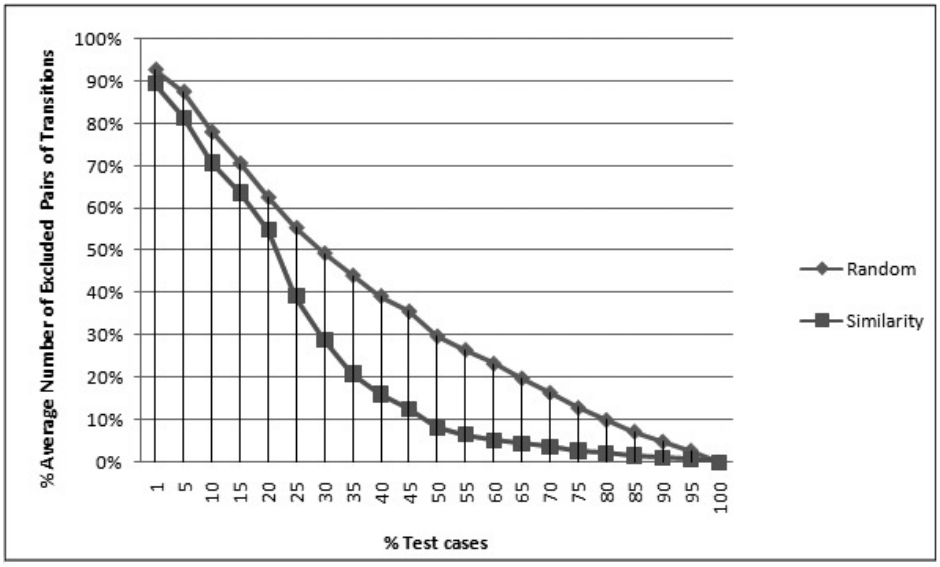
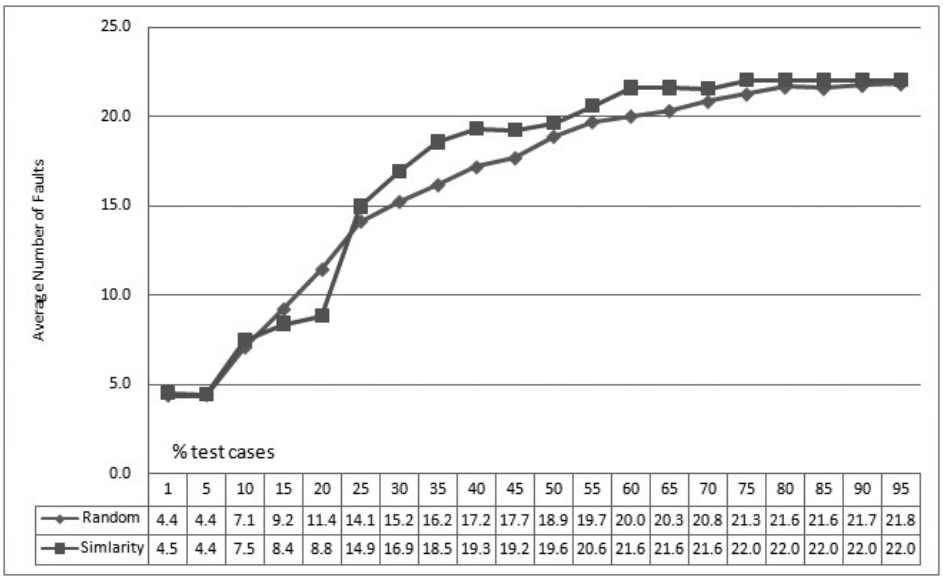


Figura A.6: Percentage of the average number of excluded pairs of transitions in all case studies for each test case selection goal.

A.4.2 Fault-based Coverage

As fault-based criteria, fault coverage and most effective test cases coverage were considered. The results are presented in the sequel.

Fault Coverage Figure A.7, Figure A.8 and Figure A.9 show the results obtained. For each one of the figures, the x-axis represents the intended percentage of test cases to be selected and the y-axis represents the average of covered faults (faults that can be revealed by one or more test cases in the suite) when a test case selection goal is applied (this data was also obtained with 100 experiments). The higher the value in the y-axis the best is the coverage obtained. Therefore, the most effective strategy regarding this criterion is the one that present the upper curve. It is important to remark that since only up to 95% of test case selection goal is considered and fault distribution is such that all test cases are generally associated with at least one fault (see Table A.4), 100% of faults coverage may not be achieved by any of the strategies.



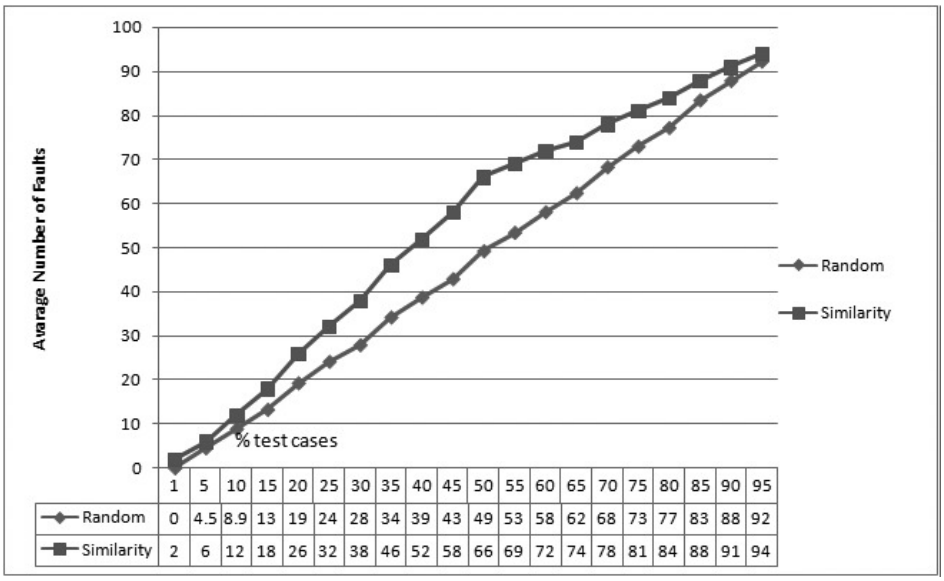
	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	4.4	4.4	7.1	9.2	11.4	14.1	15.2	16.2	17.2	17.7	18.9	19.7	20.0	20.3	20.8	21.3	21.6	21.6	21.7	21.8
Simlarity	4.5	4.4	7.5	8.4	8.8	14.9	16.9	18.5	19.3	19.2	19.6	20.6	21.6	21.6	21.6	22.0	22.0	22.0	22.0	22.0

Figura A.7: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 1.

For Case Study 1 (Figure A.7), the similarity approach is clearly more effective when 25% to 75% of the test cases are selected, with the best case for similarity achieved at 35%: 81% of faults are addressed by the selected test suite, whereas only 70% are addressed by the random strategy. From percentages 5 to 20, the performance of the similarity strategy was worse than or similar to the random one. In the worst case, at 20% selection goal, the similarity strategy addresses only 38% of the faults, whereas the random strategy addresses 50%. The best coverage achieved, 96% or 22 faults out of 23, is reached only by the similarity strategy from 75% of selection goal.

For Case Studies 2 (Figure A.8) and 3 (Figure A.9), the similarity approach is clearly more effective. The best case for similarity in Case Study 2 is achieved when 50% of the test cases are selected: 66% of faults are addressed by the selected test suite of the similarity approach, whereas only 49% are addressed by the random strategy. The best coverage achieved, 95% or 94 faults out of 99, is reached only by the similarity strategy with 95% of selection goal.

The best case for similarity in Case Study 3 is achieved when 45% of the test cases are selected: 92% of faults are addressed by the selected test suite, whereas only 60% are preserved by the random strategy. The best coverage, 100% or 127 faults, is achieved only by the similarity strategy from 65% of selection goal.

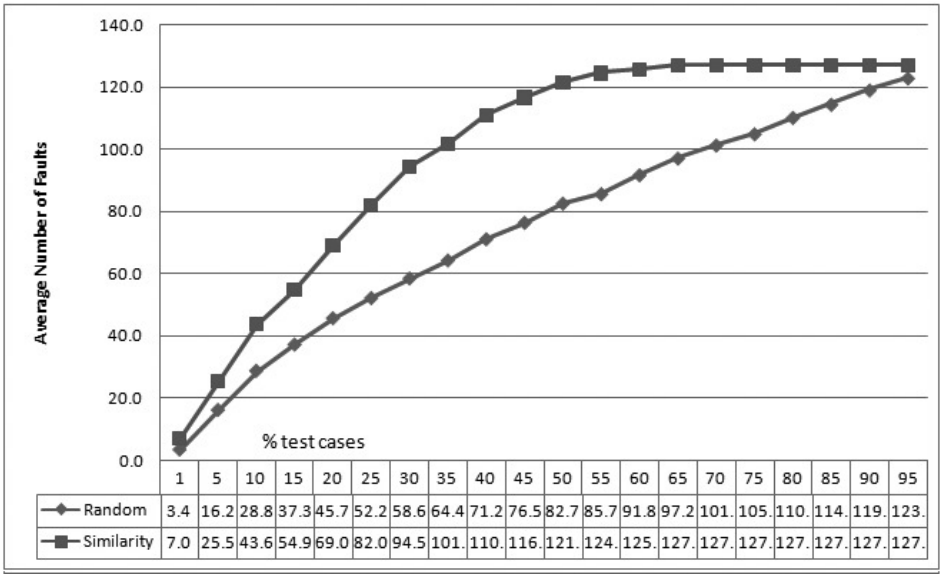


	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	0	4.5	8.9	13	19	24	28	34	39	43	49	53	58	62	68	73	77	83	88	92
Similarity	2	6	12	18	26	32	38	46	52	58	66	69	72	74	78	81	84	88	91	94

Figura A.8: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 2.

Regarding the questions put in Section 4.2.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? From Figure A.10, the average of the percentage of faults addressed by the resulting test suite in all case studies is definitely higher for the similarity strategy from 20% of test case selection goal. As mentioned above, Case Studies 1 and 3 present more redundant test cases. Therefore, the percentage of the number of faults covered is closer to 100% and the best coverage is achieved from 75% of selection goal. The similarity approach is more effective because it can more systematically select the most different faults that are associated with the most different test cases. Case Study 2, with less similar test cases, has also less similar faults. Therefore, the best coverage is only achieved at 95%. Nevertheless, there are clear gains to the similarity approach when compared to the random strategy. However, note that, Case Study 1 presents an open question: in the presence of a severe path coverage constraint (below 20%), is non-deterministic choice more effective than similarity based selection with regard to fault detection? This question is related to a claim of the random testing community (non-determinist selection is more effective than deterministic selection w.r.t. to fault detection) that deserves further investigation.



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	3.4	16.2	28.8	37.3	45.7	52.2	58.6	64.4	71.2	76.5	82.7	85.7	91.8	97.2	101.	105.	110.	114.	119.	123.
Similarity	7.0	25.5	43.6	54.9	69.0	82.0	94.5	101.	110.	116.	121.	124.	125.	127.	127.	127.	127.	127.	127.	127.

Figura A.9: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 3.

(ii) What are the limitations of the similarity strategy? Again, whenever the criterion for defining which test case to discard cannot be decided, then the similarity approach performs as the random strategy. Also, the less redundant test cases are the lower the coverage of faults is. For instance, coverage rate at Case Study 2 is, in average, 13% lower than in the other case studies up to 65%. Furthermore, as the choice for the test case to be discarded, in this paper, is based on the biggest test case, if faults distribution is more prevalent amongst the smaller test cases, then the strategy may not have a good performance. In other words, the performance of the strategy can be influenced by the choice of the criterion to discard the redundant test case.

(iii) In which circumstances is it more advisable to apply each strategy? Comparing the obtained results for similarity and random, it is clear that, in the case studies conducted, for test selection goal bigger than 20%, it is more adequate to use the similarity strategy than the random one, since the number of excluded test cases that failed for different faults is smaller than using the random strategy. In other words, the similarity strategy, by keeping the most different test cases, is more effective in selecting test suites that preserves fault detection capability of the original suite.

It is important to remark that results obtained in these experiments can also be influenced

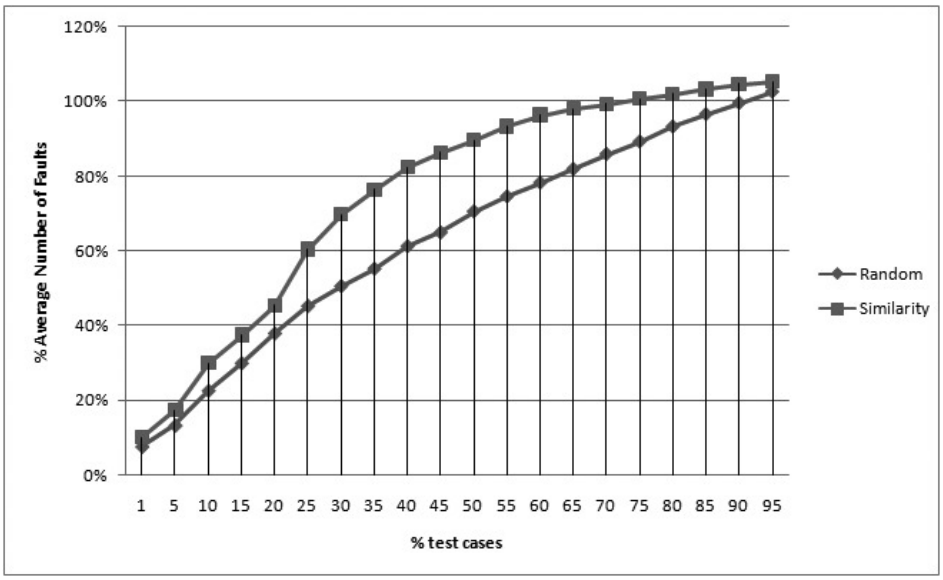


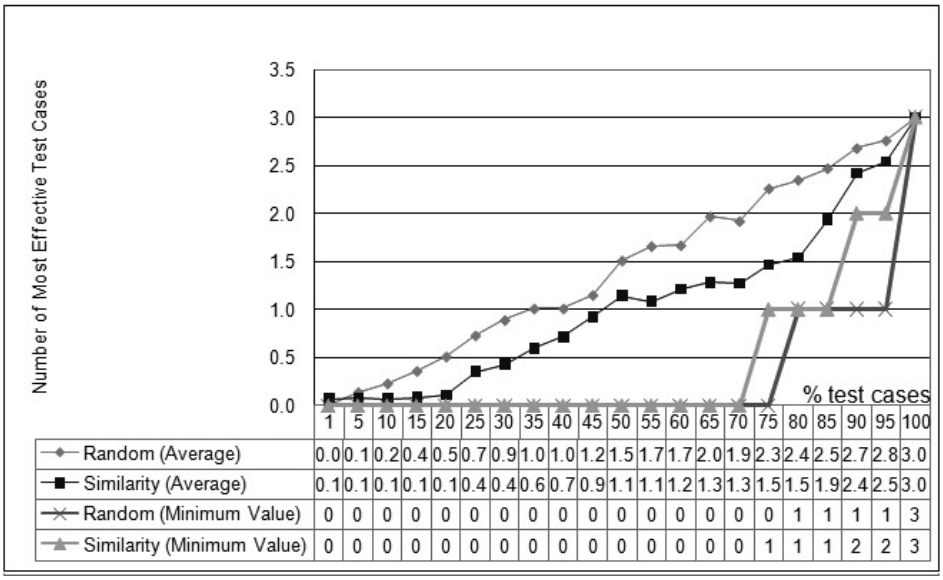
Figura A.10: Percentage of the average number of faults transitions covered in all case studies for each test case selection goal.

by the rate of faults per test cases, faults distribution and the size of the application (measured here by the number of transitions). Table A.3 summarizes these rates. For Case Study 1 (Figure A.7), there is a less significant gain of the similarity strategy over the random strategy when compared to the other case studies. On the other hand, for Case Study 3 with a similar rate of faults per transitions and faults per test case to Case Study 1, the similarity strategy had a considerable gain over the random strategy. This may be explained by the fact that Case Study 1 has more faults distributed among the smaller test cases and then the random strategy had more chances to keep them with selection goals less than or equal to 20%. Furthermore, Case Study 2 has the highest rate of faults per test case, but this is also the case with less redundant test cases and faults. Therefore, similarity is more effective for Case Study 2 than for Case Study 1.

Most Effective Test Cases For each case study, the most effective test cases were selected as the ones that are associated with the biggest number of faults. The goal is to measure how many of the best test cases are preserved at each test case selection goal. Instead of choosing a limiting number of most effective test cases (for example, 1), all test cases that achieve the biggest number of faults in the suite of a given case study were considered. Therefore, the number of most effective test cases is dependent on the case study.

For each one of the figures presented below, the x-axis represents the intended percentage of test cases to be selected and the y-axis represents the minimum number and also the average number of most effective test cases included in the selection. Again, for each selection goal, the strategies were performed 100 times.

For Case Study 1 (Figure A.11) with 4 most effective test cases out of 23, the random approach is more effective in the average case. This can be explained by the fact that the most effective test cases, in this case study, are very similar (diverging by 1-4 transitions only). The similarity approach constraints its search space by eliminating redundancy according to each selection goal, whereas the random approach freely chooses among all possible test cases for each selection goal. Therefore, the most effective test cases for Case Study 1 cannot be included in the resulting test suite when the similarity strategy is applied. However, note that, in the worst case (considering the minimum number of the most effective test cases selected at one or more of the 100 trials), the similarity approaches presented a better performance, from 75% of selection goal at least one of them is included.

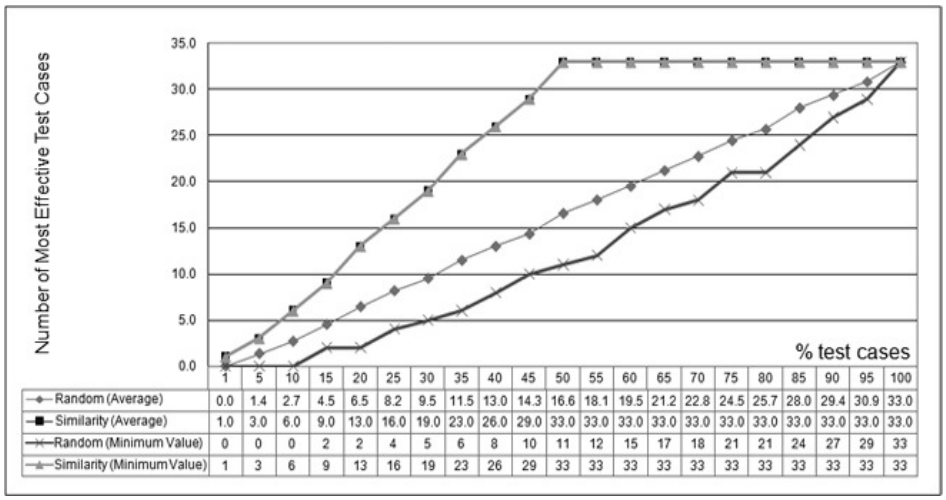


	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.0	0.1	0.2	0.4	0.5	0.7	0.9	1.0	1.0	1.2	1.5	1.7	1.7	2.0	1.9	2.3	2.4	2.5	2.7	2.8	3.0
Similarity (Average)	0.1	0.1	0.1	0.1	0.1	0.4	0.4	0.6	0.7	0.9	1.1	1.1	1.2	1.3	1.3	1.5	1.5	1.9	2.4	2.5	3.0
Random (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3
Similarity (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	3

Figura A.11: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 1.

It is also important to remark that this is the only case study where the curves of the similarity strategy for the average and the minimum value are different. The reason is that the study has similar test cases of the same size. Then random choice is very frequently applied for choosing the test case to be discarded.

For Case Study 2 (Figure A.12), with 33 most effective test cases out of 66, the similarity approach is clearly more effective, even in the worst case that coincides with the average one. In this case, the most effective test cases are completely different. Therefore, the similarity approach, by eliminating redundancy and keeping the biggest test case, selected all 33 most effective test cases from 50% of selection goal.



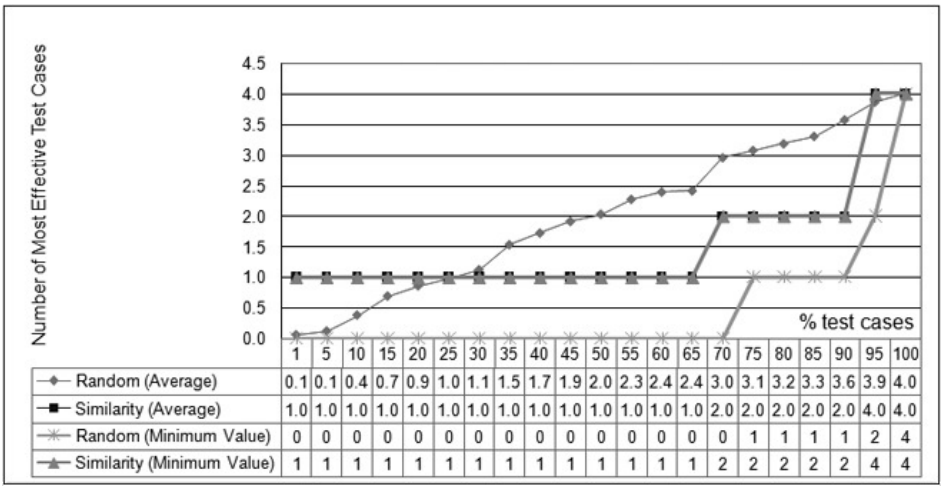
	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.0	1.4	2.7	4.5	6.5	8.2	9.5	11.5	13.0	14.3	16.6	18.1	19.5	21.2	22.8	24.5	25.7	28.0	29.4	30.9	33.0
Similarity (Average)	1.0	3.0	6.0	9.0	13.0	16.0	19.0	23.0	26.0	29.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0
Random (Minimum Value)	0	0	0	2	2	4	5	6	8	10	11	12	15	17	18	21	21	24	27	29	33
Similarity (Minimum Value)	1	3	6	9	13	16	19	23	26	29	33	33	33	33	33	33	33	33	33	33	33

Figura A.12: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 2.

Finally, for Case Study 3 (Figure A.13), with 4 out of 130 most effective test cases, the random approach is more effective in the average case, from 25% of test selection goal. As Case Study 1, the most effective test cases are similar (diverging by 4-6 transitions), but not as much as in Case Study 1. Therefore, there is a gain for the similarity approach up to 20%. From this point on, the random strategy gets more chance to select the 4 out of 130. Nevertheless, note that the similarity approach (both average and minimum number) is more effective than the worst case for the random strategy, guaranteeing that, at least one is selected for each selection goal, even the more restricted ones.

Regarding the questions put in Section A.3.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? Concerning detection of all most effective test cases, the similarity strategy is more effective whenever the most effective test cases are not so similar since the strategy focus on selecting the most different ones (Case Study 2). Nevertheless, if the worst case for random selection is considered, similarity can



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.1	0.1	0.4	0.7	0.9	1.0	1.1	1.5	1.7	1.9	2.0	2.3	2.4	2.4	3.0	3.1	3.2	3.3	3.6	3.9	4.0
Similarity (Average)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	2.0	2.0	2.0	2.0	4.0	4.0
Random (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	2	4
Similarity (Minimum Value)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	4	4

Figura A.13: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 3.

be more effective since this strategy is more deterministic and has shorter variations on results (Case Studies 1 and 3). Indeed, from Figure A.14, note that the similarity approach is more likely to select at least one of the most effective test cases for all selection goals. Furthermore, for severe selection percentages such as the ones from 5 to 20%, similarity can present better results than random selection (Case Study 3).

(ii) What are the limitations of the similarity strategy? The limitations of the strategy are on: a) the criteria for discarding test cases; and b) the similarity degree of the most effective test cases. The criteria adopted in this paper for discarding test cases (the biggest test case) may not be directly related to the criteria for choosing the most effective test cases (here are the ones that cover the biggest number of faults). In this situation, the strategy is not precisely guiding the choice towards the goal and the results may not be reasonable unless the most effective test cases are not similar. In this case, the similarity strategy will preserve them depending on the selection goal.

(iii) In which circumstances is it more advisable to apply each strategy? The similarity strategy is recommended whenever the criterion for defining the most effective test cases can influence or is related to the criterion for selecting the test cases to be discarded. If the former are semantics ones, then one possibility is to prioritise test cases, for instance, following the approach proposed by Bertolino *et al.* [6]. The random

approach is more recommended otherwise, however, to avoid the worst case, it should be applied several times. If a guarantee that at least one most effective test case is preserved is important, than the similarity strategy is recommended.

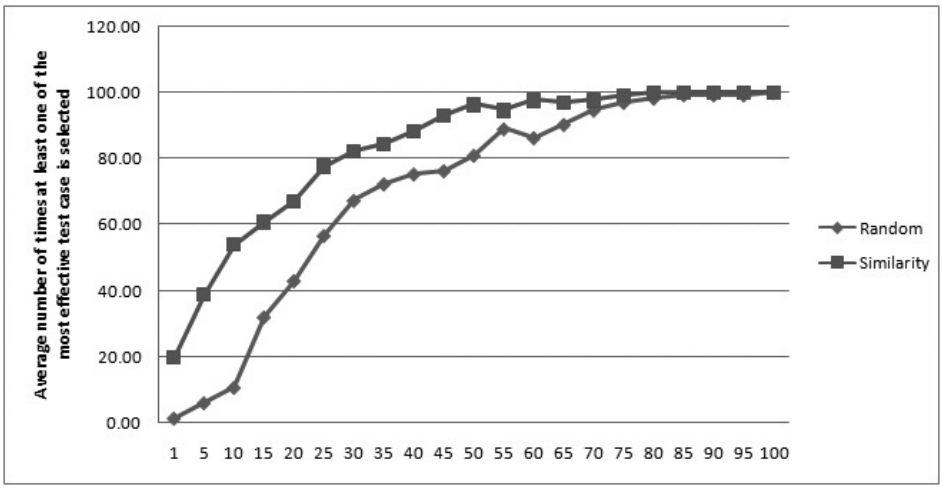


Figura A.14: Average number of the times (out of 100 executions of each strategy) at least one of most effective test cases is selected in all case studies for each test case selection goal.

A.5 Case Studies - General Remarks

Concerning transition-based coverage, for the case studies conducted, the similarity approach can be more effective than the random strategy. There are considerable advantages from 20% of test case selection goal (see Figures A.5 and A.6).

Even for very restrict selection goals such as 5%, the strategy can be more effective. Full all-transitions and all-transition-pairs is only achieved by the similarity approach.

Concerning fault coverage, for the cases studies conducted, the similarity approach has also a superior performance. For all case studies, the highest coverage is only achieved by the similarity approach. There are also considerable advantages from 20% of test case selection goal. However, the random choice had a better performance for Case Study 1 with test case selection goal of less than or equal to 20%. The similarity strategy excludes the most similar test cases, this means that for cases where it is necessary to exclude several test cases, the strategy begins well, but the last test cases that will be excluded, usually, does not have similarity, so the criteria to be used is the path length. Moreover, the criterion for discarding

similar test cases can also influence the results by guiding the strategy for selecting more faults.

Even though this is not always more effective when detecting all of the most effective test cases, the similarity approach can be more precise in detecting at least one of them. The limitations for selecting all of them occur when their degree of similarity do not allow them to be included in the selected test suite such as for Case Study 1 (see similarity of faults on Table A.4) and Case Study 3.

It is important to remark that test cases and faults are considered to be equally relevant. Different results may be reached if experiments are conducted by considering a number of attributes that may add value to specific test cases and also to the faults. Also, the similarity approach opts for keeping the biggest test case in the test suite for the sake of improving transition coverage. This decision is closely related to the fact that the focus of this work is on functional testing, where thorough coverage is critical. However, it may also be interesting to investigate the strategy when the smallest test case is kept.

Another key point to consider when analysing the results is the number of faults defined in the fault models. However, the number of faults do not influence on the results obtained: increasing or decreasing the number of faults has a very similar effect on both strategies. To confirm this claim, an experiment was performed aiming at observing the behaviour of both strategies when from a few to several faults are incrementally included in the model. The results of this experiment (that are not included in this paper for the sake of space) show that the number of faults does not bias the results. However, with more faults, clarity of the results is improved.

Regarding computational complexity, the test case generation algorithm has exponential complexity, but state space explosion is handled by requiring only *all-one-loop-paths* coverage. The selection algorithm, the main focus of this paper, is $O(n^3)$, where n is the number of test cases. For the case studies conducted, the time consumed for each algorithm (generation and similarity selection) considering one execution of the similarity strategy with 50% selection goal is presented in Table A.5. As mentioned before, the suites of these case studies are for manual test execution. By considering that the average time for executing a test case is 2 minutes, it would roughly take 48 minutes, 132 minutes, and 260 minutes for executing 100% of the test cases for Case Study 1, 2 and 3, respectively. By selecting 50% of the test

Tabela A.5: Execution time for full test case generation and also one execution of similarity selection algorithms with 50% test case selection goal.

	Case Study 1	**Case Study 2**	**Case Study 3**
Test Case Generation	16ms	15ms	32ms
Similarity Strategy (Computing the Similarity Matrix)	16ms	110ms	484ms
Similarity Strategy (Selecting Test Cases)	0ms	0ms	31ms
Total	30ms	125ms	547ms

cases, half of the time is saved with only an additional test selection time of 32ms, 125ms, and 547ms for Case Study 1, 2 and 3, respectively. Obviously, in practice the complexity of test execution grows with the size of the test suite. Also, test selection may require further analysis, for instance, running the selection algorithm more than once. Therefore, there are other gains and losses to be considered than only counting the exact time for executing each test case. Nevertheless, the difference on the magnitude of the numbers points out that the similarity strategy can be practical and indeed improve test productivity and reliability.

Apêndice B

LTS Generator

This Appendix presents the generator that was implemented to generate the inputs - LTS models - for our experiments. The goal of this generator is to generate different LTS models using a specific configuration. This configuration considers:

- **Depth:** The depth of the LTS. It is calculated by consider the biggest path (from initial state to final state - without loops);
- **Number of Loops:** One loop is an edge that goes back to any prior state or to itself;
- **Number of Forks:** A fork is a state with more than one outgoing transitions;
- **Number of Joins:** A join is a state with more than one incoming transitions.

Loops, forks and joins can add redundancy in an LTS model. The intention is to construct different LTS models that contains the specified configuration. The steps to construct the LTS models can be seen in Algorithm 4.

The inputs of this algorithm are: the number of LTS models that will be generated; and the configuration. The configuration is a number for each of the following elements: Depth, number of loops, number of joins, and number of forks.

Firstly, an initial sequence is generated following the depth (line 1) constraint. For example, if the depth is 3, the initial sequence has 4 states (0, 1, 2, 3) and 3 transitions (0 to 1, 1 to 2 and 2 to 3). The next step is to aggregate the new structures to the initial sequence (lines 2 - 11). For this, all structures are placed into a list (structures), e.g. if $NumberOfLoops = 2$, $NumberOfJoins = 1$, $NumberOfForks = 3$, then

```
   input  : NumberOfLTSModels, Depth, NumberOfLoops, NumberOfJoins, NumberOfForks
   output: LTSModels
1  buildSequenceOfTranstions(depth);
2  structures = getStructuralPatterns(NumberOfLoops, NumberOfJoins, NumberOfForks);
3  shuffle(structures);
4  for each structure:structures do
5      switch structure do
6          case join
7              putAJoin(depth);
8          case fork
9              putAFork(depth);
10         case loop
11             putALoop(depth);
```

Algorithm 4: LTS Generator - Algorithm

$structures = \{loop, loop, join, fork, fork, fork\}$ (line 2). These structures are shuffled (line 3) to increase the diversity of the generated LTS models, since aggregating these structures in different orders, increases the probability of generating different LTS models.

After shuffling the structures, each structure is aggregated to an actual LTS, observing some constraining rules:

- **Depth:** It is not allowed to aggregate any structure that violates the maximum depth;
- **Join:** Two states are randomly chosen, and these states can not be adjacent. From each of these state, one new transition is created, both going to the new state (the joining state);
- **Fork** - A state is randomly chosen, and two new transitions (and states) are created, outgoing from the chosen state;
- **Loop** - Two states are chosen randomly, however the loop must be placed from the deeper state to other state selected.

Apêndice C

Experiment - Test Suite Reduction

This Appendix shows the comparison among the strategies (G, GE, GRE and H). The variables are:

- **Dependent:** The Reduced Test Suite Size (RTSS).
- **Independent:** The test requirement percentage; the configuration chosen for the depth and amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 5 levels: G, GE, GRE, H and Dissimilarity (DSim).

The experiment definition is formalized as following:

- A null hypothesis (H_0) - $RTSS_G = RTSS_{GE} = RTSS_{GRE} = RTSS_H$: All techniques have a similar behavior in relation to the reduced test suite size;
- An alternative hypothesis (H_1) - $RTSS_G \neq RTSS_{GE} \neq RTSS_{GRE} \neq RTSS_H$: All techniques have a different behavior in relation to the reduced test suite size.

Each technique was executed 200 times. During the analysis we considered a confidence level of 95% (i.e. a significance level - α - of 0.05). The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[1]. The results can be seen in Figures C.3, C.2, C.1 and C.4.

[1] http://www.minitab.com/

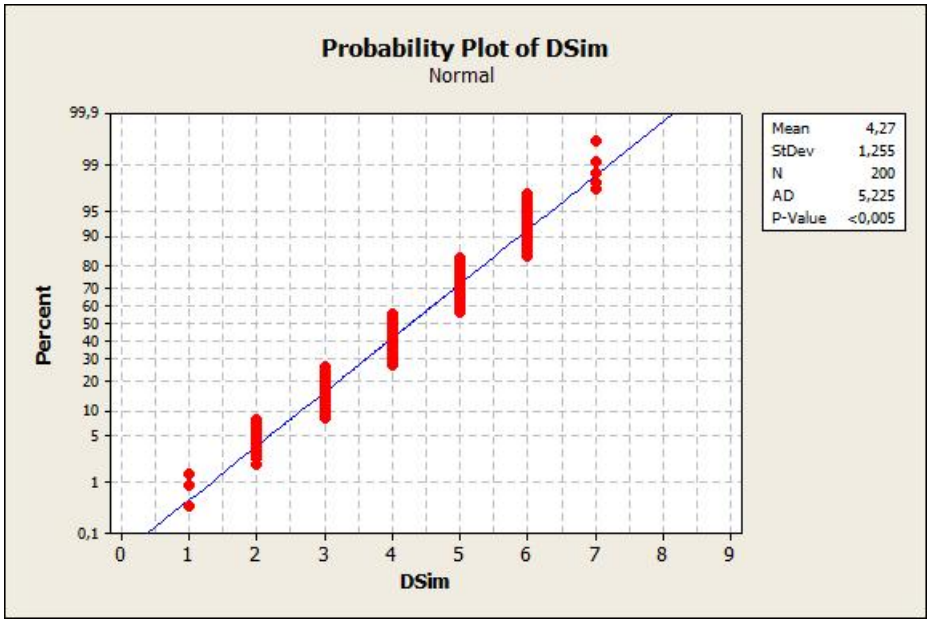


Figura C.1: Anderson-Darling normality test - GRE

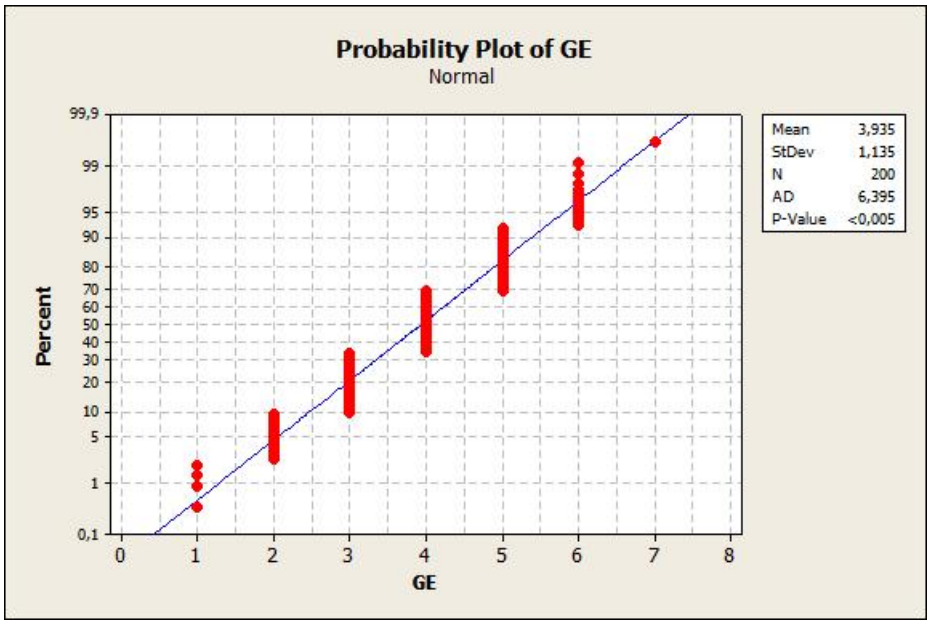


Figura C.2: Anderson-Darling normality test - GE

Observe that all $p - values$ are lower than the significance level ($\alpha = 0,05$), then the data do not show a normal distribution. Thus, it is required to apply a non-parametric tests. Since, there is only 1 factor and more than 2 treatments, a Kruskal-Wallis is applied to check the null hypothesis.

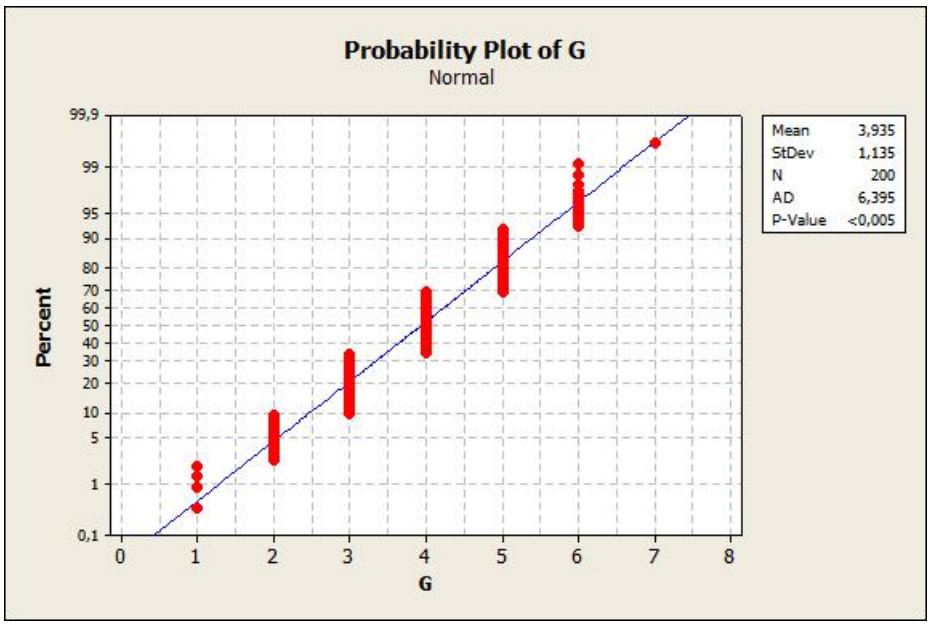


Figura C.3: Anderson-Darling normality test - G

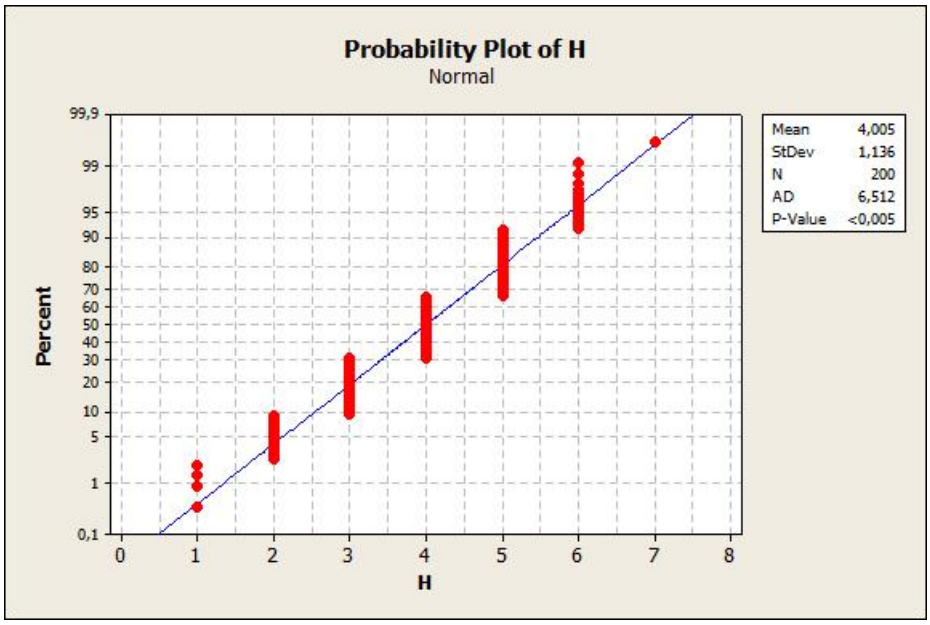


Figura C.4: Anderson-Darling normality test - H

The sample medians for the four treatments (G, GE, GRE, H) were calculated and are the equal: 4.000.

