



CBSOFT 2012

III Congresso Brasileiro de Software: Teoria e Prática
3rd Brazilian Conference on Software: Theory and Practice

23 a 28/09/2012 | Natal-RN | Brasil



XXVI Simpósio Brasileiro de Engenharia de Software (SBES 2012)	XVI Simpósio Brasileiro de Linguagens de Programação (SBLP 2012)
XV Simpósio Brasileiro de Métodos Formais (SBMF 2012)	VI Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2012)

Anais
SAST 2012
VI Workshop Brasileiro de Teste de Software
Sistemático e Automatizado



SAST 2012

VI Workshop Brasileiro de Teste de Software Sistemático e Automatizado

23 de setembro de 2012

Natal - RN - Brasil

ANAIS

Volume 13
ISSN: 2178-6097

Coordenação do SAST 2012

Roberta de Souza Coelho

Ariol Claudio Dias Neto

Coordenação do CBSOFT 2012

Nélio Cacho - Coordenador Geral

Gibeon Aquino - Vice Coordenador

Frederico Lopes - Vice Coordenador

Realização

Universidade Federal do Rio Grande do Norte (UFRN)

Departamento de Informática e Matemática Aplicada (DIMAp/UFRN)

Promoção

Sociedade Brasileira de Computação (SBC)

Patrocínio

CAPES, DataPrev, CNPq, Google, Microsoft Research, Sebrae-RN, INES, FAPERN, NatalCard

Apoio

Instituto Metrópole Digital, Cabo Telecom



SAST 2012

6th Brazilian Workshop on Systematic and Automated Software Testing

September 23, 2012
Natal - RN - Brazil

PROCEEDINGS

Volume 13
ISSN: 2178-6097

SAST 2012 Chairs

Roberta de Souza Coelho
Arilo Claudio Dias Neto

CBSOFT 2012 Chairs

Nélio Cacho - General Chair
Gibeon Aquino - Co-chair
Frederico Lopes - Co-chair

Organization

Universidade Federal do Rio Grande do Norte (UFRN)
Departamento de Informática e Matemática Aplicada (DIMAp/UFRN)

Promotion

Brazilian Computing Society (SBC)

Sponsorship

CAPES, DataPrev, CNPq, Google, Microsoft Research, Sebrae-RN, INES, FAPERN, NatalCard

Support

Instituto Metrópole Digital, Cabo Telecom

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Especializada do Centro de Ciências Exatas e da Terra – CCET.

Congresso Brasileiro de Software: Teoria e Prática (3. : 2012: Natal, RN).
SAST 2012 : anais [do] VI Workshop Brasileiro de Teste de Software Sistemático e Automatizado / III Congresso Brasileiro de Software: Teoria e Prática; coordenação [de] Nélio Cacho, Gibeon Aquino, Frederico Lopes; coordenação do SAST 2012 [de] Roberta Coelho, Arilo Dias Neto – Natal, 2012.
Xi, 82 p.

Evento no dia 23 de setembro de 2012.
Realização da Universidade Federal do Rio Grande do Norte. Promoção da Sociedade Brasileira de Computação.
Publicação composta por 14 volumes, sendo este o volume 13.
ISSN : 2178-6097.

1. Engenharia de software. I. Cacho, N., coord. II. Aquino, G., coord. III. Lopes, F., coord. IV. Coelho, R., coord. V. Dias Neto, A., coord. VI. Título.

CDD 22. ed. – 005.1

Apresentação

Teste de Software é de central importância para verificar e validar sistemas computacionais. No entanto, estudos experimentais revelam que atividades relacionadas a teste normalmente consomem cerca de 50% dos custos do desenvolvimento de um software. Abordagens sistemáticas e automáticas têm se mostrado capazes de reduzir este custo de forma significativa. Casos de sucesso na indústria têm sido cada vez mais reportados e o interesse acadêmico sobre este tópico continua a crescer, como observado pelo número crescente de pesquisadores atuando nesta área.

O principal objetivo do Workshop Brasileiro de Teste de Software Sistemático e Automático (SAST) é construir um fórum para integrar as comunidades de pesquisa e da indústria para discutir melhorias na sistematização e automação de teste de software.

Esta 6^a edição do SAST, a ser realizada em Natal no dia 23 de Setembro, será realizada em conjunto com o Congresso Brasileiro de Software: Teoria e Prática (CBSOFT). Mantendo a tradição dos anos anteriores, nesta 6^a edição foram aceitas submissões de artigos técnicos e relatos de experiência da indústria. Artigos técnicas e relatos de experiências da indústria serão apresentados em sessões de 20 minutos.

Nesta edição, temos o prazer de receber como nosso palestrante internacional o Professor Dr. Arie van Deursen (*Delft University of Technology*, Holanda). Prof. van Deursen é um pesquisador influente na área de software. Ele é ainda um dos palestrantes do CBSOFT 2012. Também temos o prazer de receber como palestrante nacional o Professor Dr. Otávio Augusto Lazzarini Lemos (UNIFESP-SJC). Prof. Otávio Lemos é um pesquisador jovem, mas já influente na área de teste de software, com resultados interessantes publicados em periódicos e conferências, como ICSE (*International Conference on Software Engineering*).

Gostaríamos de agradecer a todos os autores que submeteram artigos, aos membros do comitê de programa e de organização, às agências de fomento que apoiaram o evento, ao palestrante e todos que de uma forma direta ou indireta contribuíram para a sua realização.

Natal, Setembro de 2012.

**Roberta de Souza Coelho
Arilo Claudio Dias Neto
Coordenador do SAST 2012
CBSOFT 2012**

Foreword

Software Testing is of central importance to verify and validate software systems. However, empirical studies show that test-related activities often account for over 50% of software development costs. Systematic and automated approaches have shown capable of reducing this overwhelming cost. Industrial success cases have been openly reported and academic interest continues to grow as observed by the growing number of researchers in the field.

The main goal of Brazilian Workshop on Systematic and Automated Software Testing (SAST) is to build a forum that brings the research and industry communities together to discuss improvements in software testing systematization and automation.

This 6th edition of SAST, to happen in Natal in 23rd September, will be co-located with the Brazilian Conference on Software: Theory and Practice (CBSOFT). Following the tradition of previous editions, this 6th edition accepted technical paper and experience report submissions. Technical papers and experience reports are presented in sections of 20 minutes.

In this edition, we are pleased to welcome as international keynote speaker Professor Dr. Arie van Deursen (Delft University of Technology, Netherlands). Prof. van Deursen is an influent researcher in software testing. He is also keynote speaker of CBSOFT 2012. We also are pleased to welcome as national keynote speaker Professor Dr. Otávio Augusto Lazzarini Lemos (UNIFESP-SJC). Prof. Otávio Lemos is a young and influent researcher in software testing, with interesting results published in several journals and conferences, as ICSE (International Conference on Software Engineering).

We would like to thank all the authors who submitted papers, members of the technical program and organization committees, the funding agencies that supported the event, the speaker and all that in a way directly or indirectly contributed to their achievement.

Natal, September of 2012.

**Roberta de Souza Coelho
Arião Claudio Dias Neto**
Chair of SAST 2012
CBSOFT 2012

Breves Biografias dos Coordenadores do SAST 2012

Roberta de Souza Coelho, UFRN

É Professora Adjunto do Departamento de Informática e Matemática Aplicada (DIMAp) da Universidade Federal do Rio Grande do Norte (UFRN). Possui mestrado desde 2003 em Engenharia de Sistemas e Computação pela Universidade Federal de Pernambuco (CIn/UFPE) e obteve seu título de Doutora em Ciência da Computação pelo Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) (2008). Seus interesses de pesquisa incluem Teste de Software, Análise Estática, Tratamento de Exceção, Dependabilidade e Engenharia de Software Experimental. Lattes: <http://lattes.cnpq.br/9854634275938452>.

Arilo Claudio Dias Neto, UFAM

Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ em 2006, Doutor em Engenharia de Sistemas e Computação pela COPPE/UFRJ em 2009. Atualmente, é Professor Adjunto do Instituto de Computação da Universidade Federal do Amazonas (IComp/UFAM) e Professor Permanente do Programa de Pós-Graduação em Informática (PPGI) da UFAM. Tem atuado nos últimos 9 anos em pesquisas científicas e projetos de desenvolvimento na área de Qualidade de Software, principalmente Teste de Software, Engenharia de Software Experimental, Engenharia de Software Baseada em Busca e Desenvolvimento Distribuído de Software. Atua como revisor do Journal of the Brazilian Computer Society (JBCS), do International Journal of Information and Software Technology (IST). Lattes: <http://lattes.cnpq.br/9112415346373126>.

SAST 2012 Chairs Short Biographies

Roberta de Souza Coelho, UFRN

is an Associate Professor at Informatics and Applied Mathematics Department (DIMAp) at Federal University of Rio Grande do Norte (UFRN). She received her MSc (2003) degree in Systems Engineering and Computer Science from Federal University of Pernambuco (Cin/UFPE) and holds a PhD from the Informatics Department of the Pontifical Catholic University of Rio (PUC-Rio) (2008). Her research interests include software testing, static analysis, exception handling, dependability and empirical software engineering. Lattes: <http://lattes.cnpq.br/9854634275938452>

Arilo Claudio Dias Neto, UFAM

received his MSc (2006) degree in Systems Engineering and Computer Science from Federal University of Rio de Janeiro (COPPE/UFRJ) and his DSc (2009) degree in Systems Engineering and Computer Science from Federal University of Rio de Janeiro (COPPE/UFRJ). He is an Associate Professor at Institute of Computing (IComp) at Federal University of Amazonas (UFAM) since 2010. His research interests include Software Testing, Empirical Software Engineering, and Search-Based Software Engineering. He is reviewer of some journals, such as Journal of the Brazilian Computer Society (JBCS), International Journal of Information and Software Technology (IST). Lattes: <http://lattes.cnpq.br/9112415346373126>.

Comitês Técnicos / Technical Committees

Comitê de Programa / Program Committee

Adenilso da Silva Simão, SSC/USP
Alexandre Petrenko, CRIM, Canada
Ana Cavalcanti, University of York, UK
Ana Cavalli, GET-INT, France
Ana Maria Ambrósio, INPE
Anamaria Martins Moreira, DIMAp/UFRN
Arielo Claudio Dias Neto, IComp/UFAM (co-chair)
Arndt Von Staa, Puc-Rio
Auri Vincenzi, INF/UFG
Edmundo Spoto, UNIVEM
Eduardo Guerra, ITA
Eliane Martins, IC/Unicamp
Elisa Yumi Nakagawa, SSC/USP
Elisângela Vieira, Alcatel/Lucent
Ellen Francine Barbosa, SSC/USP
Fábio Fagundes Silveira - ICT/UNIFESP
Fatima Mattiello-Francisco, INPE
Guilherme Horta Travassos, COPPE/UFRJ
Jorge Figueiredo, DSC/UFCG
José Carlos Maldonado, SSC/USP
Juliano Iyoda, CIn/UFPE
Juliana Herbert, UNISINOS
Marcelo Fantinato, EACH/USP
Márcio Eduardo Delamaro, SSC/USP 9
Marcos Chaim, EACH/USP
Mário Jino, FEEC/UNICAMP
Otávio Lemos, ICT/UNIFESP
Patrícia Machado, DSC/UFCG
Paulo César Masiero, SSC/USP
Plínio Vilela, UNIMEP
Roberta Coelho, DIMAp/UFRN (co-chair)
Sandra Fabri, DC/UFSCAR
Silvia Regina Vergílio, DInf/UFPR
Simone do Rocio Senger de Souza, SSC/USP

Comitê Organizador / Organizing Committee

Coordenador Geral do CBSoft 2012 / CBSoft 2012 General Chair
Nélio Cacho, DIMAp/UFRN

Vice-coordenadores do CBSoft 2012 / CBSoft 2012 Co-chairs
Gibeon Aquino, DIMAp/UFRN
Frederico Lopes, ECT/UFRN

Coordenador Local do SBES 2012 / SBES 2012 Local Chair
Thaís Batista, DIMAp/UFRN

Coordenador Local do SBLP 2012 / SBLP 2012 Local Chair
Martin Musicante, DIMAp/UFRN

Coordenador Local do SBMF 2012 / SBMF 2012 Local Chair
David Deharbe, DIMAp/UFRN

Coordenadores Locais do SBCARS 2012 / SBCARS 2012 Local Chairs
Uirá Kulesza, DIMAp/UFRN
Fred Lopes, ECT/UFRN

Comitê de Apoio / Support Committee

Adilson Barbosa, DIMAp/UFRN
Alex Medeiros, DIMAp/UFRN
Anamaria Martins, DIMAp/UFRN
Carlos Eduardo da Silva, ECT/UFRN
Eduardo Aranha, DIMAp/UFRN
Fernando Figueira, DIMAp/UFRN
Jair Cavalcante Leite, DIMAp/UFRN
Lyrene Fernandes, DIMAp/UFRN
Marcel Oliveira, DIMAp/UFRN
Marcia Jacyntha, DIMAp/UFRN
Marcos Cesar Madruga, DIMAp/UFRN
Márjory da Costa-Abreu, DIMAp/UFRN
Ricardo Cacho, IFRN
Roberta Coelho, DIMAp/UFRN

Apoio / Support

Diego Oliveira, Web Development
Bernardo Gurgel, Web Designer
Arthur Souza, Web Designer
4Soft - Empresa Júnior, (website)

Índice de Artigos / Table of Contents

Technical Session 1: Empirical Evaluation and Experiences in Testing.....	1
(Experience Report) Experience in Organizing Test Teams.....	1
<i>Andrew Diniz da Costa (PUC-RJ), Soeli Fiorini (PUC-RJ), Carlos J. P. de Lucena (PUC-RJ), Gustavo Carvalho (PUC-RJ)</i>	
Technical Session 2: Mutation Testing and Testing in the Cloud.....	7
(Technical Paper) Teste de Linha de Produto de Software Baseado em Mutação de Variabilidades.....	7
<i>Marcos Antonio Quináia (UNICENTRO), Johnny Maikeo Ferreira (UFPR), Silvia Regina Vergilio (UFPR)</i>	
(Technical Paper) Execução Determinística de Programas Concorrentes Durante o Teste de Mutação.....	17
<i>Rodolfo A. Silva (ICMC-USP), Simone R. S. Souza (ICMC-USP), Paulo S. L. Souza (ICMC-USP)</i>	
(Technical Paper) Uso de análise de mutantes e testes baseados em modelos: um estudo exploratório.....	27
<i>Isaura Rennaly Souto Lima (UNICAMP), Thaise Yano (UNICAMP), Eliane Martins (UNICAMP)</i>	
(Technical Paper) Framework para Teste de Software Automático nas Nuvens.....	37
<i>Gustavo S. Oliveira (UFPB), Alexandre N. Duarte (UFPB)</i>	
Technical Session 3: Test Generation.....	47
(Experience Report) Usando o SilkTest para automatizar testes: um Relato de Experiência.....	47
<i>Thiago L. L. Lima (UFPB), Ayla Dantas (UFPB), Lívia M. R. Vasconcelos (DATAPREV), Aluizio N. S. Neto (DATAPREV)</i>	
(Technical Paper) Automating Test Case Creation and Execution for Embedded Real-time Systems.....	53
<i>Joeffison S. Andrade (UFCG), Lucas R. Andrade (UFCG), Augusto Q. Macedo (UFCG), Wilkerson L. Andrade (UFCG), Patrícia D. L. Machado (UFCG)</i>	
(Technical Paper) Controlando a Diversidade e a Quantidade de Casos de Teste na Geração Automática a partir de Modelos com Loop.....	63
<i>Jeremias D. S. de Araújo (UGCG), Emanuela G. Cartaxo (UFCG), Francisco G. de Oliveira Neto (UFCG), Patrícia D. L. Machado (UFCG)</i>	
(Technical Paper) Geração Aleatória de Dados para Programas Orientados a	

Objetos.....73

Fernando H. Ferreira (EACH/USP), Marcio E. Delamaro (ICMC/USP), Marcos L. Chaim (EACH/USP), Fátima N. Marques (EACH/USP), Auri M. R. Vincenzi (UFG)

Experience in Organizing Test Teams

Andrew Diniz da Costa, Soeli Fiorini, Carlos J. P. de Lucena, Gustavo Carvalho

Laboratory of Software Engineering - Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro – RJ – Brazil

{acosta, lucena}@inf.puc-rio.br, {soeli, guga}@les.inf.puc-rio.br

Abstract. Over time, testing software systems has become a critical activity of software development. Thus, in the last six years the Software Engineering Lab at the Pontifical Catholic University of Rio de Janeiro has carried out extensive work on coordinating and performing tests of large-scale software systems developed for a Brazilian petroleum company. From this experience, approaches were adopted to help at maintaining and creating test cases. Furthermore, a set of concerns were identified for the Lab's test teams that motivated the creation of a new software test center of excellence aiming to resolve these concerns. Thus, the goal of the paper is to describe the main experiences of the Lab's test teams over the past few years.

1. Introduction

Appropriate documentation in any project is extremely important to track the tests and the changes performed in each release of a system. Without documenting the test cases, it is difficult to plan the necessary investments to perform the tests and also to guarantee that the remaining faults were not caused due to the changes executed while maintaining or updating the software [Kaner et. al, 2001]. Therefore, there is a need for techniques that support the documentation of test related concerns [Harrold, 2008].

Over the past six years, the Software Engineering Lab at the Pontifical Catholic University of Rio de Janeiro has worked extensively on coordinating and carrying out tests of large-scale software systems developed for a Brazilian petroleum company.

There is a multitude of small tasks that must be performed and managed by the testing team. So many, in fact, that it is tempting to focus purely on the mechanics of testing a software application, and to pay little attention to the surrounding tasks that are required of a testing program. Issues, such as, the acquisition of proper test data, testability of the application's requirements and architecture, appropriate test procedure standards and documentation, and hardware and facilities are often addressed very late, if at all, in the lifecycle [Polo et. Al, 1999]. Thus, such issues should be considered when a test team is defined.

The knowledge of what constitutes a successful, end-to-end testing effort is typically gained through experience. Besides, how to organize the team could facilitate the information flow and preserve this knowledge within the team and motivate new research.