The test statistic (H) had a $p - value$ of 0.881. Since the $p - value$ is higher than the significance level ($\alpha = 0,05$), the null hypothesis can not be rejected. In other words, the obtained results, from the strategies, using a confidence level of 95%, indicate that G, GE,

Tabela C.1: Kruskal-Wallis Test - G, GE, GRE, H

Factor	N	Median	Ave Rank	Z
G	200	4.000	396.6	-0.27
GE	200	4.000	396.6	-0.27
GRE	200	4.000	396.6	-0.27
H	200	4.000	412.1	0.82
Overall	800		400,5	
H = 0.67 DF = 3 P = 0.881				

GRE and H can not be considered different.

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Strategies for Controlling the Size of Test Suite Generated from MBT Approaches

Emanuela Gadelha Cartaxo

Thesis submitted to Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande in partial fulfillment of the requirements for the degree of Doctor of Computer Science.

Area: Computer Science
Research line: Software Engineering

Patrícia Duarte de Lima Machado
(Supervisor)
Antonia Bertolino
(Co-Supervisor)

Campina Grande, Paraíba, Brazil

Abstract

Testing is the most commonly applied technique to evaluate the quality of software as part of verification & validation processes. However, it is usually an expensive activity. Promising to reduce costs as well as promoting effectiveness, Model-based Testing (MBT) approaches have been proposed, where test cases can be obtained from specifications. In MBT, the algorithms used to obtain test cases are usually based on a "search" in a behavioral model and, in most of the times, the stop decision is based on structural coverage criteria that are exhaustively applied. Therefore, in this context, the number of applicable test cases tends to be very high. On the other hand, usually, there are not sufficient resources (time and money) to execute all of them. Also, some test cases may exercise common sequences of functionalities. In this sense, redundancy is an important concept that can be considered to obtain a smaller test suite, once that redundant parts may not increase functionality coverage or fault detection.

Some strategies for controlling the size of the test suites have been proposed: test case selection and test suite reduction. The former usually considers a test purpose (to reduce a space search) and/or fix a number of test cases that are desired without taking into account the redundancy concept. On the other hand, some strategies for test suite reduction are proposed and experimented considering structural redundancy for white-box testing.

Obviously, it is necessary to seek strategies for controlling the size of the test suites generated from MBT approaches that consider the redundancy concept. Different strategies for controlling the size of test suites are proposed in this thesis focusing on selection and reduction. Results show that strategies for selection and reduction based in Similarities are good to detect faults and provide a adequate coverage. Even though the strategies proposed can be applied to different testing levels, the focus is on system testing.

Finally, a new way to evaluate test suite reduction strategies - by considering the rate of fault detection - is proposed. Even though, the rate of fault detection is a metric widely used to compare test suite prioritization strategies, it has not yet been considered to evaluate test suite reduction strategies.

Acknowledgments

First of all, I would like to thank God for everything.

Thanks to my parents and sisters for their love, understanding and support in all difficulties found during these years. Probably, I never would reach here without this.

Thanks to José Lima Júnior for his patience, understanding and love during these years.

I wish to thank my supervisor Patrícia Machado for her professional support. She was a good teacher and also a friend. Her support and patience were valuable during these years.

I wish to thank Antonia Bertolino for her productive collaboration during the course and by adopting me in her group. All discussions were valuable for this work.

I wish to thank Eda Marchetti for her professional (valuable discussions) and also personal (including all smiles and "chiacchieri") support while I was in Pisa.

Francisco Oliveira Neto for his technical support, valuable discussions during implementation of the strategies and, of course, for being always open to hear my confidences.

I wish to thank João Felipe Ouriques and Priscila Vieira for their technical support.

I wish to thank BTC-RD team for all discussions.

Finally, thanks to my friends: Ana Emília, Ana Esther, Andréia Karla, Danilo, Laísa, Neto, Rafael, Rafaelly, Ramon, Raniere, Spachson and Verlaynne by personal support and by having tolerated my moods.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing is an activity to evaluate the quality of the applications and it is usually applied in practice as an activity of the verification and validation process. This activity often consumes a significant amount of the resources in development projects [47], therefore researches have been directed to develop approaches that can contribute to decrease the costs (e.g. time and money [34; 8]) that are demanded by this activity so that it can be effectively and thoroughly applied. One of those approaches is Model-Based Testing (MBT).

MBT has become popular due to the need for quality assurance, and also due to the emerging model-centric development paradigm and test-centered development methodologies [49]. That approach promises to control software quality and to reduce the inherent costs of a testing process, since test cases can be generated from the software specification. Thus, test cases can be obtained before or during the development process and so, when the application code is available, the test cases can be executed. Summarizing, MBT has been pointed as an approach to increase reliability, effectiveness and productivity in the software process, since its promise is to control software quality and to reduce costs [49].

Generally, model-based testing approaches generate a huge number of test cases (large test suite) [42]. Since the available time and money to execute all of them are restricted [26] (particularly for manual testing), it is necessary to decrease the size of the test suite, in other words, we have to obtain a subset of test cases. This subset should contain the best test cases, *i.e.* that are able to reveal faults and provide a good coverage (e.g. transition or requirements coverage, among others). This is a difficult task, since we need to observe many variables such as functionality coverage or resource constraints. In practice, the task of reducing the

size of the test suite is a manual process that is error prone and without sufficient guaranties that the system will be effectively tested, since coverage criteria or fault detection are not used as parameter for strategy evaluation.

In addition, test suites generated from MBT approaches usually contain a considerable degree of redundancy between test cases, *i.e.*, two test cases can be so similar. It is probable that they do not add value to the suite by either guaranteeing a better coverage of a given criteria or having the capability of revealing additional defects not yet covered. In this case, we named them redundant test cases.

Researchers have investigated two approaches for addressing the test suite size problem [66]:

- Test Case Selection - Algorithms (such as proposed by Rothermel and Harrold [52; 53] and Jard and Jeron [41]) for test selection select a subset of the original test suite that may (or not) provide the same coverage as the original test suite;

- Test Suite Reduction - Algorithms (such as proposed by Wong et al.[65] Zhong et al. [68], Harrold et al. [36] Ma et al. [66] and Chen and Lau [19]) for test suite reduction select a representative subset of the original test suite that satisfies a set of test requirements (coverage criteria), so select a subset of the original test suite that provides the same coverage (according to the test requirement) as the original test suite;

Note that, both of them deal with the test suite size problem, trying to reduce the number of test cases. The difference between them is that for reducing the size of a test suite, test suite reduction considers a specific set of test requirements (coverage criteria) and the reduced test suite *must* satisfy that set as long as the original suite does.

The following sections of this chapter present: an overview of the thesis and its main contributions (Section 1.1); the adopted methodology (Section 1.2); and finally, the structure of this thesis is presented (Section 1.3).

By definition, theses strategies reduce a test suite, however the but nothing of them take into consideration the number of test cases that we can able to execute at this moment.

1.1 Overview of the Thesis

In this thesis, we seek a solution for the test suite size problem caused by redundancy in the MBT context applied to system testing. Our research questions are:

- Is it possible to reduce (test selection/test suite reduction) the size of the test suite by eliminating redundant test cases based on a similarity function and still keep reasonable coverage of a given test criteria?
- Is it possible to define a strategy that produces a smaller test suite based on that function to maximize transition coverage of the resulting test suite?
- Is it possible to combine that strategy with other strategies such as value-based that are applied to focus on usage scenarios and maximize the fault detection capability?

As answers of these questions, we proposed some strategies that deal with the test suite size problem. Then, the main contributions of our work are:

- **Similarity Function** - This function calculates the distance among test cases of a test suite. This value represents the degree of redundancy between each pair of test cases;
- **Similarity-based Selection** - A new strategy for test case *selection* is proposed based on the Similarity function. This strategy is compared to the Random selection strategy;
- **Weighted-Similarity Approach (WSA)** - A new strategy for test case *selection* is proposed based on the Similarity function and weights. This strategy is compared to other well-known strategies in the literature;
- **Dissimilarity** - A new strategy for test suite *reduction* is proposed based on the Similarity function. This strategy is compared to 4 well-known strategies in the literature;
- **New way for evaluating reduced test suites**- The main criteria used to compare strategies for test suite reduction is the test suite size where the smallest test suite would be the best one. We propose to use the selection order to compare test suite reduction strategies, since the faults are important and it is not always possible to execute all test cases (it is necessary to stop the execution of a reduced test suite).

As said before, these strategies deal with the redundancy problem and they can be applied to model-based testing approaches in order to improve the results of test case generation algorithms by eliminating redundant test cases from the generated suite. Also, they are based on the use of a similarity function to discriminate among the most different test cases. In this sense, the strategies can be effective under the following assumptions:

- Similar test cases are redundant in the sense that they cover a common set of functionalities and have similar capability of revealing faults. Therefore, some of them can be eliminated to meet resource constraints of a project;
- Probably there is no additional gain to keep (the redundant test cases) them in the test suite since they are not significantly affecting functionalities/fault model coverage.

The scope of this thesis is model-based testing approaches where the models are Labelled Transition Systems (LTSs). From LTSs, system test cases can be obtained. Then, from the test suites, test case selection/ test suite reduction strategies can be applied by considering the redundancy concept. Note that LTS models can be obtained from a number of commonly used models such as UML behavioral specifications [16; 39] and diagrams. Therefore, the results presented in this thesis can be applied to a larger scope as illustrated by the presented case studies.

1.2 Methodology

The goal of strategies for controlling the size of the test suites is to decrease the test suite size. In this thesis, these strategies are proposed according to a similarity function to select the most different test cases, obtaining the best coverage of functionalities.

The generic process to select test cases/ reduce test suites to be followed is shown in Figure 1.1. First, the test suites are extracted from the LTSs models. The test case selection/ test suite reduction strategies can be applied over those test suites. For applying test case selection strategies, it is necessary to define an objective, such as a test purpose, an intended size, among others. On the other hand, for applying test suite reduction strategies, test requirements can be automatically obtained from the model.

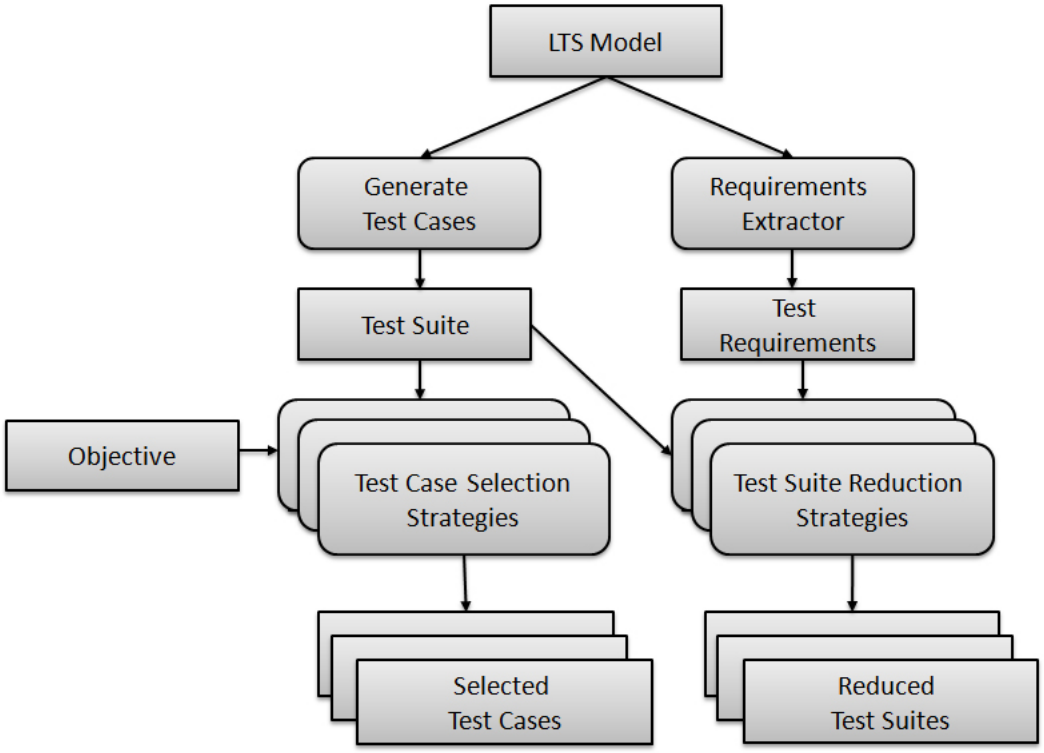


Figure 1.1: Overview of a test case selection/ test suite reduction process

After applying the test case selection or test suite reduction strategies, we obtain a subset of the test suite. In order to evaluate these strategies, some case studies and experiments are performed. Transition coverage and fault detection are used as criteria to compare and evaluate those strategies.

1.3 Outline of the Thesis

The following chapters of this thesis comprises the background, proposed strategies and their respective evaluation, related works and some concluding remarks (including future works). Part of the material in some chapters have already been published ([6; 14; 15; 18; 7]). More specifically, the topics of the thesis are organized in the chapters as following:

Chapter 2

In order to make the thesis self-contained, this chapter presents terms and concepts used in Software Testing, Model Based Testing, some Coverage Criteria, Test Case Selection,

Test Suite Reduction, Test Case Prioritization, Value Based Approach, Experimentation in Software Engineering and Statistical Analysis.

Chapter 3

In this chapter, we present our concept of redundancy and illustrate the problem in a real application. After this, we present our *Similarity Function*. This function calculates the degree of similarity between a pair of test cases by considering the number of identical transitions between two test cases and their respective lengths. To calculate all degree of similarities among all test cases of the test suite, a similarity matrix is built.

Chapter 4

Our strategy based on *Similarity for test case selection* is presented in this chapter. The algorithm of the strategy and a toy sample are presented in order to show the application of the strategy. The evaluation of this strategy is performed through: a case study (by analyzing the transition and fault coverage) and an experiment (by analyzing the transition coverage). The results are compared to random test case selection strategy.

Chapter 5

In this chapter, we propose another strategy - named *Weighted-Similarity Approach (WSA)* - for test case selection based on similarities and weights associated to test cases. The algorithm and an example to illustrate it are presented. Two case studies are executed to compare WSA, Random Selection, Guided Random and Similarity (Chapter 4) by considering fault and transition coverage).

Chapter 6

This chapter presents our strategy for test suite reduction based on similarity. This strategy, named *Dissimilarity*, is able to reduce the test suite using as test requirement, the transition coverage criteria. In order to explain the strategy, we present the algorithm and its execution in a toy example. To evaluate and compare our strategy and 4 well-known strategies for test suite reduction in the literature, a case study and an experiment are performed.

Chapter 7

In this chapter, we present a new way to analyze reduced test suites based on the order of the test cases. This work is inspired in the metric used to evaluate the test case prioritization strategies. An example is shown to illustrate our motivation. Two case studies are performed using the proposed metric.

Chapter 8

This chapter presents some related researches on strategies for test case selection and test suite reduction. The focus is on solutions that can be automated since our scope of study is on MBT and system testing.

Chapter 9

This chapter presents the concluding remarks and future works related to our contributions.

Chapter 2

Background

This Chapter presents some basic concepts about Software Testing, Model Based Testing, Coverage Criteria, Test Case Selection, Test Suite Reduction, Test Case Prioritization, Value Based Approach and Experimentation in Software Engineering and Statistical Analysis, clarifying the terminology and concepts that will be used in this Thesis.

2.1 Software Testing

Software testing is a very important activity that is part of the software development process. Since this activity can spend more than 50% of the resources of the software development [5], that is, in general, not executed properly or even skipped due to resource (cost and time) constraints [26].

There are two main reasons to execute software testing [5]: to assess the quality of the application; and to reveal problems in application under testing. Since testing is concerned with "error", "fault", and "failure", it is important to clarify these terms before presenting the other concepts about testing. According to Jorgensen [42]:

- An *error* occurs because of an incorrect or missing code;
- A *fault* or *defect* is the result of an error;
- A *failure* occurs when the fault executes, then the application does not perform the functionality as required. This is noted when the output is wrong, an abnormal termination occurs or time restrictions are violated.

Software testing is the activity of designing tests and exercising the software with them, in order to investigate on quality attributes and find defects. We can classify the test process according to its goal [57]:

- **Defect testing:** Where the goal is to reveal faults in the software;
- **Validation testing:** Where the goal is to demonstrate to the developer and the system customer that the software meets its requirements.

Our strategies are independent of the goal of the testing process, however the case studies are focused on validation testing.

The process of software testing can be divided into [42]: test planning, test case development, running test cases, and evaluating test results. Our focus is on test case development. In the next subsections, we present a definition of test case, testing methods and level of testing.

2.1.1 Test Case

The essential task of software testing is to determine a set of test cases for testing the specific system. Each test case is associated with a system behavior, and is composed by [42]:

- **An identity:** An identifier can be associated to the test case, for testing management and requirements tracing for example;
- A set of inputs, where an input can be:
  - **Pre-condition:** The system state that must hold before test case execution;
  - **Actual inputs:** Actions that should be executed;
- Set of expected outputs, where an expected output can be:
  - **Post conditions:** The system state that must hold after test case execution;
  - **Actual outputs:** An output of the system.

In order to execute one test case, the system must hold the specific state (pre-condition). Then, the system is exercised with the inputs, collecting the outputs until the post condition is

reached or a failure is detected. Finally, the obtained results are compared with the expected ones to check if the test has passed or not, i.e., if the software behaved as expected.

This is a general format of a test case. Depending on the kind or level of testing, this format is tailored. For example, a unit test is a method call, where the inputs are values that instantiate parameters. On the other hand, at a system test level is an execution scenario of the application. Then, usually, inputs are a sequence of actions executed by a user, and the respective system outputs are observed.

Figure 2.1 presents a System Test Case. This test case executes the system to validate the scenario "add phonebook contact with success". To execute this test case, the system must be in Idle, the Phonebook application must be installed in the phone and must have enough memory to add a new contact. For each input (user action) of this test case, an expected output (system state) is presented. This expected output is then compared to the real system state. If they are the same, then we "pass" the test case.

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

Input (User Action)	Output (System Response)
Press Menu Center Key	List of features is showed.
Scroll until Phonebook icon	Phonebook icon is highlighted.
Start Phonebook application.	The contact list is displayed.
Select the New Contact option.	The New Contact form is displayed.
Type the contact name and the phone number.	The new contact form is filled.
Confirm the contact creation.	A new contact is created in My Phonebook application.

Figure 2.1: System Test Case

2.1.2 Testing Methods

Testing methods are used to identify test cases. A testing method may follow a functional testing approach or a structural testing approach [5].

Functional Testing

Functional testing is based on the view that any application can be considered as a "*black box*", where only inputs and outputs are taken into consideration, and the implementation is not known. Since the implementation is not considered, only the specification is used to obtain the test cases.

For functional test cases, there are two advantages: the test cases can be obtained in parallel with the implementation, and, if there are any changes in the source code (except changes of the functionalities), the test cases do not change. In general, functional test cases may present redundancies among themselves [42], which may increase the costs of software testing.

Structural Testing

In contrast to the functional testing, the structural testing approach considers the implementation of the system to obtain the test cases. This approach is also called "*white box*", where it is necessary to observe inside the box in order to identify the test cases. In other words, to obtain the test cases, the implementation needs to be available. This approach lends to the definition and use of test coverage metrics [5]. Such coverage metrics define which part of software will be tested.

2.1.3 Level of Testing

Beizer [5] and Jorgensen [42] show three levels of testing: unit, integration, and system testing. Each level has a different goal, and thus different methods are applied to perform the test.

Unit Testing

A unit is the smallest testable piece of an application, that is usually the work of one programmer. Unit testing is performed to guarantee that the unit satisfies its functional specification and/or that its implemented structure matches the intended design structure [5].

A component can be considered an unit. This way, each component/subsystem is tested separately.

Integration Testing

Integration is a process by which components are put together to create a larger component. The goal of integration testing is to reveal faults that arise when the components are put together. This testing considers that each component has already been tested and are individually satisfactory, as demonstrated by a successful passage of component tests.

The hard task in integration testing is to locate the "faults". Usually, to make that easier we should use an incremental approach to system integration and testing [57].

System Testing

A system can be consider a big component. Then, a system testing is performed when all components are already together(i.e., after the integration occurs). The goal of system testing is to reveal issues and behaviors that can only be exposed by testing the entire integrated system.

This testing focus on capabilities and characteristics that are presented only with the entire system. System scope can be classified by the kind of conformance [10]:

- **Functional:** The goal is to find errors in the functionality of the system, in other words, this testing assess if for given inputs, the right outputs are generated;
- **Performance:** The goal is to observe the behavior of the system under heavy load;
- **Stress or load:** The goal is to find failures in the system under unexpected inputs, unavailability of dependent applications, and hardware or network failures.

Our strategies are independent of the level of testing, however the case studies are focused in functional system testing.

2.2 Model-Based Testing

Model-Based Testing (MBT) is a functional approach and consists in the automatic generation of tests using models extracted from the system specification [61]. For its application, it is necessary that the software requirements are precisely defined, in order to characterize with exactness the system behavior [5].

This section presents concepts about models (Subsection 2.2.1) and activities of Model based Testing (Subsection 2.2.2).

2.2.1 Models

System Models are an abstract view of a system. Since a model is an abstraction, it is a simplification of the reality that highlights the most important characteristics [38].

Models may represent a system from different perspectives [57]:

- **External:** The environment of the system is represented by the model;
- **Behavioral:** The behavior of the system is represented by the model;
- **Structural:** The architecture of the system is represented by the model.

Since model-based testing is a black-box approach, the behavioral perspective of the system is adopted. For behavioral perspective, the following models can be highlighted: Decision Tables, Finite State Machines (FSM), Markov Chains, Statecharts, UML diagrams, Labelled Transition System (LTS), among others. A model is chosen according to the characteristics of the system. In this work, we consider the specific type of LTS model - ALTS (Annotated LTS).

Labeled Transition System - LTS

LTS is a directed graph in which vertices are named states, and edges are named transitions. These models are largely used as the semantic formalism of several specification notations [41] and so they can be easily obtained from functional specifications by using translation tools, such as UMLAUT [39]. Several tools use LTSs as the model for obtaining test cases. Among these tools are: SPACES [3], TGV [41], LTS-BT [17] and TaRGeT [48].

Formally, an LTS can be defined as a 4-tuple $S = (Q, A, T, q0)$, where [27]:

- 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;
- q_0 is the initial state.

Usually, LTS take into account internal and external actions [41]. Since our focus is on functional testing, we show two different LTS that can be used for modeling the functional behavior of the applications: Input-Output LTS and Annotated LTS.

Annotated LTS - ALTS.

Annotated LTS (ALTS) is an LTS that has transitions actions and also, annotations. These annotations are insert in the LTS with a specific goal. In our case, this goal is to generate functional test cases, therefore, such annotations are related to this activity. Figure 2.2 shows an example of an Annotated LTS model that represents the behavior of an application where the user wants to save one phone number that is embedded in a message.

Observe that each label has an annotation for the action: *steps*, *conditions* or *expectedResults*. These correspond to, respectively, a user action, a pre-condition and a system response, and are inserted in the LTS to facilitate the test case generation.

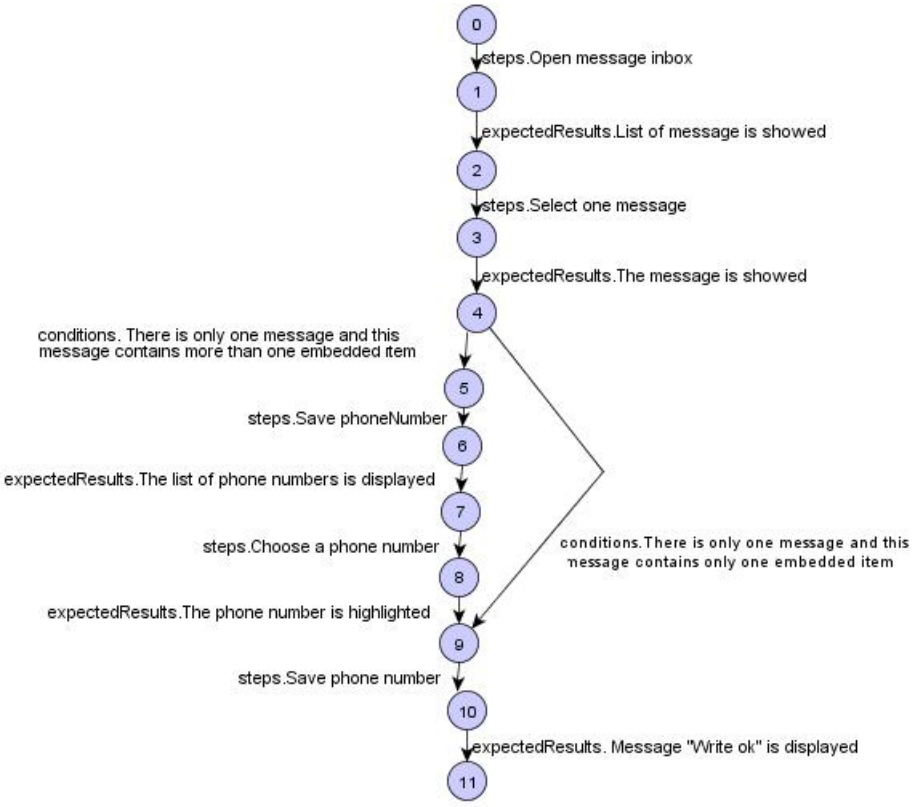


Figure 2.2: Annotated LTS model

2.2.2 Activities of MBT

The activities related to MBT can be described as follows [28]:

1. **Build the model**: The formal model is built from the software specification. This model needs to be formal, i.e., precise, consistent and unambiguous.

2. **Generate expected inputs and outputs**: The test inputs and outputs are generated from the formal model. In order to exercise the system, we need to generate the sequence of the inputs, while the expected outputs represent the expected system responses.

3. **Run tests**: The system is executed with the generated inputs, generating outputs;

4. **Compare outputs with expected outputs**: The generated outputs are compared to the expected outputs.

For example, consider we have the following requirement: *The user must be able to save a phone number that is embedded in a message*. From this requirement, we build the model (activity 1) and obtain the ALTS shown in Figure 2.2.

Using the ALTS showed in Figure 2.2, we can transverse this model by using Depth Search First (DFS) and generate 2 test cases (path coverage criteria) and its respective inputs and outputs (activity 2). The test cases to be generated from a model depends on the adopted coverage criteria (more details in Section 2.3). Figure 2.3 shows one of them.

Pre-Condition: There is only one message and this message contains only one embedded item

Input (User Action)	Output (System Response)
Open Message Inbox	List of messages is showed.
Select one message	The message is showed.
Save Phone Number	Message "Write ok" is displayed

Figure 2.3: Test Case

Figure 2.4 shows the flow of the MBT activities. As can be seen, the models are obtained from the requirements. This activity is usually done manually, and it requires a specialist

in the notation used to construct the formal model. However, there are already attempts, in practice, to automatically generated models from requirements specification written in a natural controlled language [48].

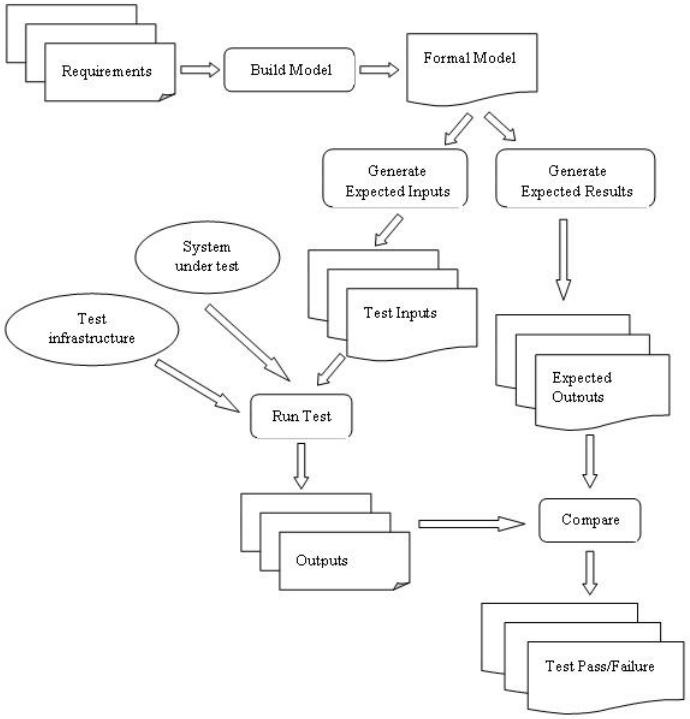


Figure 2.4: Model Based Testing

Inputs to execute the system and expected outputs are extracted from that model. Then, the system is executed with the tests (activity 3). When the system is executed, the outputs are produced. Finally, these outputs are compared to the expected outputs and the test cases are defined as Pass or Fail.

2.3 Coverage criteria

A coverage criterion is a set of rules that imposes test requirements on a test suite [2]. Coverage criteria specify the items of the system that must be exercised during testing. There are two purposes [61]:

- **Measuring the adequacy of a test suite:** The coverage level of a specific criterion is an indicator of the quality of the test suite;

- **Deciding when to stop testing:** The tests are run until reaching a coverage level of a specific criterion.

There are consolidated coverage criteria for code coverage (white-box coverage criteria), and many of these coverage criteria are used for black-box coverage [61]. Since our focus is on MBT and the model is LTS, we will present the main coverage criteria used for Transition-Based Coverage Criteria: *all-states, all-configurations, all-transitions, all-transition pairs, all-loop-free-paths, all-one-loop-paths, all-round-trips and all-paths* [61]. Here, we do not consider *all-configurations* because it is not applied to our context, since it is mostly used for *statecharts*.

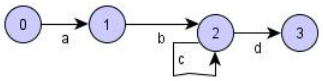


Figure 2.5: LTS model

- **All-states coverage**: Every state must be visited at least once. Considering the LTS of Figure 2.5, to reach this coverage, only one test case is required: abd;

- **All-transitions coverage**: Every transition must be visited at least once. Observing the LTS of Figure 2.5, this coverage can be reached using only one test case: $abcd$;

- **All-transition-pairs coverage**: Every pair of adjacent transition in the model must be traversed at least once. In the LTS of Figure 2.5, to reach this coverage, we need to have test cases that traverse ab, bc and bd at least once. In this case, this coverage is reached with the test cases: abd and abc;

- **All-loop-free-paths coverage**: Every loop-free path must be traversed at least once. A path is loop free when it does not have repetitions. For the LTS in Figure 2.5, only one test case is required to reach this coverage criterion: abd;

- **All-one-loop-paths coverage**: Each path is traversed at most once the loop. Therefore, we have one test for each loop. In Figure 2.5, we need two test cases to reach this coverage criterion: abd and $abcd$.

- **All-round-trips coverage**: This coverage criterion is similar to *all-one-loop-paths* since it requires that all loops are tested in the model, however it is a weaker criterion, since it only requires one path for testing one loop. For the LTS in Figure 2.5, to reach this coverage criterion, the following test cases are needed: abd and abc.

- **All paths coverage**: Every path must be traversed at least once. This corresponds to an exhaustive testing in LTS models, and the generation algorithm should have an heuristic to avoid the state space explosion, enabling the proper use of this coverage.

2.4 Test Case Selection

Test case selection is an activity to select a subset of the test suite according to a specific criterion, for example transitions or requirements coverage. The selected subset may not provide the same coverage as the original test suite [66], however, it may lower the costs of the test process. We can enumerate some strategies for test case selection:

1. **Deterministic:** Where we have a manual choice. For example, one specialist can use his or hers know-how to select test cases;

2. **Random:** The subset of test cases is randomly chosen;

3. **Statistical:** The weights are assigned to guide the choice [4]. One of them is the **Guided Random**, where the choice of the test cases is guided by probability values [50; 3]. For each decision node in an LTS, a probability is assigned. Then, this strategy tries to select the most important transitions (i.e., the transitions with the highest probability values) using a Depth-first search. The goal is to define an unbiased test suite that can be more effective for fault detection, and also to make reliability estimation possible;

4. **Test Purpose:** A test purpose denotes a scenario of a functionality of the system under testing [38]. Jard and Jéron present TGV tool [41], where the test selection is performed when the model is delimitated by the test purpose.

2.5 Test Suite Reduction

Test suite reduction is a technique that produces a representative subset of the original test suite. This subset has equivalent coverage in relation to the original test suite, concerning a specific criterion [66]. This problem can be stated as follows [36]:

Given: Test Suite TS, a Set of Test Requirements $req_1, req_2, ..., req_n$ that has to be covered to provide the desired test coverage of the program, and subsets of TS ($TS_1, TS_2, ..., TS_n$), where each test case from TS_i can be used to test req_i;
Problem: Find a representative set of test cases from TS that satisfies all of the $Req's$.

A test requirement can be a statement, a block, a decision, a requirement and so on. A representative set of test cases must have at least one test case for each Req. Therefore, these test cases satisfy all of the $Req's$. The maximum reduction occurs when the smallest representative subset is found. This is a *NP-complete* problem [22; 36].

The main advantage of test suite reduction is the reduction of the size of the test suite. However, since the test suite is reduced, we risk to decrease the capability of faults detection. Wong et al. [65] and Rothermel et al. [54] propose experimental researches to investigate this risk.

Finally, test suite reduction techniques deal with structural redundancy. The classic definition for redundant test case is: A test case is redundant if other test cases in the test suite provide the same coverage of the program [36]. For example, considering branch coverage as a test requirement, a test suite reduction strategy must find a subset that reaches 100% branch coverage.

Some heuristics for test suite reduction were proposed in the literature. These heuristics are detailed below, since they are compared to our proposed strategy (Chapter 6). For applying the proposed heuristics, it is necessary to have the satisfiability relation between the Test Suite (TS) and the Test Requirements ($Req = req_1, req_2, ..., req_n$). First we illustrate the satisfiability relation required to perform the test suite reduction with the heuristics.

Satisfiability Relation

For each req_n, there is a subset of TS ($T_1, T_2, ..., T_n$), such that all the test cases belonging to T_n can be used to test req_n. Table 2.1 presents a sample of a Satisfiability Relation.

Table 2.1: TS consists of Test Cases $t_1, ..., t_7$, Test Requirement req_n, and Associated Testing Sets are T_n - Example 1

n	Req_n	TS_n
1	req_1	$\{t_2\}$
2	req_2	$\{t_6, t_7\}$
3	req_3	$\{t_1, t_5, t_7\}$
4	req_4	$\{t_1, t_6\}$
5	req_5	$\{t_3, t_4, t_7\}$
6	req_6	$\{t_1, t_2\}$
7	req_7	$\{t_3, t_7\}$

The goal of test suite reduction, as said before, is to meet a subset (Reduced Set - $RS \subseteq TS$) that provides 100% coverage of test requirement, in other words, to satisfy *all* test requirements.

Greedy Heuristic

The Greedy heuristic [23; 25] repeatedly selects the test case t that satisfies the maximum number of unsatisfied test requirements, if there is a tie situation, a random choice is made. The selected test case is added to the Reduced Set (RS) and all test requirements that can be satisfied by that test case are marked as an already satisfied test requirement. This algorithm stops when all test requirements are satisfied. Applying this algorithm to the example showed in Table 2.1, we have:

t_7 satisfies the maximum number of unsatisfied test requirements, then $RS = \{t_7\}$, the requirements req_2, req_3, req_5 and req_7 are marked as satisfied. Now, there is a tie situation: t_1 and t_2 satisfy the maximum number of unsatisfied test requirements. This way, a random choice is made. Considering that t_1 is chosen, then $RS = \{t_1, t_7\}$, the requirements req_4 and req_6 are marked as satisfied.

Finally, the unique requirement that has not yet been marked is req_1, since this requirement is satisfied by t_2, therefore $RS = \{t_1, t_2, t_7\}$, which provides 100% coverage of the test requirements. Rehman et al. proposed TestFilter, a technique to reduce test suites based on

statement coverage [60]. This technique uses the Greedy Heuristic for test suite reduction. The authors used statement coverage as test requirement.

Heuristic Greedy - Essential (GE)

This heuristic (defined by Chen and Lau) is based on [20]:

- **Essential strategy:** Responsible for selecting all essential test cases. A test case is essential when only that specific test case covers one specific requirement;
- **Greedy heuristic:** Responsible for selecting a test case that satisfies the maximum number of not yet satisfied requirements.

Initially, all essential test cases are observed, and their respective requirements are marked. Then, the greedy heuristic is applied. The focus is on solutions that can automated since our scope of study is on MBT. Using the example from Table 2.1, the strategy execute as following.

First, t_2 is chosen, since it is an essential test case. Therefore $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now that we do not have essential test cases remaining, we must apply the greedy heuristic. Since t_7 satisfies the maximum number of unsatisfied test requirements, we add it to the reduced set. Thus, $RS = \{t_2, t_7\}$, and the requirements req_2, req_3, req_5 and req_7 are marked as satisfied.

Finally, the unique requirement that has not been marked yet is req_4. Therefore we apply a random choice between t_1 and t_6. Considering that t_1 is chosen, we obtain the reduced set $RS = \{t_1, t_2, t_7\}$ and req_4 is marked as satisfied. Now that all requirements are satisfied, the algorithm stops.

Heuristic Greedy - $1 - to - 1$ - Redundancy Essential (GRE)

This heuristic (defined by Chen and Lau) is based on [19]:

- The **Greedy** and **Essential** strategies, both presented above;
- **1-to-1 redundancy strategy:** A test case $t_{1-1} \in TS$ is said 1-to-1 redundant, if [21]:

$$\exists t \mid t \neq t_{1-1} \;\&\; t \in TS \;\&\; req(t_{1-1}) \subseteq req(t).$$

In other words, when all requirements satisfied by t_{1-1} are also satisfied by t.

The essential and 1-to-1 strategies are applied alternatively, until no essential or 1-to-1 redundant test cases can be found. That means that the greedy strategy is only applied if neither the essential or 1-to-1 redundancy can be applied. Considering the example from Table 2.1 the algorithm is applied as following.

First, t_2 is chosen, since it is an essential test case. Therefore, $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now, we do not have any other essential test cases. Thus, we have to search for 1-to-1 redundant test cases.

During this search we identify t_4, t_3 and t_5 as 1-to-1 redundant test cases, since: $req(t_4) \subseteq req(t_7)$, $req(t_3) \subseteq req(t_7)$ and $req(t_5) \subseteq req(t_7)$. The test cases t_3, t_4 and t_5 are not considered, since those are redundant in relation to t_7.

Now, at this point, t_7 becomes an essential test case, and needs to be placed in the reduced set. Thus, $RS = \{t_2, t_7\}$, and the requirements req_2, req_3, req_5 and req_6 are marked as satisfied. Finally, we have only req_4 as not satisfied. A random choice is performed, between t_1 and t_6. Considering that t_1 is chosen, the resulting subset is $RS = \{t_1, t_2, t_7\}$ and req_4 is marked as satisfied. Since all the requirements are satisfied, the algorithm stops.