This paper describes some patterns and approaches adopted by test teams of the Software Engineering Lab to maintain and create software tests. Thus, which test tools

could be used and how a test team should be organized to perform the necessary tests, are examples of questions answered in the paper.

Concerns were identified from the efforts of the past six years. Seeking to solve them, the Software Engineering Lab defined a software test center. The paper presents in detail the structure of this center and its main goals, such as to train new people to work with tests, to conduct research based on the concerns identified by the teams, etc.

The paper is organized as follows. Section 2 presents some approaches defined to maintain and create software tests in test projects. Section 3 describes concerns identified in such projects, while Section 4 presents in detail the new software test center. And finally, in Section 5, the final considerations are presented.

2. Defining Approaches in Test Projects

Over the past six years, the Software Engineering Lab has worked extensively on coordinating and carrying out tests of software systems developed for a Brazilian petroleum company. Due to space constraints of this paper, we decided to report the lessons learned that were observed in the largest, more complex and first system that we have placed in the test area. This system controls the inventory and supply of oil and derived oil products (e.g. gasoline, kerosene, etc). During the past few years, we had several meetings with project members to discuss what documentation would be appropriate to maintain and create new tests.

Some of the main goals of the chosen system are to: (i) register routes (i.e. paths) based on ducts and ships that could be used to transport the derived products (e.g. gasoline, lubricating oil, kerosene, etc); (ii) present from a 2D map the amount and which product is stored in a given refinery or terminal in Brazil; (iii) permit planning the best routes to transport a given product from one place to another; (iv) produce a set of reports and graphics, etc.

Due the complexity of the developed system, we had to define four teams responsible for the following elements: interface, database, requirements and testing. The interface team was composed of seven full-time people (40 hours per week), who developed GUI interfaces (Java). The database team was composed of three full-time people, who were responsible for creating procedures and for maintaining the database's schema (PL/SQL). The requirement team had two part-time people (20 hours per week), who created and maintained the following documents: (i) use cases, (ii) UML diagrams, (iii) business rules, and (iv) an impact spreadsheet of changes that identified the relation between requirements of the system. And, last but not least, the test team was composed of six people (four part-time and two full-time) who created, maintained and executed functional, database and performance tests.

When different test cases began to be created for different versions of a same system, we identified three test classifications: regression, new and impacted. The regression tests are those tests executed every time a new version of the system is defined. New tests are those created due to new system requirements or due to changes in the system functionalities. Impacted tests are those tests that were modified due to the definition of a new requirement or the adaptation of a given requirement. These classifications helped to perform the planning and the delegation of the tests for the system version. An example of delegation is the execution of regression tests, which

were usually performed by new members of the test team. We adopted this approach because from the regression tests the new members can better learn the requirements of the system and to understand the test coverage provided.

Besides these classifications, it is important to understand the priority of each test to a given system version. We divided this identification in two parts. The first part identifies if a test is mandatory or optional. Mandatory tests are those considered essential; i.e., they must be executed before the system is delivered to the customer. And optional tests are those with low priority that should only be executed if the test team has enough time to do so. The second part identifies the level of priority in each group (mandatory or optional); i.e., we inform a number from 1 to N (lower the number, higher the priority) to the test cases that are mandatory and to the test cases that are optional. Thus, it is possible to identify the priority of a specific test that belongs to the mandatory group, for example.

In order to control the priority of the tests, we use the Atlassian JIRA [Atlassian JIRA, 2012], an issue system that controls the tests (tasks) created for a given version, and stores the history of the changes.

However, tests have other important characteristics that are useful to test planning, such as knowing if a test is manual or automated. Therefore, the test team included the MTC acronym in the names of the test cases when they are manual, and the ATC acronym when the tests are automated.

Another important characteristic of the test cases is to identify their test types; i.e., if a test case is functional, performance, unit, etc. Depending on the test type, different tools are used and can impact the time for maintaining and executing the tests. Besides, the test team can have testers with different expertise in different test types. Thus, it is important to classify the tests according to their types. In this system we use the DBUnit API and a set of Rational tools: Rational Manual Test (RTM), Rational Functional Tester, and Rational Performance Tester. These tools and API are used to create and maintain, respectively, automated unit database tests, manual tests, automated functional tests, and automated performance tests.

In this project, the test types were identified by the following key words: GUI (functional test), DB (unit database test) or PER (performance test). For instance, ATC_PER_Login and MTC_GUI_Login are examples of names given to an automated performance test name and a manual functional test name, respectively.

The test team also agreed with the importance of documenting the pre-conditions of the tests. From them, it is possible to identify dependencies between tests and dependencies of environment configurations to execute some tests. This data allows recognizing the complexity related to the execution of a given test and to better estimate the time to be spent in the test execution. Aiming to perform such documentation and to describe the step-by-step of each test case, the project team has chosen the test management tool called Rational Test Manager (RTM).

From the patterns and the approaches adopted by the test team, we understood that the time spent in maintaining and creating tests had decreased. Before using the test acronyms (e.g. MTC, ATC, PER, DB, etc.), the time spent in the test activity was 16% above the planned time (see versions A and B in Table 1). However, when the test team started to explicitly document the newly revealed testing concerns, substantial

improvement was observed. The first time the tests were documented following this approach was in version C. As illustrated in Table 1, the test team was able to finish the test activity in the expected time in the next two versions: D and E. The data presented in Table II were collected from the issue system [Atlassian JIRA, 2012] used in the project that informs the time spent on each project task. In our discussions with the project members, all of them agreed this time reduction was directly impacted by the documentation performed and mentioned previously.

Table 1. Relation Between the Planned Testing Time and Actual Time Spent in the Project.

Versions	Planned Time (weeks)	Time used (weeks)	Time exceed (%)
Version A	6	7	16.7
Version B	7	8.5	21.4
Version C	6	6.5	8.3
Version D	7	7	0
Version E	8	8	0

3. Concerns Identified From Test Projects

After three years coordinating and carrying out tests in the system mentioned in the previous section, the Software Engineering Lab began testing another system also related to the petroleum domain. This system exchanges data with the system mentioned in Section 2. Moreover, these systems use a same set of corporation tables of different databases.

Due the complexity of the new system we defined four teams responsible for the following elements: interface, database, requirements and testing. Such structure was based on the experience described in Section 2 and we decided to adopt the same approaches for the new test team. From this, the following concerns were identified.

Training new members to work with test: When a new test team was created, the new members were trained to: (i) document the test cases created, (ii) use the test tools that allow creating and maintaining test scripts, (iii) understand how to register and to explain bugs identified, (iv) understand the entire test process adopted by the team, etc. Thus, we had to define a structure to monitor the work of the subjects and to help them when necessary.

Substituting test tools that are used: The tools used by the test team, mentioned in Section 2, were also used by the new team. However, some of these tools are no longer updated by the company that developed them. Based in this concern we had to continuously work to analyze available tools provided in the literature and that we could use to substitute these discontinued tools.

Applying Model Driven Test: The customer of the systems requested that any source code created should have UML diagrams to document them. The requirement team used the Rational Software Architecture (RSA) to create several diagrams of the system. Aiming to maintain the consistency of the requirement team, we planned to model test scripts from the RSA. However, the original UML does not represent all the relevant test concepts that could be modeled in such diagrams. Thus, we analyzed a set of

approaches [Costa et. al, 2010] in order to decide how important test concepts could be modeled from the UML.

Integrating different test teams: Due to the different experiences of these two test teams (especially the test team described in Section 2), it became important to allow the exchange of knowledge between them in order to avoid possible rework.

Allowing continuous work: Three members departed three months after the new test team was created. As a result of this situation, meeting the requests from the test project became a challenge. This situation led us to define a unique test team, which would be able to test these two systems related to the petroleum domain, and to perform integration tests, since they exchange data. Details of this organization are presented in next section.

Applying research to help test projects: Some members of the teams are undergraduate students, Master Degree and PhD candidates in software engineering. Thus, we realized that they could use their research to satisfy concerns identified in the projects. We therefore sought to align their interests to perform research that could be useful to the industry.

These concerns motivated the Software Engineering Lab to define a new test team structure: a software test center composed of 12 people, who are able to perform tests on the two systems being tested. We present this center in detail in the next section.

4. Software Testing Center of Excellence

The new software testing center of excellence defined in the Software Engineering Lab defined four main goals: (i) training of new team members, (ii) participation in industrial projects, (iii) development of plug-ins, and (iv) research.

Training: An important concern of the center is to conduct appropriate training for the members of the team. Through this, the center intends to qualify its members to maintain and create different types of tests for different systems. Because the lab is located in the Pontifical Catholic University of the Rio de Janeiro, it aims to improve the presentation and written skills of the members of the team. Nowadays, 75% of the team is composed of undergraduate students, Masters Degree and PhD candidates. Seeking to meet such concerns and to allow the identification of other test tools that could be used in the test projects, a monthly seminar was organized. During each seminar two members are chosen to study tools and, subsequently, make a presentation to the other members of the team. After the presentations, they must write a technical paper describing how to use such the tool for the first time.

Participating in Industrial Projects: Since most of the team is composed of undergraduate students, Masters Degree and PhD candidates, participation in industrial projects is an experience that is allowed to relate their research to the test projects. Furthermore, such participation demonstrates how to work collaboratively with people from different teams as well as with the customers. Meetings that involve only the test team or that involve people of other teams and customers permit this experience.

When we defined a unique team to test the two systems being tested, which are related to the petroleum domain, we were able to more easily create and maintain integration tests. This task became easier because all the members of the team

understood the systems' business rules and use cases. Thus, such understanding was important to test dependences of the systems, since they exchange data.

Research: From the industrial work, the test team identified problems and research subsequently was carried out. One example of research was related to the modeling of test concepts. This is a concern mentioned in Section 3, since the customer requested models (e.g. UML diagrams) of the source code developed. Aiming to identify which test concepts are relevant and how such concepts could be modeled, the work described in [Costa et. al, 2010] performed part of this analysis. This research makes it possible to apply a Model Driven Test (MDT) [Baker et al, 2007] in the test projects, based on a new approach proposed by a PhD candidate. Thus, the identified concern is being dealt with.

Development of plug-ins and tools: As a result of the seminar and the research mentioned, we decided to train part of the team to create plug-ins or new tools that satisfy the concerns that were identified. Thus, these people work part-time on test projects and part-time for the development of plug-ins and tools. Nowadays, the current plug-in developed is about test modeling. This plug-in was created for the Rational Software Architecture tool in order to apply MDT. The person primarily responsible for its development is a PhD candidate who is using the development knowledge gained to train other team members.

5. Final Considerations

This paper presented experiences and concerns identified from different test projects coordinated and carried out by Software Engineering Lab test teams. Since a new software test center was created from these concerns, other centers related to software engineering areas are being defined, such as: requirement, development and software processes. These areas are research fields that the Software Engineering Lab has been investigating over the past several years. Thus, through these centers, the Lab intends to conduct research that can be used in industrial projects, besides motivating the creation of new spin-offs.

References

- Atlassian JIRA (2012) <http://www.atlassian.com/software/jira/>.
- Baker, P., Dai, Z. R., Grabowski, J., Schieferdecker, I., Williams, C. (2007) "Model Driven Testing: Using the UML Testing Profile", Springer, 1st edition, ISBN: 3540725628.
- Costa, A. D., Silva, V. T., Garcia, A., Lucena, C. J. P. (2010) "Improving Test Models for Large Scale Industrial Systems: An Inquisitive Study", MODELS 2012, Oslo, Norway.
- Kaner, C., Bach, J., Pettichord, B. (2001) "Lessons Learned in Software Testing", Publisher: Wiley, 1st edition, ISBN: 0471081124.
- Harrold, M. J. (2008) "Testing Evolving Software: Current Practice and Future Promise", Proceedings of ISEC 2008, pp. 3-4.
- Polo, M., Piattini, M., Ruiz, F. and Calero, C. (1999) "Mantema: A Complete Rigorous Methodology for Supporting Maintenance Based on the ISO/IEC 12207 Standard", Proc. IEEE European Conference on Software Maintenance and Reengineering: IEEE Computer Society: Los Alamitos CA, pp. 178 -181.

Teste de Linha de Produto de Software Baseado em Mutação de Variabilidades

Marcos Antonio Quináia¹, Johnny Maikeo Ferreira², Silvia Regina Vergilio²

¹Departamento de Ciência da Computação
Universidade Estadual de Centro-Oeste (UNICENTRO) – Guarapuava, PR – Brasil

²Departamento de Informática
Universidade Federal do Paraná (UFPR) – Curitiba, PR – Brasil
quinaia@unicentro.br, {jmferreira,silvia}@inf.ufpr.br

Abstract. *The Orthogonal Variability Model (OVM) has been adopted to define and document software product line variabilities and to derive products for testing. Given the huge variability space of most applications, testing criteria based either on pair-wise combinations of variabilities or on the basis path testing are generally used to select a subset of the most representative products. However, these criteria do not consider faults that can be present in the OVM. To increase the confidence that the LPS products match their requirements, in this work, a mutation based approach is presented. Mutation operators are introduced and evaluated in a case study. The results show that other kind of faults are revealed in comparison with existing criteria.*

Resumo. *O OVM (Orthogonal Variability Model) tem sido adotado para definir e documentar as variabilidades de uma linha de produto de software (LPS), e para derivar produtos para teste. Dado o enorme espaço de variabilidades da maioria das aplicações, critérios de teste baseados em combinações par-a-par de variabilidades, ou no teste de caminhos básicos, são geralmente utilizados para selecionar um subconjunto de produtos, os mais representativos. Estes critérios, entretanto, não consideram defeitos que podem estar presentes no OVM. Para aumentar a confiança de que os produtos da LPS satisfazem a especificação, neste trabalho é apresentada uma abordagem baseada em teste de mutação. Operadores são introduzidos e avaliados em um estudo de caso. Os resultados mostram que outros tipos de defeitos são revelados em comparação com os critérios existentes.*

1. Introdução

Uma linha de produto de software (LPS) pode ser entendida como um conjunto de sistemas de software que compartilham características (*features*) comuns e que satisfazem às necessidades específicas de um segmento particular de mercado (van der Linden et al., 2007). O objetivo da LPS é a criação eficiente e sistemática de produtos por meio do reúso de artefatos. Para isto, é fundamental um suporte adequado às variabilidades da LPS (Uzuncaova et al., 2010), geralmente representadas por modelos tais como o OVM (*Orthogonal Variability Model*). Este modelo tem sido adotado por muitas metodologias de desenvolvimento de LPS, pois auxilia a definir e documentar as

variabilidades sem considerar as características comuns, o que facilita o gerenciamento das mesmas. Além disso, o OVM tem sido bastante utilizado para derivar produtos para teste (Cabral et al., 2010; Cohen et al., 2006). Um produto é dado pela combinação de características comuns e variáveis da LPS.

Idealmente todas as combinações de variabilidades representadas no OVM deveriam ser testadas. Entretanto, dada a complexidade das aplicações o número de produtos possíveis é exponencial, e o teste exaustivo é impraticável (Cohen et al., 2006). Portanto, o grande desafio é selecionar um subconjunto desses produtos que possa revelar a maioria dos defeitos e aumentar a confiança de que os produtos representados estão de acordo com os requisitos da LPS (Engström e Runeson, 2011; Silveira Neto, 2011; Oster et al., 2011; Perrouin et al., 2011). Para este desafio é necessário portanto o uso de um critério de teste que possa guiar a seleção dos produtos e também ser utilizado como uma medida de avaliação da qualidade de um dado conjunto de produtos.

Para selecionar os produtos, a maioria dos trabalhos existentes baseia-se em teste combinatorial (Cohen et al., 2006; Lamancha e Usaola, 2010; McGregor, 2001; Oster et al., 2011; Perrouin et al., 2011; Uzuncaova et al., 2010) requerendo o teste de combinações de variantes presentes no OVM. Geralmente pares de variantes (*pair-wise testing*) são utilizados para compor os produtos. Uma outra proposta (Cabral et al., 2010) adota o teste de caminhos básicos, geralmente aplicado no teste baseado em fluxo de controle de programas, na qual são requeridos caminhos independentes de um grafo que representa as características variáveis do OVM. Uma limitação dos trabalhos existentes é que eles não consideram possíveis defeitos que geralmente são introduzidos na criação do OVM para selecionar os produtos. Um critério baseado em defeitos, tal como o critério Análise de Mutantes, pode aumentar a confiança de que a LPS está de acordo com sua especificação, auxiliando a revelar outros tipos de defeitos e pode ser utilizado de uma maneira complementar às propostas existentes. No teste de programas os critérios baseados em defeitos têm se mostrado mais eficazes em termos do número de defeitos revelados (Wong et al., 1994).

Considerando este fato, neste trabalho é introduzida uma abordagem de teste baseada em mutação de variabilidades de uma LPS. São introduzidos operadores de mutação, que descrevem erros típicos do modelo OVM. Como no teste de mutação de programas, OVM mutantes são gerados a partir dos operadores propostos e devem ser mortos pelo conjunto T de produtos. Ao final o escore de mutação é calculado e utilizado como medida de avaliação do teste. A abordagem e operadores são avaliados em um estudo de caso, no qual é efetuada uma comparação com o teste *pair-wise* e com o teste de caminhos básicos. Os resultados mostram a aplicabilidade dos operadores em termos de número de mutantes gerados e a eficácia dos mesmos em revelar defeitos.

O trabalho está organizado como segue. Na Seção 2 é apresentada uma definição de OVM. Na Seção 3 são discutidos os principais trabalhos relacionados que têm como o objetivo a geração de dados de teste considerando as variabilidades da LPS. Na Seção 4 é introduzido o modelo de representação para o OVM, operadores de mutação e exemplos de mutantes gerados. Na Seção 5 é descrito o estudo de caso, LPS utilizada, passos conduzidos, seguidos da apresentação e análise dos principais

resultados obtidos. Na Seção 6 são apresentadas as conclusões e possíveis desdobramentos para este trabalho.

2. O modelo OVM

Variabilidade indica inconstância e pode levar à mudança(s). No processo de desenvolvimento de software, a variabilidade diferencia o desenvolvimento de software tradicional do desenvolvimento de LPS. No primeiro a finalidade é o desenvolvimento de um software para fins específicos, ao passo que no último procura-se desenvolver uma variedade de produtos que pertencem a uma família de produtos de software.

O OVM representa todas as partes variáveis (variabilidades) que compõem uma LPS. A Figura 1 apresenta um exemplo de OVM para a LPS Mobile Utilities, no qual são representadas relações de dependência e restrições associadas às partes variáveis da LPS. Para permitir a análise automatizada de OVMs, alguns modelos de representação foram propostos. Neste trabalho adota-se o meta-modelo proposto por Pohl et al. (2005) (Figura 2), no qual um OVM é dado por um tupla $\circ = (\text{VP}, \text{V}, \text{A}, \text{R}, \text{d})$:

- VP é o conjunto de pontos de variação vp (triângulos na Figura 1, tais como Mobile Utilities), pontos nos quais a variação irá ocorrer e que definem as características do sistema que poderão ou não estar nos produtos. Cada ponto de variação pode ser: (interno ou externo), o tipo é dado por vpType , onde $\text{vpType} \in \{\text{interno}, \text{externo}\}$. Um ponto de variação “interno” está associado a variantes que são visíveis apenas para os desenvolvedores, mas não para os clientes. Já um ponto de variação “externo” está associado a variantes que são visíveis para ambos;
- $\text{d} \in \text{VP}$ representa o domínio sendo modelado pelo OVM;
- V é o conjunto de variantes v (retângulos, por exemplo GPS, MP3, Camera) que representam, cada um, uma forma de variação, a realização de uma característica;
- A é o conjunto de associações representando uma dependência. Uma associação a é da forma $(\text{vp}, \text{aType}, \{\text{v}_1, \dots, \text{v}_n\})$, onde $\text{vp} \in \text{VP}$, $\text{aType} \in \{\text{man}, \text{opt}, \text{alt}[\text{min}, \text{max}]\}$ e $\{\text{v}_1, \dots, \text{v}_n\} \subset \text{V}$; tal que $n=1$ para $\text{aType} \in \{\text{man}, \text{opt}\}$. Cada vp deve estar associado a pelo menos um v , e cada v deve estar associado a pelo menos um vp . Com relação aos tipos de associação i) man : mandatária (linha sólida), representa um tipo de associação obrigatória caso o ponto de variação seja instanciado; o que torna o variante obrigatório; ii) opt : opcional (linha tracejada), representa um tipo de associação não obrigatória caso o ponto de variação seja instanciado; e iii) $\text{alt}[\text{min}, \text{max}]$: alternativa (linha tracejada), associação entre um ponto de variação e dois ou mais variantes que exige dois números, min e max , representando respectivamente a quantidade mínima e máxima de variantes presentes caso o ponto de variação seja instanciado, com valores $\text{default min} = \text{max} = 1$;
- R é o conjunto de restrições (linha tracejada com seta na ponta) que podem ocorrer de variante para variante (v_v), de ponto de variação para ponto de variação (vp_vp), e de variante para ponto de variação (v_vp). Uma restrição r entre x e y , é da forma (x, rType, y) , onde $\text{rType} \in \{\text{req}, \text{exc}\}$, são tipos de restrições requires (entre Camera e High Resolution) e excludes (entre GPS e Basic), indicando respectivamente que a escolha de x requer (ou exclui) y .

O OVM tem sido utilizado para derivar produtos que servem de casos de teste para a LPS. Para o exemplo da Figura 1, um produto válido para o diagrama seria dado pelos seguintes variantes (GPS, Screen, Colour, Media, MP3).

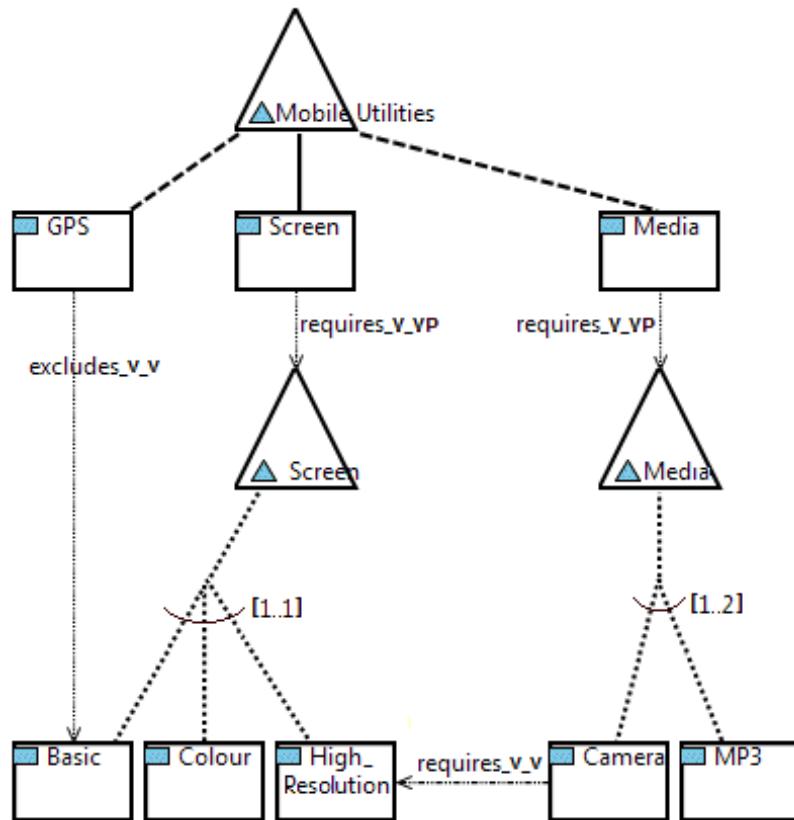


Figura 1: OVM da LPS Mobile Utilities adaptado de (Benavides et al., 2010)

3. Trabalhos Relacionados

Existem diversos trabalhos que enfatizam diferentes aspectos do teste de LPS (Engström e Runeson, 2011; Lamancha et al., 2009; Silveira Neto, 2011). Com relação ao teste de variabilidades, a maior parte dos trabalhos está baseada no teste de combinações presentes no modelo de características (FM – *feature model*) e também no OVM.

O trabalho de McGregor (2001) utiliza métodos de teste combinatorial (*orthogonal arrays*). O trabalho de Cohen et al. (2006) tem como base o OVM, *pairwise testing* e *arrays* de cobertura para selecionar os produtos. Oster et al. (2011) utilizam teste combinatorial (*pair-wise*) e teste baseado em modelos. Os trabalhos de Uzuncaova et al. (2010) e Perrouin et al. (2010) estão baseados no FM e em fórmulas *Alloy* que são utilizadas para geração de casos de teste. Perrouin et al. (2010) utilizam duas estratégias de divisão e composição para lidar com escalabilidade do *t-wise testing*. O trabalho de Lamancha and Usaola (2010) estende um algoritmo tradicionalmente utilizado no teste combinatorial, o AETG (Cohen et al., 1996) para o contexto de LPS, para que o mesmo possa considerar restrições de *requires* e *excludes*. Este algoritmo está disponível na ferramenta CombTestWeb¹. O trabalho de Cabral et al. (2010) não

¹<http://161.67.140.42/CombTestWeb>.

está baseado em teste combinatorial. A proposta, chamada *FIG Basis Path*, transforma o OVM em um grafo de *features* e de suas dependências, e requer que produtos sejam gerados de tal maneira que os caminhos básicos deste grafo sejam exercitados, similarmente ao teste de caminhos independentes a partir do grafo de fluxo de controle de programas.

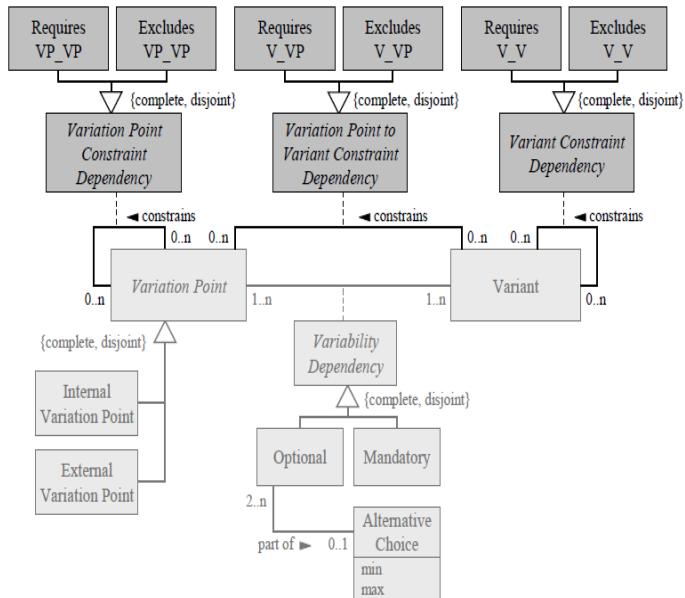


Figura 2: Metamodelo do OVM adaptado de (Pohl et al., 2005)

Os trabalhos mencionados visam selecionar produtos para cobrir as variabilidades e suas dependências. Entretanto, não é possível garantir que o subconjunto de produtos selecionado seja capaz de revelar defeitos típicos que podem estar presentes no OVM. Os critérios mencionados não são baseados em defeitos. Para considerar prováveis defeitos, neste trabalho é proposta uma abordagem baseada em mutação, a qual inclui um conjuntos de operadores descrito a seguir.

4. Operadores de Mutação para o OVM

Um OVM incorreto define incorretamente as variabilidades da LPS, fazendo com que os produtos por ela descritos não estejam de acordo com a especificação. Erros no OVM correspondem à definição incorreta de seus elementos: pontos de variação; variantes; associações; e restrições. Considerando estes tipos de defeitos e o metamodelo da Figura 2 foram propostos 17 operadores de mutação, agrupados em classes de acordo com o elemento OVM a sofrer mutação, conforme Tabela 1.

Para ilustrar o uso desses operadores, na Figura 3 são apresentados alguns exemplos de mutantes gerados. Por restrições de espaço, apenas a parte modificada do OVM para cada operador é apresentada. A Figura 3 em suas partes a), b), c), d), e), f) e g) mostram respectivamente a aplicação dos operadores MOM, RVA, MAO, DLI, MRE, RRE e MMO. Vale a pena ressaltar que os mutantes OVM gerados devem ser

diagramas válidos, ou seja consistentes, similarmente ao mutantes gerados para o teste de programas que devem compilar e não conter erros sintáticos.

Tabela 1. Operadores propostos

Nome	Definição, seja $\circ = (\text{VP}, \text{V}, \text{A}, \text{R}, \text{d})$	Descrição
Mutação em Ponto de variação, produz um mutante $m = (\text{VP}', \text{V}, \text{A}, \text{R}, \text{d})$		
MVPE	$\forall vp \in \text{VP} \mid vp\text{Type} = \text{interno}$ $vp' \in \text{VP}' \mid vp'\text{Type} = \text{externo}$	Mutação de ponto de variação interno para externo
MVPI	$\forall vp \in \text{VP} \mid vp\text{Type} = \text{externo}$ $vp' \in \text{VP}' \mid vp'\text{Type} = \text{interno}$	Mutação de ponto de variação externo para interno
Mutação em Associação, produz um mutante $m = (\text{VP}, \text{V}, \text{A}', \text{R}, \text{d})$		
RVA	$\forall a \in A, a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_{i-1}, v_i, v_{i+1}, \dots, v_n\}) \mid n > 2$ $a' \in A' \mid a' = (vp, \text{alt[min, max]}, \{v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n\})$ $a'' \in A' \mid a'' = (vp, \text{opt}, \{v_i\})$	Remove variante de uma associação alternativa (sem alterar a cardinalidade)
ILI	$\forall a \in A, a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\}) \mid \text{min} < \text{max}-1$ $a' \in A' \mid a' = (vp, \text{alt[min+1, max]}, \{v_1, \dots, v_n\})$	Incrementa limite mínimo (inferior) de uma associação alternativa
DLI	$\forall a \in A, a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\}) \mid \text{min} > 0$ $a' \in A' \mid a' = (vp, \text{alt[min-1, max]}, \{v_1, \dots, v_n\})$	Decrementa limite mínimo (inferior) de uma associação alternativa
ILS	$\forall a \in A, a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\}) \mid \text{max} < n$ $a' \in A' \mid a' = (vp, \text{alt[min, max+1]}, \{v_1, \dots, v_n\})$	Incrementa limite máximo (superior) de uma associação alternativa
DLS	$\forall a \in A, a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\}) \mid \text{max} > \text{min} \wedge \text{max} > 1$ $a' \in A' \mid a' = (vp, \text{alt[min, max-1]}, \{v_1, \dots, v_n\})$	Decrementa limite máximo (superior) de uma associação alternativa
MAO	$\forall a \in A \mid a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\})$ $\{a_1', \dots, a_n'\} \subset A' \mid a_1' = (vp, \text{opt}, \{v_1\}), \dots, a_n' = (vp, \text{opt}, \{v_n\})$	Mutação de associação alternativa para opcional
MMO	$\forall a \in A \mid a = (vp, \text{man}, \{v\})$ $a' \in A' \mid a' = (vp, \text{opt}, \{v\})$	Mutação de associação mandatária para opcional
MAM	$\forall a \in A \mid a = (vp, \text{alt[min, max]}, \{v_1, \dots, v_n\})$ $\{a_1', \dots, a_n'\} \subset A' \mid a_1' = (vp, \text{man}, \{v_1\}), \dots, a_n' = (vp, \text{man}, \{v_n\})$	Mutação de associação alternativa para mandatária
MOM	$\forall a \in A \mid a = (vp, \text{opt}, \{v\})$ $a' \in A' \mid a' = (vp, \text{man}, \{v\})$	Mutação de associação opcional para mandatária
M2A	$a_1 = (vp, \text{aType}, \{v_1\}), a_2 = (vp, \text{aType}, \{v_2\}), \dots,$ $a_n = (vp, \text{aType}, \{v_n\}) \mid n > 1 \text{ e } \text{aType} \in \{\text{opt, man}\}$ $a' \in A' \mid a' = (vp, \text{alt}[1, n], \{v_1, \dots, v_n\})$	Mutação de associação opcional e/ou mandatária para alternativa
Mutação em Restrição, produz $m = (\text{VP}, \text{V}, \text{A}, \text{R}', \text{d})$		
R2A	$\forall r \in R \mid r\text{Type} \in \{\text{req, exc}\}$ $R' = R - \{r\}$	Restrição Ausente
RRE	$r = (x, \text{req}, y) \mid x, y \in V \cup \text{VP} \text{ e } r \notin R \text{ e } x \neq y$ $R' = R + \{r\}$	Restrição Requires Extra
REE	$r = (x, \text{exc}, y) \mid x, y \in V \cup \text{VP} \text{ e } r \notin R \text{ e } x \neq y$ $R' = R + \{r\}$	Restrição Excludes Extra
MRE	$\forall r \in R \mid r\text{Type} = \text{exc}$ $r' \in R' \mid r'\text{Type} = \text{req}$	Mutação para Restrição Excludes
MRR	$\forall r \in R \mid r\text{Type} = \text{req}$ $r' \in R' \mid r'\text{Type} = \text{exc}$	Mutação para Restrição Requires

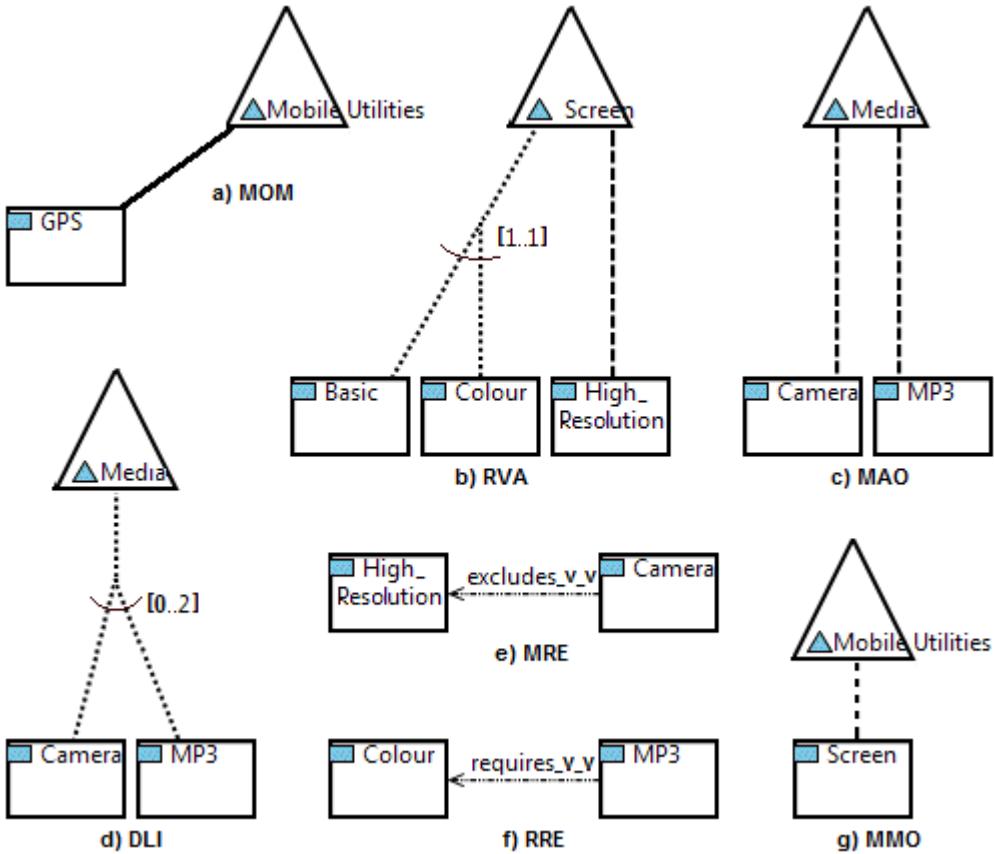


Figura 3: Exemplos de mutantes para o OVM da Figura 1

5. Estudo de Caso

O estudo foi conduzido com o OVM da Figura 1 com o objetivo de avaliar os operadores propostos, comparando-os com o *pair-wise testing* e o teste de caminhos básicos. Primeiramente foi gerado o FIG e um conjunto de 3 produtos denominado (figSet), para cobrir seus caminhos básicos. Depois foi gerado o conjunto com 4 produtos, denominado aetgSet, utilizando a ferramenta CombTestWeb. Todos os operadores foram usados considerando todas as suas possibilidades de aplicação. O número de mutantes não equivalentes gerados por operador é apresentado na Tabela 2. Os operadores RRE e REE geraram o maior número de mutantes. Os operadores ILI, ILS, DLS, MMO, M2A, MRE e MRR geraram poucos mutantes, sendo que os operadores MVPE e MVPI não puderam ser aplicados no OVM, pois neste diagrama não é feita distinção do tipo de ponto de variação. O número de mutantes é dependente das características do diagrama. Por exemplo, o número de operadores associados às restrições geram muitos mutantes com relação ao número de variantes e pontos de variação existentes. O custo deverá ser avaliado em experimentos futuros e novos estudos poderão indicar possibilidades de refinamento destes operadores.