Heuristic H

Harrold et al. [36] present a test suite reduction technique, refered as Heuristic H. Each test requirement has a cardinality, that is the number of test cases that covers that specific requirement. When a test case is added to the reduced set, all requirements covered by that test case are marked. The first step is to identify the test requirement(s) with the lowest cardinality, since they represent the most essential test cases.

Among the unmarked test requirements with the lowest cardinality, the algorithm selects the most frequently occurring test case, i.e., the test case that covers most requirements. If there is a tie, the algorithm chooses the test case that occurs most frequently at the next higher cardinality and so on (if there is another tie where the cardinality is maximum, then a random choice is applied). This algorithm stops when the reduced set has test cases that cover all test requirements.

Summarizing, the main idea is to select test cases according to their essentialness, i.e.,

Table 2.2: Cardinality

Cardinality	Req
1	req_1
2	$req_2, req_4, req_6, req_7$
3	req_3, req_5

keeping in the reduced set the test cases in the order from the most essential to the least essential. Below, we apply this algorithm to the example showed in Table 2.1.

First, we need to calculate the cardinality of each test requirement. The results can be seen in Table 2.2.

For the lowest cardinality (in this case is one), there is only one test case (t_2). Thus, $RS = \{t_2\}$, and the requirements req_1, req_6 are marked as satisfied. Now, the lowest cardinality is two, tied between req_2, req_4 and req_7. Also there is a tie between the most frequent test cases within req_2, req_4 and req_7. These test cases are t_6 and t_7.

Then we must see the next higher cardinality (in this case is 3 - req_3 and req_5) to determine which one of them occur most frequently. Observing req_3 and req_5, we identify t_7 as the most frequent test case. Therefore t_7, is chosen and then $RS = \{t_2, t_7\}$, also req_2, req_3, req_5 and req_7 are marked as satisfied.

Finally, the unique requirement that has not been yet marked is req_4. Again, we have a tie situation, however we do not have a next higher requirement cardinality, therefore we apply a random choice between t_1 and t_6. Considering that t_6 is chosen, the reduced subset is $RS = \{t_2, t_6, t_7\}$, and req_4 is marked as satisfied. Since all requirements are marked, and thus satisfied, the algorithm stops.

2.6 Test Case Prioritization

Test case prioritization is a technique that orders test cases in an attempt to maximize an objective function. The problem is defined by Elbaum et al. as follows [30]:

Given: A test suite TS Test Suite; PTS, a set of permutations of TS; and, f, a function that maps PTS to real numbers ($f : PTS \rightarrow \mathbb{R}$).

Problem: Find a $TS\prime \in PTS \mid \forall\, TS\prime\prime\, (TS\prime\prime \in PTS)\, (TS\prime\prime \neq TS\prime) \cdot f(TS\prime) \geq f(TS\prime\prime)$

The objective function is defined according to the goal of the prioritization. The manager may need to quickly increase the rate of fault detection or the coverage of the source code. Then, a set of permutations PTS is obtained and the $PTS\prime$ that has the highest value of $f(TS\prime)$ is chosen.

Note that the key point is the goal, and the success of the prioritization is measured by this goal. However, it is necessary to have some data (according to the defined goal) to calculate the function for each permutation. Then, for each test case, a priority is assigned and test cases with the highest priority are scheduled to execute first. When the goal is to increase fault detection, there is a metric largely used in the literature, named Average Percentage of Fault Detection – APFD. The highest the APFD value is, the faster and better the fault detection rates are [30].

2.7 Value Based approach

Value based software engineering has been introduced in 1981 with Boehm's Software Engineering Economics book [12] and has inspired the value based management movement in the early 1990 [9]. Its philosophy is that "*quality should not be a goal in itself in the absence of favorable economics*".

Since then, the consideration for software-related value has expanded its scope and values have been incorporated deeply into successful developments process. The common purpose has been promoting the different considerations/measures/knowledges to the foreground so that the software engineering decisions could be guided and optimized by these values.

Saying exactly once for all what "*value*" represents in this discipline is not possible. The referred numbers can be related to various topics. They can represent the economical and financial aspect, they can be percentage or probability, they can represent abstract concepts as quality, availability, usability and so on.

In software development, generally the various system functionalities do not have the same "importance" for overall system performance or dependability, and the testing effort should be planned and scheduled accordingly. Different criteria can be adopted in order to define what "importance" means for test purposes, e.g., component complexity, or usage

frequencies (such as in reliability testing [46]).

Often, these criteria are not documented or even explicitly recognized, but their use is implicitly left to the sensibility and expertise of the managers. Several criteria for assigning the importance factors could be adopted. Obviously this aspect in the proposed approach remains highly subjective, more in the realm of expert judgment than mechanizable methods.

The basic idea is that the test managers must explicit these criteria. The main task is to express, for each functionality, a value belonging to the [0,1] interval, representing its relative "importance" with respect to the other functionalities. This value, called the *weight*, must be assigned in such a manner that the sum of the weights associated to all children of one level is equal to 1; the more critical a functionality is, the greater its weight.

It is worth noticing that the process of functionalities annotation implies a beneficial side-effect: for assigning the appropriate values, the managers are forced to reflect on the relative complexity of each functionality with respect to the context in which it is inserted. Consequently, they focus on the parts where problems could be more critical and become more aware of the importance of each node for the system development.

2.8 Experimentation in Software Engineering

In this section, some basic concepts about Scientific Methods and Experimentation in Software Engineering are presented.

2.8.1 Scientific Methods in Software Engineering

There are four scientific methods that are used for doing research in software engineering. Those methods are [33; 62]:

- **Scientific:** A model is built by observing the world;
- **Engineering:** New solutions are proposed, and evaluated, from changes of a current solution;
- **Empirical:** A model is proposed and evaluated through empirical studies;
- **Analytical:** A formal theory is proposed and compared with empirical observations.

Here, we will address the empirical method. An empirical study can be conduct through:

- **Survey:** The goal is to obtain descriptive and explanatory conclusions [62] from a sample. One sample in a representative part of a population. The data used in the analysis are gathered, usually, through interviews or questionnaires. It is not possible to manipulate variables;
- **Case Study:** Data is collected for a specific purpose. Normally, a case study is used for monitoring projects or activities. From the obtained results, the statistical analysis can be applied;
- **Experiment:** It is a rigorous, formal and controlled investigation. Normally, it is executed in a laboratory environment. It is possible to manipulate variables.

The next subsection shows in details the flow of an experiment. We show elements from a process that defines each step required to perform the experimental study.

2.8.2 Experiment

In this work, we use a process for experimental studies in software engineering defined by Wohlin et al. [62]. This process is composed of the following activities: definition, planning, operation, analysis and interpretation, presentation and package. Each one of these activities are detailed below.

Definition

In this phase, the experiment is defined in terms of a problem, an objective and a goals. It is required to specify a general hypothesis relating the goal of the experiment, with the problem being addressed. In order to define the goal, the key questions proposed by Wohlin *et al.* should be answered[62]:

1. **What is studied?** Object of study: this is the entity that is studied in the experiment;
2. **What is the intention?** Purpose: intention of the experiment;
3. **Which effect is studied?** Quality focus: primary effect under study;

4. **Whose view?** Perspective: viewpoint from which the results are interpreted;

5. **Where is the study is conducted?** Context: environment which the experiment is run.

From these answers, a goal definition template is filled. This template helps organizing the main elements of the experiment, and provides a general overview of the goal and purpose of the experiment. The template is structured as follows:

Analyze *object of study*
for the purpose of *purpose*
with relation to *quality focus*
from the point of view of the *perspective*
in the context of *context*.

The environment defines the personnel involved in the experiment (subjects) and the software artifacts used in the experiment (objects). It is necessary to define the quantity, priority, know-how for each subject and quantity, size, complexity and application domain for the objects.

Planning

Once the experiment is defined, the experiment design is specified. In order to properly plan the experimental study, it is required to specify several elements, such as [62]: context selection, variable, hypothesis, design, instrumentation and threats.

Context Selection

Aiming to have more general and real results, it is necessary that the experiment is executed by professional staff in large and real software projects [62]. However, this scenario is costly. In order to reduce the costs, the project can be run off-line, being performed by students and using toys (e.g. simple models, or software application) in a specific context. Thus, the context of the experiment can be classified in four dimensions [62]:

- On-line vs. Off-line
- Student vs. Professional
- Toy vs. Real Problems

- Specific vs. General

Variables Selection

One of the main elements of the experiments are the variables. These variables comprise the elements that are modified, observed, analyzed and executed during the experimental study. The variables that will compose the experiment are defined as following:

- **Dependent:** Variables that will be observed in the experiment;
- **Independent:** Variables that will be controlled in the experiment.

Hypothesis Formulation

The experiment definition is formalized into hypotheses, that will be tested during the analysis of the experiment. The hypotheses testing is the basis for the statistical analysis of an experiment. The experiment definition is formalized as following:

- A null hypothesis, H_0: This is the hypothesis that the experimenter wishes to reject under a specific significance level;
- An alternative hypothesis, H_1: This is the hypothesis that the experimenter wishes to accept.

Selection of Subjects

The subjects of an experiment are the people involved in it. This is a very important step, since depending on the selection of the subjects, the experiment can be generalized. The larger the sample (of subjects), the lower the error becomes when generalizing the results.

Experiment Design

The Experiment design is defined from the characteristics of the experiment, such as: amount of object, subjects, factors and levels. [62; 40]. The design types are suitable for experiments with:

- **One factor with two treatments:** To compare two treatments;
- **One factor with more than two treatments:** The comparisons between more than two treatments;

- **Two factors with two treatments:** It is necessary to compare the treatments in each factor with the others (2*2 factorial design);

- **More than two factors with more than two (k) treatments:** It is necessary to compare the treatments in each one of the factors with those from the others (2^k factorial design);

From the Experiment design, the statistical resources and the number of replication are defined. The number of necessary replications (n) can be calculated using the following formula [40]:

$$n = \left(\frac{100 \cdot Z \cdot s}{r \cdot \overline{x}}\right)^2 \tag{2.1}$$

Where Z, for a 95% confidence level, is 1.96 (a standard value from the normal distribution table); s is the standard deviation from the sample; r is the desired accuracy; and $\overline{x}$ is the mean of the sample.

Instrumentation

The instrumentation comprises the elements used to automate and execute the experimental study. These elements are called instruments. During this step, three types of instruments are specified [62]:

- **Objects:** The artifacts used to execute the experiment (e.g., specification models, implementation, among others);

- **Guidelines:** The guidelines are required to properly guide the subjects in the experiment;

- **Measurements:** The method in which the data will be collected.

Threats to the Validity

Usually, threats to validity are identified during the planning phase. This is an important step because if the data are not valid, the obtained conclusions of the experiment can not be trusted. The sample needs to have an adequate validity for the population, therefore, any threat to validity need to be considered.

There are different types of validity, each one related to a specific aspect (theory, implementation, observation, among others) of the experiment. The validity can be classified in [24]:

- **Internal:** Related to the relationship between the treatments;
- **External:** Related to the ability to generalize the results;
- **Construct:** Related to the experiment setting;
- **Conclusion:** Related to draw correct conclusion about an experiment.

Each type of validity must be addressed by the experimenter, and it is necessary to identify each elements that threats the validity of the experiment. A validity threat that is not properly handled by the experiment also threats the experimental study itself. Therefore, specifying a proper validity evaluation method is one of the main elements to evaluate and validate the experimental study.

Operation

In this phase, the experiment is executed, and the measurements are collected. This is divided:

- **Preparation:** To prepare the subjects/material to collected data;
- **Execution:** The execution of the experiment is performed;
- **Data Validation:** To make sure the collected data are valid.

Analysis and Interpretation

The collected measurements are analyzed by using descriptive statistic. The interpretation is done by determining, from the analysis, if the null or alternative hypothesis are accepted or rejected. The statistical resources must be properly use, in order to avoid validity threat concerning the conclusion of the results. Therefore, it is necessary to analyze each data, and each sample obtained during the execution before using a specific statistical test (specially the parametric ones, such as Analysis of Variance, and t Test).

Presentation and Package

After analyzing the results of the experiment, the conclusions and artifacts of the experiment must be organized and be presented available, so they can be properly presented to other researches. Therefore, during this step, the entire process is organized in reports, and the data, statistical resources, should be organized in graphics and other visual resources, to ease the understanding of the performed experiment.

2.9 Statistical Analysis

In this section, some concepts about descriptive statistic, graphical visualization and hypothesis testing are presented. These resources are presented in order to provide a better understanding of the analysis performed in this work.

2.9.1 Descriptive Statistic

Descriptive Statistics is used to describe and show - graphically - characteristics of the data set. The goal is to know the data distribution (to identify abnormal data points). Usually, this is done before performing the hypothesis testing. These statistic can be measures of a central tendency, or dispersion.

The measures of central tendency provide an overview to estimate an stochastic variable. There are three measurements often used to indicate the central tendency of a data set(x) [62]:

- **Mean $\overline{x}$:** It is the sum of the values divided by the number of values;
- **Median:** It is the numeric value separating the higher half of a sample (considering the ordered data set);
- **Mode:** It is the value that occurs most frequently in a data set.

In turn, the measures of dispersion show how much variation there is from the mean. There are two measurements often used to indicate the dispersion of a data set [62]: **Variance** and **Standard Deviation** (s). A low value of variance or standard deviation indicates that

the data points tend to be too close to the mean, whereas a high value indicates that the data is spread out. The difference between **Variance** and **Standard Deviation** is that the latter is expressed in the same $unit$ as the data, whereas the variance is expressed in $(unit)^2$.

2.9.2 Graphical Visualization

By representing some measures graphically, we are, usually, able to draw conclusion about the data. A box plot shows the dispersion of a sample and the central value. An example of a box plot is illustrated is presented in Figure 2.6. The picture shows the first and third quartiles (respectively, the upper and lower edges of the box), and the mean value that is represented by the central line in each box. The whiskers extending from the quartiles represent the farthest observation lying within 1.5 times the interquartile range. The outliers (unfilled dots) represent the individual values beyond the whiskers.

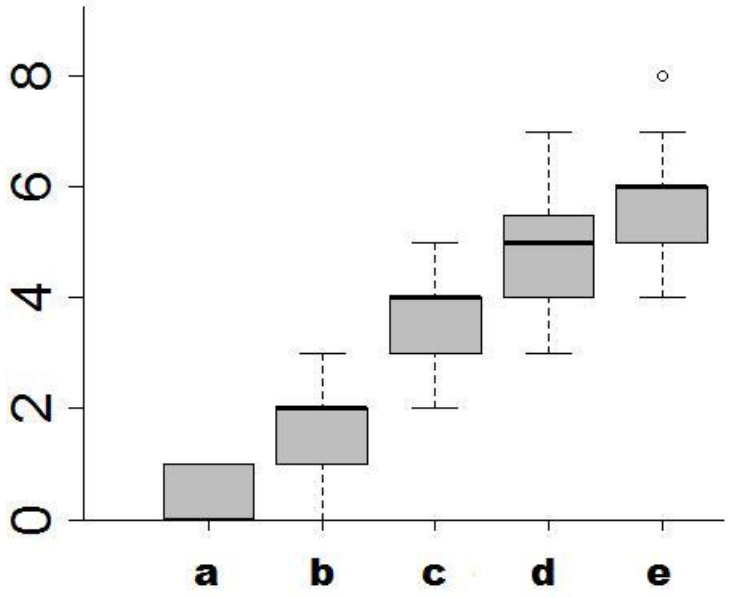


Figure 2.6: Sample of Box Plot

A confidence interval (CI) provides an estimated range of values which is likely to include an unknown population parameter. The estimated interval is calculated from a given set of sample data with a chosen confidence level. Thus, for different set of data different CI are calculated and plotted (see Figure 2.7).

We are able to use the CI from two or more samples to determine if these samples come

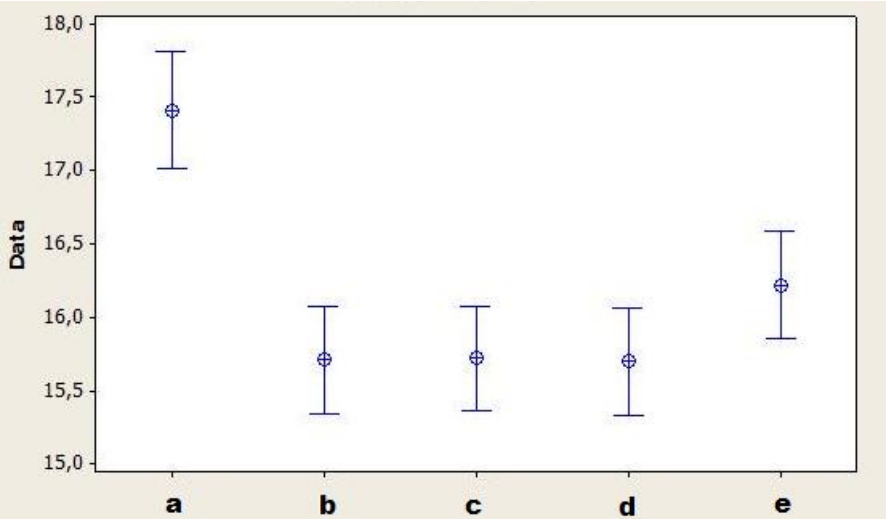


Figure 2.7: Confidence Intervals - a, b, c, d and e

from the same population. If there is no overlap among the CIs, we can conclude that the population are different and, if a hypothesis tests is performed, we are able to reject the null hypothesis (see next subsection), according to the specified confidence level.

Observing the intervals in Figure 2.7, we can state that “a” is different from “b”, “c”, “d” and “e”. However, we can not state anything about “b”, “c”, “d” and “e”, therefore further statistical investigation needs to be done.

2.9.3 Hypothesis Testing

The goal of Hypothesis Testing is to check if it is possible to reject a null hypothesis, H_0, based on a sample from some statistical distribution. Since from the graphical visualization we are not able to reject the null hypothesis, then further statistical investigation needs to be done (applying more statistical tests), aiming to obtain more significant conclusions.

First, the data distribution of the sample is checked by applying tests that investigates distributions, such as Anderson-Darling or Kolmogorov-Smirnov [40].Depending on the data distribution, the Hypothesis Testing can be classified as [62]:

- **Parametric:** The data set presents a known distribution (e.g., normal distribution). Thus, mean and standard deviation of the sample is used to calculate the results;
- **Non-parametric:** The data set does not present a known distribution.

The choice of an adequate test is done by observing the data distribution and the experimental design. Different statistical tests are presented in Table 2.3.

Table 2.3: Statistical tests for different Experimental Designs and data distribution

.

Experimental Design	**Parametric**	**Non-parametric**
One factor with two treatments	$t-test$	Mann-Whitney
One factor with two treatments (paired comparison)	Paired $t-test$	Wilcoxon
One factor with more than two treatments	ANOVA	Kruskal-Wallis
More than one factor	ANOVA	

For interpreting the statistical tests results, it is necessary to observe the resulting $p-value$ of the applied test. This value is compared to the significance level (α) in order to decide if it is possible reject the null hypothesis with the specified confidence level.

2.10 Concluding Remarks

In this chapter some important concepts were presented. According to these concepts, our testing method is functional, comprising MBT approaches that use LTS as a model are considered. The next chapters present the proposed strategies and their respective evaluation and analysis using case studies and experiments. The main variable analyzed in our case studies and experiments is transition coverage, since this variable provides an adequate overview of functionality coverage. The statistical analysis is performed through hypothesis testing.

Chapter 3

Similarity

This chapter presents the redundancy problem (Section 3.2). To better illustrate the problem, we present a simple part of model from a real application (Phonebook). After this, our similarity function (responsible for calculating the distance between two test cases) will be presented in Section 3.2, followed by the similarity matrix, in Section 3.3. Finally, in Section 3.4 are presented some concluding remarks.

3.1 Redundancy

Model-based testing is an approach that has become popular [49], however the test suites generated from MBT approaches, usually, contain a considerable degree of redundancy among test cases. Our redundancy concept considers that two test cases are redundant if they cover the same set of functionalities and present the same fault capability. Therefore, one of them can be discarded without significantly impacting the coverage and fault detection. Thus, a test case is considered redundant, if it can be discarded of the test suite without significantly affecting the fault detection and coverage of functionalities.

Additionally, if there are not redundant test cases and it is still necessary to eliminate some test cases to meet the constraints (money and time), the degree of redundancy among test cases can be observed. The higher is the degree of redundancy among test cases, probably the similar the coverage of functionalities and fault detection capability are.

To better understand, observe the LTS model presented in Figure 3.1. This LTS presents part of the behavior of a real phonebook application. This part is about “adding” a new

contact and the LTS illustrate the different flows of execution that can be considered. 3.1.

Figure 3.1: LTS Behaviour Model - Phonebook

By following this sequence (using *Depth First Search* algorithm), we obtain 6 test cases. The Table 3.1 shows these test cases generated from LTS model presented in Figure 3.1 (for the sake of simplicity, for each test case, we show the sequence of states that are covered, meaning that the correspond action/response between two states have been executed/produced).

From the main flow, there are alternative flows that characterize the behavior of the feature. Therefore, the test cases will differ mostly by a step of input and output. In this sample, the main flow is represented by the path $0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$. This path represents the addition of a contact.

Table 3.1: Test Cases generated from LTS model presented in Figure 3.1 and their respective lengths

Test Case Id	Test Case	Length
TC1	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	15
TC2	0 1 2 3 4 5 6 7 8 9 10 11 20 21	13
TC3	0 1 2 3 4 5 6 7 8 9 22 23 24 11 12 13 14 15	17
TC4	0 1 2 3 4 5 6 7 8 9 22 23 24 11 20 21	15
TC5	0 1 2 3 4 5 6 7 8 18 19	10
TC6	0 1 2 3 16 17	5

In some points of the main flow, the user of the application can chose to proceed to an alternative flow. For example, when the contact list is displayed (state 8), the user of the application can choose to "Press back" (alternative flow) and goes back to the previous screen (8 18 19) or choose to "Select a new contact option (state 9).

Observe that all test cases present the same initial transitions, by following the sequence *0 1 2 3*. Then, there is a certain degree of redundancy among all test cases, since 3 transitions of both sequences are the same. In this case, the redundancy is one step composed by the three elements (a user action, a condition and the system response).

We are able to observe that among TC1, TC2, TC3, TC4 and TC5, the degree of redundancy is higher (*0 1 2 3 4 5 6 7 8*), see that those test cases have 8 identical transitions. More specifically, among TC1, TC2, TC3 and TC4, the number of identical transitions grows to 9 (*0 1 2 3 4 5 6 7 8 9*). Between TC1 and TC2 there are 11 identical transitions (*0 1 2 3 4 5 6 7 8 9 10 11*), and between TC3 and TC4 there are 14 (*0 1 2 3 4 5 6 7 8 9 22 23 24 11 13*).

To clarify, observe Figure 3.2. Analyzing this figure, we can conclude that the same parts of the system are being executed several times with a little bit of difference. Observe also, that the difference between TC1 and TC2 is only one step, i.e., the test cases are identical until the last step, therefore, TC1 and TC2 are so similar that if we remove TC2 from the test suite, we will loose only one step. In the next section, we will present our proposal of a Similarity Function to calculate the similarity degree between a pair of test cases.

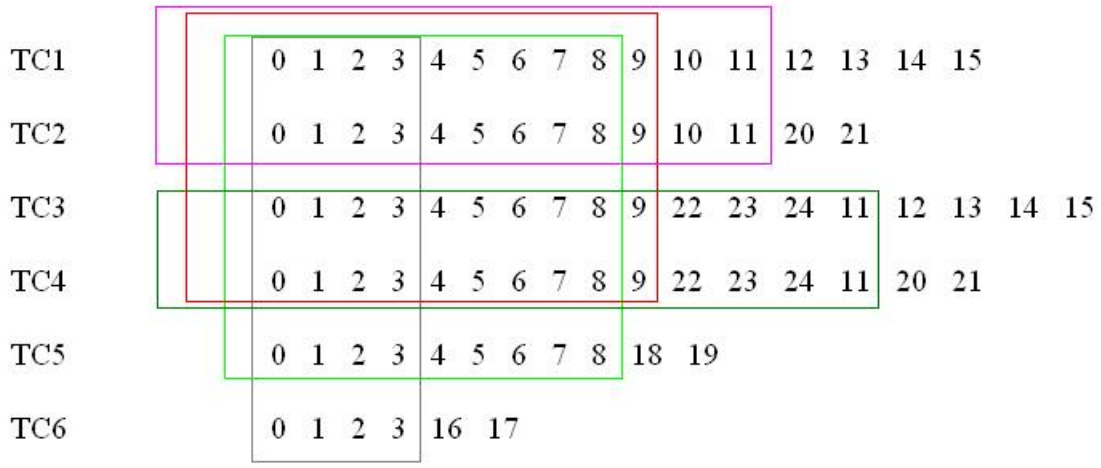


Figure 3.2: Test Cases - Redundancy

3.2 Similarity Function

By observing this real example, in order to measure the similarity degree between two test cases (two paths in an LTS), we need to count the number of identical transitions between them. Two transitions are identical (*it*) if they have exactly the same source and target states (q) and the same label (α). More formally:

$$\forall q \xrightarrow{\alpha} q', q'' \xrightarrow{\alpha'} q''' \in T \cdot it(q \xrightarrow{\alpha} q', q'' \xrightarrow{\alpha'} q''') \Longleftrightarrow q = q'' \wedge q' = q''' \wedge \alpha = \alpha'$$

Therefore, in order to obtain the similarity degree of the test cases, we need to calculate the number of identical transitions between each pair of test cases. Considering the example presented before, the number of identical transitions for each pair of test cases can be seen in Table :

The number of identical transitions (nit) that is used to calculate the similarity between two test cases discloses how much a test case is similar to another one, according to the function. Since, it is necessary to calculate the redundancy of one test case with the other ones, this number (nit) is then divided by the average between the paths length in order to balance the similarities. With this, we avoid representing a low similarity between two small test cases due to the length of the test cases and a high similarity between two very big test

Table 3.2: Pair of Test Case and Number of Identical Transitions

Pair of Test Case	nit
TC1/TC2	11
TC1/TC3	13
TC1/TC4	9
TC1/TC5	8
TC1/TC6	3
TC2/TC3	9
TC2/TC4	11
TC2/TC5	8
TC2/TC6	3
TC3/TC4	13
TC3/TC5	8
TC3/TC6	3
TC4/TC5	8
TC4/TC6	3
TC5/TC6	3

cases that are not so similar.

$$SimilarityFunction(i,j) = \frac{nit(i,j)}{avg(|i|,|j|)} \tag{3.1}$$

Next, we show the use of the similarity function for the example presented above. For example, we calculate the similarity degree between TC1 and TC2 presented in Table 3.1:

1. **Number of identical transitions (**$nit(TC1, TC2)$**):** 11;
2. **Average between Paths' Length (**$avg(|i|, |j|)$**):** 14;
3. **SimilarityFunction(TC1,TC2):** 11 / 14 = 0.78.

3.3 Similarity Matrix

Note that, to disclose the similarity among all test cases, it is necessary to apply the similarity function for each pair of test cases. Thus, we can represent similarity in a matrix, named as

Similarity Matrix, that is defined as follows:

- $n \times n$ **(square matrix)**, where n is the number of paths and each n represents one path, that is called a test case;
- **Each element of the matrix** a_{ij} is defined by computing the similarity between test cases i and j. This function is calculated by observing the number of identical transitions ($nit(i, j)$), i.e., whether states "from" and "to", and labeled transition are the same (see definition in Section 2), and the average between paths length ($avg(|i|, |j|)$).

We are able to observe that the similarity matrix is symmetric, since $a_{ij} = a_{ji}$. As a result, $a_{ij} = SimilarityFunction(i, j)$ where $i = j$, is not considered, since it is not to our interest to calculate the similarity of a test case in relation to itself. For the example presented in Figure 3.1, we obtain the Matrix 3.2. The computational complexity to build the matrix is *O*(n^2), where n is the number of test cases in the test suite.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0.78 & 0.81 & 0.60 & 0.69 & 0.30 \\ TC2 & & & 0.60 & 0.78 & 0.69 & 0.33 \\ TC3 & & & & 0.81 & 0.59 & 0.27 \\ TC4 & & & & & 0.64 & 0.30 \\ TC5 & & & & & & 0.40 \\ TC6 & & & & & & \end{pmatrix} \quad (3.2)$$

The matrix provides an overview of the similarity degree of each pair. The next step is to observe the values in the matrix, in order to draw conclusions concerning the redundancy. Observing the Matrix 3.2, we can conclude that:

- **The highest value (0.81):** This means that the test cases TC1/TC3 and TC3/TC4 are the most similar ones (they present more redundant parts). Observe that both the pairs TC1/TC3 and TC3/TC4 have 13 identical transitions (see Table 3.2).
- **The lowest value (0.30):** This means that the test cases TC1/TC6 and TC4/TC6 are the most different ones in the test suite. Note that both the pairs TC1/TC6 and TC4/TC6 have only 3 identical transitions (see Table 3.2).

3.4 Concluding Remarks

Here, a way of measuring redundancy among test cases of one test suite was presented. By observing the Similarity Matrix, some conclusions can be drawn. For the pair i and j, if the value is:

- **Zero (0):** This means that there is no similarity between the test cases i and j;
- **One (1):** This means that the test cases i and j are equal. When two test cases are equal, it is necessary to execute only one of them;
- **The highest value:** This means that the test cases i and j are the most similar ones of the test suite, i.e., the difference between the test cases is very small;
- **The lowest value:** This means that the test cases i and j are the least similar ones of the test suite, i.e., the difference between the test cases is very big.

In the next Chapters (from Chapter 4 to 6) we present three strategies that use the Similarity matrix presented here. They are:

- **Similarity Strategy (Chapter 4):** The goal is to select the most different test cases from a test suite to be executed (the number of test cases is defined by the test manager). In this case, we consider that if two test cases are very similar (the highest value of the matrix), one of them can be discarded (aiming to meet the resources constraints while having the adequate coverage of functionalities);
- **Weighted-Similarity Approach (Chapter 5):** The goal is to select the most different and important test cases from a test suite to be executed. The importance of each test case and the number that will be executed are defined by the test manager. In this case, we need to calculate a weighed-similarity matrix;
- **Dissimilarity Strategy (Chapter 6):** The goal is to reduce a test suite according to a test requirement, in this case the transition coverage is considered. Since the intention is to cover all transitions faster, the best thing is to find the lowest value of the similarity matrix (that means that the most different test cases) and place both in the reduced set.

Chapter 4

Similarity-based Selection

This Chapter presents our proposal for test case selection based on the Similarity, concept introduced in Chapter 3. This strategy considers the resource constraints (only a certain number of test cases can be executed) and thus, select the most different test cases aiming to have a better functionalities and fault coverage.

Our strategy for selection, and the algorithm, are presented in Section 4.1; a toy example used to illustrate the strategy is presented in Section 4.2; Section 4.3 contains a case study with a real application executed to compare our strategy and random selection, analyzing the transition and fault coverage; and, at last, Section 4.4 presents an experiment that uses a LTS Generator (more details in Appendix B) to generate different LTSs that are the input of the experiment.

4.1 Definition

The idea is to keep in the test suite the least similar test cases according to a goal that is defined in terms of the intended size of the test suite. The least similar test cases provide the chance of having a better coverage of both requirements and faults, once that it covers the most different transitions.

This strategy uses the similarity function to build the similarity matrix (as shown in Chapter 3). The inputs are:

- **Percentage:** The desired percentage of test cases, defined, for example, according to the resources constraints;

- **Test Suite:** The set of test cases;
- **Similarity Matrix:** The matrix that contains the information regarding the similarity among all test cases of the test suite.

The Algorithm 1 presents the steps of this strategy. The first step is to calculate the desired number of test cases (line 1) according to the percentage, i.e., the number representing the total of test cases that have to be selected. Since the idea is to keep in the Similarity Matrix, the most different test cases, the highest value of the matrix is found. The two test cases correspondent to that value are analyzed and one of them is removed from the matrix (lines 2 - 14). This procedure is repeated until the number of test cases in the matrix is equal to the desired value (line 2).

At this point, the maximum values of the matrix are found. When a tie among maximum values (more than one maximum value) is found, in the similarity matrix, the idea is to randomly choose one of them (lines 3 - 4). From the maximum value we are able to discover the correspondent test cases (lines 5 - 6) from the most similar pair.

Now, the size of the test cases are compared, and the idea is to keep in the matrix the longest test case (lines 7 - 10), i.e. the test cases with more transitions, since it can represent the highest functionality coverage. If the size between the two test cases is the same, a random choice is applied (lines 12 - 13).

Regarding the complexity analysis of Algorithm 1 we are able to observe a repeating structure (`while` command in line 2) where, within each iteration, the method `getAllMaxValue` ($O(n^2)$) is used to search the matrix for the highest similarity values. Therefore Algorithm 1 has a complexity of $O(n^3)$, where n is the number of test cases in the test suite.

It is possible to think that selecting long test cases would provide a test suite that requires a lot of time to execute. However, to our knowledge, no empirical evidence that relates the number of transitions of a test cases and the time required to execute the test suite, has been performed.

```
input  : percentage, testSuite, similarityMatrix
output: selectedTestCases
1  numberOfRequiredTestCases = calculateNumberOfDesiredTestCases(percentage, testSuite);
2  while (selectedTestCases.size() < numberOfRequiredTestCases) do
3      maxValues = getAllMaxValue(similarityMatrix);
4      chosenPair = pairs.shuffle.get(0);
5      testCase1 = chosenPair.getTestCase1();
6      testCase2 = chosenPair.getTestCase2();
7      if (testCase1.size() > testCase2.size()) then
8          similarityMatrix.remove(testCase2);
9      else if (testCase1.size() < testCase2.size()) then
10         similarityMatrix.remove(testCase1);
11         else
12             chosenTestCase = randomChoice(testCase1,testCase2);
13             similarityMatrix.remove(chosenTestCase);
14     selectedTestCases = similarityMatrix.getTestCases();
```

Algorithm 1: Similarity based Selection - Algorithm

4.2 Example - Similarity Selection

In order to illustrate the strategy, an example is presented below. An LTS model is presented in Figure 4.1. From this LTS model, 6 test cases are obtained. Both the test cases, and their respective sizes, can be seen in Table 4.1. In turn, the similarity matrix is presented in Matrix 4.1.

Table 4.1: Test Cases and Size of test cases

TC id	Path	Test Size
1	a	1
2	b c e	3
3	b d f	3
4	b d g	3
5	b d g d f	5
6	b d g c e	6

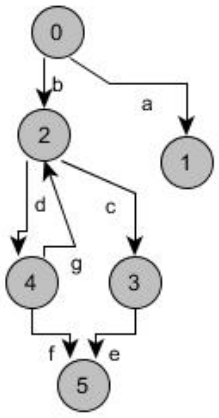


Figure 4.1: Example - LTS model

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 & 0 \\ TC2 & & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & & 0.75 & 0.75 \\ TC5 & & & & & & 0.6 \\ TC6 & & & & & & \end{pmatrix} \tag{4.1}$$

Considering that, the desired percentage is 50%, the desired number of test cases is 3 ($50\% \cdot 6$). Then, the first step is to find the highest value in the matrix. In this matrix, there is a tie among TC2/TC6, TC3/TC5, TC4/TC5 and TC4/TC6. Therefore, a random choice is done. Considering that the pair TC2/TC6 is chosen, TC2 is removed from the matrix, since $|TC6| > |TC2|$. The line and column of the removed test cases are also removed from the matrix, and the new matrix can be seen in Matrix 4.2.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 \\ TC3 & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & 0.75 & 0.75 \\ TC5 & & & & & 0.6 \\ TC6 & & & & & \end{pmatrix} \tag{4.2}$$

Following the algorithm of this strategy, again, it is necessary to find the highest value of similarity. As we can see, there is a tie among the highest values: TC3/TC5, TC4/TC5 and TC4/TC6. Thus, a random choice is performed again. Considering that the pair TC3/TC5 is chosen, since $|TC5| > |TC3|$, TC3 is removed from the matrix.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 \\ TC4 & & & 0.75 & 0.75 \\ TC5 & & & & 0.6 \\ TC6 & & & & \end{pmatrix} \tag{4.3}$$

So far, 2 test cases were excluded from the matrix (TC2 and TC3), thus we need to exclude one more. Searching for the highest value, we identify another tie between the TC4/TC5 and TC4/TC6. Randomly, TC4/TC5 is chosen and, since $|TC5| > |TC4|$, TC4 is excluded. Finally, the similarity matrix has 3 test cases (see Matrix 4.4) and thus, our set of selected test cases is composed by TC1, TC5 and TC6.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC5 & TC6 \\ TC1 & & 0 & 0 \\ TC5 & & & 0.6 \\ TC6 & & & \end{pmatrix} \tag{4.4}$$

4.3 Case Study

In order to evaluate the use of the similarity strategy, we conducted a case study. The goal of this case study is to compare Similarity and Random selection by considering fault and transition coverage. The Similarity and random strategies were applied having the percentage of the test suite (path coverage) goals ranging from 5% to 95% (increased by 5).

4.3.1 Application

The application used for this is a desktop tool named TaRGeT. This tool automatically generates test cases [48]. LTS-BT tool [17] was used for executing this case study. The input is

a use case template [48], written by Motorola experts. All test cases, generated for this case study, were manually executed by Motorola employees. The collected metrics are:

- **Transitions Coverage:** We are able to measure the coverage of transitions of the model by counting the **number of excluded transitions**. The total number of transitions that are excluded by considering all of the discarded test cases of a given test suite represents the idea of measuring whether the strategies keep a reasonable coverage of functionalities even though discarding some test cases.
- **Faults coverage:** The total number of faults that are uncovered by the test suite during test execution. For this, we considered real faults. The idea is to measure whether the strategies preserve the fault detection capability of the original test suite.

These metrics are widely used in works of the literature and they are able to provide an overview of how much the strategy is adequate to select test cases considering the specified budget constraint. We considered these metrics adequate since our interest is to cover more functionalities (this is measured through transition coverage) and faults (measured through fault coverage).

4.3.2 Case Study - Preparation

Since, Similarity and of course, Random selection present a random choice in their algorithms, then each strategy was executed one hundred times (for each percentage) and the metrics were collected.

4.3.3 Results of the Case Study

This subsection shows the results of the case study. The TaRGeT application, used in this case study has 168 transitions in its LTS model. Also, LTS-BT tool was able to generated a total of 84 test cases, and these cases were manually executed, where a total amount of 13 failures were revealed. Each failure, in this case study, corresponds to a fault, in the application (i.e., 13 failures correspond to 13 faults).

Number of Excluded Transitions

The results can be seen in Figure 4.2. In this graph, the x-axis (or abscissa) represents the intended test cases percentage and in the y-axis (or ordinate) the average of excluded transitions obtained with 100 replications. The most effective strategy regarding this criterion is the one that presents the lower curve. The results show that most of the times, the Similarity strategy discards less transitions.

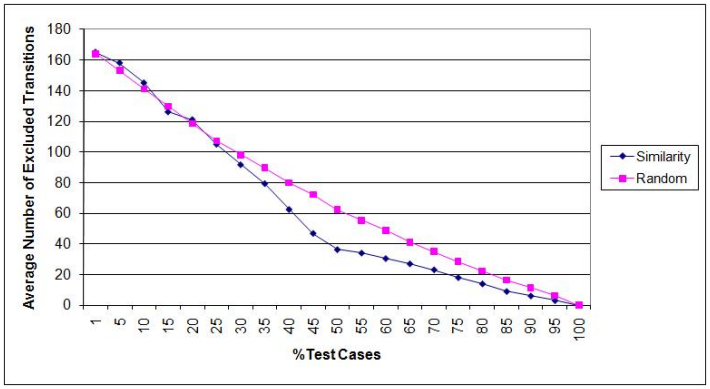


Figure 4.2: Average Number of excluded transitions by running each test selection strategy 100 times for each test selection goal

Below 20% of test cases a use of the random selection is more adequate. However, in the best case the Random selection (5% of test cases) excludes only 3.02% less transitions than Similarity. And the best case is when the percentage of desired test cases is 50%, where the Random strategy excludes 41.83% more transitions than Similarity.

Faults Coverage

For fault coverage, the results are presented in Figure 4.3. We represented in the x-axis the intended test cases percentage and in the y-axis the average of covered faults obtained with 100 replications. The most effective strategy regarding this criterion is the one that presents the highest curve. As can be seen, the Similarity strategy reveals more faults for all percentages and the best case is when the percentage of test case is 55%. In this case, Similarity is able to reveal 41.33% more faults than Random.

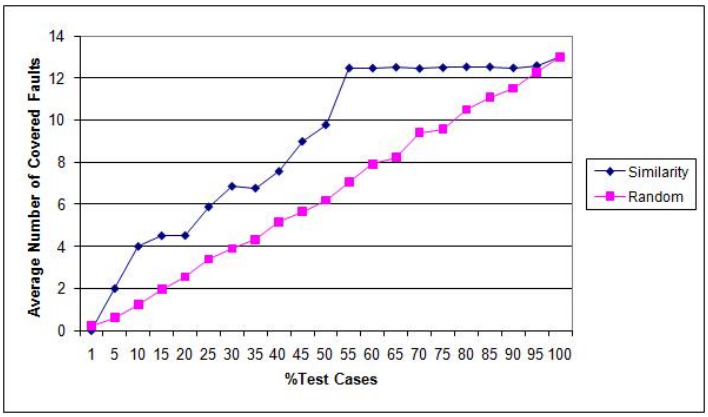


Figure 4.3: Average Number of covered faults by running each test selection strategy 100 times for each test selection goal

4.3.4 Concluding Remarks - Case Study

The case study performed in the evaluation suggests that the similarity approach can be more effective than random choice, usually by considering path coverage (test suite percentage) of more than 20%. The main threat to validity is the use of only one model, whereas the structural elements of the LTS may affect the performance of the analyzed strategies. Aside from that, it would be more appropriate to compare Similarity with other selection strategies, however our main objective with this case study was to obtain an overall perspective concerning the Similarity strategy, and not a comparative overview.

More case studies have been executed and can be seen in Appendix A. One of the executed case studies is the same presented here, however with another version of the use case document and a different level of abstraction, were used. Then, the metrics are different than the ones presented here.

4.4 Experiment - Selection

This Section presents the experiment and the obtained results of the execution. The used framework - proposed by Wohlin *et al.* - was presented in Chapter 2. The definition, the planning, the operation and the Analysis and Interpretation results of the experiment are showed in the next subsections.

Our general hypothesis is that Similarity presents the best performance in relation to the

number of excluded transitions, considering 50% of the test cases. This percentage was considered because the highest difference between Similarity and Random strategy in case study is 50%, as presented before (Section 4.3).

4.4.1 Definition

The first step is to define the experiment. Therefore, the key questions proposed by Wohlin *et al.* were answered:

1. **What is studied?** Selection strategies;
2. **What is the intention?** To investigate;
3. **Which effect is studied?** Number of excluded transitions;
4. **Whose view?** The tester;
5. **Where is the study is conducted?** Model-Based Testing (MBT).

From these answers, the goal definition template is filled. In summary, the goal of this experiment is:

Analyze *selection strategies*
for the purpose of *investigating*
with relation to *number of excluded transitions*
from the point of view of the *tester*
in the context of *MBT*.

For this experiment, the (input) objects are LTS models. Since the strategies execute automatically, there is no need for subjects to be involved in this experiment.

4.4.2 Planning

Once the elements of the experiment are properly defined, the following steps of the study must be planned. Following the chosen framework, the context selection, the variables (dependent and independent), hypothesis, design and instrumentation were defined.

Context Selection

The context of this experiment can be characterized as a "toy vs. real" problem. In this case the objects are LTS models, randomly generated from a configuration. This configuration is characterized by a specific number for the depth of the LTS, the number of loops, branches and joins (these elements are detailed in Appendix B.

Variables Selection

In order to characterize the experiment, the variables must be defined. The variable chosen to observe (dependent variables) and to control (independent variables) are:

- **Dependent:** The Number of Excluded Transitions (NET).
- **Independent:** The test cases percentage; the configuration chosen the depth and the amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 2 levels: Similarity (Sim) and Random.

Hypothesis Formulation

Once the variables are defined, we are able to structure our null and alternative hypothesis. Their definition is formalized as following:

- **A null hypothesis (**H_0**)**: $NET_{Sim} = NET_{Random}$ - The two strategies exclude the same number of transitions, in another words, the strategies present the *same behavior*;
- **An alternative hypothesis, (**H_1**)**: $NET_{Sim} \neq NET_{Random}$ - The two strategies exclude a different number of transitions, i.e., the strategies present *different behavior*.

Experiment Design

As seen before, there is one factor (test case selection strategy) with 2 levels (or treatments). Thus, there is one factor and 2 treatments, where, for each object, the two treatments are applied. The chosen confidence level is 95% (significance level is $\alpha = 0.05$), as suggested by statistical literature [40].

Aiming to define the number of replications, necessary to guarantee statistical significance for the specified level of confidence (95%), 40 replications were performed, collecting the number of excluded transitions. These data are presented in Table 4.2.

Table 4.2: Mean, Standard Deviation and number of necessary replications for each technique.

Technique	Similarity	Random
Mean ($\overline{x}$)	17.9	20.0
Standard Deviation (s)	6.99	4.61
Number of Necessary Replications (n)	235	82

Observing the Table 4.2, we are able to see that 235 and 82 replications for Similarity and Random selection, respectively, provide a statistical significance for the obtained data. Therefore, this experiment design will consider 300 replications for each strategy.

Instrumentation

The next step of the planning is to specify the instruments of the experiment. In this step, there are three types of instruments [62]:

- **Objects:** The objects are LTS models randomly generated from a configuration (depth, number of loops, forks and joins).
- **Guidelines:** This experiment uses no guidelines, since the strategies do not require subjects to configure them.
- **Measurements:** The NET will be collected for each treatment. The tool LTS-BT provides support for both executing the experiments and collecting the data.

Validity Evaluation

The objects used in this experiment can be considered the main threat to validity. These objects are automatically generated, and therefore, they can not represent a real behavior. Besides, since they are randomly generated from a specific configuration, both the traceability and controllability of the elements of the model (transitions and states) are reduced.

On the other hand, we are able to obtain an overview of the execution of the strategies in several models, since they are randomly generated. Thus, we avoid being presented with a conclusion that is specific to only one LTS (if we would have used the same LTS in every execution). A proper scenario would be to have several real applications to execute the strategies. However, most real applications and their respective specification are not available to the open public.

4.4.3 Operation

To execute this experiment, it was necessary to implement the two strategies and the LTS generator (see Appendix B). Both the LTS generator and the strategies are implemented in the Java programming language[1].

The objective, in using this LTS generator, is to automatically generate different models. Therefore, a specific configuration for the depth and structure of the LTS is specified and the generator is able to place these structures (loops, forks and joins), in different ways. A deeper LTS provides more option to place the structures (branches, loops and joins).

Through executions of the generator, we were able to observe that an LTS, generated with a smaller depth and several structures, generates a lot of test cases with the same size. This fact does not represent real applications, since the test cases of real applications vary the size (they have different number of flows). Besides, when there are several test cases with the same size, the Similarity strategy applies a random selection between each pair of test cases (most similar), whereas random selection can pick any test case from the test suite. That scenario would not provide a fair comparison of the strategies. Therefore we chose the following configuration:

- **Depth:** 15;
- **Number of loops:** 2;
- **Number of branches:** 3;
- **Number of joins:** 3.

[1] http://www.sun.com/java/

This configurations provides a wide range of possible LTS. The generated LTS begins with 16 states, where the generator can choose 15 states, out of the 16, to place the structures according to the constraints described in Appendix B. Aiming to have different size of test cases, we decide to have a higher depth in relation to the number of structures.

There is only one experimental design (with only one factor - test suite reduction strategy) with a null and an alternative hypothesis, where the intention is to reject the null hypothesis. Each strategy was executed 300 times, using a machine with the following configurations:

- Intel Core 2 quad 2.33 GHz;
- 4GB RAM;
- 1TB for Hard Disk Memory.

4.4.4 Analysis and Interpretation

The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[2]. The results can be seen in Figures 4.4 and 4.5. In this graph, the red dots, should overlap the blue line, in order to indicate that the data fit a normal distribution.

As can be seen, the data do not fit a normal distribution, once the *p-values* are less than 0.05. Then, it is necessary to apply a non-parametric test. Since we have only one factor and two treatments, we can apply a Mann-Whitney testing to check the null hypothesis. The results are presented in Table 4.3.

Table 4.3: Mann-Whitney Test - Sim and Random

Technique	N	Median
Sim	300	14.000
Random	300	19.000
Point estimate for ETA1-ETA2 is -1.000		
95.0 Percent CI for ETA1-ETA2 is (-2.000;-1.001)		
The test is significant at 0.0004		

[2]http://www.minitab.com/

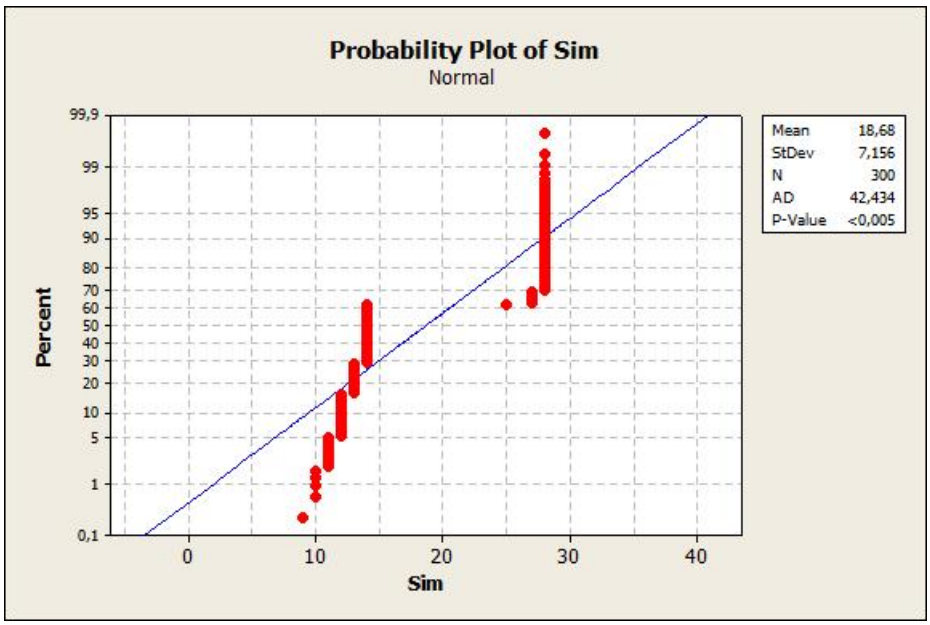


Figure 4.4: Anderson-Darling normality test - Similarity

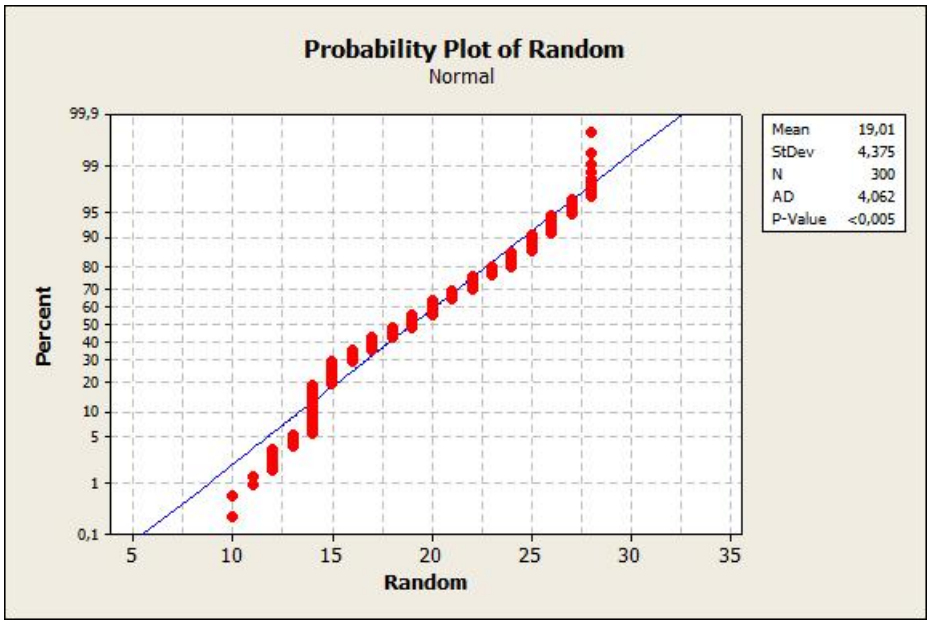


Figure 4.5: Anderson-Darling normality test - Random

Since $p - value = 0.0004$, and this is less than 0.05 (α), H_0 (the null hypothesis) can be rejected. Therefore, the data support the hypothesis that there is a difference between the population medians ($ETA1 - ETA2$). The difference between the two population medians is greater than or equal to -2.000 and less than or equal to -1.001. As we can see, the difference between the population medians of Similarity and Random is negative, therefore, the NET

from Similarity is less than the NET from Random selection.

4.4.5 Concluding Remarks - Experiment

With 95% of confidence level, we are able to reject our null hypothesis. Therefore, our general hypothesis is confirmed, since the behavior of the Similarity - considering 50% of the number of the test cases - is better than the one observed with Random. In another words, by applying the Similarity strategy we obtain better results than by applying Random.

4.5 Concluding Remarks

In this Chapter, we presented one of our strategies for test case selection. This strategy tries to meet the resources constraints and then, keeps in a test suite, only the test cases that will be executed. The focus is to increase functionality and fault coverage as high as possible. A Case study and an experiment were performed, and the results show evidence that:

Case studies:

- **Concerning transition-based coverage:** For the conducted case study, the similarity strategy can be more effective than the random strategy. There are considerable advantages when the desired coverage is higher or equal to 20% of the test case;

- **Concerning fault coverage:** For the conducted case studies, the similarity strategy has also a superior performance. For all case studies (the ones presented in this chapter, and the ones presented in Appendix A, the highest coverage is only achieved by the similarity approach.

Experiment:

Considering a desired coverage of 50% of the number of the test cases, Similarity is better than the Random strategy. In another words, by applying the Similarity strategy we obtain better results than by applying Random.

Note that the obtained results in cases studies and in the experiment are not contradictory. It is important to say that when the model presents test cases with the same length, the behavior of similarity strategy can be the same or worse than the one observed in the Random

strategy. That situation presents itself, in the Similarity strategy, when the test cases to be discarded are chosen by the length, and this length is the same for both test cases.

The drawback of this strategy is that important test cases could be eliminated. For instance, a test case may be focused on either a frequent used functionality, or a very important functionality with respect to the user needs (or on critical path of the application). Since this strategy does not distinguish the importance of test cases, crucial ones may be discarded. Therefore, we integrate the similarity method with a value-based test strategy, called the Weighted-Similarity Approach, that will be presented in next Chapter (Chapter 5).

Chapter 5

Weighted-Similarity Approach (WSA)

This Chapter presents our proposal for test case selection based on Similarity (Chapter 3) and that also considers weights that are assigned to test cases. Our strategy for selection is defined in Section 5.1 and an example to illustrate it is presented in Section 5.2. Section 5.3 presents two case studies that were executed to compare Weighted-Similarity Approach (WSA), Random Selection, Guided Random (Random choice guided by the same transition probabilities applied with the WSA approach) and Similarity by considering fault and transition coverage. Finally, in Section 5.4 the concluding remarks for this chapter are drawn.

5.1 Definition

In Chapter 4, the Similarity strategy was presented. This strategy considers the resource constraints and thus, selects the most different test cases aiming to have a better functionality and fault coverage. The results of both the case study and the experiment showed that Similarity is an efficient strategy, in relation to transition and fault coverage. However, when applying this strategy, important test cases (e.g., test cases that discover faults) could be eliminated, since no information about this "importance" is considered by that strategy.

However, if this information is available, this can be used to get most important and different test cases. For this, we integrate similarity based selection strategy with a value-based test strategy, named Weighted-Similarity Approach (WSA). The main goal of this approach is to exploit the software engineering knowledge and experience, in order to minimize the size of a test suite by keeping in it only the test cases that can be feasibly executed according

to the user behavior and resources available to the testing process. To accomplish this goal, we use the concept of *similarity* and show how it can be used for test cases selection. WSA foresees the test cases selection by prioritizing accordingly with the probabilities set by the test manager. The idea is that when choosing between two similar test cases to be discarded, the one that has a greater probability is kept. In the original strategy (Similarity-based selection), a random choice is performed when they have the same size. Therefore, we aim at testing suites that have the most different test cases and yet, these are also the most important (according to the probability) ones.

Using an LTS model, the user can set the desired path coverage (i.e. amount of desired test cases) and provide the weights to be associated to each possible flow of the LTS. We can consider that each weight indicates the expected frequency of use as a concept of "importance" for a given flow. Taking in consideration that the sum of weights for every flow of a branch is 1 (the total flow - before the branch - is 1, so the sum should be 1).

Given the LTS model with weights assigned, the test cases similarity and *final weight* for each flow is computed. The similarity between two test cases is computed as the number of common steps in the two test cases. The *final weights* of a test case is obtained by multiplying the attributed weight in their branches. As result, a weighted similarity matrix is built with test cases as columns and rows, where each element is defined as the similarity between two test cases divided by the weight of the test case of the respective row. Note that, the most similar test cases must be eliminated and the most important must be kept. For this, we balance the similarity with the weight of each test case.

With the LTS behavior model and probabilities (weights), we can calculate the similarity and test case weights (this is obtained by multiplying the attributed probabilities in their branches), build the weighted-similarity matrix, where each $a_{ij} = WeightedSimilarityFunction(i, j)$. The $WeightedSimilarityFunction(i, j)$ is defined as follows:

$$WeightedSimilarityFunction(i, j) = \frac{SimilarityFunction(i, j)}{W(i)} \tag{5.1}$$

As can be seen, the equation 5.1 uses the equation 3.1 as its numerator. Similarly to the Similarity Selection, we apply this equation to each pair of test cases, in order to obtain the weighted similarity matrix. Therefore, the weighted-similarity matrix is:

- $n \times n$ (square matrix), where n is the number of paths and each n represents one path, that is called a test case;
- Each element of the matrix $a_{ij} = WeightedSimilarityFunction(i, j)$.

The weighted-similarity matrix is not symmetric, since the similarity between i and j ($SimilarityFunction(i, j)$) is balanced by the weight of the i ($W(i)$). The highest value represents the test case that is most similar to the other ones, and least important. Thus, in order to choose a test case to be removed, we search for the highest value in the matrix. This corresponds to a test case that is very similar to other test case, and has a lower weight. If there is a tie, then the smallest test case must be eliminated. If there is a tie again, random choice is applied.

This strategy uses the weighted-similarity function to build the weighted-similarity matrix. The inputs are:

- **Percentage:** The desired percentage of test cases. This percentage is defined according resources constraints;
- **Test Suite:** The set of test cases;
- **Weights:** The weights of the test cases;
- **Weighted-Similarity Matrix:** The matrix that contains the information about similarity and importance among all test cases of the test suite.

The Algorithm 2 presents the steps of this approach. The first step is to calculate the desired number of test cases (line 1) according to the percentage, that number represents the total of test cases that have to be selected. Since the idea is to keep in the Weighted-Similarity matrix, the most different and most important test cases. Then, the highest value of the matrix is found and discarded of the matrix (lines 2 - 13). This procedure is repeated until the number of test cases in *wsMatrix* is equal to the desired value (line 2).

In line 3 of the algorithm, the maximum values are found. When a tie among maximum values is found (more than one maximum value), the idea is to randomly choose the least important (line 4), then the least important is discarded of the matrix (lines 5 - 6). If there is also a tie among the weights (importance) of test cases (lines 7 - 12), the idea is to discard the

smallest test case (lines 8 - 10) to guarantee the highest coverage of functionalities. Finally, if there is a tie between the smallest test cases, then the random choice is applied (lines 11 - 12).

```
   input  : percentage, testSuite, weights, wsMatrix
   output: selectedTestCases
1  numberOfRequiredTestCases = calculateNumberOfDesiredTestCases(percentage, testSuite);
2  while (selectedTestCases.size() < numberOfRequiredTestCases) do
3      maxValues = getAllMaxValue(wsMatrix);
4      choosedTestCases = getLessImportant(maxValues, weights);
5      if (choosedTestCases.size()=1) then
6          wsMatrix.remove(choosedTestCases.get(0));
7      else
8          choosedSmalls=getSmallTestCase();
9          if (choosedSmalls.size()=1) then
10             wsMatrix.remove(choosedSmalls.get(0));
11         else
12             wsMatrix.remove(randomChoice(choosedSmalls));
13     selectedTestCases = wsMatrix.getTestCases();
```

Algorithm 2: WSA - Algorithm

The complexity analysis of Algorithm 2 is similar to the one presented for the Similarity-based Selection algorithm (Algorithm 1 in Section 4.1, Chapter 4). Performing the analysis we are able to obtain a computational complexity $O(n^3)$, where n is the number of test cases in the test suite, for Algorithm 2.

5.2 Example - WSA

In this Section, we will show one simple example to illustrate the proposed strategy. For this example, the use case (main and alternative flows) and the respective LTS model are presented (this approach is not strictly related to a specific model-based approach - use case). Then, the approach is applied.

5.2.1 Example - Description

The use case describes the creation of a new contact in the Contact list (the use case model presented here was presented by Nogueira and his colleagues [48]). In Figure 5.1 the main flow of this use case is presented. This flow represents an user that successfully adds a contact to a phone book.

Main Flow

flow: Description: description: Create a new contact

From Step: fromSteps: START

To Step: toSteps: END

Step Id	User Action	System State	System Response
step id: 1M	action: Start My Phonebook application.	condition: My Phonebook application is installed in the phone.	response: My Phonebook application menu is displayed.
step id: 2M	action: Select the New Contact option.	condition:	response: The New Contact form is displayed.
step id: 3M	action: Type the contact name and the phone number.	condition:	response: The new contact form is filled.
step id: 4M	action: Confirm the contact creation.	condition: There is enough phone memory to insert a new contact.	response: A new contact is created in My Phonebook application.

Figure 5.1: Creating a New Contact - Main Flow

In Figure 5.2 two alternative flows are presented. The first alternative flow describes the scenario where the user is able to cancel the creation of the new contact. This can happen in two cases: the form is opened (Step ID 2M) and is filled (Step ID 3M). The second alternative flow describes the scenario where the new contact is not created because there is not available phone memory, this can happen when the user tries to add the contact (after filling the form in Step ID 3M).

By following the Step IDs, we can obtain the Labeled Transition System (LTS) Behavior model (this can be seen in Figure 5.3). Observing this figure, we can see that, after the user executes the Step 2M, the user can execute the Steps 3M or 1B. The same way for Step 3M,

Figure 5.2: Creating a New Contact - Alternative Flows

Table 5.1: Test Cases generated from LTS model presented in Figure 5.3 and their respective lengths

Test Case Id	Test Case	Length
TC1	1M 2M 3M 4M	4
TC2	1M 2M 3M 1C	4
TC3	1M 2M 3M 1B	4
TC4	1M 2M 1B	3

the user can execute the Steps 4M, 1C or 1B.

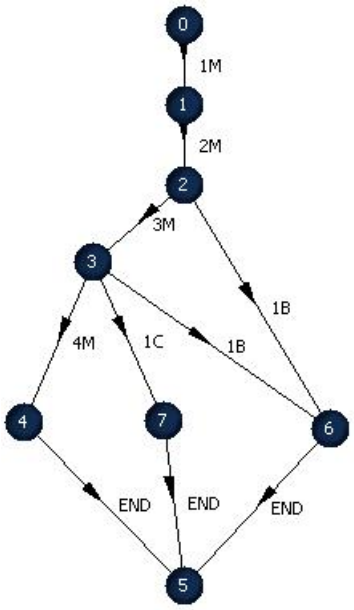


Figure 5.3: Labeled Transition System (LTS) Behavior model

We can use a test case generation algorithm, to traverse the model, where each path in the LTS is a test case. In this case, we used LTS-BT tool, to generate the test cases from the LTS in Figure 5.3. Thus, we obtain 4 test cases, as seen in Table 5.1.

For applying the weighted-similarity approach, it is necessary to define the meaning of "importance". Then, we define weights for each branch that is originated by alternative flows after step 2M and 3M (Figure 5.4). In this case, the weight is related to the expected frequency of execution (by the end user) of that specific step.

For calculating the weighted-similarity matrix, it is necessary to calcu-

BRANCHES		Probabilities
2M	3M	0,7
	1B	0,3
3M	4M	0,6
	1C	0,2
	1B	0,2

Figure 5.4: Probabilities

Table 5.2: Weights of the test cases obtained from LTS Model 5.3 and assigned probabilities 5.4

Test Case Id	Weight
TC1	$0.7 \times 0.6 = 0.42$
TC2	$0.7 \times 0.2 = 0.14$
TC3	$0.7 \times 0.2 = 0.14$
TC4	0.3

late the weighted-similarity function for all a_{ij}. Then, for example, the $WeightedSimilarityFunction(TC1, TC2)$ is given by:

1. **Number of identical transitions** (nit): 3;
2. **Average between Paths' Length** $(avg(|TC1|, |TC2|))$: 4;
3. **SimilarityFunction(TC1,TC2)**: 3 / 4 = 0.75.
4. **WeightedSimilarityFunction(TC1,TC2)**: 0.75 / 0.42 = 1.78.

Calculating every a_{ij}, we are able to obtain the weighted-similarity matrix, and thus, the information regarding the similarity and importance of each test case. The complete matrix for this example is presented in Matrix 5.2.

$$WeightedSimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 \\ TC1 & & 1.78 & 1.78 & 1.36 \\ TC2 & 5.35 & & 5.35 & 4.08 \\ TC3 & 5.35 & 5.35 & & 4.08 \\ TC4 & 1.90 & 1.90 & 1.90 & \end{pmatrix} \tag{5.2}$$

Considering that there are enough resources to execute only 50% of the test cases, we need to discard two test cases. The highest values of the Matrix is between rows TC2 and TC3. Since we need to choose one of them, and the importance and length of both test cases are the same, a random choice is applied. Supposing that TC2 is discarded, the new Matrix is presented in Matrix 5.3.

$$WeightedSimilarityMatrix = \begin{pmatrix} & TC1 & TC3 & TC4 \\ TC1 & & 1.78 & 1.36 \\ TC3 & 5.35 & & 4.08 \\ TC4 & 1.90 & 1.90 & \end{pmatrix} \tag{5.3}$$

Observing the new matrix, the highest value is in the row of TC3. Therefore, TC3 is discarded. The test cases the are kept to execute are TC1 and TC4. Observe that TC2 and TC3 are so similar to TC1, but TC1 is kept because it is more important than the other ones. And TC4 is also kept because it is the most different and has the highest weight.

5.3 Case Study

In order to evaluate the use of WSA, we conducted two case studies. The goal of these case studies is to compare WSA, Random Selection, Guided Random (Random choice guided by the same transition probabilities applied with the WSA approach) and Similarity by considering fault and transition coverage. All strategies were applied having percentage of test suite (path coverage) goals ranging from 10% to 90% (increased by 10).

5.3.1 Applications

Two real applications provided by Motorola Software Engineers have been selected for the case studies. They are briefly described as follows:

- **Application 1:** TaRGeT is a desktop application that supports the model-based testing process, where it automatically generates test cases from use case documents.
- **Application 2:** Direct License Acquisition (DLA) Support is a feature for mobile phone applications that handles acquisition of the WMDRM License (Windows Media

Table 5.3: Number of Test Cases and Faults

	# Test Cases	# Faults
Application 1	84	13
Application 2	28	2

Digital Rights Management) for the Windows Media platform. This License provides secure delivery of audio and/or video content.

For each one of these applications, Motorola Software Engineers elaborated the use case documents [48]. From these documents, LTS-BT generated the test suites that have the metrics showed in Table 5.3. The test cases were manually executed and the captured failures were associated with faults that can be detected by the suite.

5.3.2 Metrics

Once the applications used to apply the strategies have been explained, our next step, is to present the metrics considered in our analysis. Aiming to do a comparison among WSA, Random Selection, Guided Random, Similarity, and a manual selection, the same metrics used in the case study performed for the Similarity-based Selection (Section 4.3, Chapter 4) will be observed. These metrics are the **Transitions Coverage**, and the **Fault Coverage**.

5.3.3 Case Study - Preparation

Since, all strategies, besides the manual selection, present a random choice in their algorithms, then each one of them was executed one hundred times and the metrics were collected. The results are presented in the next subsection. For each case study, the following activities were performed:

- Reading and understanding the related documents, use cases (Target templates), and LTS model (generated from use cases);
- Assigning probabilities to branches in the LTS model;

- Manually selecting test cases to produce a minimized test suite from 10% to 90% of selection goal, starting from 10% (performed by the test designer only).

Assigning Probabilities - Subjects

Case studies have been conducted by one WSA designer and by one test designer. In both cases, the goal is to reduce the test suite, maximizing the ability for fault detection and also providing an adequate coverage of functionalities.

- **The test designer:**
  - Has experience on test selection;
  - Has experience with all the approaches considered in this study;
  - Has no previous knowledge of the defects presented in the case studies.
- **The WSA designer:**
  - Proposed the WSA approach;
  - Has experience with all the approaches considered in this study;
  - Knows all defects presented in the case studies.

Our assumption is that by using experience on test selection and also knowledge of the application domain, the test designer can assign probabilities to guide the selection of the most effective test cases. By knowing the defects, the WSA designer can maximize the results obtained by WSA. Therefore, we should be able to assess the limits of the approach in the best case. From this, we may identify strategies for assigning probabilities as well as uncover limitation of practical use.

Assigning Probabilities - How to specify the values

Each subject (one test designer and one WSA designer) received the use case document and the respective LTS of each application. For each branch of the LTS, the subject assigned the probabilities. The assigned value considers the probability to discover faults. The subjects were instructed to assign weights (probabilities) between 0 and 1, whereas the closer to zero, the lower the chance of reveal faults.

5.3.4 Results of the Case Study

This subsection presents the results of the case study. As a reminder, the variables considered in the study are the size of the reduced test suite (RTS) and the fault detection.

Faults Coverage

Application 1

The results obtained - for Application 1 - by applying WSA, Random Selection, Guided Random, Similarity and Manual selection (done by the test designer), considering the probabilities assigned by WSA and the test designer can be seen, respectively, in the Figures 5.5 and 5.6.

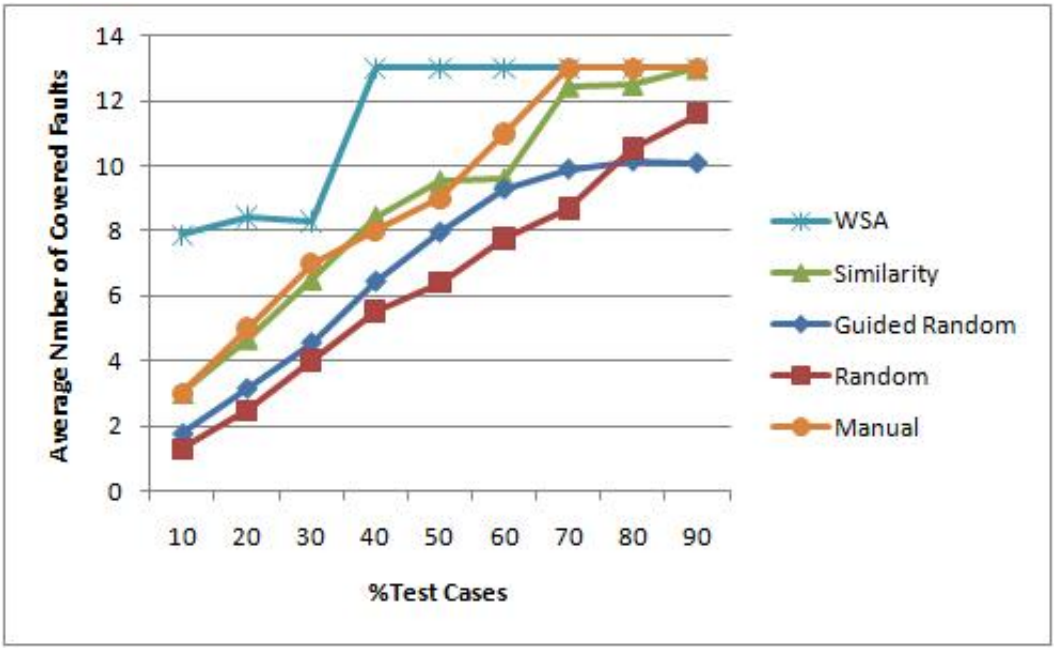


Figure 5.5: Average number of covered faults by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 1.

Observing the results, we can conclude that when the probabilities are well assigned, it's better to use WSA for any desired percentage of test cases. In another words, effective probability distribution can contribute to better results. Observe that, when the probabilities are not well distributed, the manual selection can perform better, and the Similarity, that uses

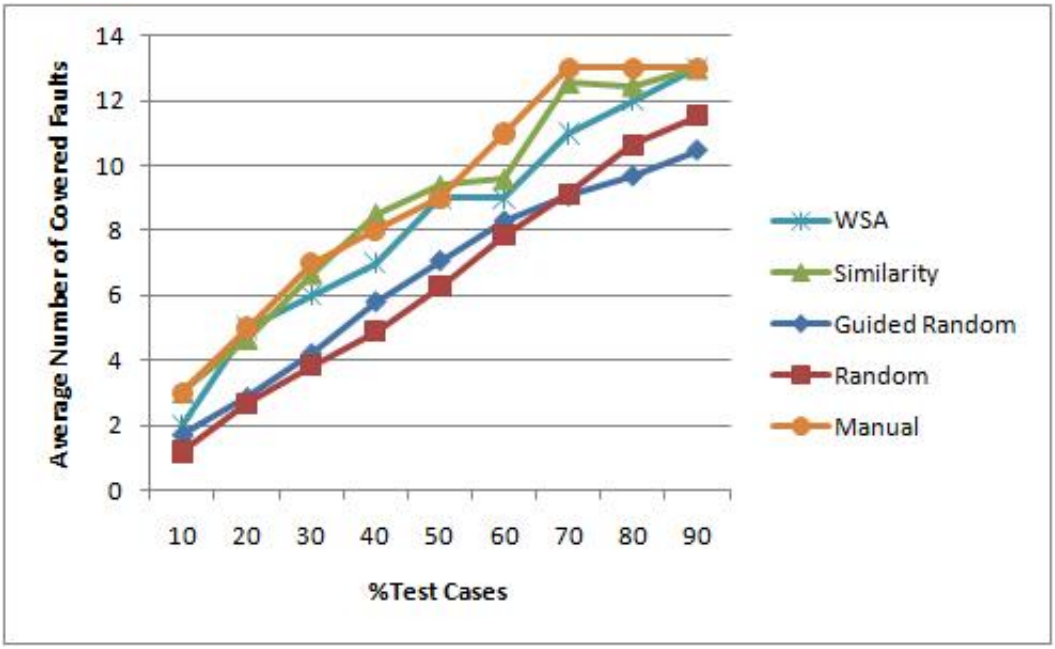


Figure 5.6: Average number of covered faults by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 1.

no probabilities values, presents a better behavior, but it is necessary to consider the overhead to apply it.

Application 2

The results obtained - for Application 2 - by applying WSA, Random Selection, Guided Random, Similarity and Manual selection (done by test designer), considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.7 and 5.8.

Again, WSA brings better results depending on probabilities assignment and also on the ability of the tester to pinpoint faults. Note that the manual selection for Application 2 was not as successful as for Application 1. Moveover, for faults distributed among the longest test cases selected from several branches, the guided random presents a better behavior since the selection is always based by the best local choices. On the other hand, WSA computes the product of probabilities to choose the test case to select, favoring the choice of shorter test cases.

Fault Coverage - Concluding Remarks

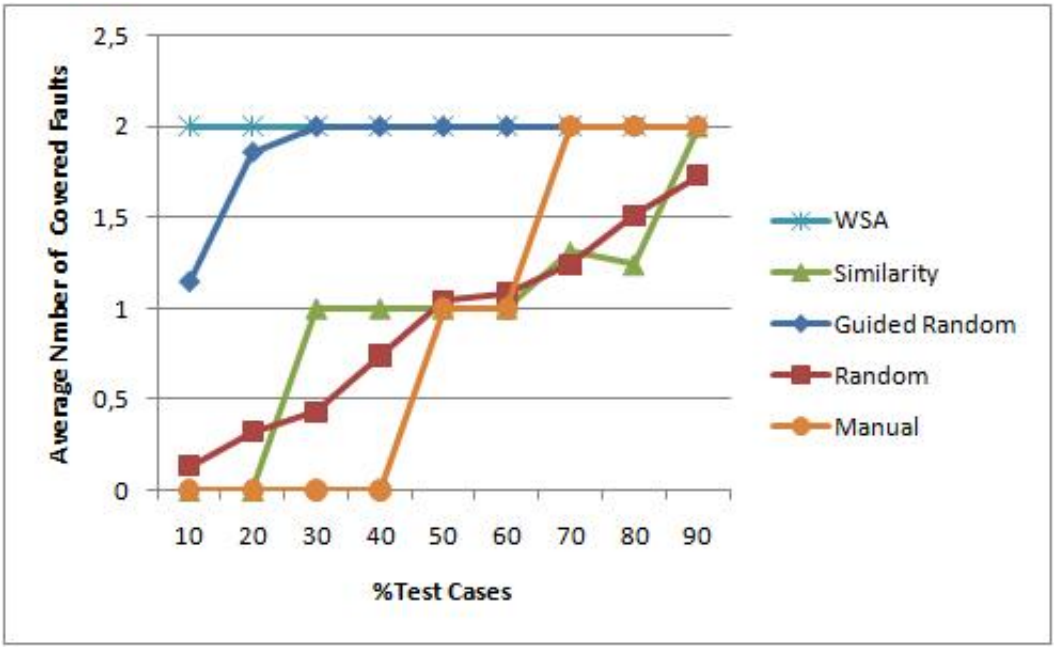


Figure 5.7: Average number of covered faults by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 2.

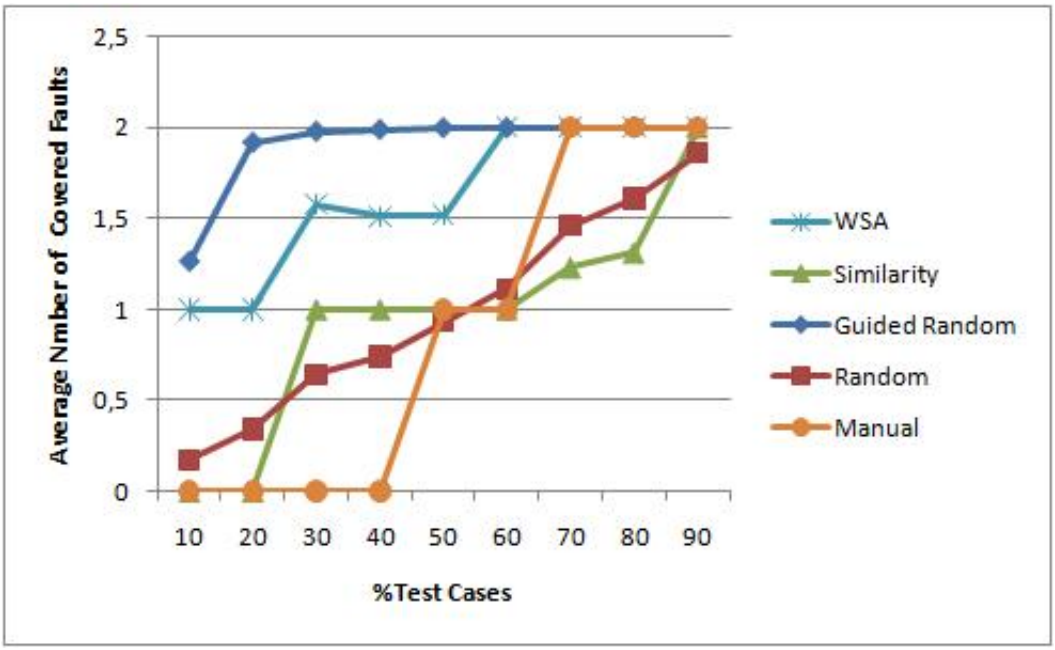


Figure 5.8: Average number of covered faults by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 2.