Para avaliar o *strength* dos conjuntos figSet e aetgSet, os produtos destes conjuntos foram avaliados utilizando os mutantes gerados e o framework FaMa (Benavides et al., 2007). O mutante foi considerado morto quando os resultados diferentes foram obtidos com a avaliação do produto no FaMa para o mutante e para o OVM original, ou seja, o produto é válido para o mutante e inválido para o original, ou

o contrário, ele é valido para o original e inválido para o mutante. O número de mutantes mortos pelos conjuntos, por operador, é apresentado na Tabela 2. Entre parênteses é apresentado o número de produtos de cada conjunto necessários na ordem executada. Ou seja o produto é incluído neste número somente se ele contribuiu para matar um mutante ainda não morto pelos produtos executados anteriormente. Portanto este número depende da ordem de avaliação dos produtos, que seguiu a ordem de geração.

Pode-se observar que, de um total de 77 mutantes gerados, o conjunto figSet matou 40 (escore 0,48), e o conjunto aetgSet matou 49, um escore de 0,58. Foram necessários todos os casos de teste de ambos conjuntos (na ordem dada) para alcançar este escore. O escore do conjunto aetgSet é melhor, ou seja, o *pair-wise testing* pode revelar uma maior quantidade de defeitos que o teste de caminhos básicos. Mas percebe-se que existem operadores para os quais nenhum mutante pôde ser morto usando os casos de teste destes conjuntos; são os operadores DLI, ILS, DLS, MAO, MMO, M2A e R2A. A maioria destes operadores altera a cardinalidade das associações. Isto significa que estes tipos de critérios não são capazes de revelar os defeitos descritos por estes operadores e que a abordagem aqui proposta deve ser usada como complementar para aumentar a confiança de que os produtos da LPS estão de acordo com a especificação.

Para avaliar os custos da aplicação dos mutantes, foram gerados 11 casos de teste adicionais para matar os mutantes restantes (penúltima coluna da Tabela 2). No total foram necessários 16 casos de teste (considerando a ordem de execução figSet +aetgSet+adicionais). A maioria dos mutantes requereu apenas um caso de teste. O operador REE foi o que mais requereu casos de teste. Este operador está dentre os que mais geraram mutantes. Entretanto, percebe-se que o número de casos de teste não cresce de maneira proporcional ao número de mutantes gerados, o que viabiliza a utilização prática da abordagem baseada em mutação e dos operadores propostos. Outros experimentos com outros diagramas OVM deverão ser conduzidos para que o custo possa ser melhor avaliado.

Foram encontrados 7 mutantes equivalentes, 3 gerados pelo operador RRE e 4 pelo REE. Estes mutantes foram gerados pela inclusão de uma relação (requires ou excludes) entre MP3 e algum variante relacionado a escolha alternativa do ponto de variação Screen. Para matar estes mutantes seria necessário criar um produto que contrarie as restrições, muitas vezes incluindo o variante Camera que possui uma relação de requires com High_Resolution. Isto requer que este último seja incluído, fazendo com que não seja possível incluir outros variantes Basic e Colour, devido a escolha alternativa ter cardinalidade [1,1]. Portanto qualquer produto será inválido para estes mutantes e original, não existe um produto capaz de diferenciar seus comportamentos.

6. Conclusões

Este artigo apresentou uma abordagem de teste baseada em mutação de variabilidades da LPS a partir do OVM. A ideia é a utilização um critério baseado em defeitos para

selecionar um conjunto de produtos, considerando erros comuns que podem estar presentes no OVM, descritos por um conjunto de operadores de mutação introduzido.

Para aplicar os operadores utilizou-se um processo de mutação semelhante ao teste de mutação de programas. Um OVM mutante é considerado morto se o seu resultado da validação de um caso de teste (produto) é diferente do resultado produzido pelo OVM original. Ao final, um escore de mutação é obtido, e pode ser usado para guiar a geração de produtos, ou para avaliar a qualidade de um conjunto disponível.

Tabela 2: Número de mutantes e de produtos necessários para cada critério

operador	# mutantes	# aetgSet	# figSet	# adicionais	# casos de teste
MVPE	0	0	0	0	0
MVPI	0	0	0	0	0
RVA	3	3 (3)	3(3)	0	3
ILI	1	1 (1)	1 (1)	0	1
DLI	2	0	0	2	2
ILS	1	0	0	1	1
DLS	1	0	0	1	1
MAO	2	0	0	2	2
MMO	1	0	0	1	1
MAM	2	2 (1)	2 (1)	0	1
MOM	2	2(2)	2(1)	0	1
M2A	1	0	0	1	1
R2A	2	0	0	2	2
RRE	29	24(4)	19 (3)	2	6
REE	26	15(4)	11(3)	4	8
MRE	1	1(1)	1(1)	0	1
MRR	1	1(1)	1(1)	0	1

Os operadores foram utilizados em um estudo de caso. Os resultados mostram a aplicabilidade da abordagem. O número de casos de teste não cresce proporcionalmente ao número de mutantes gerados. A abordagem foi comparada com o teste *pair-wise* e o teste de caminhos básicos. Ambos tipos de teste não foram capazes de matar diversos mutantes, ou seja, não revelam os defeitos descritos por eles. Os principais defeitos estão associados à cardinalidade das associações. Este estudo preliminar mostra que a abordagem baseada em mutação deve ser utilizada de uma forma complementar às existentes, sendo capaz de revelar outros tipos de defeitos e contribuindo para aumentar a confiança de que os produtos descritos usando o OVM estão de acordo com a especificação.

Como trabalho futuro, pretende-se implementar os operadores introduzidos e oferecer um suporte automatizado à geração dos produtos. Isto permitirá a condução de novos experimentos, com LPS (diagramas) maiores e avaliar escalabilidade. Isto levará ao refinamento dos operadores propostos e a proposição de estratégias para redução do custo, analogamente ao que acontece com o teste de mutação de programas.

Agradecimentos

Os autores agradecem ao CNPq e à Fundação Araucária pelo apoio financeiro.

Referências

- Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortés, A. (2007). FAMA: Tooling a framework for the automated analysis of feature models. In: International Workshop on Variability Modelling of Software Intensive Systems (VAMOS). pp. 129-134.
- Benavides, D., Segura, S., Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*. Vol. 35 (6), pp. 615–636.
- Cabral, I., Cohen, M.B., Rothermel, G. (2010). Improving the testing and testability of software product lines. In: International Conference on Software Product Lines: going beyond. pp. 241-255. SPLC'10, Springer-Verlag, Berlin, Heidelberg.
- Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C. (1996). The combinatorial design approach to automatic test generation. *IEEE Software* 13(5), 83-88.
- Cohen, M.B., Dwyer, M.B., Shi, J. (2006). Coverage and adequacy in software product line testing. In: ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis. pp. 53-63.
- Engström, E., Runeson, P. (2011). Software product line testing: a systematic mapping study. *Information and Software Technology* 53(1), 2 – 13.
- Lamancha, B.P., Usaola, M.P., Velthius, M.P. (2009). Software product line testing, a systematic review. In: Int. Conf. on Software Paradigm Trends. vol. 49, pp. 78 – 81.
- Lamancha, B.P., Usaola, M.P. (2010). Testing product generation in software product lines using pairwise for features coverage. In: Intern. Conf. on Testing Software and Systems. pp. 111-125. ICTSS'10.
- van der Linden, F., Schimd, K., Rommes, E. (2007) Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer.
- McGregor, J.D. (2001). Testing a software product line. Tech. rep., Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-2001-TR-022.
- Oster, S., Zink, M., Lochau, M., Grechanik, M. (2011). Pairwise feature-interaction testing for SPLs: potentials and limitations. In: Proceedings of the 15th International Software Product Line Conference, Volume 2. pp. 6:1-6:8. SPLC '11.
- Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.L. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In: Third Intern. Conf. on Software Testing, Verification and Validation. pp. 459-468. ICST '10.
- Pohl, K., Böckle, G. and van der Linden, F. (2005). Software Product Line Engineering Foundations, Principles, and Techniques, p.84. Springer.
- Roos, F. C. (2009). Automated Analysis of Software Product Lines - An Approach to Deal with Orthogonal Variability Models. University of Seville. (Thesis Project). p.5.
- Silveira Neto, P.A. da M., Carmo Machado, I.d., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R. (2011). A systematic mapping study of software product lines testing. *Information and Software Technology* 53(5), 407-423.
- Uzuncaova, E., Khurshid, S., Batory, D.F. (2010) Incremental test generation for software product lines. *IEEE Transactions on Software Engineering* pp. 309-322.
- Wong, W., Mathur, A., Maldonado, J. (1994) Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In: Software Quality and Productivity, Theory, Practice, Education and Training.

Execução Determinística de Programas Concorrentes Durante o Teste de Mutação

Rodolfo A. Silva, Simone R. S. Souza, Paulo S. L. Souza

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo (ICMC/USP)
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brasil

{adamshuk, srocio, pssouza}@icmc.usp.br

Abstract. *Mutation testing has been successfully used due its high effectiveness to reveal faults. However, when this criterion is applied in context of concurrent programs, new challenges are presented. Due to non-determinism, the execution of the program with one test input may traverses different and correct synchronization sequences. The challenge is to find the set of all possible synchronization sequences and evaluate the mutants behavior with them. In this paper is proposed the use of two approaches to deal with it. The results indicate that both approaches contribute in this direction, supporting the mutant analysis and the generation of feasible synchronization sequences.*

Resumo. *O Teste de Mutação tem sido usado com sucesso devido o sua alta eficácia em revelar defeitos. No entanto, quando este critério é aplicado no contexto de programas concorrentes, novos desafios surgem. Devido ao não determinismo, a execução do programa com um dado de teste pode percorrer diferentes sequências de sincronização corretas. O desafio é encontrar o conjunto de todas as sincronizações possíveis e, a partir delas, avaliar o comportamento dos mutantes. Neste artigo é proposto o uso de duas abordagens para apoiar a análise comportamental dos mutantes. Os resultados indicam que as abordagens contribuem nessa direção, colaborando com a análise dos mutantes e com a geração de sequências de execução executáveis.*

1. Introdução

A atividade de teste de software tem o objetivo de identificar defeitos no produto em teste e consiste em uma análise dinâmica do mesmo. O Teste de Mutação tem tido grande aceitação na comunidade de teste de software, que o considera um dos critérios mais efetivos para revelar defeitos, uma vez que melhorando o conjunto de casos de teste aplicados ao programa em teste, maiores são as chances de revelar defeitos [Souza, 2007]. Esse critério se baseia na construção dos mutantes a partir de um modelo de falhas específico para o domínio do problema. Esse modelo de falhas representa erros típicos normalmente cometidos no domínio. O objetivo desse critério é apoiar a geração de um conjunto de teste capaz de indicar que os defeitos inseridos não estão presentes no programa em teste, aumentando a confiança sobre a corretude desse programa.

Diferentemente de programas sequenciais, programas concorrentes possuem comportamento não determinístico, o que permite que diferentes execuções de um

programa concorrente com o mesmo valor de entrada possam produzir diferentes saídas corretas. Esse comportamento é produzido devido à comunicação, à sincronização e ao paralelismo existente entre os processos que fazem parte do programa concorrente. Por este motivo, a definição do teste de mutação para o contexto de programas concorrentes apresenta desafios extras em relação aos programas sequenciais. Uma vez que o programa possui mais de uma resposta diferente e correta, o fato da execução do mutante gerar um resultado diferente do apresentado pelo programa original não é o suficiente para matá-lo, pois a discrepância dos resultados pode ter ocorrido devido ao não determinismo e não pela presença da mutação no código.

Quando um mutante M é executado com um caso de teste t , ele deve ser morto se o comportamento apresentado é incorreto em relação à especificação do programa. No contexto de programas sequenciais, isso equivale a dizer que o resultado da execução do mutante com t é diferente da execução do programa original com t . Quando são considerados programas concorrentes, seria necessário comparar o conjunto de todos os resultados possíveis de M com t em relação a todo conjunto possível de resultados de P com t . Se os comportamentos não são os mesmos, então M pode ser considerado morto. Para isso, é necessário conhecer todo o conjunto possível de resultados de um programa concorrente para uma determinada entrada.

Este artigo propõe uma abordagem para realizar a análise comportamental de mutantes no contexto de programas concorrentes, baseando-se em abordagens definidas previamente para o teste determinístico [Carver 1993] e para o teste de alcançabilidade [Lei e Carver 2006]. Essas abordagens buscam contemplar dois aspectos diferentes durante a análise comportamental de mutantes que são: 1) a seleção das sequências de sincronização a serem executadas e 2) a garantia de que as mesmas sequências de sincronização sejam executadas tanto pelo programa em teste, como pelo programa mutante. Além disso, outro objetivo é garantir que o mutante e programa original sejam avaliados considerando sempre a mesma sequência de sincronização. A aplicabilidade da abordagem é ilustrada por meio de um programa concorrente que utiliza o ambiente de passagem de mensagens MPI (*Message Passing Interface*).

O artigo está organizado da seguinte forma: a Seção 2 apresenta conceitos importantes sobre programação concorrente e descreve trabalhos relacionados a este artigo. A Seção 3 apresenta as duas abordagens utilizadas e a proposta deste artigo, juntamente com um exemplo ilustrando sua aplicação. A Seção 4 apresenta as conclusões e desdobramentos deste trabalho.

2. Teste de Mutação para Programas Concorrentes

Um programa concorrente é formado por processos paralelos que interagem para resolver um problema complexo. A comunicação entre os processos paralelos pode ser realizada usando o paradigma de memória compartilhada ou passagem de mensagens. Quando memória compartilhada é usada, um processo modifica o valor de uma variável que pode ser lida e/ou atualizada por outro processo que compartilha o mesmo espaço de endereçamento. Já em passagem de mensagens, um processo envia uma mensagem que é recebida por outro, através do uso das primitivas básicas, como *send()* e *recv()*.

O padrão MPI [Forum 1995] é uma especificação extremamente popular que suporta o desenvolvimento de programas concorrentes com memória distribuída (ou

comunicação por passagem de mensagem). Em MPI uma computação é formada por um ou mais processos que se comunicam através de chamadas de sub-rotinas de envio e recebimento de mensagens. A comunicação é realizada através de duas formas. A primeira é conhecida como comunicação ponto a ponto e utiliza operadores *send* e *receive*. A segunda é chamada de comunicação coletiva e utiliza funções como *broadcast*, *barrier*, *gather*, entre outras. O MPI 2, no qual este artigo se relaciona, estende a especificação anterior com novas operações coletivas, comunicação unilateral, E/S, criação e gerência de processos e interfaces externas.

Conforme descrito anteriormente, o teste de mutação tem por objetivo avaliar a qualidade de um conjunto de casos de teste. Para isso, defeitos típicos são inseridos sistematicamente no programa em teste, criando os mutantes. O intuito é demonstrar que caso esses defeitos estivessem nesse programa eles seriam revelados pelo conjunto de teste. Assim, quanto mais próximo da realidade for o modelo de falhas usado para inserir defeitos, mais eficaz será o conjunto de testes gerado. Para a aplicação eficaz desse critério é importante identificar os erros comuns que são cometidos no domínio de aplicação considerado.

A aplicação de teste de mutação em programas concorrentes foi abordada inicialmente por Carver (1993), onde é apresentado um procedimento chamado DEMT (*Deterministic Execution Mutation Testing*) para a execução determinística dos mutantes, mostrado na Figura 1. Neste procedimento são geradas sequências de sincronizações aleatórias e, então, se aplica a execução determinística, selecionando-se casos de teste formados por uma entrada *X* e uma sequência de sincronização *S*. Assim, os mutantes devem ser executados pelo par (*X, S*). Esta abordagem é eficiente quanto à execução dos mutantes deterministicamente, porém encontrar as sequências de sincronização é um problema, uma vez que não se pode ter certeza que todas as sequências foram executadas. Para a criação dos mutantes, o autor adapta para a linguagem Ada alguns operadores de mutação definidos na ferramenta Mothra [King and Offut 1991], definida para o teste de mutação de programas em Fortran.

```

gerar mutantes (M1, M2, ..., Mn) a partir de P;
repita {
    aplicar teste não determinístico para executar aleatoriamente P com a entrada x;
    armazenar a sync-seq Sp gerada pela execução e a saída de P com x, output-x-P;
    analisar quais das condições ocorrem:
        (a) Sp é válida e output-x-P é correta;
        (b) Sp é válida e output-x-P é incorreta;
        (c) Sp é inválida e output-x-P é correta;
        (d) Sp é inválida e output-x-P é incorreta;
    se (b), (c) ou (d) então
        localizar e corrigir o erro de P;
        aplicar teste determinístico, forçando a execução de P com (x, Sp)
        para avaliar se o comportamento de P corrigido está correto;
    senão
        para cada mutante Mi, i <=n, faça
            aplicar teste determinístico, forçando a execução de Mi com (x, Sp)
            e produzindo a saída output-x-Mi;
            se Sp é não executável para Mi ou output-x-P <> output-x-Mi então
                marcar mutante Mi como morto(ou distinguido);
    fim-se;
} até escore de mutação atingir valor desejado;

```

Figura 1. Procedimento para o teste de mutação segundo Carver (1993)

Silva-Barradas (1998) propõe operadores de mutação para tratar aspectos de concorrência em programas na linguagem Ada. Para isso, foram identificados todos os comandos (ou construtores) da linguagem que se relacionam à concorrência, tais como: *accept* e *entry*. O autor propõe 12 operadores de mutação classificados como operadores para funções concorrentes e operadores para expressões condicionais. Para tratar o não determinismo, a instrumentação do programa é empregada para armazenar as sequências de sincronização do programa P . A sequência S gravada pode ser usada para reproduzir o mesmo comportamento em uma posterior execução de P com o caso de teste t . Juntando esta sequência de sincronização como parte do caso de teste é esperado que $M(t,S) = P(t,S)$, ou seja, o resultado da execução do mutante M com t reproduzindo a mesma sequência de sincronização deve ser o mesmo do programa original. A técnica considera que o mutante é distinguido caso uma das condições seja verificada: 1) $M(t,S) \neq P(t,S)$ ou 2) o mutante não consegue reproduzir a sequência S .

Delamaro et al. (2001) definem um conjunto de operadores de mutação para testar aspectos de concorrência e sincronização da linguagem Java. Foram identificadas as principais estruturas relacionadas à concorrência e, então, definidos os operadores de mutação para essas estruturas. Delamaro (2004) apresenta uma abordagem para reproduzir a execução de um programa concorrente em Java utilizando instrumentação. Esta abordagem baseia-se na técnica de *record and playback*, onde, na fase de *record* é gravada a sequência de sincronização ocorrida durante a execução de métodos e objetos sincronizados. Já na fase de *playback*, antes de uma *thread* entrar em um ponto sincronizado é necessário checar na sequência de sincronização se este é o próximo evento a ser executado. A abordagem apresentada pelos autores foi eficaz para a reprodução de programas concorrentes em Java usando a instrumentação, similar a abordagem de Silva-Barradas, porém utilizando uma estratégia de instrumentação diferente, visto que os aspectos de concorrência de Ada e Java são diferentes.

Bradbury et al. (2006) propõem a criação de 24 operadores de mutação para programas desenvolvidos em Java (J2SE 5.0). Os autores apresentam cinco categorias de operadores de mutação concorrentes para Java: 1) modificar parâmetros de métodos concorrentes; 2) modificar a ocorrência de chamadas a métodos concorrentes (removendo, substituindo e trocando); 3) modificar palavras-chave; 4) mudar objetos concorrentes e 5) modificar região crítica.

Os trabalhos anteriores são propostas para programas concorrentes com memória compartilhada. Considerando programas concorrentes com memória distribuída, as contribuições identificadas definem o teste de mutação para programas em PVM (*Parallel Virtual Machine*) [Giacometti et al. 2003] e para programas em SystemC [Sen 2009]. Nessa direção, em Silva et al. (2012) foram definidos operadores de mutação para o teste de programas concorrentes em MPI. Tais operadores exercitam as funcionalidades do padrão MPI em relação à comunicação e sincronização de processos concorrentes, considerando funções que realizam comunicação ponto a ponto e coletivas. Os operadores definidos são divididos em três categorias principais: 1) operadores aplicados em funções coletivas; 2) operadores aplicados em funções ponto a ponto e 3) operadores que podem ser aplicados a todas as funções MPI. A abordagem para apoio à análise de mutantes proposta neste artigo é relacionada com as pesquisas anteriores apresentadas em Silva et al. (2012).

3. Uma Abordagem para a Análise Comportamental de Mutantes

A solução para o teste de mutação no contexto de programas concorrentes apresentada por Carver (1993) engloba a avaliação dos mutantes em relação ao programa original, usando teste determinístico, ou seja, fazendo com que o mutante seja avaliado em relação à mesma sequência de sincronização aplicada ao programa original. Nesse procedimento considera-se um par (entrada, sincronização) e avalia o resultado do mutante com o mesmo par. O autor estende a proposta incluindo a necessidade de testar com as outras possibilidades de sincronização para a mesma entrada de teste, entretanto, o problema de como gerar todas essas possíveis sincronizações não é tratado.

Considerando tal cenário, dois aspectos precisam ser explorados: 1) a geração de todas as possíveis sincronizações para cada entrada de teste; 2) o teste determinístico do programa original e dos mutantes, forçando a ocorrência de determinadas sincronizações. Este trabalho investiga a utilização de duas abordagens para tratar esses aspectos, considerando o teste de mutação para programas concorrentes com memória distribuída. Com relação ao teste determinístico, a abordagem utilizada já é amplamente conhecida e vem sendo usada com sucesso para reproduzir sequências de sincronização [Carver 1993, Silva-Barradas 1998, Delamaro 2004, Hausen 2005]. Com relação à geração de todas as possíveis sincronizações, investigou-se uma abordagem conhecida como teste de alcançabilidade (*reachability testing*), a qual foi proposta com o intuito de gerar, em tempo de execução, variações de execuções de sincronização. O diferencial dessa abordagem em relação às demais é que ela gera as variações dinamicamente, com isso, são geradas somente sincronizações executáveis. Com base nessas abordagens, um exemplo é utilizado para ilustrar como essas abordagens podem ser aplicadas no contexto do teste de mutação. Funcionalidades da ferramenta de teste ValiMPI [Hausen 2005] são empregadas para apoiar a aplicação das abordagens.

3.1. Teste de Alcançabilidade

O teste de alcançabilidade, proposto por Lei e Carver (2006), obtém todas as sincronizações executáveis a partir de uma dada execução inicial. Seu objetivo é reduzir o número de sincronizações redundantes, gerando as possíveis combinações de sincronização. Diferentemente das abordagens anteriores, o teste de alcançabilidade determina, durante a execução, quais sincronizações são possíveis de ocorrer em uma nova execução. A técnica de teste baseado em prefixo é empregada para executar o programa deterministicamente até certo ponto e após esse ponto permitir a execução não determinística. As sequências de sincronização (*sync-sequences*) são derivadas automaticamente e *on-the-fly*, sem construir nenhum modelo estático.

A abordagem funciona da seguinte forma. A partir de uma execução inicial, gerada de maneira não determinística, uma sequência de sincronização inicial Q_0 é produzida. Com base em Q_0 , identificam-se “condições de disputa” entre pares de sincronizações e essas condições de disputa dão origem às variantes de Q_0 . As variantes são prefixos de Q_0 com uma ou mais sincronizações modificadas. Essas variantes são utilizadas para realizar execuções controladas baseadas em prefixo com a mesma entrada, exercitando as sincronizações da variante. Com isso, novas sequências de sincronização Q_i são geradas de modo a gerar todas as variações a partir de Q_0 e, a partir delas, todas as sincronizações possíveis para uma entrada de teste t .

3.2. Teste Determinístico

Várias abordagens podem ser utilizadas para forçar a re-execução de um programa P seguindo a mesma sequência de sincronização gravada em uma prévia execução. Como exemplo, temos o escalonador de processos do sistema operacional utilizando as mesmas escolhas ao determinar o processo a executar. Além disso, pode-se monitorar externamente a primeira execução de P e com o auxílio do monitor, forçar a execução da sequência de sincronização ao re-executar P . Por fim, tem-se a abordagem onde P é instrumentado com o objetivo de gravar a sequência de sincronização na primeira execução e seguir essa sequência de sincronização na re-execução [Delamaro 2004].

A terceira opção é a que foi escolhida pela abordagem proposta aqui. Ela é a mais simples de implementar, porém é mais restrita que as outras. Ela funciona inserindo funções de controle (*check-points*) em determinados pontos do código para que seja possível a gravação da sequência de execução dos processos e também da sequência de sincronização ocorrida durante a execução.

3.3. Abordagem Proposta

As duas abordagens anteriores são empregadas para apoiar a análise comportamental de programas concorrentes, adaptando o procedimento de Carver (1993). Nessa adaptação (Figura 2), o teste de alcançabilidade é empregado para gerar todas as sequências de sincronização a partir de uma entrada de teste X . Novas entradas de teste são adicionadas à medida que é necessário melhorar o escore de mutação. Com isso, é possível avaliar cada mutante com todas as combinações de sincronização que podem ocorrer a partir de X . Isso é importante porque é possível avaliar os mutantes com todas as sequências de sincronização possíveis para o programa original e, com isso, todas as possibilidades são cobertas.

```
gerar mutantes (M1, M2, ..., Mn) a partir de P;
repita {
    aplicar teste não determinístico para executar aleatoriamente P com a entrada x;
    armazenar a sync-seq Sp gerada pela execução e a saída de P com x, output-x-P;
    analisar quais das condições ocorrem:
        (a) Sp é válida e output-x-P é correta;
        (b) Sp é válida e output-x-P é incorreta;
        (c) Sp é inválida e output-x-P é correta;
        (d) Sp é inválida e output-x-P é incorreta;
    se (b), (c) ou (d) então
        localizar e corrigir o erro de P;
        aplicar teste determinístico, forçando a execução de P com (x, Sp)
        para avaliar se o comportamento de P corrigido está correto;
    senão
        aplicar o teste de alcançabilidade e gerar o conjunto Seq(x) composto por
        todas as sync-seqs Spi geradas a partir de Sp;
        para cada Spi pertencente a Seq(x) faça
            para cada mutante Mi, i <=n, faça
                aplicar teste determinístico, forçando a execução de Mi com (x, Spi)
                e produzindo a saída output-x-Mi;
                se Spi é não executável para Mi ou output-x-P <> output-x-Mi então
                    marcar mutante Mi como morto (ou distinguido);
            fim-se;
} até escore de mutação atingir valor desejado;
```

Figura 2. Procedimento do teste de mutação adaptado

Com intuito de avaliar a aplicabilidade dessa abordagem, a mesma foi utilizada considerando o programa *Greatest Common Divisor* (GCD). Esse programa calcula o maior divisor comum entre três números fornecidos, usando quatro processos concorrentes. O processo mestre ($p0$) recebe os três valores x , y , z e chama os três processos escravos ($p1$, $p2$ e $p3$). O processo $p0$ envia x e y para $p1$ e y e z para $p2$. Os dois processos escravos recebem os valores, encontram o máximo divisor comum entre os dois valores e enviam o resultado para o processo mestre. Se um dos valores retornados for 1, então $p3$ é finalizado. Caso contrário, esses valores são enviados para $p3$ que calcula o máximo divisor comum final, retornando o resultado para $p0$.

A execução da abordagem proposta consiste em cinco passos principais: 1) Instrumentação e compilação do programa original; 2) Execução dos casos de teste, gravando a sequência de sincronização entre os processos; 3) Geração das variações das sequências de sincronização, aplicando o teste de alcançabilidade; 4) Criação dos mutantes a partir do programa instrumentado e 5) Execução determinística dos mutantes com os casos de teste, forçando a execução das mesmas sequências de sincronização obtidas pelo programa original.

Algumas funcionalidades da ferramenta de teste ValiMPI [Hausen 2005] foram utilizadas na aplicação do teste de alcançabilidade e na execução determinística. A ValiMPI é uma ferramenta de teste estrutural desenvolvida para o teste de programas concorrentes em MPI. As funcionalidades disponíveis nessa ferramenta são distribuídas em módulos, sendo que, para este trabalho são utilizados os módulos Vali-Inst, Vali-Exec e Vali-Sync.

O primeiro passo consiste em utilizar o módulo Vali-Inst para a instrumentação do programa original, que é realizada por meio da inserção de *check-points* (Figura 3), necessários para a gravação da sequência de sincronização entre os processos e da sequência de comandos executados em cada processo. Por meio dos *check-points* é possível, por exemplo, obter informações sobre qual par *send-receive* sincronizou, armazenando o número dos processos e a linha de código em que está cada primitiva.

O segundo passo consiste em utilizar o módulo Vali-Exec para criar uma versão executável do programa instrumentado. Com isso, é possível executar casos de teste no programa gerando: a saída do teste, o traço de execução e a sequência de sincronização de cada processo. A Figura 4 apresenta o conteúdo do arquivo que contém a sequência de sincronização do processo 1. Cada linha do arquivo segue a sintaxe “*recv_no n :send_no m processo x com_a_tag t*”, significando que o *receive* do nó **n** do processo ao qual o arquivo se refere sincroniza com o *send* do nó **m** pertencente ao processo **x** e que tem a *tag* de mensagem **t**. No exemplo da Figura 4, o *receive* do nó 2 do processo 1 sincroniza com o *send* do nó 2 pertencente ao processo 0 e que tem a *tag* de mensagem 1. Esses arquivos armazenam as sincronizações para cada primitiva *receive* de cada processo. No caso do programa da Figura 3, tem-se um *receive* na linha 10 e um *send* na linha 28; a sincronização desse *send* está descrita no arquivo de trace do processo, o qual recebe a mensagem enviada por esse processo.

Após a execução do programa original, utiliza-se o módulo Vali-Sync para obter todas as sequências de sincronização para o dado de teste inserido. A Figura 5 (A) apresenta a sequência de sincronização do programa GCD quando executado com a entrada “10 20 30”. Nos nós 5 e 7 de $P0$ existem *receives* não determinísticos, os quais

recebem os resultados dos processos P_1 e P_2 , não importando a ordem. Com isso, duas possibilidades de sincronizações podem ocorrer para esses nós: $\{(P_2, P_0), (P_1, P_0)\}$ e $\{(P_1, P_0), (P_2, P_0)\}$, as quais são ilustradas na Figura 5 (A) e (B) respectivamente.

```

01 void
02 Slave(int rank)
03 {
04     int buf[2];
05     VALIMPI_TRACE DECL;
06     VALIMPI_REQ_LIST DECL;
07     Valimpi_Req_list_init(&valimpiListReq);
08     Valimpi_Init_trace(&valimpiTrace, "Slave");
09     Valimpi_Check_trace(&valimpiTrace, 1);
10    Valimpi_Check_trace(&valimpiTrace, 2);
11    Valimpi_Recv_trace(&valimpiTrace, buf, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
12    Valimpi_Check_trace(&valimpiTrace, 3);
13    while (Valimpi_Check_trace(&valimpiTrace, 4), buf[0] != buf[1])
14    {
15        Valimpi_Check_trace(&valimpiTrace, 5);
16        if (buf[0] < buf[1])
17        {
18            Valimpi_Check_trace(&valimpiTrace, 6);
19            buf[1] = buf[1] - buf[0];
20        }
21        else
22        {
23            Valimpi_Check_trace(&valimpiTrace, 7);
24            buf[0] = buf[0] - buf[1];
25        }
26    }
27    Valimpi_Check_trace(&valimpiTrace, 9);
28    Valimpi_Send_trace(&valimpiTrace, buf, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
29    Valimpi_Check_trace(&valimpiTrace, 10);
30    Valimpi_End_trace();
31 }

```

Figura 3. Processo escravo do programa GCD instrumentado pela Valimpi

```
recv_no 2 :send_no 2 processo 0 com_a_tag 1
```

Figura 4. Sequência de sincronização do processo 1 do programa GCD

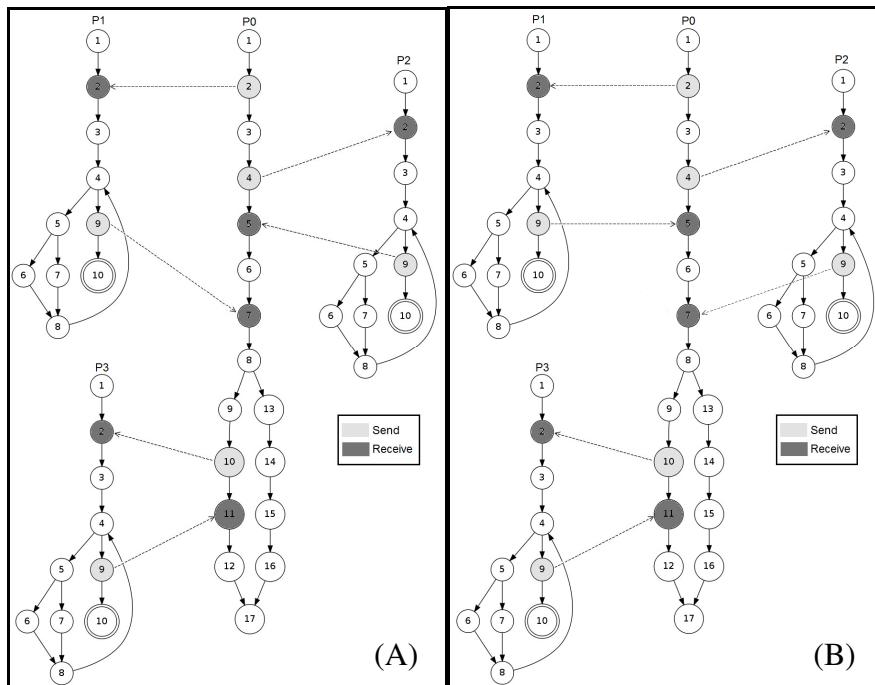


Figura 5. Sequência de sincronização original (A) e alternativa (B)

O quarto passo consiste em criar e executar os mutantes. Com base nos operadores de mutação definidos por [Silva et al. 2012] é possível criar um conjunto de mutantes. Um exemplo é o operador de mutação *DelRecv*, que tem como objetivo

excluir todas as ocorrências de funções de recebimento de mensagens (funções *receive*). Ao utilizar este operador no programa exemplo, um dos mutantes criados é o apresentado na Figura 6. Onde antes havia uma função instrumentada para recebimento de mensagens, agora há uma linha vazia (linha 10). Um aspecto importante aqui é que, por realizar a mutação diretamente no programa instrumentado, têm-se as mesmas informações de trace obtidas pelo programa original, facilitando a análise do comportamento do mutante.

```

01 void
02 Slave(int rank)
03 {
04     int buf[2];
05     VALIMPI_TRACE DECL;
06     VALIMPI_REQ_LIST DECL;
07     ValiMPI_Req_list_init(&valimpiListReq);
08     ValiMPI_Init_trace(&valimpiTrace, "Slave");
09     ValiMPI_Check_trace(&valimpiTrace, 1);
10     ValiMPI_Check_trace(&valimpiTrace, 2);
11
12     ValiMPI_Check_trace(&valimpiTrace, 3);
13     while (ValiMPI_Check_trace(&valimpiTrace, 4), buf[0] != buf[1])
14     {
15         ValiMPI_Check_trace(&valimpiTrace, 5);
16         if (buf[0] < buf[1])
17         {
18             ValiMPI_Check_trace(&valimpiTrace, 6);
19             buf[1] = buf[1] - buf[0];
20         }
21         else
22         {
23             ValiMPI_Check_trace(&valimpiTrace, 7);
24             buf[0] = buf[0] - buf[1];
25         }
26         ValiMPI_Check_trace(&valimpiTrace, 8);
27     }
28     ValiMPI_Check_trace(&valimpiTrace, 9);
29     ValiMPI_Send_trace(&valimpiTrace, buf, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
30     ValiMPI_Check_trace(&valimpiTrace, 10);
31     ValiMPI_End_trace();
32 }
```

Figura 6. Mutante gerado com a aplicação do operador DelRecv

O quinto e último passo consiste em utilizar o teste determinístico (módulo ValiExec, opção re-execução) para executar os mutantes com os casos de teste e sequências de sincronização obtidas na execução do programa original. Considerando as possíveis sincronizações (Figura 5) e o mutante da Figura 6 observa-se que esse mutante não consegue executar essas sincronizações, pois a linha com o *receive* foi eliminada. O resultado disso é a ocorrência de um *deadlock*, uma vez que é esperada uma comunicação que nunca irá ocorrer. Com isso, esse mutante é marcado como morto. O mesmo processo é realizado para os demais mutantes gerados. Vale ressaltar que a determinação que o mutante é morto é feita automaticamente. Para a análise, irão restar somente aqueles que não foram mortos. Para esses, novos casos de teste podem ser adicionados, caso não sejam equivalentes ao programa original.