For both applications, we can see through the graphics showed in Figures 5.5, 5.6, 5.7 and 5.8 that, in order to obtain good results it is necessary to know the points of possible faults. The results obtained when WSA designer assigned the probabilities show that WSA is an effective technique.

Transition Coverage

Application 1

The results obtained - for Application 1 - by applying WSA, Random Selection, Guided Random, and Similarity, considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.9 and 5.10.

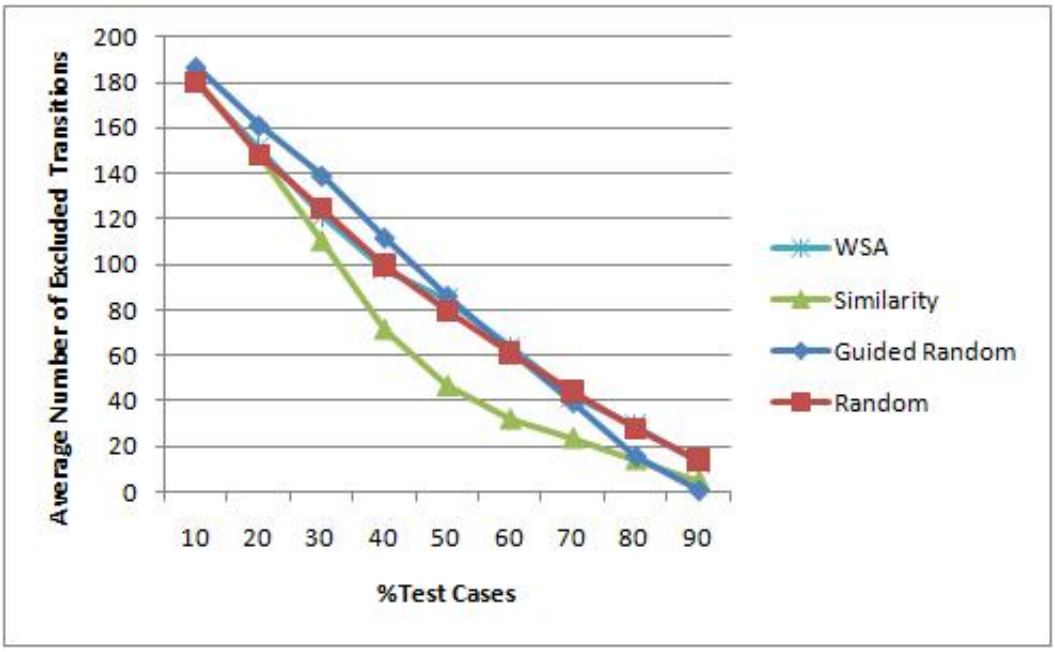


Figure 5.9: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 1.

By observing the results, we can conclude that the similarity presents better performance in relation to the others, since it excludes less transitions, and therefore, it presents the best coverage. This fact can be explained because similarity does not consider the importance of the test cases. Particularly, similarity keeps in the test suite the biggest test cases.

Application 2

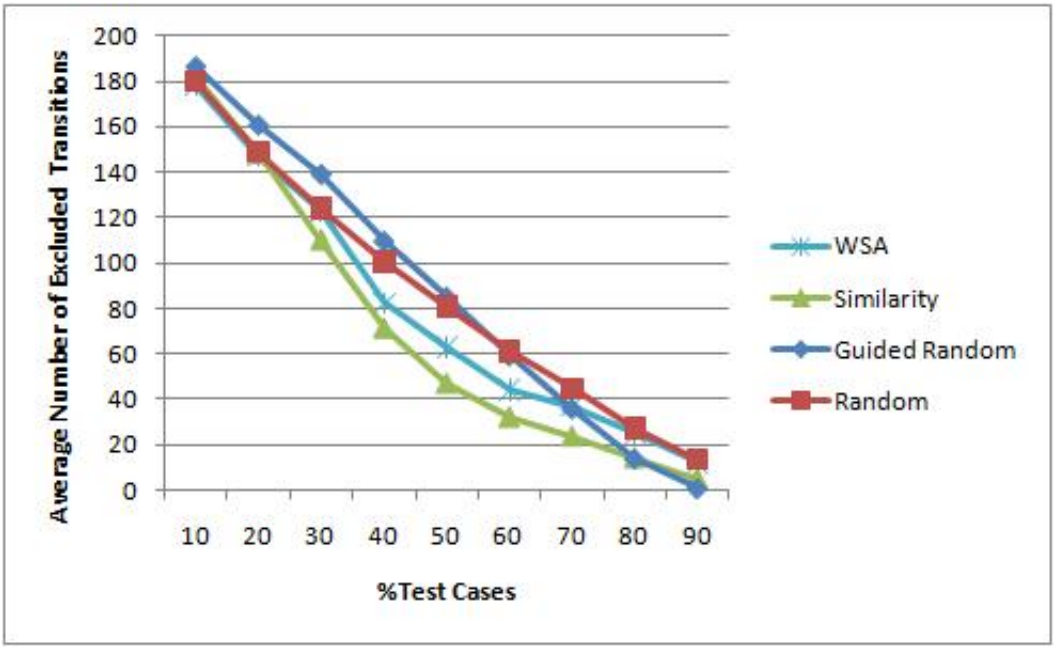


Figure 5.10: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 1.

The results obtained - for Application 2 - by applying WSA, Random Selection, Guided Random, and Similarity, considering the probabilities assigned by WSA and test designer can be seen, respectively, in the Figures 5.11 and 5.12.

By observing these graphics, we can conclude that the similarity presents better performance in relation to the others in most of the cases. However, WSA can be better when considering a path coverage bellow 30%. This can be an evidence that - bellow 30% - the importance and length of the test cases are the same and random choice is applied.

Transition Coverage - Concluding Remarks

For both applications, we can see through the graphics showed in Figures 5.9, 5.10, 5.11 and 5.12 that Similarity presents a better performance - in most of the cases. However, when considering the probabilities based approaches (WSA and Guided Random), WSA presents generally a better performance since it inherits the similarity principles.

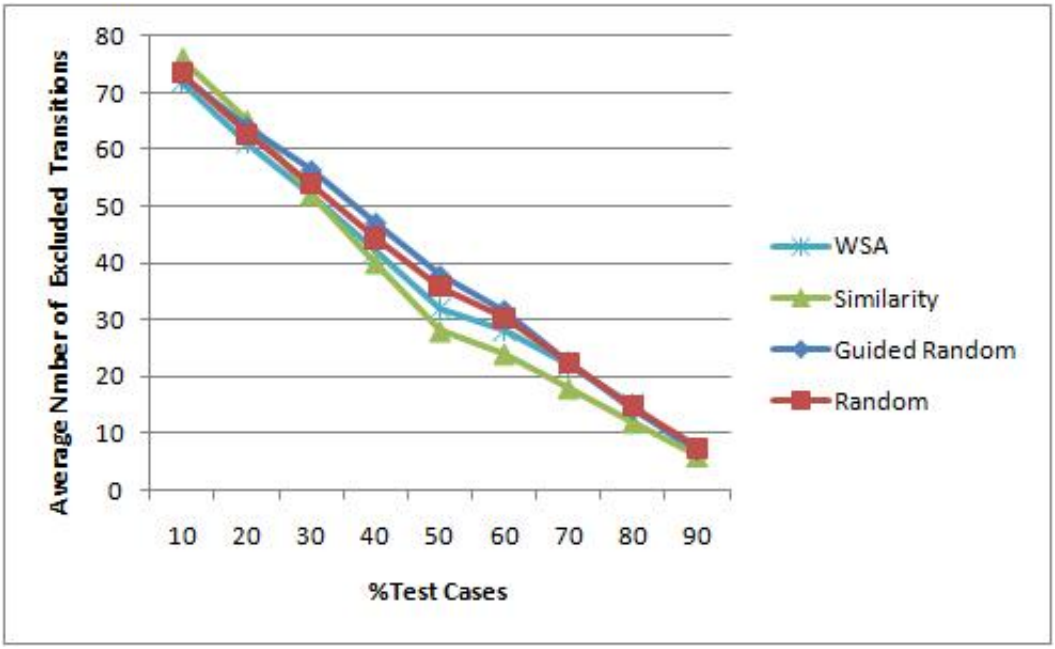


Figure 5.11: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **WSA designer** - 100 times for each test case selection goal - Application 2.

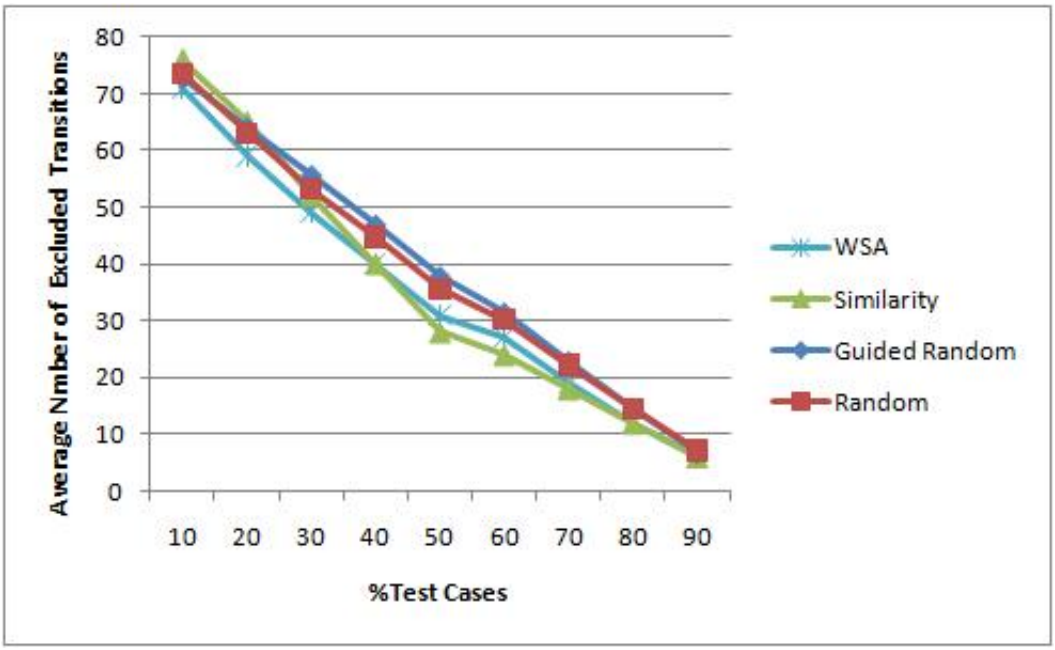


Figure 5.12: Average number of excluded transitions by running each test selection strategy - with probabilities assigned by **test designer** - 100 times for each test case selection goal - Application 2.

5.4 Concluding Remarks

In this Chapter, we presented our proposal for test case selection based on Similarity (Chapter 3) that also considers weights that are assigned to test cases (WSA). The goal of WSA is to keep in test suite the most important and different test cases. Case studies were performed, and the results show evidence that:

- **Fault Coverage:** WSA is an effective strategy, but it is necessary to know and/or guess the point of the faults;
- **Transition Coverage:** Similarity - in most of the cases - presents a better performance.

The main threats to validity of the performed case study are related to the weights assigned. In the case study the weights were assigned by only two subjects. One of them (WSA designer) had knowledge about the faults and the strategies. The use of more subjects to assign different weights would provide a more general overview of the analyzed strategies. Also no guidelines were used to assign the probability values, which may affect the performance of some strategies (e.g. WSA and Guided Random). Aside from those threats the use of only two models affect the generalization of the obtained results.

Chapter 6

Dissimilarity-based Reduction

This Chapter presents our proposal for test suite reduction based on Similarity (Chapter 3). As said before, the goal of test suite reduction is decrease the size of test suite while keeping the coverage of a specific test requirement. However, the fault coverage, in general, is affected. Since we obtained adequate results with Similarity strategies considering fault coverage, our goal is to apply the similarity function to choose the most different test cases, that possibly will cover the most test requirements while keep a good fault coverage.

6.1 Definition

The idea is to keep in the reduced test suite the most different test cases while providing 100% coverage of one defined test requirement (in our case, transitions coverage). Then, the test cases are chosen according to the degree of similarity and are placed in a reduced set named as the Reduced Test Suite (RTS).

Overall, this strategy uses the similarity function to build the similarity matrix (as showed in Chapter 3). The inputs are:

- **Test Suite:** the set of test cases that should be reduced;
- **Test Requirements:** the set of transitions that should be covered;
- **Similarity Matrix:** the matrix that contains the information about similarity among all test cases of the test suite.

Since the proposal of any test suite reduction technique is to cover 100% test requirements, then it is important to identify all essential test cases, since an essential test case is the only one that covers a specific requirement. Therefore, all essential test cases should be in the RTS for reaching 100% of test requirements.

The algorithm of this strategy is presented in Algorithm 3. The first step of this strategy is to remove all essential test cases from the similarity matrix, and add all of them to the RTS (lines 1 and 2). Then, all test requirements satisfied by those test cases are marked (line 3). After that, the idea is to find, in the matrix, the minimum value that represents the most different test cases and try to keep them in the RTS, always verifying if all test requirements have already been covered (lines 4 - 30).

When a tie among minimum values (more than one minimum value) is found, in the similarity matrix, the idea is to choose the pair that covers the maximum number of not yet covered requirements, however if there is a new tie, a random choice is applied (lines 5 - 7).

The next step (lines 10 - 17) is to verify if the test cases (of the chosen pair) are 1-to-1 redundant. This occurs when the set of covered requirements of one of them is contained within the other one. In this case, the idea is to place in the RTS, the test case that covers more not yet covered requirements so far. Since the two test cases were already analyzed, they are removed from the similarity matrix, and all new covered requirements are placed in the marked requirements set.

If the pair is not 1-to-1 redundant, then it is necessary to check if the requirements covered by each one of them are already covered (lines 19 - 20 and 25 - 26), because if the requirements have already been covered, these test cases are redundant. Otherwise, the test cases are added to the RTS and the new satisfied requirements are added to marked requirements set (lines 21 - 24 and 27 - 30).

Regarding the complexity analysis of Algorithm 3 we are able to observe a repeating structure (`while` command in line 4) that repeats *m* times, where *m* is the number of the test requirements. The method `getAllMinValue` searches the matrix for the lowest values of similarity, having, thus, a complexity $O(n^2)$, where n is the number of test cases in the test suite. Considering that the method `getAllMinValue` is executed m times, we obtain a complexity $O(m \times n^2)$ for Algorithm 3.

```
   input  : testSuite, testRequirement, similarityMatrix
   output: reducedTestSuite
1  similarityMatrix.removeEssentialTestCases();
2  reducedTestSuite.add(essentialTestCases);
3  markedRequirements.add (satisfiedRequirements(essentialTestCases));
4  while (!(satisfyAllRequirements(testRequirements,markedRequirements)) do
5      minValues = getAllMinValue(similarityMatrix);
6      pairs = PairsCoverMaxNumOfNotCoveredRequirements(markedRequirements,minValues);
7      choosedPair = pairs.shuffle.get(0);
8      testCase1 = choosedPair.getTestCase1();
9      testCase2 = choosedPair.getTestCase2();
10     if (containsRequirements(testCase1,testCase2)) then
11         similarityMatrix.remove(testCase1,testCase2);
12         markedRequirements.add(satisfiedRequirements(testCase1));
13         reducedTestSuite.add(testCase1);
14     else if (containsRequirements(testCase2,testCase1)) then
15         similarityMatrix.remove(testCase1,testCase2);
16         markedRequirements.add(satisfiedRequirements(testCase2));
17         reducedTestSuite.add(testCase2);
18         else
19             if (containsRequirements(markedRequirements,testCase1)) then
20                 similarityMatrix.remove(testCase1);
21             else
22                 similarityMatrix.remove(testCase1);
23                 markedRequirements.add(satisfiedRequirements(testCase1));
24                 reducedTestSuite.add(testCase1);
25             if (containsRequirements(markedRequirements,testCase2)) then
26                 similarityMatrix.remove(testCase2);
27             else
28                 similarityMatrix.remove(testCase2);
29                 markedRequirements.add(satisfiedRequirements(testCase2));
30                 reducedTestSuite.add(testCase2);
```

Algorithm 3: Dissimilarity-based Reduction - Algorithm

6.2 Example - Dissimilarity

In order to illustrate the strategy, an example is presented below. An LTS model is presented in Figure 6.1. From this LTS model, 6 test cases are obtained as can be seen in Table 6.1.

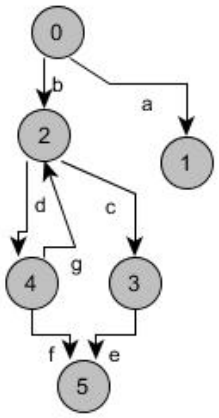


Figure 6.1: Example - LTS model

Table 6.1: Test Cases and Size of test cases

TC id	Path	Test Size
1	a	1
2	b c e	3
3	b d f	3
4	b d g	3
5	b d g d f	5
6	b d g c e	6

Considering that, the test requirement (coverage criteria) is transition coverage, the RTS should cover all transitions. By applying the presented idea, the matrix is showed in Matrix

6.1. The size of test cases are also presented in Table 6.1.

$$SimilarityMatrix = \begin{pmatrix} & TC1 & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC1 & & 0 & 0 & 0 & 0 & 0 \\ TC2 & & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & & 0.75 & 0.75 \\ TC5 & & & & & & 0.6 \\ TC6 & & & & & & \end{pmatrix} \quad (6.1)$$

Since the test requirements are transitions, then $testRequirements = \{a, b, c, d, e, f, g\}$. The first step is to find essential test cases. For this example, TC1 is essential. Therefore, the variables $markedTestRequirements$ and $reducedTestSuite$ are updated: $markedTestRequirements = \{a\}$; $reducedTestSuite = \{TC1\}$. The new matrix is presented in Matrix 6.2.

$$SimilarityMatrix = \begin{pmatrix} & TC2 & TC3 & TC4 & TC5 & TC6 \\ TC2 & & 0.33 & 0.33 & 0.25 & 0.75 \\ TC3 & & & 0.66 & 0.75 & 0.5 \\ TC4 & & & & 0.75 & 0.75 \\ TC5 & & & & & 0.6 \\ TC6 & & & & & \end{pmatrix} \quad (6.2)$$

There is only one minimum value (0.25) found between TC2 and TC5. They are not 1-to-1 redundant test cases, and the satisfied requirements (covered transitions) by each one of them have not been covered yet. Thus, both test cases are added to RTS, and, the variables are updated again: $markedTestRequirements = \{a, b, c, d, e, f, g\}$; $reducedTestSuite = \{TC1, TC2, TC5\}$.

Finally, all requirements are satisfied by the RTS composed by TC1, TC2, TC5. In other words, $markedTestRequirements = testRequirements$.

6.3 Case Study

A case study was executed. The goal of this case study is to compare the fault detection capability and the RTS size for the Heuristics - G, GE, GRE and H - and Dissimilarity. The defined test requirement is 75% of transitions coverage. This number was defined because the stop criteria is the amount of covered transitions, and in this case if all transitions are covered, all faults will be revealed (the faults are linked to transitions). Therefore, decreasing the test requirements from 100% to 75%, provides a more realistic and fair comparison.

6.3.1 Application

The application used for this is a desktop tool named TaRGeT, that automatically generates and selects test cases [48]. In order to execute the case study the tool LTS-BT [17] was used. For this case study, the input is a use case template [48], written by Motorola experts. This template contains information regarding the use cases and the scenarios, in the application, that can be executed by a user. All test cases, generated for this case study, were manually executed by Motorola employees.

The objective of this case study is to analyze the behavior of the strategy concerning aspects of the reduced test suite. Therefore we need to observe how much of the test suite the strategy is able to reduce, and also how many faults were able to be detected by executing the reduced test suite. These information can be obtained from the following metrics:

- **Test Suite Size:** The size of a test suite is measured by the number of test cases that it contains. In this study, the test suite size is 84 test cases;
- **Fault Coverage:** As said before, all test cases are executed manually. From the 84 test cases, 13 revealed failures. And each one of these failures were caused by a different fault. Summarizing, the complete test suite revealed 13 different faults.

6.3.2 Case Study - Preparation

As said before, the test requirement is 75% of the transitions of the model. Thus, for each execution, 75% of transitions are randomly chosen, from the model, in order to establish the test requirement set. Since the set of test requirements is different, the results can change.

Therefore, 3 sets of test requirements, i.e. three different sets that cover 75% of the transitions, were applied to analyze each strategy.

In other hand, the Heuristics and the Dissimilarity strategy present a random choice in their algorithms. Thus, each one of them was executed one hundred times, for each set of test requirement. The results are presented in the next subsection.

6.3.3 Results of the Case Study

This subsection presents the results of the case study. As a reminder, the variables considered in the study are the size of the RTS and the fault coverage.

Reduced Test Suite Size

By applying Dissimilarity and the heuristics to reduce the test suite, the different RTS size can be observed. The Table 6.2 presents the obtained results (for one hundred executions) for each one of the 3 different, randomly defined, set of test requirements.

Table 6.2: Average of RTS size (100 executions) for all 3 sets of test requirements

	Dissimilarity	GRE	GE	G	H
1	31.78	29	29	29	29.52
2	32.08	30	30	30	31.89
3	29.83	27	27	27	29.36
Average	31.23	28.66	28.66	28.66	30.25

The results show that GRE, GE and G present better results, followed by H. In this case, Dissimilarity presents the worst results. In average, Dissimilarity reduces the test suite 62.82 %; G, GE and GRE, 65.88%; and H, 63.99%. The Figure 6.2 presents the graph that shows the obtained average for the 3 sets of test requirements, in each one of the 100 executions.

Fault Coverage

The obtained results for each strategy, considering the 3 different sets of test requirements, are presented in Table 6.3.

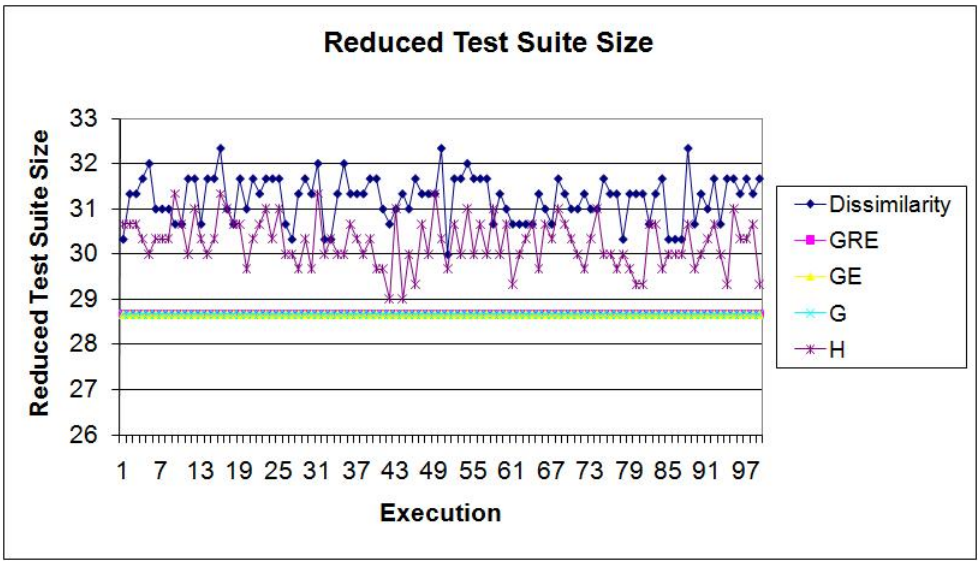


Figure 6.2: TaRGeT - Reduced Test Suite Size

Table 6.3: Average of test suite reduced size (100 executions) for all 3 sets of test requirements

	Dissimilarity	GRE	GE	G	H
1	8.92	7	7	7	7
2	4.51	4	3.58	3.48	3.99
3	5.2	4.68	5.17	5.26	5.26
Average	6.21	5.22	5.25	5.24	5.41

The results show that most of the times, the Dissimilarity strategy reveals more faults. In average, Dissimilarity reveals 51.28% of the total amount of faults; H reveals 43.58%; GRE and GE, 42.02%; and G, 38.46%.

The Figure 6.3 presents the graph that shows the obtained average by considering the 3 sets of test requirements in each one of 100 executions. The conclusion is that, by considering each execution (i.e., the average among the 3 sets of test requirements), Dissimilarity reveals more faults.

Concluding Remarks - Case Study

The Table 6.4 presents the summary of this case study, in terms of the percentage of reduction and fault coverage for Dissimilarity, G, GE, GRE and H.

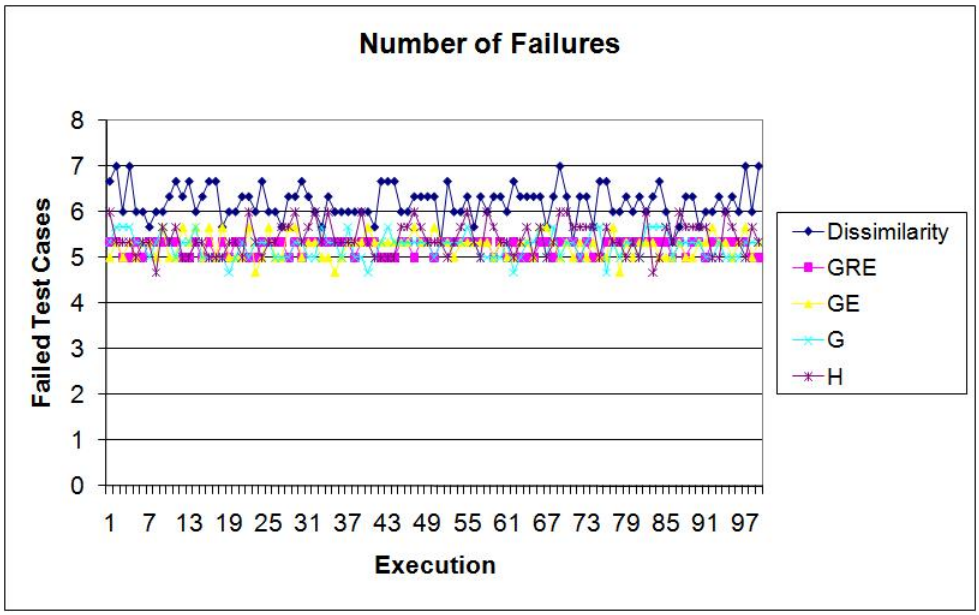


Figure 6.3: TaRGeT - Failures

Table 6.4: Summary - Percentage of Reduction and Fault Coverage

	Dissimilarity	GRE	GE	G	H
Percentage of Reduction (%)	62.82	65.88	65.88	65.88	63.99
Percentage of Fault Coverage (%)	51.28	42.02	42.02	38.46	43.58

Summarizing the data of the table: Dissimilarity presents the worst percentage of test suite reduction (the best is reached by applying GRE ,GE or G), however it presents the best percentage of the revealed faults (the worst is obtained by applying G). Analyzing the data from Dissimilarity, it is necessary - in average - to execute 3.06% more test cases, than the best strategy. In other hand, the percentage of the faults is increased by 9.26%. For this case study, this means that executing 2.57 more test cases, we are able to reveal 1.2 more faults.

The use of only one model can be considered a threat to validity of this case study. Since we measure structural elements from the LTS, mainly the transitions, using more models would provide a more general overview of the strategies. Another threat to validity is the chosen test requirement. In order to generalize the results, it would be necessary to use different test requirements (e.g. fault coverage, requirements coverage, among others) and observe the behavior of each strategy.

6.4 Experiment - Reduction

This Section presents the executed experiment and results. The used framework was presented in Chapter 2. The process was followed and the results are presented in the next subsections.

Our general hypothesis is that Dissimilarity presents the best performance in relation to the rate of reduction, considering, as test requirement, 75% of transitions.

6.4.1 Definition

To define the goal, the key questions proposed by Wohlin *et al.* were answered[62]:

1. **What is studied?** reduction stragegies;
2. **What is the intention?** to investigate;
3. **Which effect is studied?** reduced test suite size;
4. **Whose view?** the tester;
5. **Where is the study is conducted?** Model-Based Testing (MBT).

From these answers, the goal definition template is filled. In summary, the goal of this experiment is:

Analyze *reduction strategies*
for the purpose of *investigating*
with relation to *reduced test suite size*
from the point of view of the *tester*
in the context of *MBT*.

The (input) objects are LTS models. In this case, this experiment does not present subjects.

6.4.2 Planning

For this phase, it is necessary to define [62]: context selection, variable, hypothesis, design and instrumentation.

Context Selection

The context of this experiment can be characterized as a "toy vs. real" problem [62]. In this case the objects are LTS models, randomly generated from a configuration. This configuration is characterized by a specific number for the depth of the LTS, the number of loops, branches and joins (these elements are detailed in Appendix B.

Variables Selection

The variable chosen to observe (dependent variables) and to control (independent variables) are:

- **Dependent:** The Reduced Test Suite Size - RTSS .
- **Independent:** The test requirement percentage; the configuration chosen for the depth and amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 5 levels: G, GE, GRE, H and Dissimilarity (DSim).

Hypothesis Formulation

The experiment definition is formalized as following:

- **A null hypothesis (H_0):** $RTSS_G = RTSS_{GE} = RTSS_{GRE} = RTSS_H = RTSS_{DSim}$: All strategies have a similar behavior in relation to the reduced test suite size;
- **An alternative hypothesis, H_1:** $RTSS_G \neq RTSS_{GE} \neq RTSS_{GRE} \neq RTSS_H \neq RTSS_{DSim}$: All strategies have a different behavior in relation to the reduced test suite size.

Experiment Design

As seen before, there is one factor (test suite reduction strategy) with 5 levels (or treatments). Thus, there is one factor and 5 treatments, where, for each object, all five treatments are applied. The chosen confidence level is 95% (significance level is $\alpha = 0,05$).

Aiming to define the number of replications, necessary to guarantee statistical significance for the specified level of confidence (95%), 40 replications were executed and the data are presented in Table 6.5.

Table 6.5: Mean, Standard Deviation and number of necessary replications for each technique.

Technique	G	GE	GRE	H	DSim
Mean ($\overline{x}$)	4.3	3.75	3.75	3.75	3.85
Standard Deviation (s)	1.26	1.05	1.05	1.05	1.02
Number of Necessary Replications (n)	133	122	122	122	110

Observing the Table 6.5, we are able to see that 133 replications provide a statistical significance for the obtained data. Therefore, this experiment design will consider 200 replications for each technique, in order to better explain the results (since they are expressed using percentages).

Instrumentation

In this step, there are three types of instruments [62]:

- **Objects:** The objects are LTS models randomly generated from a configuration (depth, number of loops, branches and joins).
- **Guidelines:** This experiment uses no guidelines, since the strategies do not require subjects to configure them.
- **Measurements:** The RTS size will be collect for each treatment. The tool LTS-BT provides support for executing the experiments and collecting the data.

Validity Evaluation

The objects used in this experiment, can be considered the main threat to validity. They can not represent a real behavior since these are automatically generated. From a specific configuration, the objects are randomly generated from a specific configuration, both the traceability and controllability of the elements of the model (transitions and states) are reduced.

On the other hand, we are able to obtain an overview of the execution of the strategies in several models that has a same configurations, since they are randomly generated. Then, we avoid presenting a conclusion that is specific to only one LTS of a configuration. A proper scenario would be to have several real applications to execute the strategies. However, most real applications and their respective specification are not available to the public.

6.4.3 Operation

To execute this experiment, it was necessary to implement the 5 strategies and the LTS generator (see Appendix B). Both the LTS generator and the strategies are implemented in Java programming language[1].

The objective is to automatically generate different models using the LTS generator. In order to depict a real application in these generated LTS, the configuration used for the generator was chosen from observing a real application (used in the case study - Section 6.3). Then the chosen configuration is:

- **Depth:** 15;
- **Number of loops:** 2;
- **Number of branches:** 3;
- **Number of joins:** 3.

There is only one experimental design (with only one factor - test suite reduction strategy) with a null and an alternative hypothesis, where the intention is to reject the null hypothesis. Each strategy was executed 100 times, using a machine with the following configurations:

- Intel Core 2 quad 2.33 GHz;
- 4GB RAM;
- 1TB for Hard Disk Memory.

[1] http://www.sun.com/java/

6.4.4 Analysis and Interpretation

The confidence intervals are plotted for each strategy. The graph can be seen in Figure 6.4. Since the confidence intervals of G, GE, GRE and H overlap, a statistical test is required [40] to test the hypothesis. In this case, since the confidence interval for G, GE, GRE and H is overlapped, we will do first a comparison among them. In order to show the comparison among the four strategies, the statistical tests are applied and the results are presented in Appendix C. In this Appendix, by applying the statistical test, we concluded that the obtained results can not be considered different, for the heuristics (G, GE, GRE and H).

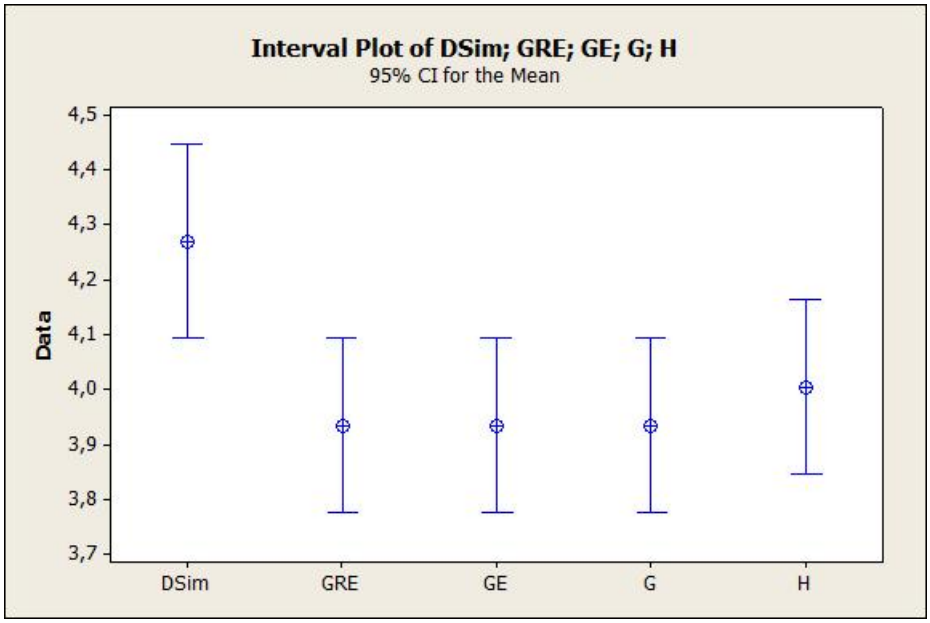


Figure 6.4: Interval Plot

Since the behavior of the heuristics can not be considered different, the Dissimilarity strategy needs to be compared with only one of them. Once that Dissimilarity includes some concepts from GRE, the comparison will be between GRE and Dissimilarity. The experiment definition is formalized as following:

- **A null hypothesis (H_0):** $RTSS_{GRE} = RTSS_{DSim}$: All techniques have a similar behavior in relation to the reduced test suite size;
- **An alternative hypothesis (H_1):** $RTSS_{GRE} \neq RTSS_{DSim}$: All techniques have a different behavior in relation to the reduced test suite size.

The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[2]. The results can be seen in Figures 6.5 and 6.6. In this graph, the red dots, should overlap the blue line, in order to indicate that the data fit a normal distribution.

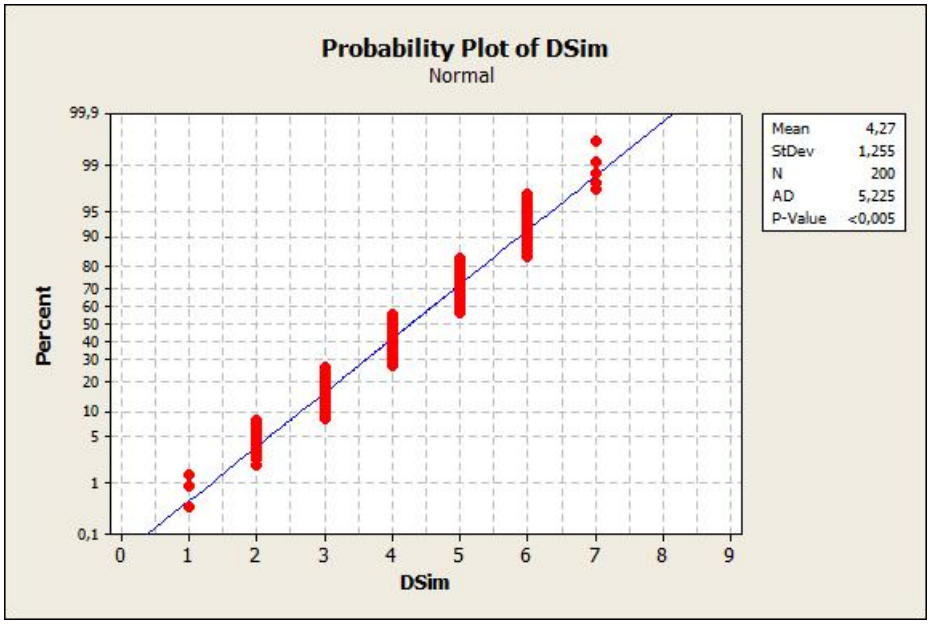


Figure 6.5: Anderson-Darling normality test - Dissimilarity

The $p - Values$ are smaller than the significance value ($\alpha = 0.05$), thus the data do not show a normal distribution. In this case, it is necessary to use a non-parametric test. Since this experiment has a factor with two treatments (GRE and Dissimilarity), the Mann-Whitney is applied to check the null hypothesis. The results for the test can be seen in the Table 6.6.

Table 6.6: Mann-Whitney Test - GRE and DSim

Technique	N	Median
GRE	200	4.000
DSim	200	4.000
The test is significant at 0.0074		

Since $p - Value = 0.0074$, and the $p - value$ is lower than 0.05 (α), H_0 (the null hypothesis) can be rejected. The box plot that presents the difference of the RTSS between

[2]http://www.minitab.com/

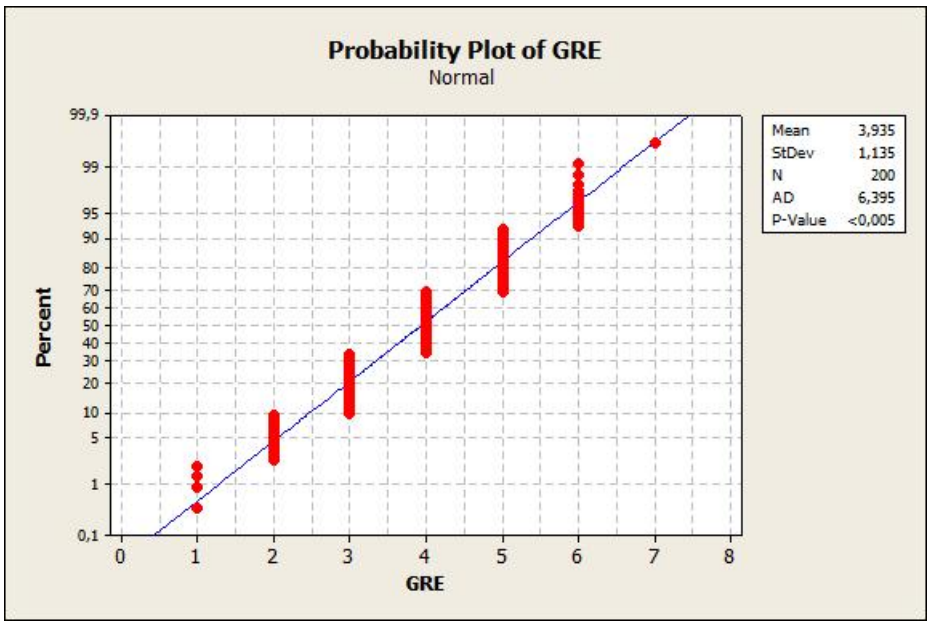


Figure 6.6: Anderson-Darling normality test - GRE

DSim and GRE can be seen in Figure 6.7. By this figure, we can conclude that in the most cases, the difference is 0 or 1, but in some cases (not often) can be 2.