4. Conclusão

Este trabalho apresenta uma abordagem para análise comportamental de mutantes no contexto de programas concorrentes. Dois aspectos importantes são explorados que é a execução de todas as combinações de sincronização e a execução determinística dos mutantes. Os resultados obtidos indicam que a utilização da instrumentação para apoiar a execução determinística facilita a análise dos mutantes. Com relação à geração das combinações de sincronização, o uso do teste de alcançabilidade preenche uma lacuna até então não explorada no contexto de teste de mutação, resolvendo o problema de gerar somente sincronizações executáveis e não redundantes. As funcionalidades presentes na ferramenta de teste ValiMPI mostraram-se adequadas para essa

investigação e deverão ser utilizadas para o desenvolvimento da ferramenta de apoio ao teste de mutação.

Uma limitação dessa abordagem é que alguns mutantes só são distinguidos se forem executados com sincronizações não executáveis no programa original. Como o teste de alcançabilidade só gera sincronizações executáveis, essa sequência não estará presente no conjunto de possibilidades e esse mutante permanecerá vivo. Uma solução é permitir que o testador analise esse mutante e, a partir disso, decida se o mutante é equivalente ou se o mesmo representa um erro no programa original. Esse aspecto será explorado na continuidade deste trabalho.

Agradecimentos

Os autores gostariam de agradecer à FAPESP pelo apoio financeiro sobre o processo de número 2010/04935-6.

Referências

- Bradbury, J. S., Cordy, J. R. and Dingel, J. (2006) “Mutation Operators for Concurrent Java (J2SE 5.0)”, In: Proceedings of 2nd Workshop on Mutation Analysis, USA, p. 11-20.
- Carver, R. (1993) “Mutation-based Testing of Concurrent Programs”, In: Test Conference, USA, pages 845-853.
- Delamaro, M. E. (2004) “Using Instrumentation to Reproduce the Execution of Java Concurrent Programs”, In: Simpósio Brasileiro de Qualidade de Software, Brasil.
- Delamaro, M., Maldonado, J. C., Pezzè, M. and Vincenzi, A. (2001) “Mutant Operators for Testing Concurrent Java Programs”, In: Simpósio Brasileiro de Engenharia de Software.
- Forum, M. P. I. (1995) “MPI: A Message-passing Interface Standard”, Relatório Técnico.
- Giacometti, C., Souza, S. R. S. and Souza, P. S. L. (2003) “Teste de Mutação para a Validação de Aplicações Concorrentes Usando PVM”. In: Revista Eletrônica de Iniciação científica, volume 2.
- Hausen, A. C. (2005) “ValiMPI : Uma Ferramenta de Teste Estrutural para Programas Paralelos em Ambiente de Passagem de Mensagem”, Dissertação de Mestrado, UTFPR.
- King, K. N. and Offutt, A. J. (1991) “A Fortran Language System for Mutation-based Software Testing”, In: Software-Practice and Experience, p. 685–718.
- Lei, Y. and Carver, R. H. (2006) “Reachability Testing of Concurrent Programs”, IEEE Transactions on Software Engineering, p.382-403.
- Sen, A. (2009) “Mutation Operators for Concurrent SystemC Designs”, In: 10th International Workshop on Microprocessor Test and Verification, USA, p. 27-31.
- Silva, R. A., Souza, S. R. S. and Souza, P. S. L. (2012) “Mutation Testing for Concurrent Programs in MPI”, In: 13th Latin American Test Workshop, Quito, Equador, p 69-74.
- Silva-Barradas, S. (1998). “Mutation Analysis of Concurrent Software”, Tese de Doutorado, Dottorato di Ricerca in Ingegneria Informatica e Automatica.
- Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L. (2007). “Teste de programas concorrentes”, In: Maldonado, J. C.; M., J.; Delamaro, M. E., eds. “Introdução ao Teste de Software”, v. 1, Elsevier Editora Ltda.

Uso de análise de mutantes para avaliação de uma abordagem de testes baseados em modelos: um estudo exploratório

Isaura Rennaly Souto Lima¹, Thaise Yano¹, Eliane Martins¹

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

isaurarennalysl@gmail.com, thaiseyano@gmail.com, eliane@ic.unicamp.br

Abstract. In this paper, we discuss the use of mutant analysis to evaluate a set of test cases generated from state models. The use of this technique to evaluate the fault-revealing effectiveness of test cases is no novelty, but in this paper we apply a slightly different approach. We defined a executable model, in which a compiler generates the behavioral model source code, which is platform independent. The paper presents the results of an exploratory study, in which the code of an executable model has been mutated. This study served not only to determine how to evaluate a test set, but also the factors that could affect the fault-revealing power of a test case.

Resumo. Nesse artigo apresentamos o uso de análise de mutantes para avaliar os casos de teste gerados a partir de modelos de estados. O uso desta técnica para avaliar o potencial de casos de teste em revelar erros não é uma novidade, mas o diferencial deste trabalho está na abordagem utilizada. Apresentamos os resultados de um estudo exploratório em que utilizamos mutação do código de um modelo executável. Esse estudo serviu de base para determinar a forma de avaliar o conjunto de testes, e também para determinar alguns fatores que podem afetar o potencial dos casos de teste em revelar erros.

1. Introdução

Atualmente, um dos grandes desafios no desenvolvimento de software consiste em melhorar a produtividade, e ao mesmo tempo garantir um produto de boa qualidade. Dado que os prazos para a produção de software são cada vez mais curtos, técnicas que ajudem na melhoria da qualidade devem, também ser realizadas de forma a causar um mínimo de impacto nos prazos de entrega.

A realização de testes é uma atividade incontornável para determinar se um software apresenta as qualidades desejadas. Um ponto importante consiste, portanto, em automatizar as atividades de teste, para que a qualidade não acabe sendo sacrificada em prol do ganho em produtividade. Uma das atividades mais dispendiosas é a derivação dos objetivos de teste, isto é, determinar quais itens serão testados e como [Jino et al. 2007]. A derivação manual é uma forma pouco estruturada, mal documentada e difícil de reproduzir. Além disso, a qualidade dos casos de teste gerados é muito dependente da engenhosidade do testador.

Testes baseados em modelos visam mitigar esses problemas através da derivação de testes a partir de modelos representando o comportamento do sistema (ou de seu ambiente). Máquinas de Transição de Estados (MTE) representam sequências de ações (ou cenários de uso) que podem ocorrer durante a execução do sistema em teste.

Dentre as várias formas de representar MTEs, as máquinas finitas de estado (MFE) são muito utilizadas para o aspecto de controle, ou seja, a ordem em que as ações podem ser realizadas, e inúmeras técnicas para geração automática de testes a partir desse tipo de modelo foram propostas [Petrenko and Bochmann 1996]. No entanto, esse modelo apresenta limitações para a representação de sistemas mais complexos, pois o número de estados pode se tornar muito grande, levando ao problema de explosão de estados. Esse problema é causado, em parte, quando se quer representar o aspecto de dados [Bochmann and Petrenko 1994]. Com isso, utiliza-se uma extensão da MFE, as máquinas finitas de estado estendida (MFEE), para representar esses aspectos, em que são introduzidas as variáveis de contexto e parâmetros de eventos. Esse modelo é a base para muitas técnicas de especificação usadas na prática, dentre elas, o modelo de estados da UML.

No entanto, derivar casos de teste a partir de MFEE ainda é um desafio. Aplicar técnicas existentes para MFE, que só leva em conta aspectos de controle, leva à produção de muitos caminhos infactíveis, que correspondem a cenários de uso impossíveis de serem executados, devido à existência de parâmetros e variáveis de contexto. O que acontece é que a maioria das técnicas propostas produzem caminhos de teste e, em seguida, os dados que ativam os caminhos [Derderian et al. 2005, Kalaji et al. 2009, Zhao et al. 2010].

Em um trabalho prévio foi proposto um método para geração de casos de teste a partir de MFEE, denominado MOST (*Mult-Objective Search Based Testint*) [Yano 2011], em que aspectos controle e dados são considerados simultaneamente, para, assim, evitar a geração de caminhos infactíveis. A técnica é dinâmica, e se baseia no uso de modelos executáveis.

De acordo com Mellor e Balcer [Mellor and Balcer 2002], modelos executáveis constituem um dos pilares da Arquitetura Dirigida por Modelos MDA (*Model Driven Architecture - MDA*). Em MDA existe a noção de Modelo Independente de Plataforma (*Platform Independent Model - PIM*), no qual o modelo do sistema independe de aspectos de implementação. O PIM pode ser especificado com o uso de modelo executável. Sendo executável, o modelo pode ser testado antes mesmo de se ter o código final. Assim sendo, é possível a geração dinâmica de casos de teste, em que o modelo é executado e os caminhos observados durante a execução são possíveis candidatos a serem selecionados. Caminhos infactíveis não são produzidos, pois não são observados durante a execução do modelo.

Neste trabalho o objetivo é avaliar, experimentalmente, os casos de teste gerados pelo método MOST. Para tanto, aplicamos o teste de mutação. Essa técnica é muito comum nos testes de unidade, em que falhas típicas de implementação são introduzidas no código, produzindo diversos mutantes. Após a aplicação do conjunto de testes aos mutantes gerados, obtém-se um escore de mutação, que pode ser usado para avaliar a adequação do conjunto de testes em revelar a presença de erros. O diferencial neste trabalho é que utilizamos o código do modelo executável, ao invés do código da implementação final, para avaliar o conjunto de testes. O estudo exploratório apresentado neste artigo visou não só avaliar a eficácia do conjunto de testes em revelar erros, mas também determinar os fatores que afetam essa eficácia. As seguintes questões foram consideradas para este estudo:

1. *Quais tipos de erros são mais difíceis de serem revelados?*

2. *Os casos de teste mais longos revelam mais erros do que os casos de teste mais curtos?*
3. *Os caminhos de teste com maior número de transições diferentes revelam mais erros?*

O restante do artigo está organizado da seguinte forma, a seção 2 apresenta uma fundamentação teórica sobre o assunto. A seção 3 descreve a abordagem experimental utilizada, o estudo do caso e a ferramenta utilizada. Os resultados obtidos a partir da abordagem desenvolvida estão localizados na seção 4. A seção 5 apresenta alguns trabalhos relacionados. Finalmente, a seção 6 conclui o trabalho.

2. Fundamentação Teórica

Nesta seção são apresentados os principais conceitos envolvidos neste trabalho, tais como MFEE, testes baseados em busca e teste de mutação.

2.1. Máquina Finita de Estado Estendida

Máquinas Finitas de Estado Estendidas (MFEE) pode ser definida como uma tupla (S, s_0, I, O, T, V, P) , sendo S, I, O, V, P e T um conjunto finito de estados, eventos de entrada, eventos de saída, variáveis, parâmetros de entrada e transições, respectivamente, e $s_0 \in S$ o estado inicial. Eventos de entrada podem conter parâmetros que pertencem a P . Cada transição $t_x \in T$ é descrita por: $i_x[g_x]/a_x$, sendo $i_x \in I$, g_x uma condição de habilitação da transição chamada de guarda e a_x a ação executada quando a transição é disparada. As partes g_x e a_x podem ser opcionais. Para disparar t_x , a máquina deve receber a entrada i_x no estado de origem de t_x e satisfazer g_x . Se g_x é verdadeiro, a_x é executada e o estado corrente se torna o estado destino de t_x . Uma ação pode conter comandos de atribuição ou produzir eventos de saída, que podem mudar os valores das variáveis do modelo.

Um caminho em uma MFEE representa uma sequência adjacente de transições no modelo. No contexto de geração de casos de teste, nos interessam caminhos que passem por uma ou mais transições-alvo. Os eventos de entrada e os valores dos parâmetros que permitem que um caminho seja executado formam uma sequência de entrada. Várias sequências de entrada podem executar um dado caminho, variando-se os valores dos parâmetros. O objetivo da geração dos testes é encontrar pelo menos uma sequência de entrada que faça com que um determinado caminho seja executável.

2.2. Teste Baseado em Busca

No teste baseado em busca (em inglês, *Search Based Software Testing*, SBST), os casos de teste são automaticamente gerados utilizando-se técnicas de otimização. As possíveis entradas do sistema são consideradas o espaço de busca da técnica de otimização. O objetivo é selecionar os dados que mais se aproximam de atender critérios de teste, tais como cobertura de instruções, caminhos e predicados.

Embora a maioria dos trabalhos em SBST esteja relacionada ao teste baseado em código, existem também iniciativas para o teste baseado em modelo, em que sequências de teste são geradas para cobrir, por exemplo, estados, transições e caminhos no modelo. Neste trabalho, utilizamos uma abordagem SBST para o teste baseado em MFEE, em que o espaço de busca é constituído pelas sequências de entrada e pelos valores dos

parâmetros dos eventos de entrada. Dado que o número de possíveis soluções é muito grande, procura-se então explorar esse espaço para encontrar as soluções que melhor atendam aos objetivos dos testes.

2.3. Teste de Mutação

No teste de mutação, são realizadas pequenas alterações sintáticas em um programa P , gerando um conjunto de programas mutantes P' similares a P [DeMillo et al. 1978]. Tais alterações são geradas por operadores de mutação que, em geral, estão associados a um tipo ou classe de erros que se pretende revelar em P [Fabbri et al. 1999].

O processo de gerar dados de entrada capazes de diferenciar o comportamento do programa original e de seus mutantes auxilia a geração de um conjunto de CT . Uma outra vantagem do teste de mutação é que pode ser utilizado para avaliar a adequação de um conjunto de CT em revelar determinadas falhas típicas cometidas ao desenvolver um sistema.

Todos os mutantes são executados utilizando-se um conjunto de CT . Se o programa mutante P' apresenta diferentes resultados do programa original P , ele é considerado morto, caso contrário, ele é dito vivo. Quando um mutante permanece vivo, é preciso verificar se P' e P são equivalentes. Nesse caso, não existe caso de teste capaz de distinguir P' e P .

A avaliação da adequação do conjunto de CT é calculada por um escore de mutação:

$$ms(P, CT) = \frac{DM(P, CT)}{M(P) - EM(P)} \quad (1)$$

onde: $DM(P, CT)$ representa o número de mutantes mortos em CT ; $M(P)$ equivale a quantidade de mutantes gerados; e $EM(P)$ consiste no número de mutantes equivalentes a P . Escore de mutação é a relação entre o número de mutantes mortos e o número total de mutantes não equivalentes gerados. O escore varia no intervalo de 0 e 1. Quanto maior for o seu valor, mais adequado será o conjunto de casos de teste para o teste do programa.

2.4. Geração dos casos de teste

A abordagem utilizada para a geração de testes foi descrita em outros artigos [Yano 2011, Yano et al. 2011]. Aqui faremos uma breve apresentação da abordagem, para deixar o texto auto-contido, mas recomendamos ao leitor que consulte as referências supra-citadas caso queira obter maiores detalhes. A abordagem consiste basicamente na realização dos seguintes passos: (i) obter o modelo de estados que representa o comportamento do sistema; (ii) validar o modelo, criando sua versão executável para determinar se atende ao comportamento esperado do sistema; (iii) instrumentar o modelo executável do sistema, de modo a obter o caminho percorrido por uma determinada sequência de teste; (iv) gerar as sequências de entrada que percorram caminhos que passem por uma determinada transição esperada. Neste passo é utilizado um algoritmo de otimização para produzir as sequências de teste. A abordagem é geração, ou seja, o modelo é executado para uma dada sequência de entrada, obtendo-se o caminho percorrido com a instrumentação verificando-se se a transição alvo faz parte do caminho ou não. Dado

que várias sequências de teste podem satisfazer ao critério, procura-se também selecionar aquelas que tenham menor comprimento. O algoritmo tenta então obter soluções (i.e., sequências de entrada) que satisfaçam a ambos os objetivos, quais sejam: cobrir a transição-alvo com menor comprimento de caminho. É possível encontrar várias soluções ótimas para uma dada transição, conforme está mostrado na seção 3.2.

Visto que os caminhos de transições são dinamicamente obtidos durante a execução do modelo, ao invés de realizar, de forma estática, uma análise de alcançabilidade, o problema de geração de caminhos infactíveis é evitado. Esse é um importante aspecto a ser considerado no teste a partir de MFEE devido aos conflitos de dados ao longo de um caminho.