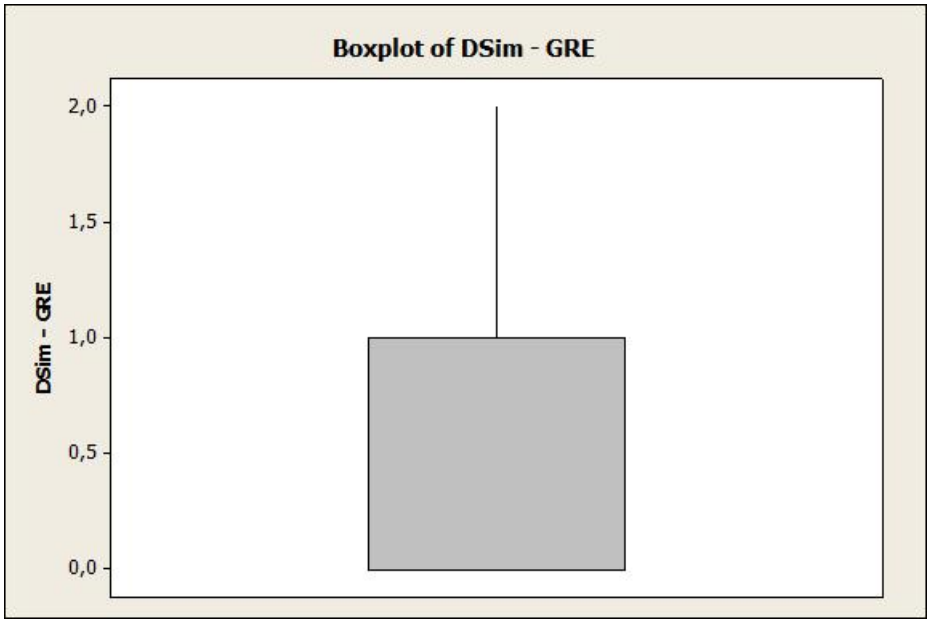


Figure 6.7: Box Plot RTSS of DSim - GRE

6.4.5 Concluding Remarks - Experiment

Summarizing the data of the experiment: Dissimilarity presents the worst percentage of test suite reduction. The best is reached by applying GRE ,GE or G. However, we can conclude that the difference is 0 or 1 or 2.

6.5 Concluding Remarks

In this Chapter, we presented our proposal for test suite reduction based on Similarity (Chapter 3) named as Dissimilarity. A case study and an experiment are performed. The results of the case study show evidence that Dissimilarity presents the worst percentage of test suite reduction (the best is reached by applying GRE ,GE or G), however it presents the best percentage of the revealed faults (the worst is obtained by applying G). Analyzing the data from Dissimilarity, it is necessary - in average - to execute 3.06% more test cases, than the best strategy. In other hand, the percentage of the faults is increased by 9.26%. For this case study, this means that executing 2.57 more test cases, we are able to reveal 1.2 more faults.

The experiment confirms one of the obtained results of the case study: Dissimilarity presents the worst percentage of test suite reduction. In this case, considering 75% of transition coverage, as the test requirement.

Chapter 7

Analysing Reduction based on Selection Order

In this chapter, a new perspective in assessing test suite reduction techniques based on their rate of fault detection is introduced. This criterion, which is standard in assessing test-suite prioritization, has never been used for reduction. Our proposal stems from the consideration that under pressure testing could be stopped before all tests in the reduced test-suite are run, and in such cases the ordering in the reduced test-suite is also important. We compare four well-known reduction heuristics showing that by considering the rate of fault detection, the reduction technique to be chosen when time is an issue might be different from the one performing the best when testing can be completed.

7.1 Motivation

Different test reduction heuristics have been proposed. The common practice to compare them is by considering that the whole reduced test suite will be executed [36; 20]. Unfortunately during development managers might be forced for many reasons to stop the testing earlier than planned, and thus a lower number of test cases is run than those in the reduced test-suite. The ideal solution would be to maximize the number of failures detected while selecting a subset of non-redundant test cases that covers all requirements.

To motivate such an approach, let us consider the same toy example presented in [30], where a program is supposed to contain 10 faults and a test suite of 5 test cases, called for

simplicity (A,B,C,D,E), is available. Table 7.1 shows the fault detection capability of each test case. When all test cases are executed, independently of their order, the percentage of fault detection is always 100%. To select the best prioritization technique the Average Percentage of Fault Detection (APFD) measure reached by the associated combination of test cases has been proposed. For instance, as described in [30], if three different prioritization techniques are applied that produce the sets (A,B,C,D,E), (E,D,C,B,A), (C,E,B,A,D), they yield respectively the following APFD: 50%, 64 % and 84%. Hence the third one gives the best ordering.

Now, let us suppose to apply three different reduction techniques on the same test suite, obtaining the following reduced sets:

- TS1 = (B,E,D)
- TS2 = (A,E,B,C)
- TS3 = (B,A,C,D,E)

We are assuming that all of the three sets reach 100% requirements coverage, hence TS1 performs the best in terms of test size reduction. However, it should be considered that with only 3 test cases TS1 discovers the 70% of the faults, while TS2 and TS3 detect the 100%. We believe that in test reduction a practical compromise that takes into consideration both the number of test cases executed and the rate of fault detection should be defined. For instance if we use the APFD measure for comparing the respective rates of fault detection of the three

Table 7.1: Test Suite and Faults exposed

Test	faults									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

Table 7.2: APFD of the considered reduction heuristic

#TC	TS1	TS2	TS3
1	40	20	40
2	35.5	22.5	30
3	47.85	34.65	37.95
4	47.85	47.5	46.25
5	47.85	47.5	62

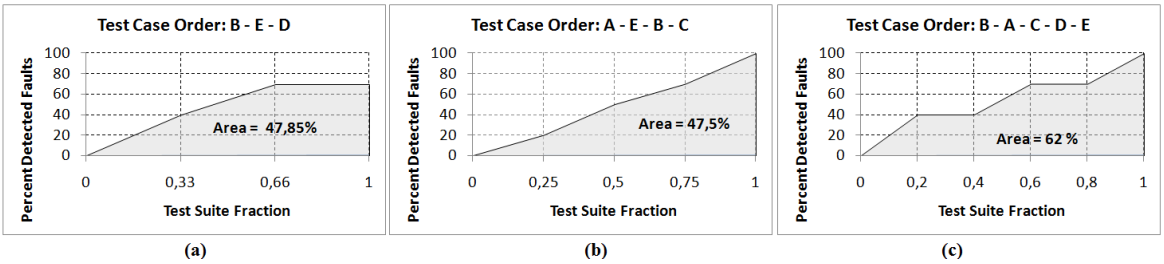


Figure 7.1: Test Case Order

test suites, we get the results reported in Figure 7.1: TS1 has 47.85% of APFD, TS2 has 47.5% and TS3 has 62%. Thus the best technique in terms of quickly detecting faults would be TS3, but it is the one having the worst performance in terms of test suite reduction.

On the other hand, if for any reasons the test phase needs to be stopped after only two test case are executed, which is the relative performance of the three test suites? If the APFD of TS1, TS2, TS3 are again evaluated considering only their first two tests, the following results can be observed:

- TS1_r = (B,E) detects 7 faults and reaches the 35.5% of APFD
- TS2_r = (A,E) detects 5 faults and reaches the 22.5% of APFD
- TS3_r = (B,A) detects 4 faults and reaches the 30% of APFD

This changes the previous measures. The best APFD when only two test cases have been executed belongs to TS1 and not anymore to TS3. Thus if testing is stopped before all test cases in the reduced test-suite are run, the ordering in the reduced test-suite is important for selecting the most effective test strategy. This is why we propose to mix the concepts of reduction and prioritization.

It is important to notice that the kind of data analyzed in the above example would only be available a posteriori. This example is used to evidence the existence of different points of view in the evaluation of a test reduction strategies when taking in consideration also realistic problems. However, the fault detection of each test case is available only after the test case is executed and not before, and constraints forcing the manager to stop testing in advance cannot be foreseen.

From a practical point of view the knowledge about the rate of fault detection of reduction heuristics could be derived by the application of fault seeding or from the history of test

execution for similar products.

7.2 General definition

From the example discussed in the previous section, we think that test reduction and test prioritization can be seen as two aspects of a more general problem that is to select an optimal set of test cases under the existing constraints that minimizes redundancy (via reduction) and maximizes fault detection (via prioritization). In this section we formalize the procedure followed for deriving Table 7.2 and consequently provide a general definition of a criterion to select the reduction heuristics to be applied.

In particular, considering the definition of APFD of [58], given a program having a number of faults equal to m, a number of requirements equal to q and an ordered test suite $TS = (T_1, ...T_n)$ of cardinality n, the following functions can be defined:

- $TF(i)$ for $1 \leq i \leq m$ represents the position of the first test case in TS that exposes the fault i
- For $1 \leq h \leq n$ $req(T_h) = \{r_p | 1 \leq p \leq q$ and r_p is a requirement covered by $T_h\}$ represents the set of requirements covered by T_h
- For $1 \leq h \leq n$ $r(T_h) = |req(T_h)|$ represents the number of requirements covered by T_h

Consequently if j, $1 \leq j \leq n$, represents cumulative number of test cases executed during a testing phase till a certain point in time, and f, $1 \leq f \leq m$ is the cumulative number of faults in $T_1, ..T_j$, the following cumulative functions can be defined:

- $APFD(j) = 1 - \frac{TF(1)+...+TF(f)}{jf} + \frac{1}{2j}$ represents the incremental APFD after the execution of j test cases and the detection of f faults
- $REQ(j) = \bigcup_{i=1}^{j} req(T_i)$ represents the set of requirements covered by the execution of the subset $T_1, ..T_j$
- $R(j) = |REQ(j)|$ represents the cumulative number of requirements discovered by the execution of the first j test cases

In this case if q represents the cumulative amount of requirements to be covered for a specific system, and m is the cumulative amount of faults, then when $j = n$ the previous formulas become:

- $APFD(n) = 1 - \frac{TF(1)+...+TF(m)}{nm} + \frac{1}{2n}$ represents the standard formula of APFD
- $REQ(n) = \bigcup_{i=1}^{n} req(T_i) \leq q$ represents the set of requirements covered by the reduced test suite $T_1, ..T_n$
- $R(n) = |REQ(n)|$ represents the cumulative number of requirements discovered by the execution of the reduced test suite

In general, given q and m as above, let us assume that k different heuristics, $H_1, ...H_k$ are available for test reduction, such that the reduced test suites have cardinality $h1, ..., hk$ respectively and are represented by $(T_{h1_1}, ..., T_{h1_{h1}})$, ..., $(T_{hk_1}, ..., T_{hk_{hk}})$.

To take into account the number of executed test case, we denote by $S \in (H_1, .., H_k)$ an index representing the heuristic having the best fault detections effectiveness, then:

for every j such that $1 \leq j \leq (max(h1, .., hk)$

- $\bigvee 1 \leq i \leq k$ Calculate the $APFD(j)_{Hi}$. If $j > hi$ than $APFD(j)_{Hi} = APFD(hi)_{Hi}$
- Determine $MAX(j) = max(APFD(j)_{H1}, ...APFD(j)_{Hk})$
- Define $S = Hs$ such that $APFD(j)_{Hs} = MAX(j)$.
- If there are two (or more) heuristics, Hp and Hq, such that $MAX(j) = APFD(j)_{Hp} = APFD(j)_{Hq}$ then

  if $R(j)_{Hp} \geq R(j)_{Hq}$ then $S = Hp$

  otherwise $S = Hq$

The last rule simply says that in case the two heuristics have the same fault detection effectiveness the heuristic yielding the highest requirements coverage is selected.

The procedure above construct the referring table of the different APFD and provide a guideline for manager to select the heuristic having the best performance depending on the number of test cases to be executed.

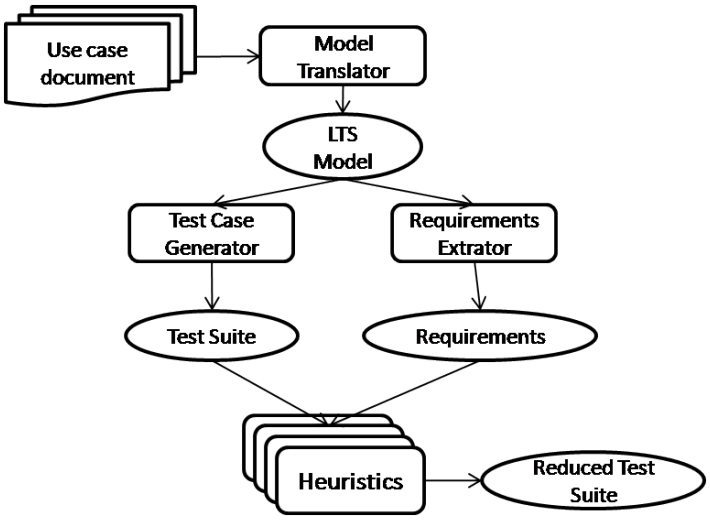


Figure 7.2: Overview of a test suite reduction process

7.3 Case Studies

In two real-world case studies we compared four well-known test suite reduction heuristics – G, GE, GRE and H – from the new viewpoint. The idea is to measure the rate of fault detection in order to show that the techniques based on these heuristics may present a different performance when considering coverage in different intervals of the ordering of selection up to 100% coverage of the test requirements. As mentioned before, the reason is that the heuristics may pick test cases in a different order. Depending on the order, the rate of fault detection can be maximized or not with the first few test cases selected.

7.3.1 Case Studies Design

The heuristics were implemented in Java programming language using the LTS-BT tool [17] as execution environment. These heuristics receive a test suite TS and a satisfiability relation. After processing the suite, the output is the reduced test suite.

Figure 7.2 illustrates the whole process for obtaining a test suite and reducing it using a heuristic. In this figure, the round-edge rectangles represent components of LTS-BT and oval forms represent the artifacts produced by the components. These are used as input for the next component. LTS-BT is a model-based testing tool that automatically generates test cases from use case documents. Initially the use case document is translated into an LTS from which the test suite TS is derived and the requirements mapping $ReqM$ is defined.

Then, for the reduction process, the heuristics receive TS and $ReqM$ as input and after processing the output, the reduced test suite is produced.

For the case studies, we considered transition coverage as requirement. This means that each transition in the LTS represents a test requirement and the test cases that contain a given transition satisfy the corresponding test requirement.

Applications under testing

Two real-world applications provided by Motorola Software Engineers have been selected for the case studies. They are briefly described as follows.

Application 1: TaRGeT is desktop application that supports the model-based testing process, where it automatically generates test cases from use case documents.

Application 2: Direct License Acquisition (DLA) Support is a feature for mobile phone applications that handles acquisition of the WMDRM License (Windows Media Digital Rights Management) for the Windows Media platform. This License provides secure delivery of audio and/or video content.

For each one of the applications, Motorola Software Engineers elaborated the use case documents [48]. From these documents, LTS-BT generated the test suites that have the following particularities.

Application 1: 84 test cases that present redundancy, taking into account the transitions as test requirements, i.e., there are some test cases that cover the same requirement. Therefore the heuristics are able to reduce this test suite.

Application 2: 28 test cases that present redundancy, but each test case has at least one transition that is only covered by it. In this case, the heuristics are not able to reduce the test suite since 100% coverage of the test requirements is needed and this can only be achieved if all test cases are considered.

The test cases were manually executed and the failures captured were associated with faults that can be detected by the suite. For Application 1, 13 faults have been defined, whereas for Application 2, 2 faults have been defined.

Evaluation Metrics

As said before, in general, to evaluate a prioritized test suite taking into account the fault detection, the APFD metric is calculated. However, as discussed in Section 7.1 the APFD metric applied to the complete reduced suite is not suitable here: we have 4 reduced test suites (one for each heuristic) and we need to compare them and analyze which of them presents the best ordering. Therefore, we measure the number of faults detected by the test cases selected up to a given position in the ordering. For the case studies considered, there is exactly one test case associated with each fault. Then, we need to identify whether the test case has been included in the selection in order to count the fault.

For presenting the results, we consider groups of 5 test cases from the first one to be selected up to the last one and count the faults that can be detected up to the group.

Implementation

Output data in the form of the reduced test suites are collected only at the end of the heuristic processing. This means that the instrumentation does not influence the heuristic performance. The code instrumentation was done by inserting Java code to store the information in a data structure that access entries in constant time, because this is indexed by the heuristic and metric names.

The experiment consists in executing the four heuristics for each application. The inputs are the test suite TS, generated from the use case document, and the requirement mapping $ReqM$ to produce the reduced test suite, with the additional information on the position of each test case that is associated with a fault.

This process is repeated 20 times for each heuristic since these heuristics may have a random choice.

7.3.2 Results

Table 7.3 presents the obtained metrics for Application 1: the average of reduced test suite size and the number of faults obtained with 20 executions for each heuristic.

For Application 2, as said before, the heuristics did not reduce the test suite, since each of the 28 test cases has at least 1 transition that is covered only by it. So, all heuristics kept on

Table 7.3: Application 1: Reduced Test Suite Size and Number of Faults.

Heuristic	**Reduced Test Suite Size (Average)**	**Number of Faults (Average)**
Greedy	74	10.4
GE	74	10.4
GRE	74	10.5
H	74.45	10.75

the reduced test suite the same number of test cases of original test suite and, consequently, the rate of fault detection is not decreased.

To evaluate fault detection effectiveness, we construct box plots to show the distribution of faults in 20 executions. In the following figures, the x-axis represents the ordered test cases that are grouped from 5 to 5 and the y-axis represents the number of detected faults. The edges of the box mark the first and third quartiles. The mean value is represented by the central line in each box. The whiskers extend from the quartiles represent the farthest observation lying within 1.5 times the interquartile range. The outliers (unfilled dots) represent the individual values beyond the whiskers. Figures 7.3, 7.4, 7.5 and 7.6 show the box plots obtained for Application 1 respectively for heuristics GE, GRE, Greedy and H. Figures 7.7, 7.8, 7.9 and 7.10 show the box plots obtained for Application 2 respectively for heuristics GE, GRE, Greedy and H.

7.3.3 Threats to validity

To prevent threats to internal validity we have replicated the reported case studies 20 times. So we believe that the resulted reported in Figures 7.3 – 7.10 are reliable as concerns the two applications TaRGeT and DLA. We see however important threats to external validity, i.e., the results observed cannot of course be generalised to other applications different from the two case studies considered here. The main problem, as observed in Section 7.1, is that the comparison of the fault detection capabilities of the test cases involves the knowledge can only be carried out a posteriori. The only conclusion we can safely draw is that the ordering of test cases in the reduced test suite is important, but we cannot deduce which heuristic is

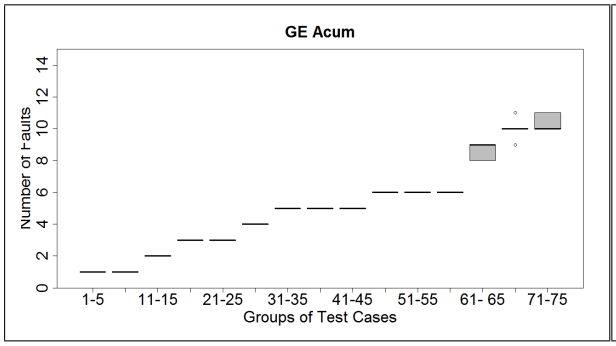


Figure 7.3: Application 1 - GE

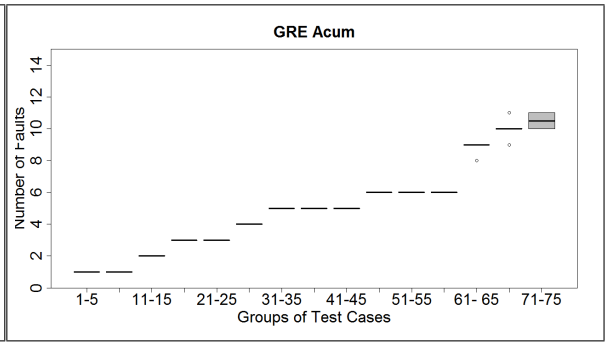


Figure 7.4: Application 1 - GRE

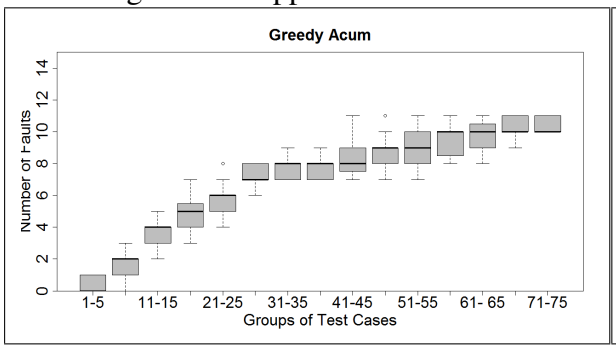


Figure 7.5: Application 1 - Greedy

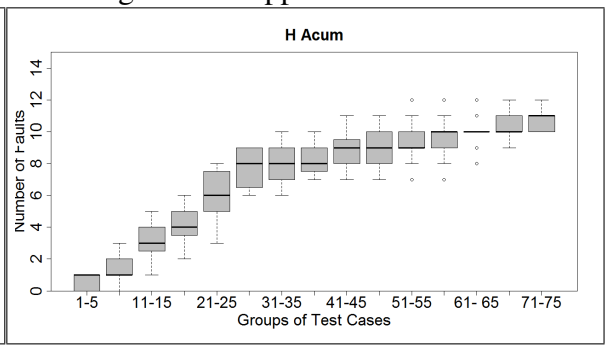


Figure 7.6: Application 1 - H

superior.

7.4 Discussion

For Application 1, it can be noticed, from Figures 7.3, 7.4, 7.5 and 7.6, that:

- 1 - 5 test cases – For this group, GE and GRE present the best fault detection effectiveness (always 1); H and Greedy present a variation between 0 and 1, but the order obtained by H can be better than the one obtained by Greedy because its median value is 1;

- 6 - 20 test cases – For this group, Greedy presents the best fault detection effectiveness (the minimum number of faults is 3 and the maximum is 7, whereas the mean value is 5); H presents the second best fault detection effectiveness (the minimum number of faults is 2 and the maximum is 6, whereas the mean value is 4). The fault detection effectiveness for GE and GRE are the same (always 3);

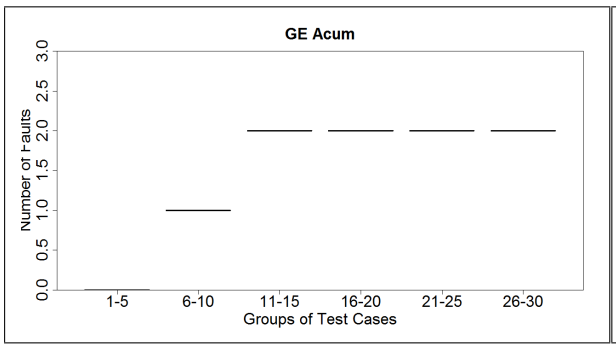


Figure 7.7: Application 2 - GE

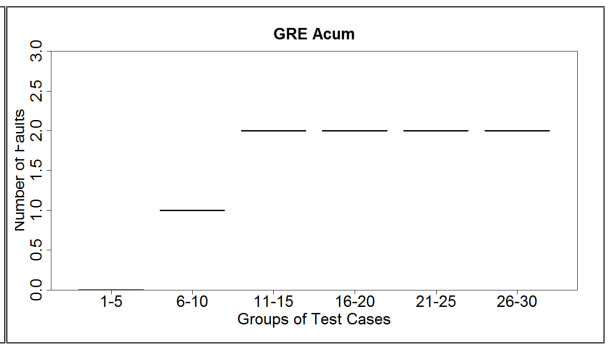


Figure 7.8: Application 2 - GRE

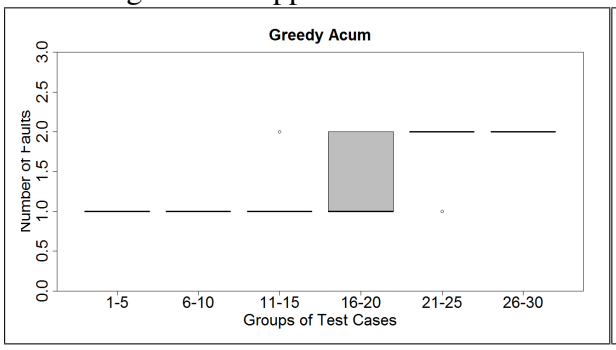


Figure 7.9: Application 2 - Greedy

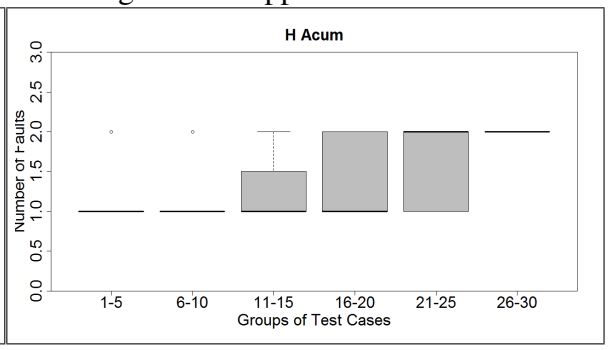


Figure 7.10: Application 2 - H

- More than 20 test cases – For these groups, H presents the best fault detection effectiveness.

Therefore, for this application and the faults considered, if the tester is able to execute only 1-5 test cases, then GE and GRE is the best choice. From 6-20 test cases, Greedy is the best choice, whereas for more than 20 test cases, H is the best choice. Note that GE and GRE present the same behavior until 60 test cases.

For Application 2, it can be noticed, from Figures 7.7, 7.8, 7.9 and 7.10, that:

- 1 - 5 test cases – For this group, H presents the best fault detection effectiveness where the number of detected faults is – most of the time – 1, but it can occur that the 2 faults considered are detected; Greedy presents the second best fault detection effectiveness as it always detects 1 fault. Finally, the behaviour of GE and GRE is the same, none of them detect faults;

- 6 - 10 test cases – For this group, H presents the best fault detection effectiveness where the number of detected faults is – most of the time – 1, but it can occur that the

2 faults considered are detected. Greedy, GE and GRE presents the same behavior, always detecting 1 fault.

- More than 10 test cases – For these groups, GE and GRE presents the best fault detection effectiveness (always 2). About Greedy and H:
  - 11 - 15 test cases – For this group, H presents the highest chance of detecting 2 faults;
  - 16 - 20 test cases – For this group, H and Greedy present the same behaviour;
  - 21 - 25 test cases – For this group, H and Greedy present a similar behaviour (they can detect 1 or 2), but Greedy detects 2 faults at most of the executions.

Therefore, for this application and the faults considered, if the tester is able to execute only 1-10 test cases, then H is the best choice. For more than 10 test cases, GE and GRE are the best choice. It is important to highlight that GE and GRE heuristics present the same behaviour for all positions. From Figures 7.7, 7.8, the first failure is detected after executing 6-10 test cases.

From these case studies, it can be noticed that by analysing the rate of fault detection, it is possible to observe which technique can be more effective, depending on the goals of the tester. Some techniques are more effective when only the first selected test cases can be handled, whereas others improve their performance as more test cases are considered.

Note that the differences on the best technique at different points between the two applications may have been caused by the fact that we are not controlling important factors such as fault distribution and redundancy level. Nevertheless, the purpose of this study is to illustrate the importance and the information that can be gained if reduction techniques are also evaluated from this new viewpoint rather than by the size of the reduced test suite only. This may lead to more effectiveness on selection strategies. As mentioned before, drawing a general conclusion of which heuristic is the best in each circumstance is out of the scope of this study (and probably cannot be ever stated).

7.5 Concluding Remarks

In literature different studies have been developed for comparing prioritization and reduction techniques considering a unique point of view: for instance the efficacy in decreasing test-suite size or the impact on fault detection effectiveness. A common practice is to compare and then select the methodologies for test generation considering that all the test cases will be executed during the testing phase. However, if under budget constraints a lower number of test cases than scheduled have to be run, the test methodology chosen for deriving the test cases during the planning could not be the best choice anymore.

We presented a criterion that generalizes the APFD (Average Percentage of Fault Detection) metric for evaluating the performance of test suite reduction heuristics in subsequent moments of the test activity. The purpose is to analyze how the fault detection effectiveness of the reduction heuristics could change when testers are forced to drastically reduce the number of test cases scheduled for a certain software. Thus we considered four well-known test suite reduction heuristics – G, GE, GRE and H – and measured their rate of fault detection using two real-world applications, namely TaRGeT and DLA, by varying the number of test cases executed and by picking one-by-one the test cases in the generation order of the four heuristics.

The analysis of the case studies evidences how the performance of the four heuristics can be really influenced by the number of the test cases executed: it is possible that the heuristic having the best performance after the execution of few test cases is not the best when the all the reduced test suite is executed. Of course the studies performed so far cannot be used for general conclusions and further investigations are necessary. In particular, since we considered real applications we could not have the complete controlling of important factors: fault distribution and redundancy level. Probably using techniques for seeding faults on the model or comparing applications that have similar level of redundancy could have provided more effective results. However the task of this work was not to conclude which technique is the best in every situation. As shown in our case studies the heuristics having the best performance in a case study have not the same behavior in the other one. Thus no general conclusion can be derived. Our goal was to show that the metrics adopted so far for assessing the relative effectiveness of various reduction approaches probably do not completely match

a reality in which testing phase can be shorten depending on time and cost constraints.

Our results suggested that probably a new way of measuring the performance of various heuristics could be necessary, which takes into consideration the variability of the number of test case to be executed and the faults detected so far. Giving such a new assessment approach is part of our future work as well as to execute more case studies and experiments.

Chapter 8

Review of Work on Test Case Selection and Test Suite Reduction

This chapter presents some related works on strategies for test case selection and test suite reduction. The focus is on solutions that can be automated since our scope of study is on Model-based Testing and system testing.

8.1 Review of Work on Test Case Selection

Related works are presented in this section according to the general kind of the strategy followed to select test cases. For the sake of simplicity, the focus is on works that are more closely related to ours, particularly in the model-based testing area.

Combinatory Selection.

Grindal et al. [34] present an evaluation of strategies for test case selection (the All Combination Strategy, the Each Choice Strategy, the Base Choice Strategy, Automatic Efficient Test Generator and Orthogonal Arrays). These strategies are mostly devoted to testing activities where a number of combinations, between parameters and values, need to be considered.

The combination strategies were evaluated by considering the number of test cases, the number of revealed faults, failure size, and the number of decisions covered. Our strategies, for test case selection (Similarity and WSA) can be applied to refine the test suite produced

by those other strategies, whenever applicable, as some of the selected combinations may still be redundant.

Test Purpose Selection.

Based on a test purpose, that may denote a functionality, or scenario of a functionality to be tested, test case generation algorithms can reduce their search space by considering only sequences that are related to the test purpose. This is a strategy used by the TGV tool [41]. The inputs of this tool are a specification in an Input/Output LTS (IOLTS) model and a test purpose. The outputs are test cases that cover the functionality that was modelled in a test purpose.

Therefore, a selection of part of the model that meets the test purpose is performed. Even though, this can greatly minimize the search space, the redundancy problem is not addressed. LTS-BT tool [17] also implements this idea. The inputs are annotated LTS and the outputs are test cases that cover the test purpose.

Statistical Testing

The Cleanroom software certification process [50] is based on statistical usage testing that consists in selecting a sample of test cases from a Markov chain model whose probabilities are defined to reflect a usage profile. The goal is to define an unbiased test suite that can be more effective for fault detection and also to make reliability estimation possible.

Following a similar idea, the Cow Suite tools focus on specifications in Unified Modeling Language (UML), such as UML sequence and use case diagrams [4] for integration testing. The test case selection is done by considering a weight function, i.e., for each diagram, it is attributed a weight regarding its functional importance.

Also, for testing from UML models, the SPACES tool [3] uses UML diagrams. This tool is used for functional component testing. For each model's transition, a weight is attributed. According to the weights, the most important set of test cases are selected. The main disadvantage of these strategies is the need for attributing weights or probabilities.

These presented strategies do not cope directly with redundancy. Our strategy can be integrated into both tools (Cow Suite and SPACES) to handle redundancy between test cases as a test case selection strategy.

Remarks - Test Case Selection

Here, we present some remarks on the related works concerning test case selection. The Table 8.1 presents a comparison between the test case selection strategies of the literature and our test selection strategies (Similarity selection and WSA).

In general, test selection strategies are guided by some purpose or number of test cases and they do not consider a redundancy concept. As advantages, all of the selection techniques addressed by these works can be fully automated and are based on sound theory. However, MBT presents several limitations in practice that cannot be completely addressed by them.

Test purposes alone can reduce the scope of search, but the TGV tool usually produces a huge number of test cases even for a simple test purpose. Statistical testing (Cleanroom, CowSuites and Spaces) take into account the cost restriction (size - we can define the number of test cases that we wish) and can lead to unbiased choices. However, the redundancy concept is not taken in consideration. Similarity and WSA strategies are adequate for MBT approaches and deal with the redundancy concept. The inputs are LTSs that can be obtained from some specifications such as UML [39].

8.2 Review of Work on Test Suite Reduction

There is a growing interest among the testing community in strategies for test suite reduction. As said before, this is an *NP-complete* problem. Therefore, algorithms based on clusters and some heuristics have been proposed. A number of experimental studies have been conducted to compare different strategies proposed in the literature.

Clustering Test Cases.

Simão et al. [56] present a technique to reduce the test suites for regression testing. This technique uses ART-2A self-organizing neural network architecture to classify test cases (feature vector). These test cases are classified into clusters. When the new source code is available, the modified arcs are evaluated and the most important clusters are selected.

Also to address regression testing, Ma et al. [45] investigate the use of genetic algorithms for defining a minimum test suite for regression testing. The algorithm builds the initial population based on test history, calculates the fitness value using coverage and cost

Table 8.1: Kinds of strategies for selecting test cases compared to the Similarity strategy and WSA strategy

	Goal	**Coverage Criterion**	**Scope of Application and Experimentation**	**Input Required**	**Redundancy Concept**
Combinatory Selection	Select a minimal number of combinations of different factors according to a pattern defined as criterion	All combinations according to a combinatory pattern	Data selection at Integration Testing Level	Table where columns are factors and lines are values associated with the factors	NO
Test Purpose	Select a minimal part of the model that covers the test purpose	All paths in the model that are related to the test purpose	Model-based testing	Labelled Transition Systems: a specification and a test purpose	No
Statistical Testing	Select a given number of test cases randomly guided by a usage profile that weights the importance of certain execution scenarios	A number of test cases to be selected	Model-based testing	Markov Chain	No
Similarity Strategy	Select the most different test cases according to a percentage of test selection goal	A percentage of all-one-loop-paths coverage	Model-based testing	Paths generated from a Labelled Transition System	Yes
WSA	Select the most different and important test cases according to a percentage of test selection goal	A percentage of all-one-loop-paths coverage	Model-based testing	Paths generated from a Labelled Transition System and probabilities	Yes

information, and selectively produces the successive generations using genetic operations until found a minimized test suite.

Clustering based strategies are complementary to ours: if we do not have enough resources to execute all test cases, since the test cases in clusters are very similar, we are able to apply our strategy, within the cluster, and choose only the most different test cases.

Heuristics.

The heuristics are based on the notion of defining the minimal test suites that covers 100% of testing requirements. A set of test requirements is defined for satisfying a given testing objective/criterion. Some heuristics were presented in Chapter 2:

- Greedy Heuristic (G);
- Heuristic H;
- Heuristic GE;
- Heuristic GRE.

As these heuristics focus on coverage of a specific testing objective, they may be too strict and discard test cases that are important for other criteria. Our strategy is more flexible in this sense, since it relies on the general similarity of the test cases that may favor one or more criteria. However, these heuristics can be used to extend our proposed strategy as a criterion for discarding similar test cases.

Experimental works.

A number of experimental studies have been conducted to compare different reduction strategies proposed in the literature. Chen and Lau [20] present the results of a simulation study of four heuristics - H, G, GE and GRE - applied to compute a representative test suite that covers a given testing requirement. The results can be used as a guideline for choosing the most appropriate one for test suite reduction. For using this guideline, it is necessary to know the satisfiability relation and the overlapping ratio (the average number of test cases that satisfy one requirement). Since these results were obtained from simulation data, they may not reflect the real situation [68].

Zhong et al. [68] present an experimental comparison of test suite reduction techniques - H, GRE, genetic algorithm-based approach and ILP-based Approach [11]. The conclusion is that the four techniques can dramatically reduce the test suite size. However H, GRE and ILP have a better behavior since they can reduce more and the test suite sizes are almost the same. The smallest test suite is obtained from ILP-approach. This is applied for test regression, because it requires that error detection information are available.

Wong et al. in [64; 65] present empirical studies conducted to evaluate the effect of reducing the size of the test suite, keeping the block and all-uses criteria coverage. The idea is to evaluate the effect on fault detection of reducing the size of a test suite, while keeping coverage constant. The conclusion is that representative sets have almost the same capability to reveal defects as the original test suite.

Rothermel et al. [55] present empirical studies of test suite reduction for heuristic H [36]. The test suite size and fault detection capability for the reduced test suite are analyzed and reveal that the test suite reduction can compromise fault-detection capability.

Heimdahl and George [37] present an experiment where they investigate the effects of test suite reduction in test suites generated from model based testing. The algorithm used to reduce the test suite randomly and retrieve a test case in a test suite. This test case is added to the reduced set if the coverage criterion is improved. The used coverage criteria were: Variable Domain, Transition, Decision, Decision Usage, MCDC, MCDC usage. They concluded that there is an unacceptable loss in terms of test suite quality.

Zhang et al. [67] present an empirical evaluation of test suite reduction (heuristics G', H', GE' and GRE') for boolean specification-based testing. They show a guideline to choose among these strategies.

Remarks - Test Suite Reduction

To sum up, we can see that test suite reduction has been extensively experimented, but results are not conclusive and also divergent. Results depend on the techniques and choice of requirements and also on the context of application. Further research is needed to identify the most appropriate ones for MBT. Fraser [32; 31] proposed an algorithm to optimize the total costs of a test suite with respect to two factors: the test suite size and the test suite length, using concepts from model checkers. In the resulting test suite, individual test cases

Table 8.2: Kinds of strategies for reduction test suites compared to the Dissimilarity strategy

	Goal	**Coverage Criterion**	**Scope of Application and Experimentation**	**Input Required**	**Redundancy Concept**
Heuristics for Test Suite Reduction	Select a minimal suite that keeps 100% of requirements coverage	All testing requirements determined by a testing objective/criterion	White-box testing	Traceability Matrix (test requirements $\times$ test cases)	Yes
Clustering Test Cases	Grouping of similar test cases in a cluster	All test cases that can be grouped in the cluster based on a fitness function	White-box testing Unit testing	Test cases and code excerpt that is targeted	No
Dissimilarity Strategy	Select the most different test cases	All test requirements	Model-based testing	Paths generated from a Labelled Transition System	Yes

can be longer than in the original test suite.

A related topic is that of test case prioritization. In contrast to test suite reduction strategies which attempt to discard test cases from the test suite, the test case prioritization techniques (such as the ones presented by Elbaum *et al.* [29], Wong *et al.* [63], Korel *et al.* [44], Kim and Porter [43]) that only re-order the execution of test cases within a suite with the goal of maximizing some objective function [51].

The Table 8.2 presents a comparison among the test suite reduction strategies of the literature and Dissimilarity - our test reduction strategy.

For test suite reduction, related works focus on code-based criteria. Test suite reduction

strategies are guided by a test requirement defined in terms of some coverage criteria. They reduce the size of a test suite by fixing some coverage criterion, in this case a chosen criterion is favored, however other important test cases (to reveal faults) can be excluded by being considered redundant in relation to the chosen criteria.

8.3 Concluding Remarks

In general, test selection strategies are guided by some purpose or number of test cases and they do not consider a redundancy concept. However, test suite reduction take in account that concept, but, by reducing the test suite, the fault detection capability can be decreased. The choice of a test requirement can favor or not the fault detection.

Chapter 9

Conclusions and Future Works

This is the concluding Chapter of this thesis. Here, conclusions are drawn in Section 9.1 and, in Section 9.2 the possible future works are presented.

9.1 Conclusions

In this thesis, we proposed a similarity function to measure redundancy among test cases of a test suite in the context of model-based testing, focusing on LTSs and test case generation algorithms guided by structural coverage criteria such as the *all-one-loop-paths* coverage. From that function, strategies for test case selection and test suite reduction are proposed. The idea is to decrease the test suite size by observing redundant parts. We have contribution in two angles, as follows:

Test Case Selection Strategies

Two strategies for test case selection are proposed and evaluated. These strategies are:

- **Similarity-based Selection:**

  The goal is to apply a similarity function to help on the selection of the most different test cases among the set of automatically generated ones according to a target number of test cases (represented by a percentage of the full test suite). To evaluate this strategy, two kinds of coverage criteria (transition and fault coverage) were considered in order to compare it with a random selection strategy that is largely used in practice.

The results show that the similarity strategy is usually more effective to eliminate redundant test cases according to these criteria. In particular, for the conducted case studies (system testing of reactive mobile phone and desktop applications), for test case selection goal equal or higher than 20%, the similarity strategy is clearly more appropriate for selection considering the investigated criteria. When the percentage is too small, the use of a random selection can be more appropriate. An experiment considering a specific configuration of LTS and 50% of the number of test cases shows that by applying the Similarity strategy we are able to obtain better results than by applying Random, considering transition coverage.

- **Weighted-Similarity Approach:**

  The main goal of this approach is to exploit the knowledge (expertise) and the similarity concept to minimize the size of a test suite. Those concepts are combined and a desired number of test cases are selected. To evaluate this strategy two case studies were executed, considering fault and transition coverage.

  The results show evidences that WSA can be an effective strategy for faults detection, however the results depend on probabilities assignment and also on the ability of the tester to pinpoint faults. In other hand, for transition coverage Similarity-based selection presents a better performance - in most of the cases. Considering only the probabilities based approaches, WSA presents generally a better performance since it inherits the similarity principles.

Since these strategies are defined for LTSs, they can be largely applied in practice and adapted to be implemented in different tools. Both strategies are currently implemented in the LTS-BT Tool [17]. TaRGeT tool [48] implements a version of Similarity-based Selection.

Test Suite Reduction

One strategy for test suite reduction is proposed in this work. This strategy, named Dissimilarity, is detailed below.

- **Dissimilarity:** The main goal of this approach is to reach transition coverage as fast as possible, considering the redundancy concept. As how less similar are the test cases, more different they are. A case study and an experiment were performed.

The results of the case study and experiment showed evidences that the Dissimilarity strategy presents the worst percentage of test suite reduction. However, the case study show that Dissimilarity presents the best percentage of the revealed faults. The experiment confirms the results of the case study: Dissimilarity presents the worst percentage of test suite reduction. In this case, considering 75% of transition coverage, as the test requirement.

A new perspective in assessing test suite reduction

We propose a new perspective in assessing test suite reduction techniques based on their rate of fault detection. Our proposal takes in account that, under pressure, testing could be stopped before all tests in the reduced test suite are run, and in such cases the ordering in the reduced test suite is also important, since the APFD of an entire test suite is different from the APFD of a part of the same test suite. Two case studies were performed and they show that, by considering the rate of fault detection, the reduction strategy to be chosen, when time is an issue, might be different from the one presenting the best performance, when testing is completed.

9.2 Future works

Many other problems need to be solved and improvements in our strategies are possible. In the future, we wish to evaluate the strategies more exhaustively by executing more industrial case studies and experiments, by considering different LTSs configurations and fault models.

- Test Case Selection
  - Similarity-based Selection

    A random choice is applied when there is a tie between values of the similarity matrix. This point can be improved by considering, for example, transition coverage or other criteria, rather than a simple random choice. The other point that can be improved is when discarding the test cases. In our proposal, the biggest test cases are kept. We performed a parallel work (presented in Cartaxo et al.

[14]) that kept the smallest. Further investigation addressing this concern can be conducted in future works.

  - Weighted-Similarity Approach

    A guideline to assign the probabilities needs to be proposed, since this is essential to the performance of the WSA strategy. Other point that needs more investigation is how to combine the probabilities that are assigned by the test manager. The current strategy considers the multiplication of the values. When the considered path (test cases) passes by several branches (where the probabilities are assigned), the final weight of the test case can be very small due to the multiplication.

- Test Suite Reduction

  - Dissimilarity

    The fact of placing two test cases at a time in the reduced test suite has hampered the performance, considering the percentage of the test suite reduction. It is probably that, by analyzing the placement of the test cases, one by one, this aspect might be improved.

- A new perspective in assessing test suite reduction

  A new perspective in assessing test suite reduction was presented and a new way to choose a test suite reduction strategy is showed. It is necessary to perform further investigations of this new way of choosing a reduction strategy, since it is still necessary to execute all strategies in order to pinpoint the best one.

Finally, our scope is MBT approaches and system testing. It is probable that these strategies are good also for regression test. Then more experimentation can be done in this direction. Besides, we provide, through LTS-BT, a tool support to execute the proposed strategies presented in this work. Therefore, other strategies can also be implemented in the tool, so they can be executed, compared and analyzed in regards to our strategies.

Bibliography

[1] Bernhard K. Aichernig. Mutation testing in the refinement calculus. *Formal Aspects of Computing*, 15(2-5):280–295, 2004.

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

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

[4] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow suite approach to planning and deriving test suites in uml projects. In *UML'02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 383–397, London, UK, 2002. Springer-Verlag.

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

[6] A. Bertolino, E. Cartaxo, P. Machado, and E. Marchetti. Weighting influence of user behavior in software validation. In *19th International Conference on Database and Expert Systems Application - DEXA 2008 Workshops*, pages 495–500. IEEE Computer Society, 2008.

[7] A. Bertolino, E. Cartaxo, P. Machado, E. Marchetti, and Jo ao Ouriques. Test suite reduction in good order: Comparing heuristics from a new viewpoint. In *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*, pages 13–18. CRIM, 2010.

[8] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[9] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher. *Value-Based Software Engineering*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2005.

[10] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[11] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society.

[12] B.W. Boehm. *Software Engineering Economics*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[13] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, 2008.

[14] E. G. Cartaxo, P. D. L. Machado, F. G. O. Neto, and J. F. S. Ouriques. Usando funções de similaridade para redução de conjuntos de casos de teste em estratégias de teste baseado em modelos. In *Simposio Brasileiro de Engenharia de Software 08 (SBES 2008)*, Campinas, Sao Paulo, October 2008. SBC.

[15] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Automated test case selection based on a similarity function. In *Model-based Testing 07 (Motes'07)*, Bremen, Germany, September 2007. Lecture Notes in Informatics.

[16] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of uml sequence diagrams and labeled transition systems. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1292 –1297, 2007.

[17] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. Oliveira Neto, and Patrícia D. L. Machado. LTS-BT: a tool to generate and select functional test cases for em-

bedded systems. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA, 2008. ACM.

[18] Emanuela Gadelha Cartaxo, Patricia Duarte Lima Machado, and Francisco Gomes Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *STVR Journal of Software Testing, Verification, and Reliability*, 2009.

[19] T. Y. Chen and M. F. Lau. A new heuristic for test suite reduction. *Information & Software Technology*, 40(5-6):347–354, 1998.

[20] T. Y. Chen and M. F. Lau. A simulation study on some heuristics for test suite reduction. *Information & Software Technology*, 40(13):777–787, 1998.

[21] T. Y. Chen and M. F. Lau. On the completeness of a test suite reduction strategy. *COMPJ: The Computer Journal*, 42, 1999.

[22] T. Y. Chen and M. F. Lau. On the divide-and-conquer approach towards test suite reduction. *Inf. Sci.*, 152(1):89–119, 2003.

[23] V. Chvatal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[24] T D Cook and D T Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.

[26] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[27] Rene G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. In *Proceedings of Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, pages 115–128, 1998.

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

[29] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28:159–182, 2002.

[30] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *In Proc. of the Int. Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[31] Gordon Fraser. *Automated Software Testing with Model Checkers*. PhD thesis, Graz University of Technology, October 2007.

[32] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 291–305, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] Robert L. Glass. The software-research crisis. *IEEE Software*.

[34] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[35] Dick Hamlet. When only random testing will do. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 1–9, New York, NY, USA, 2006. ACM.

[36] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[37] Mats P. E. Heimdahl and Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 176–185, Washington, DC, USA, 2004. IEEE Computer Society.

[38] Anders Hessel. Model-based test case generation for real-time systems, 2007.

[39] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: An extendible uml transformation framework. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, Washington, DC, USA, 1999. IEEE Computer Society.

[40] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[41] Claude Jard and Thierry Jeron. 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, 2005.

[42] Paul Jorgensen. *Software Testing: A Craftman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 2001.

[43] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, New York, NY, USA, 2002. ACM.

[44] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, pages 34–43, New York, NY, USA, 2007. ACM.

[45] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. Test-suite reduction using genetic algorithm. In *Advanced Parallel Processing Technologies*, volume 3756 of *Lecture Notes in Computer Science*, pages 253–262, 2005.

[46] John D. Musa. Software-reliability-engineered testing. *Computer*, 29(11):61–68, 1996.

[47] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[48] Sidney Nogueira, Emanuela Cartaxo, Dante Torres, Eduardo Aranha, and Rafael Marques. Model based test generation: An industrial experience. In *1st Brazilian Workshop*

on Systematic and Automated Software Testing - SBBD/SBES 2007, Joao Pessoa, PB, Brazil, 2007.

[49] Alexander Pretschner. Model-based testing. In *Proceedings of International Conference on Software Engineering - ICSE*, pages 722–723, 2005.

[50] Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[51] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. *(ICSM '99) Proceedings. IEEE International Conference on Software Maintenance*, pages 179–188, 1999.

[52] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–184, New York, NY, USA, 1994. ACM.

[53] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[54] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *In Proceedings of the International Conference on Software Maintenance*, pages 34–43, 1998.

[55] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability*, 12:219–249, 2002.

[56] Adenilso da Silva Simao, Rodrigo Fernandes de Mello, and Luciano Jose Senger. A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02*, pages 93–96, Washington, DC, USA, 2006. IEEE Computer Society.

[57] I. Sommerville. *Software Enginnering*. Van Nostrand Reinhold, eighth edition, 2007.

[58] Praveen Ranjan Srivastava. Test case prioritization. *Journal of Theoretical and Applied Information Technology*, pages 178–181, 2008.

[59] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[60] Saif ur Rehman Khan, A. Nadeem, and A. Awais. Testfilter: A statement-coverage based test case reduction technique. In *Multitopic Conference. INMIC '06. IEEE*, pages 275–280. IEEE, 2006.

[61] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[62] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[63] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. *Software Reliability Engineering, International Symposium on*, 0:264, 1997.

[64] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[65] W. Eric Wong, Joseph Robert Horgan, Aditya P. Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software*, 48(2):79–89, 1999.

[66] Xue ying MA, Zhen feng He, Bin kui Sheng, and Cheng qing Ye. A genetic algorithm for test-suite reduction. In *IEEE International Conference on System, Man and Cybernetics*, pages 133–139, 2005.

[67] Xiaofang Zhang, Baowen Xu, Zhenyu Chen, Changhai Nie, and Leifang Li. An empirical evaluation of test suite reduction for boolean specification-based testing (short paper). In Hong Zhu, editor, *QSIC*, pages 270–275. IEEE Computer Society, 2008.

[68] Hao Zhong, Lu Zhang, and Hong Mei. An experimental comparison of four test suite reduction techniques. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 636–640, New York, NY, USA, 2006. ACM.

Appendix A

Similarity based Selection - Case Studies

The content of this chapter have been published in the paper Cartaxo et al. [18].

A.1 Introduction

In order to evaluate the use of the similarity strategy, three different case studies were conducted, where test cases were selected by the similarity strategy and also by a plain random selection strategy. The evaluation focused on assessing whether the similarity strategy (when compared to the random strategy) is suitable for test case selection with the goal of producing test suites of a given smaller size that still keep effective coverage. The random choice strategy basically consists in randomly selecting a test case at a time to be removed from the test suite according to a random function with an even probability distribution. The similarity strategy has been applied by using the LTS-BT tool [17].

The random selection strategy was chosen because, when coverage and diversified choices are of concern, random choice has been accepted to be more effective than deterministic choice in the model-based testing area [50]. Therefore, this strategy has compatible expectations when compared to the similarity ones. Also random testing methods have proven to be more effective in practice in situations where information is lacking to make systematic approaches applicable [35]. When selecting test cases from plain labelled transition systems (the context of this work), which is different from selection from high-level specifications, this situation arises. For example, at system level (the scope of the case studies presented here), systematic approaches will often require assumptions or information such as opera-

tional profile and domain partition. Moreover, random selection is usually applied by other selection, reduction and prioritisation strategies when they reach an undecided situation and get blocked due to a tie among test cases to be selected. This makes random selection a very important and representative selection strategy in practice. Furthermore, random selection is usually considered as fundamental basis of comparison in most of the empirical studies in the area.

For the sake of simplicity when explaining the case studies, the term “percentage of test cases selected" is used with the same meaning as “percentage of *all-one-loop-paths* coverage". This comes from the fact that the selected paths are indeed the test cases.

Next sections present an overview of the case studies applications (Section A.2), how the case studies were defined and conducted (Section A.3), and the results obtained during case studies execution (Sections A.4 and A.5).

A.2 Overview of Case Study Applications

In this section, the case studies chosen are briefly described. They are named, for further reference, as Case Study 1, Case Study 2 and Case Study 3. All of them are reactive applications i.e., applications that react to stimuli of their environment [41]. Particularly, Case Studies 1 and 2 are mobile phone applications, whereas Case Study 3 is a desktop application. The focus is on system testing and test suites are for manual execution. Therefore, the LTS models represent use scenarios of the applications. In summary, the case studies are described as follows:

- Case study 1 is an application for adding contacts in a mobile phone’s contact list;
- Case study 2 is a message application that deals with embedded items. An embedded item can be an URL, phone number or e-mail. For each embedded item, it is possible to execute certain tasks (See Table A.1);
- Case study 3 is an application that generates test cases automatically from use case scenarios - the TaRGeT tool [48].

It is important to remark that, for all case studies, the same basic process has been followed to obtain the LTS model. Basically the process has the following activities:

Table A.1: Embedded item and available Tasks

Embedded item	**Tasks**
URL	Store, Go to
Phone number	Send message, Send voice message, Store, Call
E-mail	Send message, Store

- Writing use cases from natural language requirement documents according to the format defined by Cabral and Sampaio [13];
- Use cases inspection and review for consistency, completeness and conformance;
- LTS model generation that combines the behaviours of all use cases [48]. Basically, each LTS transition represents a use case step. The use case flows may be branched, resulting in different paths in the LTS.

All case studies have also been conducted by the same team. Therefore, this evaluation assumed that the level of details and consistency of the LTS models obtained is similar for all of them. Also, the LTS models cover 100% of the known requirements for each application.

Table A.2 shows some metrics on the case studies in order to illustrate their complexity such as the number of use cases, the number of branch nodes, the maximum level of nested sub-branches, the number of loops and the number of transitions. At system use case level specification, loops are quite rare, unless repetitive interactions are needed in a use. Actually, since all-one-loop-paths coverage has been considered, loops are not significant here. Table A.2 also presents the number of test cases, transitions and faults for each case study considering 100% *all-one-loop-paths* coverage. Faults are defined according to a fault model (Subsection A.3.3). Each fault model makes reference to faults that can be included in an implementation in case programmers do erroneously interpret requirements and, as result, the implementation produces responses that are not in conformance with the LTS model. The reason for this is that it is important to obtain a similar level of fault distribution in all case studies. For Case Study 3, actual faults detected by test execution and debugging were also considered.

Table A.2: Case Studies - Metrics

	Case Study 1	**Case Study 2**	**Case Study 3**
Number of Use Cases	1	33	25
Number of Branches	7	34	21
Maximum level of Sub-Branches	5	1	3
Number of loops	1	0	0
Number of Transitions	121	826	631
Number of Test Cases	24	66	130
Smallest Test Case (Number of Transitions)	6	11	6
Biggest Test Case (Number of Transitions)	29	27	38
Most Common Test Case Size	24	19	14
Number of Most Effective Test Cases (Associated with more faults)	3	33	4
Number of Faults	23	99	127

Table A.3: Faults per Number of Transitions and Test Cases and Test Cases per Transitions (Similarity Rate)

	Case Study 1	**Case Study 2**	**Case Study 3**
Faults/Number of Transitions	0,190	0,120	0,201
Faults/Test Cases	0,958	1,500	0,977
Test Cases/Transitions	0,198	0,080	0,206

Table A.3 presents the rates of faults per number of transitions and test cases as well as the rate of test cases per transition that may characterize the similarity degree of paths in the application and, consequently, the similarity degree of test cases.

For the sake of confidentiality and also for the sake of simplicity, the LTS models of these case studies are not presented in this paper. However, it is important to comment that Case Studies 1 and 3 have more redundant test cases than Case Study 2, since the latter has 3 disjoint groups of functionalities that are handled in isolation. This can also be observed by the rate of test cases per transitions for Case Study 2 (see Table A.3), that is, less test cases for more transitions. The most effective test cases of Case Study 1 are relatively more redundant than the ones of Case Study 3 since the former is about a single and cohesive use case.

A.3 Overview of Case Studies Definition

This section presents the evaluation criteria, path selection strategy and fault model structure defined for conducting and evaluating the case studies.

A.3.1 Evaluation Criteria

In the general research area on test case selection which is the main focus of the case studies, the main criteria used to evaluate the resulting test suite are:

(i) Structural coverage;

(ii) Fault-coverage;

(iii) The number of faults detected by the most effective test contained in it;

Regarding (i), for the case studies presented in this paper that are of system level testing (abstracting from code), transition-based coverage criteria are the most appropriate ones (actually the only ones that make sense), since the focus of this work is on plain labelled transition systems without either guards or datatypes or parallel composition. In this case, the only observable behaviours are transitions that represent outputs. Therefore, transition coverage is a very important metric, since the number of observable behaviours that are

going to be evaluated at testing time can be measured. The most popular transition-based coverage criteria that have been applied to model-based testing are: *all-states* (every state must be visited at least once), *all-configurations* (every configuration of a statechart is visited at least once), *all-transitions* (every transition must be visited at least once), *all-transition-pairs* (every pair of adjacent transition in the model must be traversed at least once), *all-loop-free-paths* (every loop-free path must be traverse at least once) , *all-one-loop-paths* (all the loop-free paths through the model must be visited at least once, plus all the paths that loop once), *all-round-trips* (requires a test for each loop in the model, but do not require that all the paths the precede or follow a loop to be tested) and *all-paths* (every path must be traversed at least once) [61]. For them, it is valid to say that:

- *all-paths* subsumes all of them, but this is not applicable to the case studies since the test generation algorithm only guarantees *all-one-loop-paths* coverage in order to avoid the state space explosion problem;
- *all-transitions* subsumes *all-states*;
- *all-transition-pairs* subsumes *all-transitions*;
- *all-configurations* is not observable in the context of this work;
- *all-one-loop-paths* subsumes *all-round-trips* and *all-loop-free-paths*;
- *all-round-trips* is based on breadth search that selects the shortest test cases guided by this search. Even though, the similarity strategy is independent of whether the generation algorithm is based on depth or breadth search, we used a depth search one for these case studies. Therefore, all-round-trips is not applicable here.

Regarding (ii) and (iii), faults were abstracted by possible observable failures during test execution. Faults are associated with test cases that are capable of exhibiting the corresponding failure behaviour. Finally, for (iii), instead of counting the number of faults revealed by one most effective test case, the most effective test cases were defined and counted (how many of them are included in the selected suite) - a stronger criterion.

In summary, the following criteria were considered to evaluate the test suites obtained from each strategy:

- **Transition-based coverage** - The total number of transitions and pairs of transitions that are covered by considering all of the selected test cases of a given test suite. The idea is to measure whether the strategies keep a reasonable coverage of functionalities.

- **Fault-based coverage** - The total number of faults that are uncovered by the test suite during test execution. For this, versions of the case studies that include faults were considered and also fault models were defined. The idea is to measure whether the strategies preserve the fault detection capability of the original test suite. It was also measured whether the most effective test cases are kept in the minimised suites.

The reason for choosing these criteria is to make it possible to investigate, in the context of the case studies, questions such as: (i) Is test case selection based on the similarity strategy more effective than random selection regarding a given criterion? (ii) What are the limitations of the similarity strategy? (iii) In which circumstances is it more advisable to apply each strategy?

A.3.2 Test Case Selection Goals

For each case study, the similarity and random selection strategies were applied having test selection goals ranging from 5% to 95% (increased by 5) of the test cases. The purpose is to identify which strategy (similarity or random) assures the best selection according to the criteria mentioned above. This is reflected in the final transition coverage and observable failures. Also, due to the random choice that is presented in both strategies, for each path coverage goal, the selection algorithm has been executed 100 times for each strategy. In this case, the average of the values obtained for each metric is considered.

A.3.3 Fault Model

For each case study, test cases were associated with the faults that they are capable of revealing. Then, the number of faults covered by a test suite is computed by the total number of different faults covered by its test cases.

As mentioned before, all case studies are reactive applications. Moreover, their LTS models represent system level scenarios that are derived from use case specifications. Branches

in the LTS represent either different expected inputs to the system or different outputs that can be produced by the system. A choice of input is made by the environment (and this is usually controlled/pre-defined for each test case), whereas the actual output produced during a test case execution is defined by internal behaviour of the system that may or not depending on conditions to be met. Therefore, outputs are the central information to be observed for deciding on the success of a test execution [59].

In this context, a fault model aimed at generic system scope may consider faults that lead to: 1) undesirable feature interactions, 2) incorrect output, 3) abnormal termination, 4) inadequate response time [10]. Since the models considered in the studies do not express requirements on feature interaction and timing, only faults of type 2 and 3 were considered to build the fault models.

At system testing specification level, a fault can only be viewed through the failure that expose it, i.e., an output produced by the system during test execution that is different from the expected ones. Therefore, the first step is to identify possible points of failure and them to assume that one or more faults in the code cause them (assuming also that test cases are sound, i.e., they would not produce false positives). For example, consider the excerpt of the LTS model of Case Study 1 presented in Figure A.1. Transitions leaving Node 1 are input actions (input action have "?" as prefix). If "`?go to main menu`" action occurs, then the expected output is "`!main menu is displayed`" (output actions have "!" as prefix). When a different output is produced, this indicates a failure that is caused by one or more faults at different points in the code. For the sake of simplicity, the procedure for constructing the fault model described below assumes that only one fault is associated with each failure.

One possible way of identifying points of failure, is to consider the possible mutations that could be made upon output transitions at specification level aiming at anticipating design errors that could lead to failures [1]. That is, possible ways of mutating the specification in order to investigate on possible non-conformant implementations that could be produced. Based on this, the general procedure applied to construct the fault model is as follows. The main idea is to mark, in the LTS model, all failure-prone occurrences of output transitions, i.e., the ones where failures are more likely to occur (the expected output is not the one produced). All marks represent a different fault that can be uncovered. Then, test cases

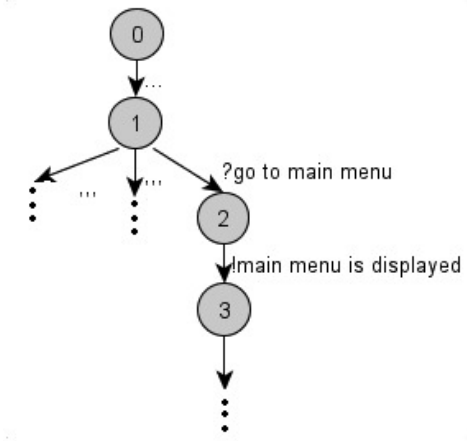


Figure A.1: An excerpt of the LTS model for Case Study 1.

that include the transition are marked as the ones that can reveal the corresponding fault. In summary, the main steps are as follows:

1. Transitions that are part of a branch of outputs are marked. The reason is that these outputs are likely to be defined by a combination of conditions. Since whether a condition is completely faulty cannot be decided (fails for any combination of values in the input domain), the fact that all outputs can be erroneously produced must be considered. Therefore, there is a chance of failure at each one.

2. Transitions that represent a single expected output (only one output transition out of a node) are marked if: 1) there is a chance of abnormal termination; 2) the output is produced as a result of a risk operation; 3) a failure is very likely to occur.

3. A matrix of faults and test cases is constructed where the intersection of fault i and test case t is marked if and only if the output transition represented by i is included in test case t.

The output transition in Figure A.1 is marked in Step 2, because at this point it is likely that by failure a different menu is displayed. This is often caused by pointer manipulation faults.

Even though, for the purposes of the case studies, it is important to define as many faults as possible in order to favour a more coherent evaluation of the results, not all possible occurrences of outputs were considered since this is not realistic for real systems: test cases

are usually designed for revealing specific faults and stable functionality is rarely faulty (Step 2). An example of a fault model defined according to these steps is presented in Table A.4.

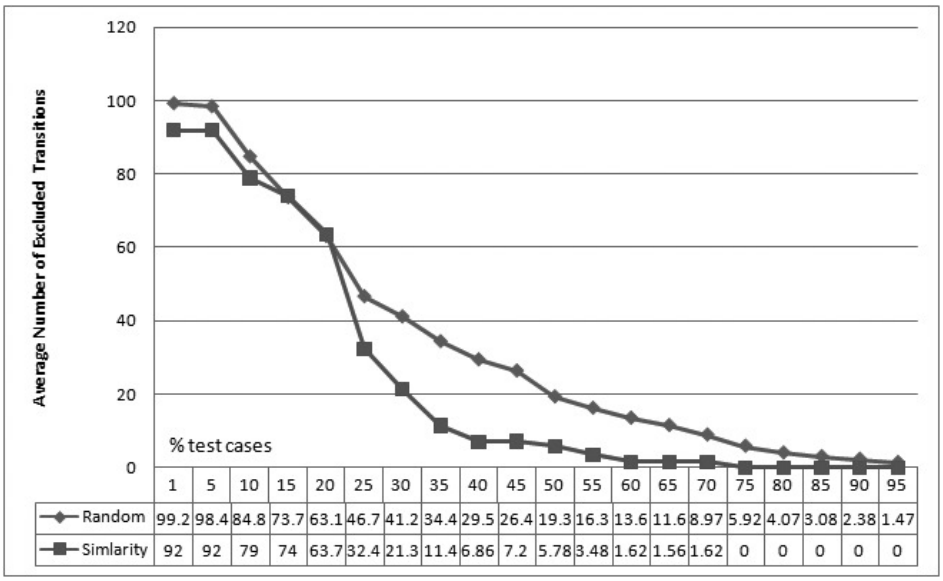
A.4 Case Studies Results

This section presents and discusses the results obtained by considering transition-based and fault-based coverage respectively. For each criterion, results are analysed considering the questions posed at the end of Subsection A.3.1.

A.4.1 Transition Based Coverage

On the LTS model of the applications, the functionalities are represented as labelled transitions. Then, transition coverage when we applied the similarity and random strategies was observed. As transition-based criteria, *all-transitions* and *all-transition-pairs* coverage were considered (see Subsection A.3.1).

Transition Coverage. Figure A.2, Figure A.3 and Figure A.4 illustrate the obtained results for the similarity strategy and the random choice strategy for Case Studies 1, 2 and 3, respectively.



% test cases	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	99.2	98.4	84.8	73.7	63.1	46.7	41.2	34.4	29.5	26.4	19.3	16.3	13.6	11.6	8.97	5.92	4.07	3.08	2.38	1.47
Simlarity	92	92	79	74	63.7	32.4	21.3	11.4	6.86	7.2	5.78	3.48	1.62	1.56	1.62	0	0	0	0	0

Figure A.2: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 1.

Table A.4: Fault Model - Case Study 1. Test cases 04, 12, 18 are the most effective test cases w.r.t. the number of faults covered

Faults/ Test Cases	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
F001	-	-	-	-	-	-	-	-	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-
F002	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-
F003	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-
F004	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X
F005	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
F006	-	-	-	-	-	-	-	X	X	X	X	X	X	X	-	-	-	-	-	-	-	-	-	-
F007	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	-	-	-
F008	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
F009	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-
F010	-	-	X	X	-	-	-	-	-	-	X	X	-	-	-	-	X	X	-	-	-	-	-	-
F011	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-
F012	-	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-
F013	X	X	X	X	X	X	-	-	-	-	-	-	-	-	X	X	X	X	X	X	-	-	-	-
F014	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-
F015	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X
F016	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X
F017	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-
F018	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
F019	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	-	-
F020	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-
F021	-	-	-	-	X	-	-	-	-	-	-	-	X	-	-	-	-	-	X	-	-	-	-	-
F022	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-
F023	-	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
# Faults	4	5	5	6	5	4	4	3	4	5	5	6	5	4	4	5	5	6	5	4	4	1	2	3

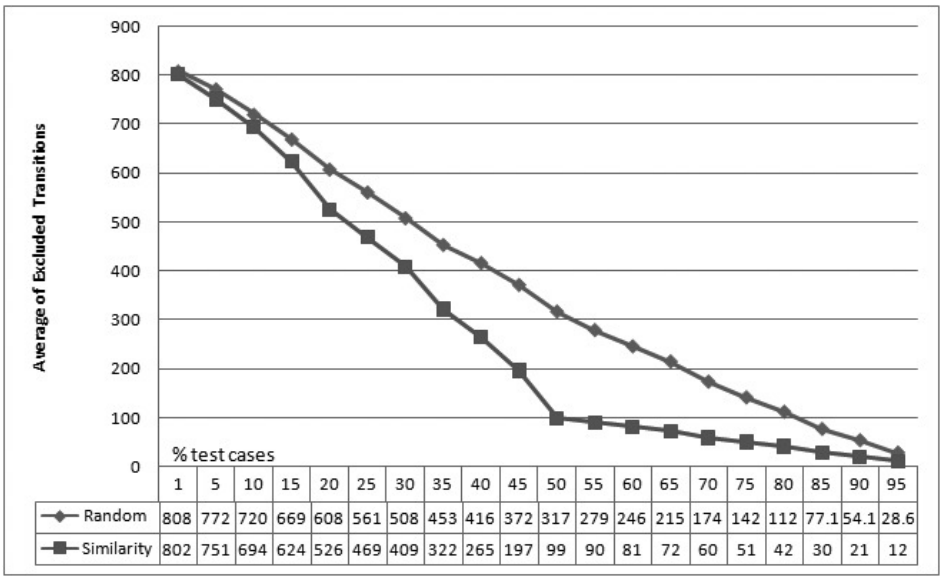
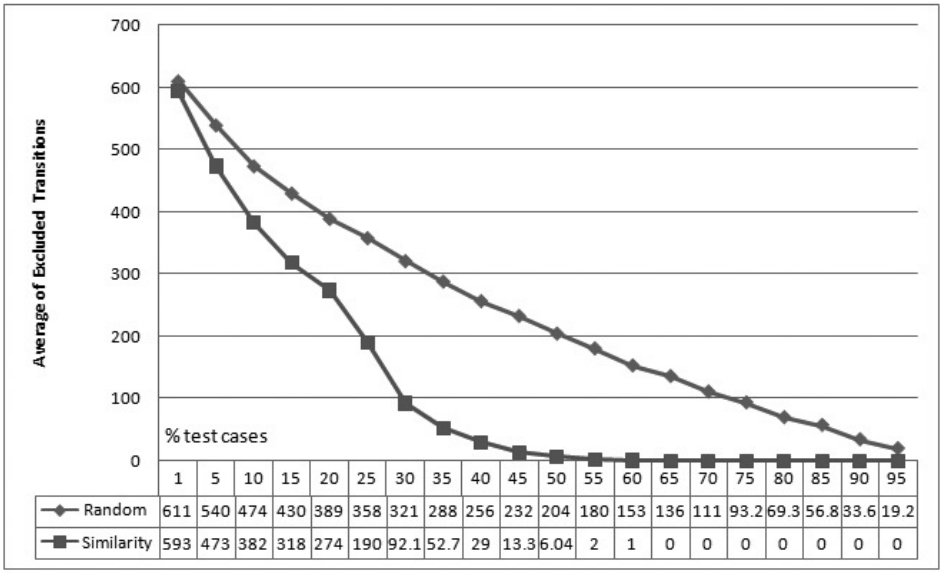


Figure A.3: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 2.

For each of the figures, the x-axis (or abscissa) represents the intended percentage of test cases to be selected and the y-axis (or ordinate) represents the average number of excluded transitions obtained with 100 experiments. The higher the value in the y-axis the worse is the coverage obtained. Therefore, the most effective strategy regarding this criterion is the one that present the lower curve.

For Case Study 1 (Figure A.2), the similarity approach is clearly more effective when 25% to 75% of the test cases are selected, with the best case for similarity achieved at 35%: 91% of transitions are preserved in the selected test suite, whereas only 82% are preserved by the random strategy. From percentages 5 to 20, the performance of the similarity strategy was similar to the random one. In the worst case, at 5% goal, the similarity strategy kept 24% of the transitions, whereas the random strategy kept only 19%. Particularly, in this case study, a number of similar paths have the same size. Therefore, when such strict selection percentages are applied, the choice for discarding one test case is mostly a random one. From 75% of test case selection goal, *all-transitions* coverage is achieved by the similarity strategy.

For Case Studies 2 (Figure A.3) and 3 (Figure A.4), the similarity approach is clearly more effective. The best case for similarity in Case Study 2 is achieved when 50% of the test cases are selected: 88% of transitions are preserved in the selected test suite, whereas



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	611	540	474	430	389	358	321	288	256	232	204	180	153	136	111	93.2	69.3	56.8	33.6	19.2
Similarity	593	473	382	318	274	190	92.1	52.7	29	13.3	6.04	2	1	0	0	0	0	0	0	0

Figure A.4: Average number of excluded transitions by running each test selection strategy 100 times for each test case selection goal - Case Study 3.

only 62% are preserved by the random strategy. The best case for similarity in Case Study 3 is achieved when 35% of the test cases are selected: 92% of transitions are preserved in the selected test suite, whereas only 54% are preserved by the random strategy. Also, for this case study, all-transitions coverage is achieved from a selection of 65% of the test cases.

Regarding the questions put in Section A.3.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? From Figure A.5, the average of the percentage of excluded transitions in all case studies is lower for the similarity strategy. As mentioned above, Case Studies 1 and 3 present more redundant test cases. Therefore, the percentage of excluded transitions is lower for these two case studies when compared to Case Study 2. From 75% of test cases selected, all-transitions coverage is achieved. This indeed shows that the similarity approach is more effective than random choice for these case studies. The reason is that the similarity strategy is more systematic and more precisely pinpoints the similarity, keeping the most different test cases and therefore the best transition coverage, even for Case Study 2 with less redundant test cases than the other ones.

(ii) What are the limitations of the similarity strategy? The similarity strategy performs

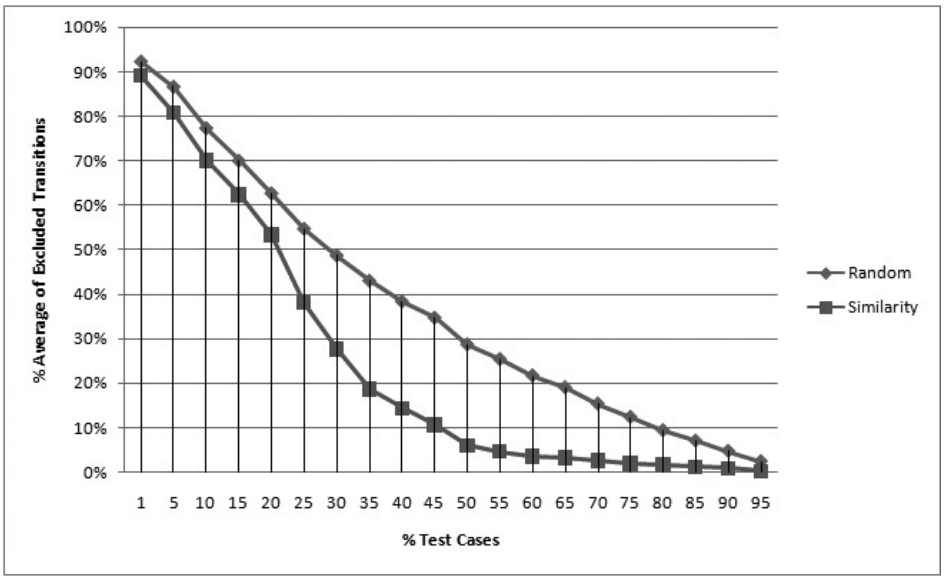


Figure A.5: Percentage of the average number of excluded transitions in all case studies for each test case selection goal.

as well as random selection whenever the criterion for choosing one test case to be discarded cannot be decided. In this case, random selection is applied. This happened in Case Study 1, where some similar paths have the same size and the criterion is based on keeping the one with the biggest size.

(iii) In which circumstances is it more advisable to apply each strategy? By generally comparing the results obtained, the figures suggest that, usually for test case selection goal from 20%, it can more adequate to use the similarity strategy than the random, even in the case where the application presents a considerable number of redundant test cases as in Case Studies 1 and 3, where redundancy may favour the random choice strategy performance.

Transition-Pairs Coverage In order to apply this strategy, all pairs of transitions at each node of the LTS model are computed. The aim is to check whether the strategies preserve combinations of transitions in the test cases. The evaluation was conducted in the same way as for transition coverage and the results are very similar with the same advantages and limitations for each case study. Therefore, for the sake of simplicity, only a summary is presented in Figure A.6. In the best case, the similarity approach preserves 23% more

pairs of transitions than the random approach. For transition coverage, in the best case, the similarity approach preserves 24% more transitions than the random approach.

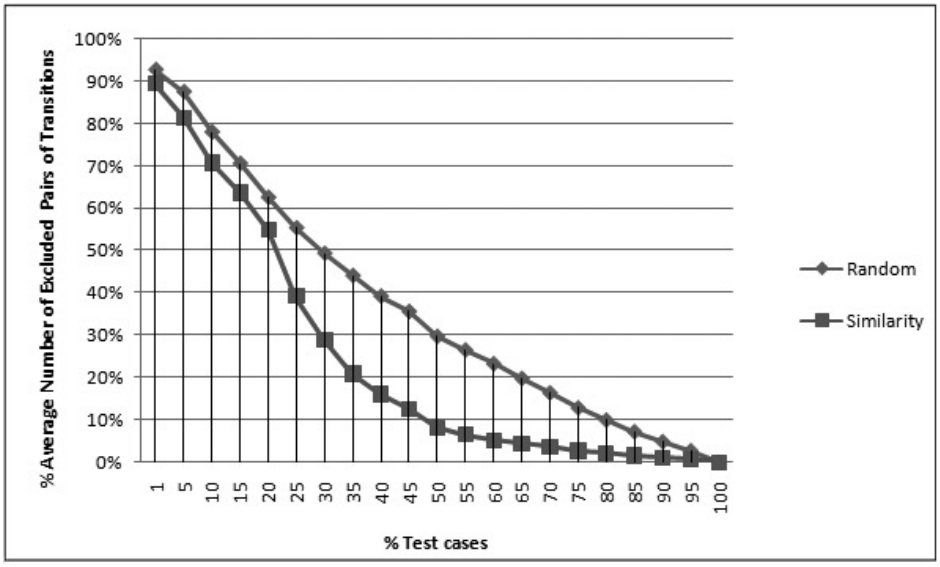
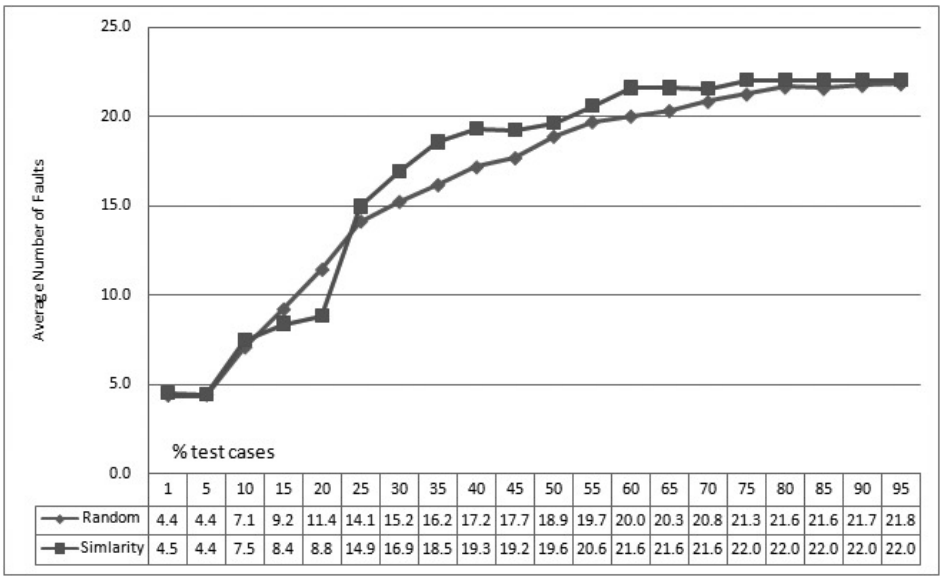


Figure A.6: Percentage of the average number of excluded pairs of transitions in all case studies for each test case selection goal.