3. Abordagem experimental utilizada

Esta seção apresenta a abordagem utilizada neste estudo, cujo objetivo foi o de levantar subsídios para experimentos subsequentes.

3.1. Objetivos

O objetivo dos experimentos foi avaliar a capacidade do conjunto de testes em revelar a presença de erros e, principalmente, determinar quais fatores podem afetar essa eficácia. Para tanto, os experimentos foram realizados visando responder as seguintes questões:

1. *Quais tipos de erros são mais difíceis de serem revelados?* Para responder a essa questão, analisamos quais operadores geram mutantes que foram mais difíceis de matar. Um mutante M_1 é mais difícil de matar do que um mutante M_2 , se M_1 é morto por um número menor de casos de teste do que M_2 . A análise é feita a partir da porcentagem calculada do número de mutantes mortos, com os casos de teste sobre o total de mutantes gerados, por operador de mutação.
2. *Os caminhos mais longos revelam mais erros?* Para isso determinamos o número de mutantes mortos pelas sequências de teste mais longas e comparamos com o número de mutantes mortos pelas sequências mais curtas.
3. *Os caminhos com maior número de transições diferentes revelam mais erros?* Utilizamos nesse caso uma abordagem similar à da questão 2, considerando-se o número de transições diferentes que existem em um caminho.

3.2. Estudo de caso

A partir do método MOST [Yano 2011], foram gerados casos de teste para cada transição do modelo de estados. Neste trabalho, utilizamos um modelo que representa o comportamento de um sistema de caixa eletrônico (figura 1).

Para produzir o modelo executável foi utilizada a ferramenta SMC (*State Machine Compiler*)¹. Essa ferramenta aceita um modelo em um formato de entrada (.sm) e o converte para diferentes linguagens de programação. A figura 2 ilustra o processo de criação do modelo. A classe *AtmContext.java* implementa o modelo de estados utilizando o padrão State [Gamma et al. 1995]. A classe *ATM.java* foi criada para os testes, e funciona como um driver, fornecendo as entradas, coletando e armazenando em um log as saídas produzidas pelo modelo.

¹Disponível em: <http://smc.sourceforge.net>

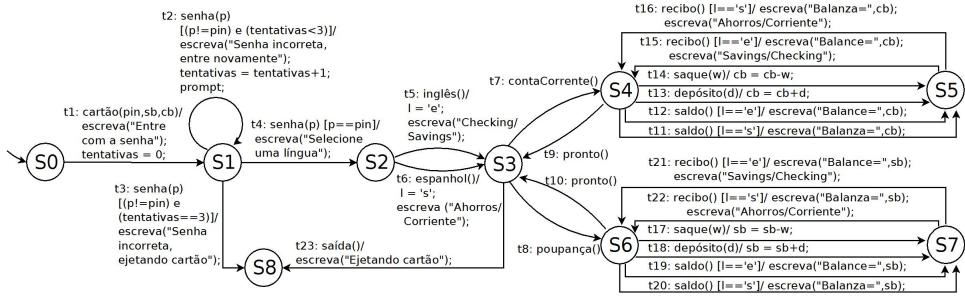


Figura 1. Sistema de caixa eletrônico representado em uma MFEE [Korel et al. 2003].

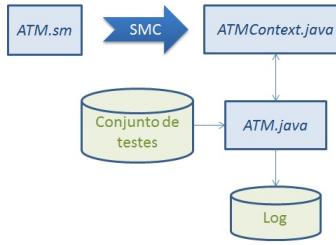


Figura 2. Visão esquemática do modelo executável.

3.3. Ferramenta utilizada

Para realizar o teste de mutação fizemos uso de uma ferramenta comumente utilizada, μ Java. Esta é uma ferramenta de mutação para programas Java, desenvolvida por Ma et al [Ma et al. 2005]. Ela gera mutantes automaticamente de acordo com o operador de mutação selecionado. Tais operadores são classificados em operadores de classe e operadores de métodos, conforme mostra a figura 3. Os operadores de métodos são as tradicionais mudanças sintáticas no código, tais como troca de um operador relacional por outro (operador ROR), ou substituição de uma variável escalar por outra (operador SVR). Já os operadores de classes afetam características de orientação a objetos, tais como herança e polimorfismo, como por exemplo, substituição de um método de acesso (operador EAM), chamada de método com o tipo da classe filho (operador PNC). Dado que a classe *ATMContext.java* é a que implementa os estados e transições, esta foi escolhida como classe-alvo para as mutações.

O primeiro passo, para a avaliação dos casos de testes gerados, consiste da geração de mutantes. Para gerar os mutantes, inicialmente são selecionados todos os operadores de mutação disponíveis na ferramenta.

Depois de selecionados os operadores e as classes-alvo, são gerados os mutantes para os operadores correspondentes. A figura 4 ilustra a geração de mutante tradicional (métodos) (a), enquanto a parte (b) mostra um mutante gerado para um operador de classe. O próximo passo consiste no cálculo do escore para o conjunto de testes, conforme equação 1.

4. Resultados

Na tabela 1, são apresentados os escores de mutação obtidos de acordo com o tipo de operador. Cada mutante foi executado com todos *CT* gerados por MOST. O *ms₁* corres-

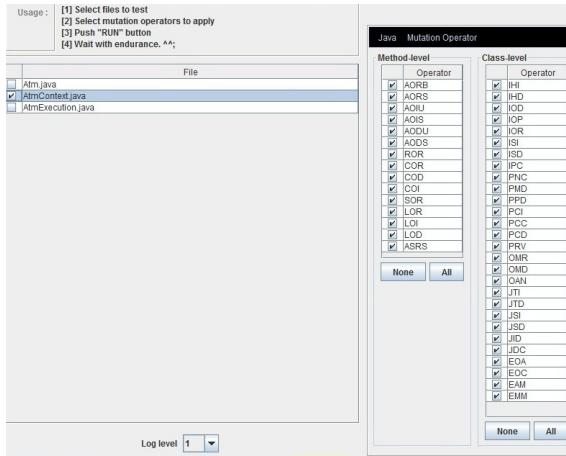


Figura 3. Operadores de métodos e classes.

Op	#	Summary
AQIS_53	1	Original
AQIS_54	1	(line 513) cbt.getL() == 0 => cbt.getL() > 0
AQIS_21	1	if (cbt.getL() == 1){
AOI_4	1	context.getState().Exit(context);
AOIU_31	1	context.clearState();
AODS_86	1	try {
AOI_2	1	cbt.mprint("Balance = ", cbt.getCb());
AOI_16	1	cbt.write("Ahorros Corriente");
AOI_17	1	} finally {
ROR_182	1	context.setState(AtmEFSM.S4);
COR_8	1	}
COD_0	1	context.getStat().Entry(context);
COI_40	1	}
SOR_0	1	}
LOD_0	1	}
LOI_37	1	} else {
ROR_38	1	super.Receipt(context);
ROR_39	1	}
ROR_40	1	}
ROR_41	1	}
ROR_42	1	}
ROR_53	1	if (cbt.getL() > 0) {
ROR_54	1	context.getState().Exit(context);
ROR_55	1	context.clearState();
ROR_56	1	try {
ROR_57	1	cbt.mprint("Balance = ", cbt.getCb());
ROR_58	1	cbt.write("Salvings Checking");
ROR_59	1	} finally {
ROR_60	1	context.setState(AtmEFSM.S4);
ROR_61	1	context.getStat().Entry(context);
ROR_62	1	}
AQIS_55	1	} else {
AQIS_56	1	if (cbt.getL() == 1){
AQIS_22	1	context.getState().Exit(context);
LOI_25	1	context.clearState();

Op	#	Summary
IH1_0	1	Original
EAM_1	1	(line 303) cbt.getPin() => cbt.getAttempts()
EAM_10	1	/
EAM_11	1	EAM_12
IHD_0	1	289
EAM_13	1	290
IOD_0	1	291
IOP_0	1	292
JID_0	1	private static final class AtmEFSM_S1 extends AtmContext.AtmEFSM_Default
ISI_0	1	293
ISD_0	1	{
IPC_0	1	294
PHC_0	1	private AtmEFSM_S1(java.lang.String name, int id)
PMD_0	1	295
PCI_0	1	296
PCCD_0	1	297
PCD_0	1	super(name, id);
PRV_0	1	298
EAM_19	1	299
EAM_20	1	300
EAM_21	1	protected void PIN(AtmContext context, int p)
EAM_22	1	301
EAM_23	1	{
EAM_24	1	302
EAM_25	1	Atm cbt = context.getOwner();
OAN_0	1	303
EAM_26	1	if (p != cbt.getPin() && cbt.getAttempts() < 3) {
JTI_0	1	}
EAM_27	1	290
JTD_0	1	291
JSI_1	1	292
JSD_0	1	private static final class AtmEFSM_S1 extends AtmContext.AtmEFSM_Default
EAM_28	1	293
EAM_29	1	294
EAM_30	1	295
JDC_0	1	296
EAO_0	1	297
EAM_31	1	298
EAM_32	1	299
EAM_33	1	300
EAM_34	1	protected void PIN(AtmContext context, int p)
EAM_35	1	301
EAM_36	1	302
EAM_37	1	Atm cbt = context.getOwner();
EAM_38	1	if (p != cbt.getAttempts() && cbt.getAttempts() < 3) {

Figura 4. Geração de mutantes. (a) Mutantes de métodos; (b) Mutantes de classe.

ponde ao valor obtido pela equação 1. Analisando-se os mutantes vivos, viu-se que alguns deles afetam partes do código que não são relativas aos estados e transições do modelo. Esses mutantes foram considerados como irrelevantes para o modelo, designados como $IM(P)$.

Sendo assim, o escore de mutação é obtido conforme a equação 2, e seus valores são mostrados na linha ms_2 da tabela 1.

$$ms(P, CT) = \frac{DM(P, CT)}{M(P) - EM(P) - IM(P)} \quad (2)$$

A última linha da tabela 1 apresenta o escore final considerando os operadores de classe e método a partir dos valores obtidos em ms_2 . A tabela 2 apresenta os operadores que geraram os mutantes mais difíceis de matar, mostrando-se o escore de mutação correspondente. Nesses resultados considera-se apenas mutantes relevantes ao modelo.

Com os resultados apresentados na tabela 2, pode-se observar que os operadores

Escore de mutação (%)		
	Classe	Método
ms_1	91%	42%
ms_2	93%	73%
Escore Final	79.62%	

Tabela 1. Escore.

	Operador	% mutantes vivos
Op. de Método	AOD (<i>Arithmetic Operator Deletion</i>)	0.0%
	AOI (<i>Arithmetic Operator Insertion</i>)	86.54%
	COI (<i>Conditional Operator Insertion</i>)	80.46%
	COR (<i>Conditional Operator Replacement</i>)	75.35%
	LOI (<i>Logical operator Insertion</i>)	83.82%
	ROR (<i>Relational Operator Replacement</i>)	91.67%
Op. de Classe	EAM (<i>Accessor Method Change</i>)	79.11%
	EMM (<i>Modifier Method Change</i>)	35.28%
	JSI (<i>Static Modifier Insertion</i>)	0.0%

Tabela 2. Mutantes difíceis de matar.

de métodos AOI e ROR são os mais difíceis de matar. Os mais fáceis foram gerados pelos operadores AOD, JSI e EMM. AOD e JSI encontram-se em partes do código que foram consideradas irrelevantes ao modelo, logo foram descartados. Já o EMM de fato altera as transições dos estados.

Pode-se observar na figura 5 a relação entre o número de mutantes mortos e o tamanho da sequência de entrada para os 70 casos de teste gerados. Pode-se perceber que o tamanho da seqüência não está diretamente relacionado à quantidade de mutantes mortos, visto que uma sequência pode gerar um caminho que passe por transições repetidas.

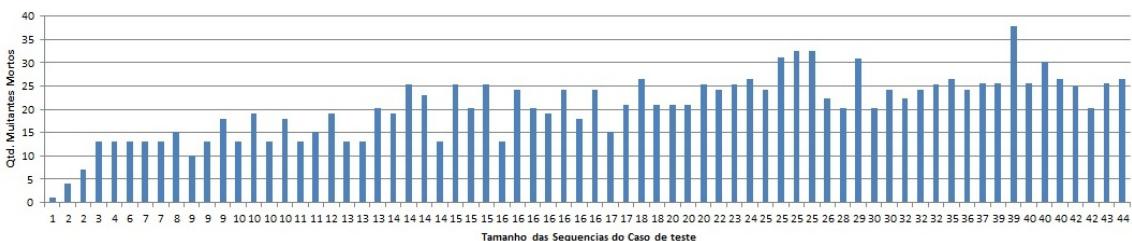


Figura 5. Mutantes Mortos x Tamanho da Sequência do Caso de Teste.

A figura 6 representa a relação entre o número de transições distintas percorridas pelo caminho gerado pelos casos de teste e o número de mutantes mortos. Este resultado indica que não é o tamanho da seqüência, mas o maior número de transições distintas no caminho, que faz com que um caso de testes tenha maior potencial para revelar erros.

5. Trabalhos relacionados

A técnica de mutação foi originalmente proposta para o código, e nesse aspecto, nosso trabalho se assemelha a muitos outros que utilizam essa técnica para avaliar conjuntos de teste baseados na implementação, ou de caixa branca. No entanto, dado que o código, no nosso caso, representa o modelo de comportamento de um sistema, nosso trabalho também está relacionado com mutações aplicadas a artefatos de mais alto nível.

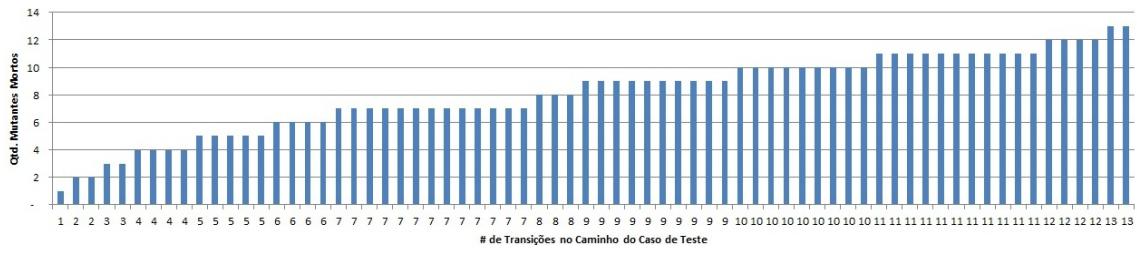


Figura 6. Mutantes Mortos x Número de Transição no Caminho do Caso de Teste.

A mutação da especificação foi proposta inicialmente para avaliar a eficácia de casos de teste usando especificações na forma de cálculo de predicados. Foram depois propostos diferentes operadores para diversos modelos, como por exemplo MFEs e Statecharts. Somente para citar alguns, o trabalho de Fabbri et al. [Fabbri et al. 1999], apresenta a definição de operadores para esses tipos de modelos. A ferramenta Proteum/RS foi construída para dar suporte a esses operadores. Guarneri et al. [Guarnieri et al. 2011] utilizaram mutação de MFEE para avaliar conjuntos de testes construídos para verificar descrições de sistemas em *System C*. Fakih et al. [Fakih et al. 2008] apresentam um método de derivação de casos de testes para EFSM baseado em erros definidos para esse tipo de modelo. O diferencial do nosso trabalho está no fato de que utilizamos o código fonte do modelo executável para inserir os mutantes, enquanto nos trabalhos citados os operadores de MFEE são aplicados ao modelo.

6. Conclusões

Este trabalho apresentou um estudo exploratório sobre o uso de mutação do código fonte de um modelo executável para avaliar o conjunto de testes gerados pela abordagem MOST, proposta em trabalho prévio do grupo. Essa abordagem utiliza um algoritmo de otimização para a geração de testes a partir de Máquinas Finitas de Estado Estendidas.

Esse estudo exploratório nos permitiu determinar as diretrizes para futuros experimentos. Por exemplo, temos indícios de que casos de teste mais longos e que cobrem maior número de transições diferentes matam mais mutantes. Pode-se constatar também que muitos mutantes não afetam a parte do código referente aos estados e transições, e são portanto, irrelevantes na análise. Esse resultado nos levou ao próximo passo, que será o uso de operadores para máquinas de estados estendidas, mencionados na seção 5. Estes operadores serão mapeados para os operadores de código-fonte para serem utilizados no modelo executável.

Referências

- Bochmann, G. and Petrenko, A. (1994). Protocol Testing: Review of Methods and Relevance for Software Testing. In *International Symposium on Software Testing and Analysis (ISSTA '94)*, pages 109–124.
- DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41.
- Derderian, K., Hierons, R. M., Harman, M., and Guo, Q. (2005). Generating Feasible Input Sequences for Extended Finite State Machines (EFSMs) using Genetic Algo-

- rithms. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*, pages 1081–1082, Washington, D.C., USA.
- Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., and Masiero, P. C. (1999). Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering, ISSRE '99*, Washington, DC, USA. IEEE Computer Society.
- Fakih, K., Kolomeez, A., Prokopenko, S., and Yevtushenko, N. (2008). Extended Finite State Machine Based Test Derivation Driven by User Defined Faults. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST '08*, pages 308–317, Washington, DC, USA. IEEE Computer Society.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Guarnieri, V., Bombieri, N., Pravadelli, G., Fummi, F., Hantson, H., Raik, J., Jenihhin, M., and Ubar, R. (2011). Mutation analysis for SystemC designs at TLM. *Latin American Test Workshop*, 0:1–6.
- Jino, M., Maldonado, J. C., and Delamaro, M. E. (2007). *Introdução ao teste de software*. CAMPUS. Capítulo 3.
- Kalaji, A. S., Hierons, R. M., and Swift, S. (2009). Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM). *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:230–239.
- Korel, B., Singh, I., Tahat, L., and Vaysburg, B. (2003). Slicing of State-Based Models. In *Proc. ICSM'03: Int. Conf. on Software Maintenance*, page 34, Washington, DC, USA. IEEE Computer Society.
- Ma, Y., Offutt, J., and Kwon, Y. R. (2005). MuJava: an automated class mutation system: Research Articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133.
- Mellor, S. J. and Balcer, M. J. (2002). *Executable Uml: A Foundation for Model-Driven Architecture*. Addison-wesley Object Technology Series. Addison-Wesley.
- Petrenko, A. and Bochmann, G. V. (1996). On Fault Coverage of Tests for Finite State Specifications. *Computer Networks and ISDN Systems*, 29:81–106.
- Yano, T. (2011). *Uma abordagem evolutiva multiobjeto para geração automática de casos de teste a partir de máquinas de estados*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, Brasil.
- Yano, T., Martins, E., and De Sousa, F. L. (2011). MOST: a multi-objective search-based testing from EFSM. In *Proc. SBST'11: 4th Int. Workshop on Search-Based Software Testing*, pages 164–173.
- Zhao, R., Harman, M., and Li, Z. (2010). Empirical Study on the Efficiency of Search Based Test Generation for EFSM Models. In *ICST Workshops*, pages 222–231.