A.4.2 Fault-based Coverage

As fault-based criteria, fault coverage and most effective test cases coverage were considered. The results are presented in the sequel.

Fault Coverage Figure A.7, Figure A.8 and Figure A.9 show the results obtained. For each one of the figures, the x-axis represents the intended percentage of test cases to be selected and the y-axis represents the average of covered faults (faults that can be revealed by one or more test cases in the suite) when a test case selection goal is applied (this data was also obtained with 100 experiments). The higher the value in the y-axis the best is the coverage obtained. Therefore, the most effective strategy regarding this criterion is the one that present the upper curve. It is important to remark that since only up to 95% of test case selection goal is considered and fault distribution is such that all test cases are generally associated with at least one fault (see Table A.4), 100% of faults coverage may not be achieved by any of the strategies.



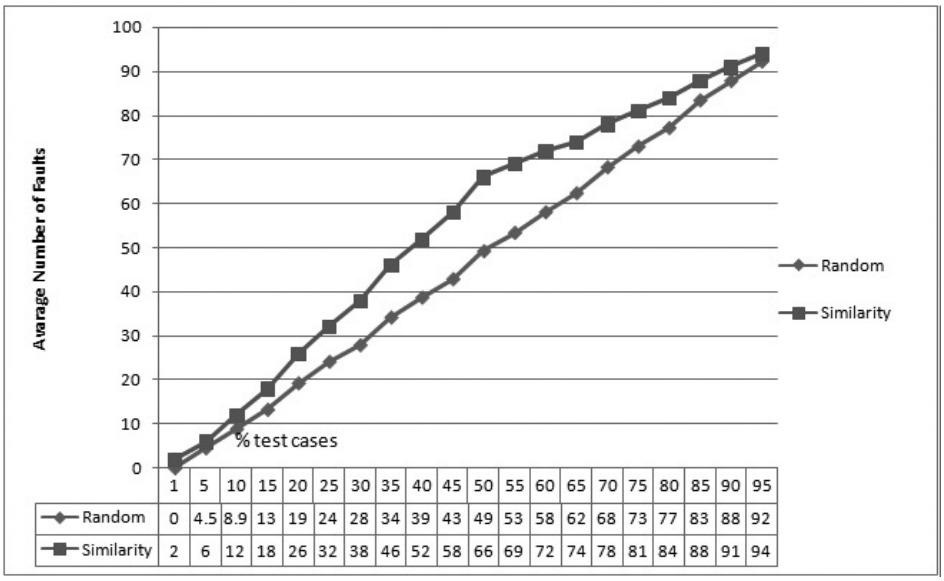
% test cases	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	4.4	4.4	7.1	9.2	11.4	14.1	15.2	16.2	17.2	17.7	18.9	19.7	20.0	20.3	20.8	21.3	21.6	21.6	21.7	21.8
Similarity	4.5	4.4	7.5	8.4	8.8	14.9	16.9	18.5	19.3	19.2	19.6	20.6	21.6	21.6	21.6	22.0	22.0	22.0	22.0	22.0

Figure A.7: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 1.

For Case Study 1 (Figure A.7), the similarity approach is clearly more effective when 25% to 75% of the test cases are selected, with the best case for similarity achieved at 35%: 81% of faults are addressed by the selected test suite, whereas only 70% are addressed by the random strategy. From percentages 5 to 20, the performance of the similarity strategy was worse than or similar to the random one. In the worst case, at 20% selection goal, the similarity strategy addresses only 38% of the faults, whereas the random strategy addresses 50%. The best coverage achieved, 96% or 22 faults out of 23, is reached only by the similarity strategy from 75% of selection goal.

For Case Studies 2 (Figure A.8) and 3 (Figure A.9), the similarity approach is clearly more effective. The best case for similarity in Case Study 2 is achieved when 50% of the test cases are selected: 66% of faults are addressed by the selected test suite of the similarity approach, whereas only 49% are addressed by the random strategy. The best coverage achieved, 95% or 94 faults out of 99, is reached only by the similarity strategy with 95% of selection goal.

The best case for similarity in Case Study 3 is achieved when 45% of the test cases are selected: 92% of faults are addressed by the selected test suite, whereas only 60% are preserved by the random strategy. The best coverage, 100% or 127 faults, is achieved only by the similarity strategy from 65% of selection goal.

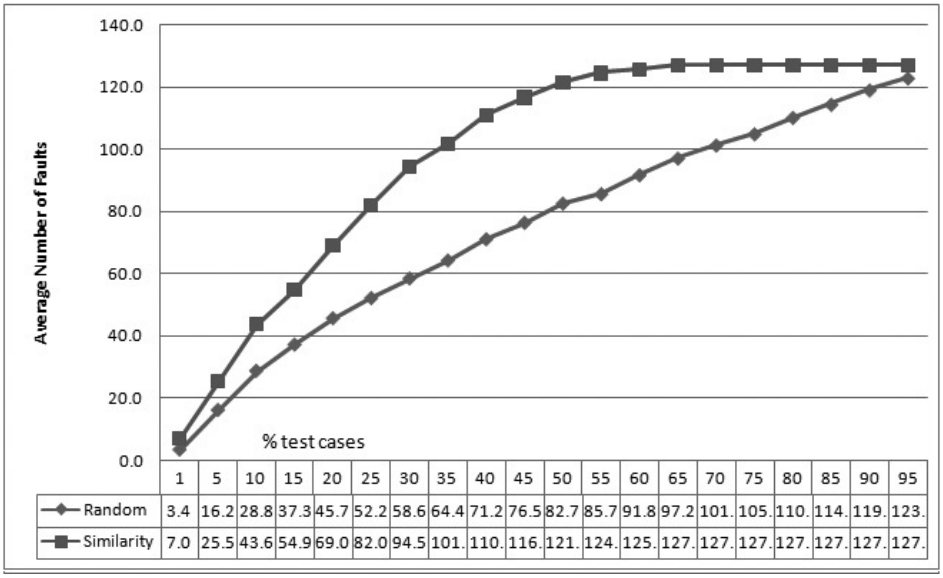


	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	0	4.5	8.9	13	19	24	28	34	39	43	49	53	58	62	68	73	77	83	88	92
Similarity	2	6	12	18	26	32	38	46	52	58	66	69	72	74	78	81	84	88	91	94

Figure A.8: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 2.

Regarding the questions put in Section 4.2.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? From Figure A.10, the average of the percentage of faults addressed by the resulting test suite in all case studies is definitely higher for the similarity strategy from 20% of test case selection goal. As mentioned above, Case Studies 1 and 3 present more redundant test cases. Therefore, the percentage of the number of faults covered is closer to 100% and the best coverage is achieved from 75% of selection goal. The similarity approach is more effective because it can more systematically select the most different faults that are associated with the most different test cases. Case Study 2, with less similar test cases, has also less similar faults. Therefore, the best coverage is only achieved at 95%. Nevertheless, there are clear gains to the similarity approach when compared to the random strategy. However, note that, Case Study 1 presents an open question: in the presence of a severe path coverage constraint (below 20%), is non-deterministic choice more effective than similarity based selection with regard to fault detection? This question is related to a claim of the random testing community (non-determinist selection is more effective than deterministic selection w.r.t. to fault detection) that deserves further investigation.



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
Random	3.4	16.2	28.8	37.3	45.7	52.2	58.6	64.4	71.2	76.5	82.7	85.7	91.8	97.2	101.	105.	110.	114.	119.	123.
Similarity	7.0	25.5	43.6	54.9	69.0	82.0	94.5	101.	110.	116.	121.	124.	125.	127.	127.	127.	127.	127.	127.	127.

Figure A.9: Average number of covered faults by running each test selection strategy 100 times for each path coverage goal - Case Study 3.

(ii) What are the limitations of the similarity strategy? Again, whenever the criterion for defining which test case to discard cannot be decided, then the similarity approach performs as the random strategy. Also, the less redundant test cases are the lower the coverage of faults is. For instance, coverage rate at Case Study 2 is, in average, 13% lower than in the other case studies up to 65%. Furthermore, as the choice for the test case to be discarded, in this paper, is based on the biggest test case, if faults distribution is more prevalent amongst the smaller test cases, then the strategy may not have a good performance. In other words, the performance of the strategy can be influenced by the choice of the criterion to discard the redundant test case.

(iii) In which circumstances is it more advisable to apply each strategy? Comparing the obtained results for similarity and random, it is clear that, in the case studies conducted, for test selection goal bigger than 20%, it is more adequate to use the similarity strategy than the random one, since the number of excluded test cases that failed for different faults is smaller than using the random strategy. In other words, the similarity strategy, by keeping the most different test cases, is more effective in selecting test suites that preserves fault detection capability of the original suite.

It is important to remark that results obtained in these experiments can also be influenced

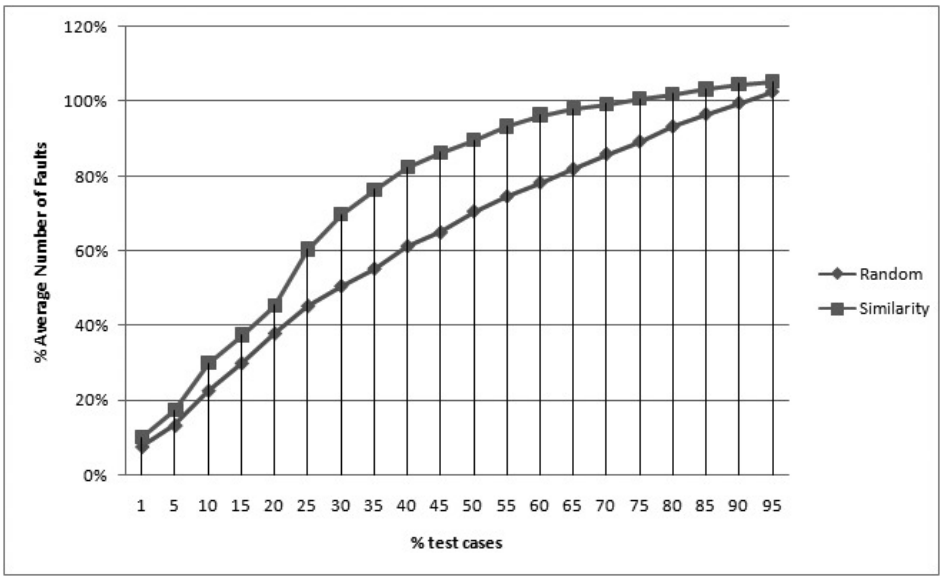


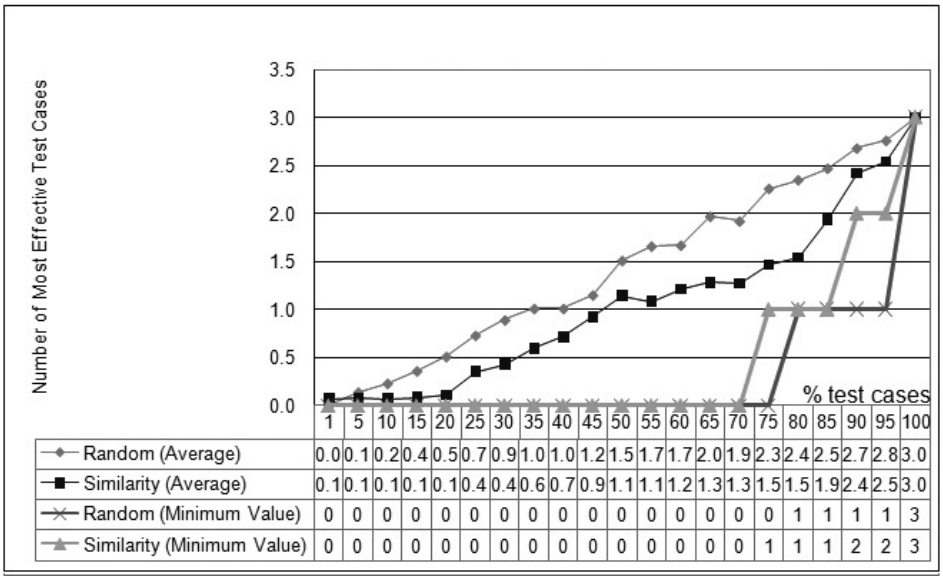
Figure A.10: Percentage of the average number of faults transitions covered in all case studies for each test case selection goal.

by the rate of faults per test cases, faults distribution and the size of the application (measured here by the number of transitions). Table A.3 summarizes these rates. For Case Study 1 (Figure A.7), there is a less significant gain of the similarity strategy over the random strategy when compared to the other case studies. On the other hand, for Case Study 3 with a similar rate of faults per transitions and faults per test case to Case Study 1, the similarity strategy had a considerable gain over the random strategy. This may be explained by the fact that Case Study 1 has more faults distributed among the smaller test cases and then the random strategy had more chances to keep them with selection goals less than or equal to 20%. Furthermore, Case Study 2 has the highest rate of faults per test case, but this is also the case with less redundant test cases and faults. Therefore, similarity is more effective for Case Study 2 than for Case Study 1.

Most Effective Test Cases For each case study, the most effective test cases were selected as the ones that are associated with the biggest number of faults. The goal is to measure how many of the best test cases are preserved at each test case selection goal. Instead of choosing a limiting number of most effective test cases (for example, 1), all test cases that achieve the biggest number of faults in the suite of a given case study were considered. Therefore, the number of most effective test cases is dependent on the case study.

For each one of the figures presented below, the x-axis represents the intended percentage of test cases to be selected and the y-axis represents the minimum number and also the average number of most effective test cases included in the selection. Again, for each selection goal, the strategies were performed 100 times.

For Case Study 1 (Figure A.11) with 4 most effective test cases out of 23, the random approach is more effective in the average case. This can be explained by the fact that the most effective test cases, in this case study, are very similar (diverging by 1-4 transitions only). The similarity approach constraints its search space by eliminating redundancy according to each selection goal, whereas the random approach freely chooses among all possible test cases for each selection goal. Therefore, the most effective test cases for Case Study 1 cannot be included in the resulting test suite when the similarity strategy is applied. However, note that, in the worst case (considering the minimum number of the most effective test cases selected at one or more of the 100 trials), the similarity approaches presented a better performance, from 75% of selection goal at least one of them is included.

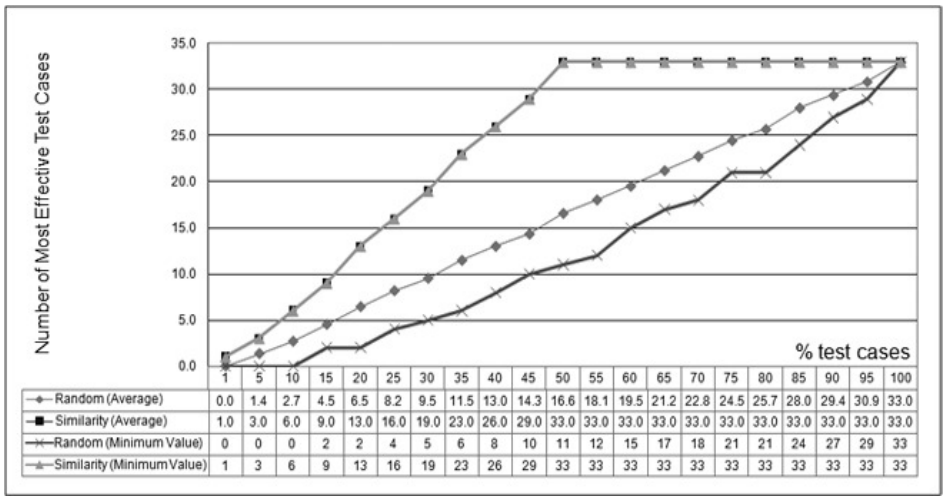


% test cases	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.0	0.1	0.2	0.4	0.5	0.7	0.9	1.0	1.0	1.2	1.5	1.7	1.7	2.0	1.9	2.3	2.4	2.5	2.7	2.8	3.0
Similarity (Average)	0.1	0.1	0.1	0.1	0.1	0.4	0.4	0.6	0.7	0.9	1.1	1.1	1.2	1.3	1.3	1.5	1.5	1.9	2.4	2.5	3.0
Random (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	3
Similarity (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	2	2	3

Figure A.11: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 1.

It is also important to remark that this is the only case study where the curves of the similarity strategy for the average and the minimum value are different. The reason is that the study has similar test cases of the same size. Then random choice is very frequently applied for choosing the test case to be discarded.

For Case Study 2 (Figure A.12), with 33 most effective test cases out of 66, the similarity approach is clearly more effective, even in the worst case that coincides with the average one. In this case, the most effective test cases are completely different. Therefore, the similarity approach, by eliminating redundancy and keeping the biggest test case, selected all 33 most effective test cases from 50% of selection goal.



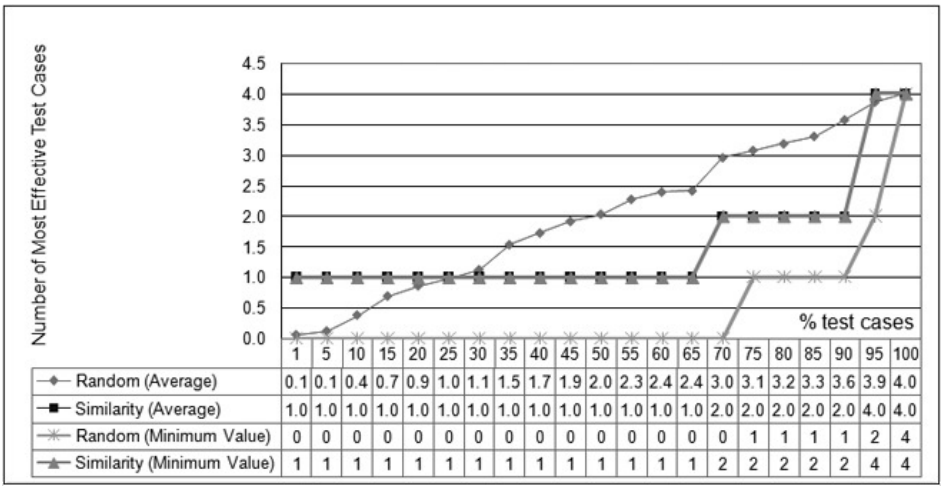
	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.0	1.4	2.7	4.5	6.5	8.2	9.5	11.5	13.0	14.3	16.6	18.1	19.5	21.2	22.8	24.5	25.7	28.0	29.4	30.9	33.0
Similarity (Average)	1.0	3.0	6.0	9.0	13.0	16.0	19.0	23.0	26.0	29.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0	33.0
Random (Minimum Value)	0	0	0	2	2	4	5	6	8	10	11	12	15	17	18	21	21	24	27	29	33
Similarity (Minimum Value)	1	3	6	9	13	16	19	23	26	29	33	33	33	33	33	33	33	33	33	33	33

Figure A.12: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 2.

Finally, for Case Study 3 (Figure A.13), with 4 out of 130 most effective test cases, the random approach is more effective in the average case, from 25% of test selection goal. As Case Study 1, the most effective test cases are similar (diverging by 4-6 transitions), but not as much as in Case Study 1. Therefore, there is a gain for the similarity approach up to 20%. From this point on, the random strategy gets more chance to select the 4 out of 130. Nevertheless, note that the similarity approach (both average and minimum number) is more effective than the worst case for the random strategy, guaranteeing that, at least one is selected for each selection goal, even the more restricted ones.

Regarding the questions put in Section A.3.1:

(i) Is test case selection based on the similarity strategy more effective than random selection regarding this criterion? Concerning detection of all most effective test cases, the similarity strategy is more effective whenever the most effective test cases are not so similar since the strategy focus on selecting the most different ones (Case Study 2). Nevertheless, if the worst case for random selection is considered, similarity can



	1	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
Random (Average)	0.1	0.1	0.4	0.7	0.9	1.0	1.1	1.5	1.7	1.9	2.0	2.3	2.4	2.4	3.0	3.1	3.2	3.3	3.6	3.9	4.0
Similarity (Average)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	2.0	2.0	2.0	2.0	4.0	4.0
Random (Minimum Value)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	2	4
Similarity (Minimum Value)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	4	4

Figure A.13: Average and minimum number of the most effective test cases that are included for each test case selection goal - Case Study 3.

be more effective since this strategy is more deterministic and has shorter variations on results (Case Studies 1 and 3). Indeed, from Figure A.14, note that the similarity approach is more likely to select at least one of the most effective test cases for all selection goals. Furthermore, for severe selection percentages such as the ones from 5 to 20%, similarity can present better results than random selection (Case Study 3).

(ii) What are the limitations of the similarity strategy? The limitations of the strategy are on: a) the criteria for discarding test cases; and b) the similarity degree of the most effective test cases. The criteria adopted in this paper for discarding test cases (the biggest test case) may not be directly related to the criteria for choosing the most effective test cases (here are the ones that cover the biggest number of faults). In this situation, the strategy is not precisely guiding the choice towards the goal and the results may not be reasonable unless the most effective test cases are not similar. In this case, the similarity strategy will preserve them depending on the selection goal.

(iii) In which circumstances is it more advisable to apply each strategy? The similarity strategy is recommended whenever the criterion for defining the most effective test cases can influence or is related to the criterion for selecting the test cases to be discarded. If the former are semantics ones, then one possibility is to prioritise test cases, for instance, following the approach proposed by Bertolino *et al.* [6]. The random

approach is more recommended otherwise, however, to avoid the worst case, it should be applied several times. If a guarantee that at least one most effective test case is preserved is important, than the similarity strategy is recommended.

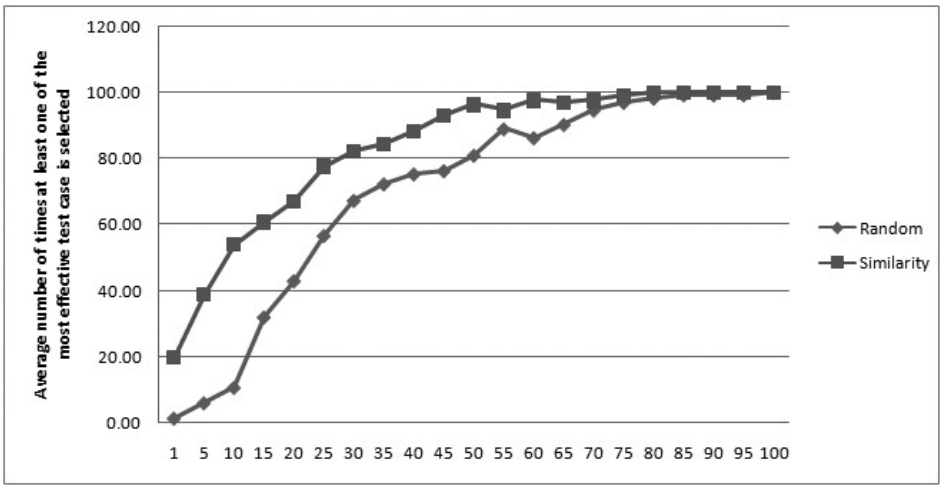


Figure A.14: Average number of the times (out of 100 executions of each strategy) at least one of most effective test cases is selected in all case studies for each test case selection goal.

A.5 Case Studies - General Remarks

Concerning transition-based coverage, for the case studies conducted, the similarity approach can be more effective than the random strategy. There are considerable advantages from 20% of test case selection goal (see Figures A.5 and A.6).

Even for very restrict selection goals such as 5%, the strategy can be more effective. Full all-transitions and all-transition-pairs is only achieved by the similarity approach.

Concerning fault coverage, for the cases studies conducted, the similarity approach has also a superior performance. For all case studies, the highest coverage is only achieved by the similarity approach. There are also considerable advantages from 20% of test case selection goal. However, the random choice had a better performance for Case Study 1 with test case selection goal of less than or equal to 20%. The similarity strategy excludes the most similar test cases, this means that for cases where it is necessary to exclude several test cases, the strategy begins well, but the last test cases that will be excluded, usually, does not have similarity, so the criteria to be used is the path length. Moreover, the criterion for discarding

similar test cases can also influence the results by guiding the strategy for selecting more faults.

Even though this is not always more effective when detecting all of the most effective test cases, the similarity approach can be more precise in detecting at least one of them. The limitations for selecting all of them occur when their degree of similarity do not allow them to be included in the selected test suite such as for Case Study 1 (see similarity of faults on Table A.4) and Case Study 3.

It is important to remark that test cases and faults are considered to be equally relevant. Different results may be reached if experiments are conducted by considering a number of attributes that may add value to specific test cases and also to the faults. Also, the similarity approach opts for keeping the biggest test case in the test suite for the sake of improving transition coverage. This decision is closely related to the fact that the focus of this work is on functional testing, where thorough coverage is critical. However, it may also be interesting to investigate the strategy when the smallest test case is kept.

Another key point to consider when analysing the results is the number of faults defined in the fault models. However, the number of faults do not influence on the results obtained: increasing or decreasing the number of faults has a very similar effect on both strategies. To confirm this claim, an experiment was performed aiming at observing the behaviour of both strategies when from a few to several faults are incrementally included in the model. The results of this experiment (that are not included in this paper for the sake of space) show that the number of faults does not bias the results. However, with more faults, clarity of the results is improved.

Regarding computational complexity, the test case generation algorithm has exponential complexity, but state space explosion is handled by requiring only *all-one-loop-paths* coverage. The selection algorithm, the main focus of this paper, is $O(n^3)$, where n is the number of test cases. For the case studies conducted, the time consumed for each algorithm (generation and similarity selection) considering one execution of the similarity strategy with 50% selection goal is presented in Table A.5. As mentioned before, the suites of these case studies are for manual test execution. By considering that the average time for executing a test case is 2 minutes, it would roughly take 48 minutes, 132 minutes, and 260 minutes for executing 100% of the test cases for Case Study 1, 2 and 3, respectively. By selecting 50% of the test

Table A.5: Execution time for full test case generation and also one execution of similarity selection algorithms with 50% test case selection goal.

	Case Study 1	**Case Study 2**	**Case Study 3**
Test Case Generation	16ms	15ms	32ms
Similarity Strategy (Computing the Similarity Matrix)	16ms	110ms	484ms
Similarity Strategy (Selecting Test Cases)	0ms	0ms	31ms
Total	30ms	125ms	547ms

cases, half of the time is saved with only an additional test selection time of 32ms, 125ms, and 547ms for Case Study 1, 2 and 3, respectively. Obviously, in practice the complexity of test execution grows with the size of the test suite. Also, test selection may require further analysis, for instance, running the selection algorithm more than once. Therefore, there are other gains and losses to be considered than only counting the exact time for executing each test case. Nevertheless, the difference on the magnitude of the numbers points out that the similarity strategy can be practical and indeed improve test productivity and reliability.

Appendix B

LTS Generator

This Appendix presents the generator that was implemented to generate the inputs - LTS models - for our experiments. The goal of this generator is to generate different LTS models using a specific configuration. This configuration considers:

- **Depth:** The depth of the LTS. It is calculated by consider the biggest path (from initial state to final state - without loops);
- **Number of Loops:** One loop is an edge that goes back to any prior state or to itself;
- **Number of Forks:** A fork is a state with more than one outgoing transitions;
- **Number of Joins:** A join is a state with more than one incoming transitions.

Loops, forks and joins can add redundancy in an LTS model. The intention is to construct different LTS models that contains the specified configuration. The steps to construct the LTS models can be seen in Algorithm 4.

The inputs of this algorithm are: the number of LTS models that will be generated; and the configuration. The configuration is a number for each of the following elements: Depth, number of loops, number of joins, and number of forks.

Firstly, an initial sequence is generated following the depth (line 1) constraint. For example, if the depth is 3, the initial sequence has 4 states (0, 1, 2, 3) and 3 transitions (0 to 1, 1 to 2 and 2 to 3). The next step is to aggregate the new structures to the initial sequence (lines 2 - 11). For this, all structures are placed into a list (structures), e.g. if $NumberOfLoops = 2$, $NumberOfJoins = 1$, $NumberOfForks = 3$, then

```
   input  : NumberOfLTSModels, Depth, NumberOfLoops, NumberOfJoins, NumberOfForks
   output: LTSModels
1  buildSequenceOfTranstions(depth);
2  structures = getStructuralPatterns(NumberOfLoops, NumberOfJoins, NumberOfForks);
3  shuffle(structures);
4  for each structure:structures do
5      switch structure do
6          case join
7              putAJoin(depth);
8          case fork
9              putAFork(depth);
10         case loop
11             putALoop(depth);
```

Algorithm 4: LTS Generator - Algorithm

$structures = \{loop, loop, join, fork, fork, fork\}$ (line 2). These structures are shuffled (line 3) to increase the diversity of the generated LTS models, since aggregating these structures in different orders, increases the probability of generating different LTS models.

After shuffling the structures, each structure is aggregated to an actual LTS, observing some constraining rules:

- **Depth:** It is not allowed to aggregate any structure that violates the maximum depth;
- **Join:** Two states are randomly chosen, and these states can not be adjacent. From each of these state, one new transition is created, both going to the new state (the joining state);
- **Fork** - A state is randomly chosen, and two new transitions (and states) are created, outgoing from the chosen state;
- **Loop** - Two states are chosen randomly, however the loop must be placed from the deeper state to other state selected.

Appendix C

Experiment - Test Suite Reduction

This Appendix shows the comparison among the strategies (G, GE, GRE and H). The variables are:

- **Dependent:** The Reduced Test Suite Size (RTSS).
- **Independent:** The test requirement percentage; the configuration chosen for the depth and amount of structures (loops, forks and joins) in the objects; and the strategies for test case selection (factor). For this factor, there are 5 levels: G, GE, GRE, H and Dissimilarity (DSim).

The experiment definition is formalized as following:

- A null hypothesis (H_0) - $RTSS_G = RTSS_{GE} = RTSS_{GRE} = RTSS_H$: All techniques have a similar behavior in relation to the reduced test suite size;
- An alternative hypothesis (H_1) - $RTSS_G \neq RTSS_{GE} \neq RTSS_{GRE} \neq RTSS_H$: All techniques have a different behavior in relation to the reduced test suite size.

Each technique was executed 200 times. During the analysis we considered a confidence level of 95% (i.e. a significance level - α - of 0.05). The first step is to analyze if the obtained data, for each strategy, present a normal distribution. For this, we applied the Anderson-Darling normality test, using the Minitab tool[1]. The results can be seen in Figures C.3, C.2, C.1 and C.4.

[1] http://www.minitab.com/

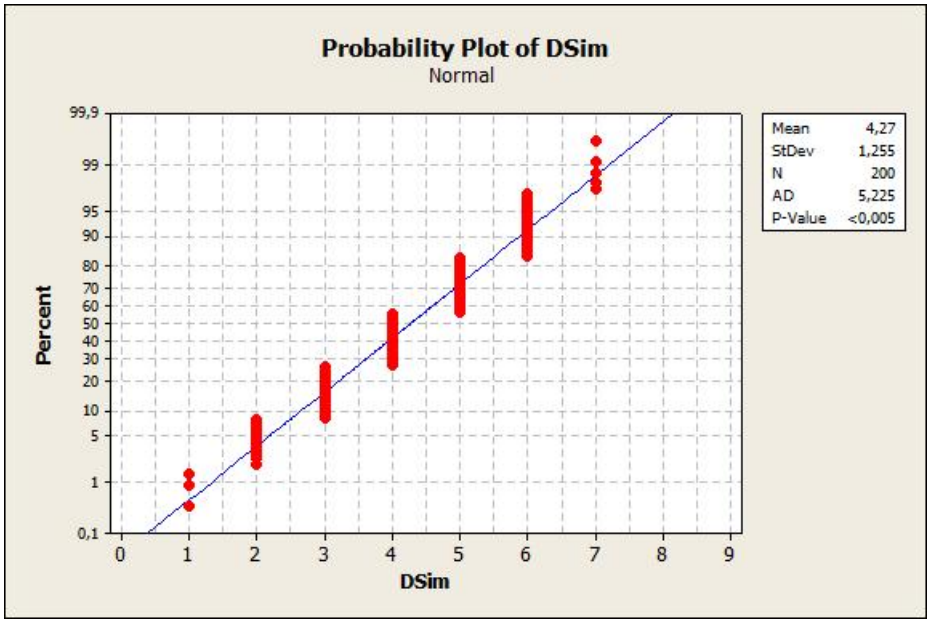


Figure C.1: Anderson-Darling normality test - GRE

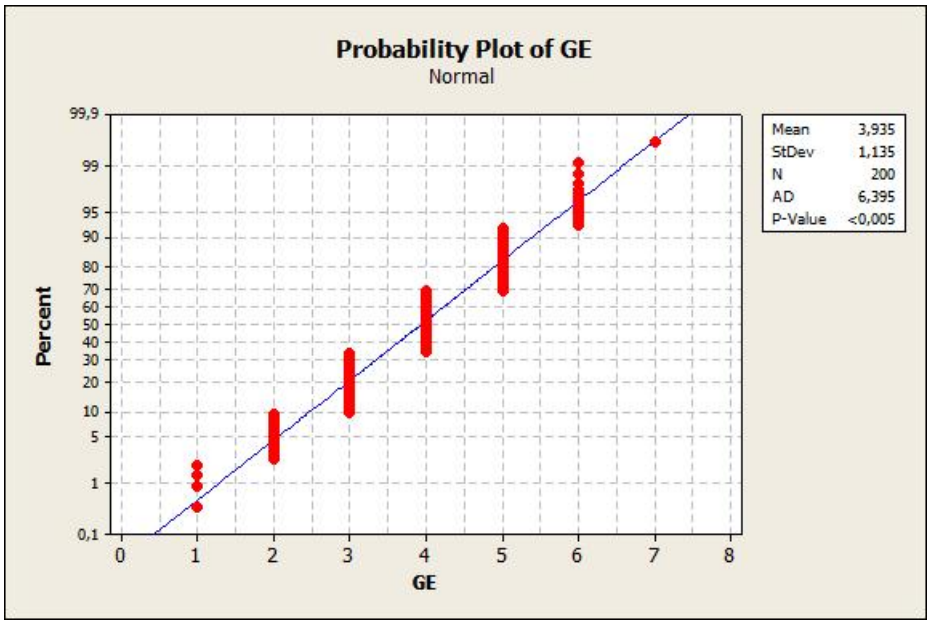


Figure C.2: Anderson-Darling normality test - GE

Observe that all $p - values$ are lower than the significance level ($\alpha = 0,05$), then the data do not show a normal distribution. Thus, it is required to apply a non-parametric tests. Since, there is only 1 factor and more than 2 treatments, a Kruskal-Wallis is applied to check the null hypothesis.

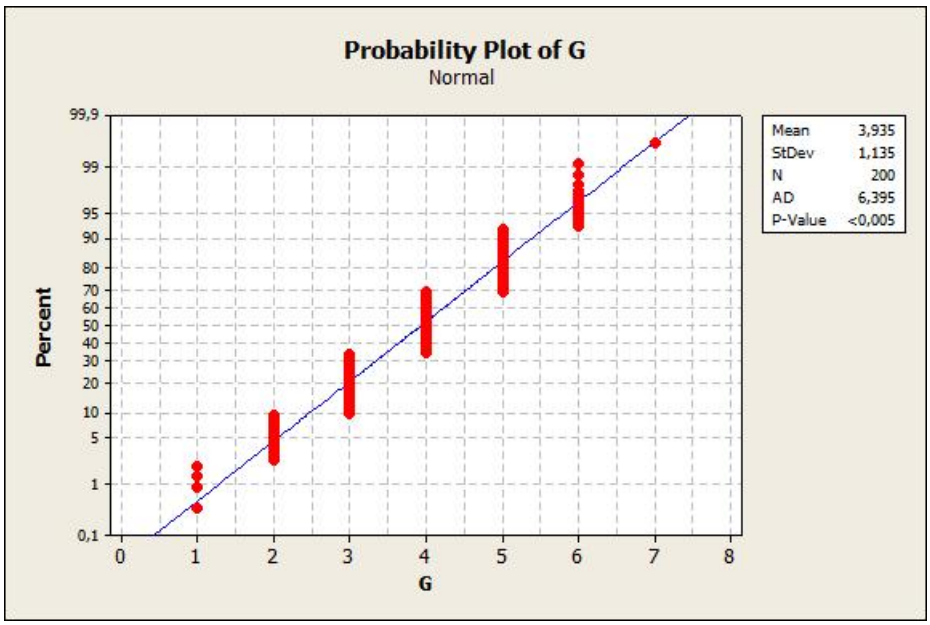


Figure C.3: Anderson-Darling normality test - G

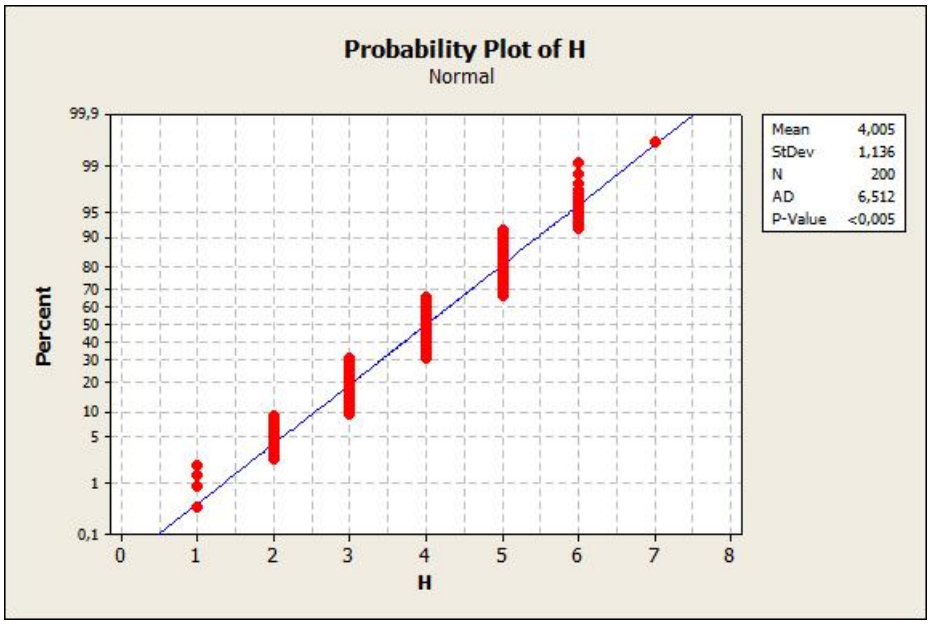


Figure C.4: Anderson-Darling normality test - H

The sample medians for the four treatments (G, GE, GRE, H) were calculated and are the equal: 4.000.

The test statistic (H) had a $p - value$ of 0.881. Since the $p - value$ is higher than the significance level ($\alpha = 0,05$), the null hypothesis can not be rejected. In other words, the obtained results, from the strategies, using a confidence level of 95%, indicate that G, GE,

Table C.1: Kruskal-Wallis Test - G, GE, GRE, H

Factor	N	Median	Ave Rank	Z
G	200	4.000	396.6	-0.27
GE	200	4.000	396.6	-0.27
GRE	200	4.000	396.6	-0.27
H	200	4.000	412.1	0.82
Overall	800		400,5	
H = 0.67 DF = 3 P = 0.881				

GRE and H can not be considered different.