Framework para Teste de Software Automático nas Nuvens

Gustavo S. Oliveira¹, Alexandre N. Duarte¹

¹Centro de Informática

Universidade Federal da Paraíba (UFPB) – João Pessoa, PB – Brasil

gso@di.ufpb.br, alexandre@ci.ufpb.br

Abstract. This paper presents a framework called *CloudTesting* that distributes and coordinates the parallel execution of automated testing of software in distributed and heterogeneous environments, including different cloud infrastructures. The use of cloud computing platforms as execution environment for automated testing of software increase tests efficiency and effectiveness when compared to traditional methods. The goal of this work is to obtain practical and real numbers related to the speedup achieved with the use of the framework, allowing other applications to use infrastructures of distributed execution in an uncomplicated way.

Resumo. Este trabalho apresenta o framework *CloudTesting* que se trata de uma solução que efetua execução paralela de testes automáticos de software em ambientes distribuídos e heterogêneos. O uso de plataformas de computação nas nuvens como ambiente de execução de testes automáticos de software proporciona testes mais eficientes e eficazes se comparados aos métodos tradicionais. O objetivo deste trabalho é obter números práticos e reais relacionados ao ganho de desempenho alcançado com o uso do framework, possibilitando que outras aplicações utilizem infraestruturas de execução distribuída de modo simples e descomplicado.

1. Introdução

Um processo de teste de *software* eficaz deve ser rápido e realizado de forma automática. Existem soluções bem estabelecidas [Gamma e Beck 1999] para automatizar o processo de teste de *software* e outras soluções que visam principalmente acelerar o processo, distribuindo a execução em um conjunto de processadores [Kapfhammer 2001][Hughes e Greenwood 2004]. Nesse mesmo contexto, há também esforços para explorar as características de plataformas de computação distribuída como as grades computacionais, como seu amplo paralelismo e vasta heterogeneidade de ambientes, como forma de limitar os efeitos do ambiente de desenvolvimento sobre os resultados dos testes [Duarte *et al.* 2005]. Porém, apenas recentemente novos estudos têm adotado a computação nas nuvens como plataforma para testes de *software* de larga escala [Hanawa *et al.* 2010][Oriol e Ullah 2010].

A exploração de plataformas de computação nas nuvens como ambiente para execução de testes de *software* tem apresentado ganhos tanto em eficiência quanto na eficácia dos testes, quando comparados aos métodos tradicionais [Riungu-Kalliosaari, Taipale e Smolander 2012], diversos fatores contribuem para essa afirmação, tais como

a redução de custos de implantação, manutenção e licenciamento de ambientes de testes internos, flexibilidade para aquisição e montagem de ambientes de testes customizados em instantes e escalabilidade de maneira oportuna e econômica [Grundy *et al.* 2012].

Entretanto, desenvolver e processar testes na nuvem requer esforços na configuração, distribuição, automatização e execução de testes e scripts [Hanawa 2010]. Com o objetivo de facilitar a exploração de plataformas de computação na nuvem como ambientes de testes de *software* propomos neste trabalho um *framework* denominado *CloudTesting*. Nossa solução possibilita a execução paralela de testes automáticos de *software* em ambientes heterogêneos, diminuindo o tempo gasto no período dos testes, por meio de uma camada de abstração para os usuários, eliminando a necessidade de efetuar configurações complexas, para que se possa fazer uso de tais infraestruturas de execução distribuída. Uma característica muito importante da solução proposta neste trabalho e que pode facilitar sua adoção na comunidade de desenvolvedores de *software* é o fato de a ferramenta não exigir qualquer modificação no código fonte dos testes para sua execução na nuvem.

Apresentamos neste trabalho os resultados dos experimentos realizados com o *framework* *CloudTesting* adotando o provedor de nuvem Amazon EC2 (*micro, small e medium instances*) em comparação com a execução dos mesmos testes executados localmente. A análise quantitativa realizada indica ganhos significativos de tempo na execução dos testes de *software* utilizando a infraestrutura de computação na nuvem.

2. O Framework *CloudTesting*

Projetos de *software* de grande porte possuem geralmente uma quantidade proporcionalmente grande de casos de testes [Duarte *et al.* 2005]. Muitas vezes, tais testes consomem muito tempo para serem executados, inviabilizando qualquer processo de desenvolvimento que se baseie fortemente no uso de testes automáticos, como o *Extreme Programming*, por exemplo [Wu e Sun 2010]. Uma vez que cada teste consome um determinado período de execução, dependendo do tamanho e da complexidade do *software*, a única maneira de decrementar o tempo gasto no processo de teste de *software* é paralelizar massivamente a sua execução [Banzai e Hitoshi 2010].

O *framework* *CloudTesting*¹ facilita este processo, obtendo de forma dinâmica os recursos computacionais necessários para a execução dos testes, abstraindo do testador/desenvolvedor toda a complexidade envolvida no processo de execução dos testes sem exigir qualquer modificação no código fonte dos testes para sua utilização. Quando se decide distribuir e paralelizar a execução dos testes, obtém-se uma redução significativa do tempo necessário para executar um grande conjunto de testes, diminuindo, por sua vez, o tempo despendido na identificação e correção de falhas no *software*, o que representa um grande impacto no custo total de desenvolvimento. Além disso, o *framework* possui vantagem ao aumentar a confiabilidade nos resultados dos testes de unidade por meio da utilização de ambientes heterogêneos e sabidamente não contaminados para a execução dos testes, facilitando a exposição de falhas que de outra maneira só seriam encontradas durante a fase de execução do sistema.

¹ Disponível em: <http://code.google.com/p/cloudtestingdi/>

Conforme apresenta a Figura 1, fundamentalmente o *framework* distribui o projeto que contém o conjunto de testes de unidade para a nuvem, após isso efetua reflexão nas classes locais desse conjunto de testes enviando requisições de execução para um método de modo independente, paralelizando ao máximo a execução das unidades. O balanceamento de carga de recursos da rede é baseada na implementação do Algoritmo Round-Robin [Ramabhadran e Pasquale 2003]. Assim, todas as requisições são distribuídas de modo uniforme entre as máquinas da nuvem participantes do平衡amento.

No entanto, existem situações em que a ferramenta ainda não pode ser aplicada. O *framework* apresenta restrições relacionadas à paralelização da execução de conjuntos de testes que, necessariamente devem ser executados em sequência previamente definida, visto que, como citado anteriormente, cada método é distribuído e executado de modo autônomo. Além disso, a ferramenta ainda não oferece suporte a aplicações que apresentam dependências externas de API, banco de dados e interface com outras aplicações.

A Figura 2 apresenta os componentes da arquitetura do *CloudTesting* que é composta pelos componentes de *configuration*, *reflection*, *distribution*, *connection*, *log* e *main*.

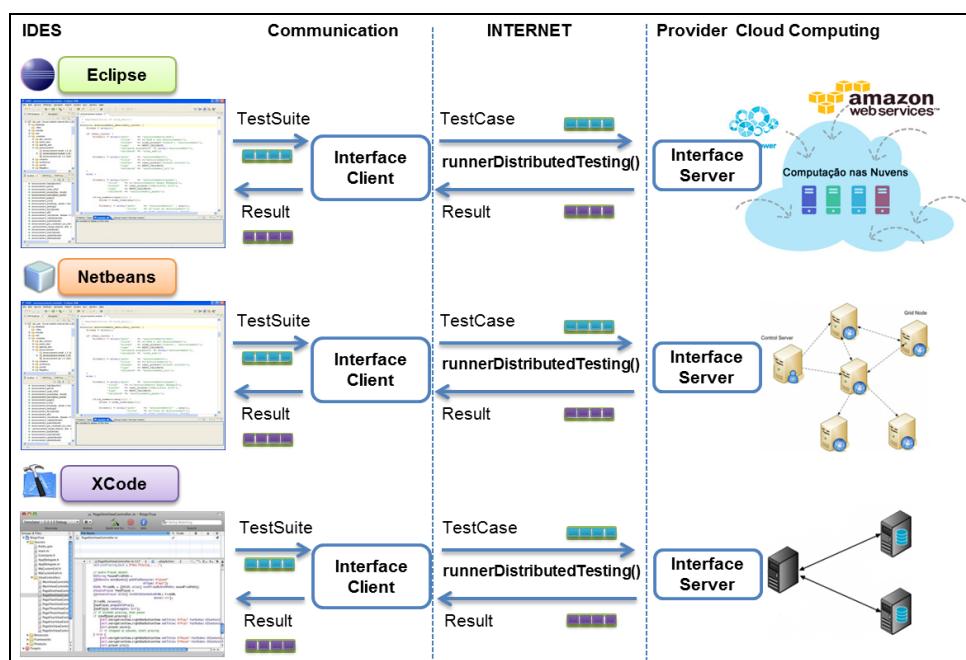


Figura 1. Fluxo de execução do Teste de Software de Unidade utilizando o framework *CloudTesting*.

O componente *configuration* auxilia na definição de informações referentes a *paths*, *hosts* e balanceamento de carga, ou seja, local de armazenamento da biblioteca do *framework* de unidade na nuvem, projeto que deverá ser enviado para a nuvem, arquivo de permissão de acesso do provedor da nuvem e local para armazenamento dos projetos na nuvem. O componente *reflection* efetua leituras nos conjuntos de testes de unidade desenvolvidos pelo usuário, com o intuito de informar ao componente *distribution* quais

métodos devem ser executados na nuvem. Por sua vez, o componente de distribuição efetua *upload* de todo o projeto do usuário para a nuvem. O componente *connection* possui interface do lado do cliente que disponibiliza uma interface para comunicação com a nuvem. Na nuvem esse componente estabelece um serviço, que gerencia a execução de cada teste e envia o resultado dos testes para o cliente em tempo real. O componente de *log* grava os eventos de todo o processo. Já o *Main* serve de fachada para encapsular todos os componentes anteriores.

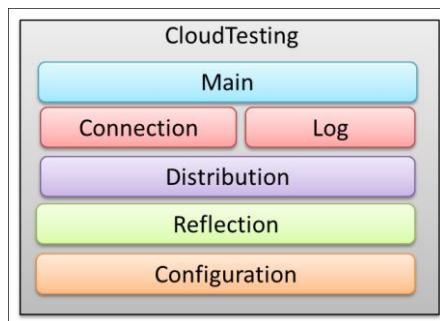


Figura 2. Componentes da Arquitetura do *CloudTesting*.

3. Trabalhos Relacionados

Sistemas de testes de *software* automático distribuído ou de larga escala foram propostos nos últimos anos, explorando as características de amplo paralelismo e vasta heterogeneidade da computação distribuída como forma de diminuir o tempo de execução dos testes.

A ferramenta GridUnit [Duarte *et al.* 2005] investiga o uso de grades computacionais como ambiente de teste, fornecendo uma extensão do JUnit [Gamma e Back 1999] rodando unicamente sobre o OurGrid [Andrade 2003], sendo capaz de distribuir a execução de um conjunto de testes de unidade na grade sem a necessidade de qualquer modificação no código fonte.

O *CloudTesting* acrescenta melhorias em relação ao *GridUnit*, podendo ser estendido e customizado para diferentes infraestruturas de execução, em vez de manter forte acoplamento com o OurGrid. Posteriormente, por utilizar computação em nuvens ao invés de grades computacionais, o *CloudTesting* herda todos os avanços da área de computação distribuída e paralela.

4. Experimentos

Por se tratar de um *framework* o *CloudTesting* pode ser utilizado em vários ambientes de desenvolvimento integrado (IDE), além de diversificadas plataformas de execução. Entretanto, para esta análise, foi desenvolvida uma instanciação do *framework* para o IDE Eclipse e o provedor de nuvem Amazon AWS. Contudo, essa foi apenas uma escolha de projeto, outras opções poderiam ter sido utilizadas, como apresenta a Figura 1.

Para realizar os experimentos, desenvolveu-se um conjunto de testes que possuem tempo de processamento médio previamente conhecido quando executado em uma máquina local, objetivando efetuar uma comparação com os resultados obtidos por

meio do uso do *framework CloudTesting*. O conjunto de teste adotado para os exames possuem 1800 testes de unidade, onde cada unidade efetua o cálculo da proporção numérica entre a relação das grandezas do perímetro e o diâmetro de uma circunferência com 5000000 amostras com pontos dentro do círculo definidos de modo aleatório para o eixo X e eixo Y.

Basicamente os experimentos se dividem em dois cenários: (01) onde o conjunto de testes é executado 45 vezes em uma máquina local; (02) onde o conjunto de testes é distribuído e executado 45 vezes na nuvem.

A análise dos resultados engloba uma série de subtarefas. Utiliza-se o Critério de Chauvenet [Pop e Pitica 2010] buscando erradicar anomalias nos resultados. Em seguida, calcula-se a média do tempo de execução, o desvio padrão e identifica-se o melhor e pior tempo de execução. Essas informações permitem o cálculo do *speedup* ($SP = T_1/T_p$, onde T_1 é o tempo de execução do programa sequencial e T_p é o tempo de execução do mesmo programa executado em paralelo) e da eficiência do algoritmo paralelo ($EF = Sp/N_p$, onde Sp é o *speedup* alcançado e N_p é o número de processadores utilizados para executar o programa em paralelo). Após isso, garante-se o tempo de execução em intervalos de confiança de 95% e 99% [Dillard 1997].

4.1. Cenário 01

Os testes executados localmente seguiram uma política rígida em relação ao uso do equipamento durante o período de testes. Visando evitar resultados anômalos e obter resultados reais, utilizou-se uma máquina dedicada para executar os testes. Após o término de cada teste efetuava-se um *reboot* na máquina com o intuito de higienizar dados armazenados na memória RAM e *cache* do processador. A configuração da máquina possuía processador Core 2 Duo 2.20 GHz, 4 Gigabytes de RAM e Sistema Operacional Linux 32 bits.

Para capturar o tempo de execução real do conjunto de teste, utilizou-se o ambiente de desenvolvimento integrado Eclipse juntamente com o *plugin JUnit PDE* que possui um perfilador e gerencia por padrão todos os testes de unidade de *software* executados com o *JUnit* na plataforma. A Figura 3 apresenta um trecho do conjunto de testes utilizado para realizar a experiência.

```
@Test
public void testAdd0() {
    long amostras = 5000000;
    long pontos = 0;

    for (int i = 0; i < amostras; i++) {
        double px = 2 * Math.random() - 1;
        double py = 2 * Math.random() - 1;

        if (Math.pow(px, 2) + Math.pow(py, 2) <= 1) {
            pontos++;
        }
    }
    assertEquals(314, Math.round(((4 * (double) pontos / (double) amostras) * 100)/ 1d));
}
```

Figura 3. Trecho do código fonte do conjunto de testes.

Cada teste de unidade obteve um tempo de execução médio próximo de 1 segundo. O conjunto foi composto por 1800 testes, assim um único conjunto de testes gastou em média 30 minutos para completar seu processamento. A execução total do conjunto em 45 vezes resultou em 22:54:51. O tempo médio de execução foi de 00:30:33 por conjunto, desvio padrão de 00:00:27, coeficiente de variação de 1,47%, melhor e pior tempo de execução 00:29:58 e 00:31:09 consecutivamente. Com um intervalo de confiança de 95% obteve-se limite inferior de 00:30:25 e limite superior de 00:30:41 e com um intervalo de 99% obteve-se limite inferior 00:30:23 e limite superior de 00:30:43. Esses dados servirão de base para análise de *speedup* da execução na nuvem.

4.2. Cenário 02

O teste na nuvem confrontou-se com características encontradas no contexto de sistemas distribuídos: Latência e flutuação de rede. Por esse motivo, foi estabelecida uma política de escala de horário para a execução dos testes, com o intuito de obter em tese as mesmas condições de largura de banda para todos os experimentos. Assim, todos os testes foram realizados em uma faixa de tempo que varia entre 00:00 e 04:00, comumente a fatia de tempo selecionada reflete o horário de menor utilização da rede no laboratório. Devido à quantidade de testes, os experimentos não foram executados em um único dia, ao invés disso, foram executados em dias alternativos seguindo a política supramencionada.

Todos os 45 exames foram realizados na Amazon nos tipos de instâncias: *Micro*, *Small* e *Medium instance*. Para capturar o tempo de execução real do conjunto de teste, utilizou-se o ambiente de desenvolvimento integrado Eclipse juntamente com o *plugin TPTP* que possui um perfilador. Adotou-se uma largura de banda nominal de 15mbps para *download* e 1mbps para *upload*. Algumas configurações foram feitas nas máquinas da nuvem de modo prévio para a execução do *framework CloudTesting*: (1) Os diretórios log e lib foram criados para armazenamento de logs e bibliotecas respectivamente; (2) As bibliotecas dos frameworks *JUnit* e *CloudTesting* foram distribuídas para o diretório lib; (3) O serviço do *CloudTesting* foi ativado por meio do comando: `java -XX:PermSize=128m -XX:MaxPermSize=256m -classpath lib/cloudtesting.jar:. cloudtesting.connection.Server`.

4.3. Cenário 02 - Micro Instance

Micro instances possuem recursos limitados de CPU, embora permitam o aumento da capacidade computacional quando ciclos adicionais estão disponíveis, são compostas por Sistema Operacional Linux 32 bits com até 2 unidades computacionais EC2 (para estouros periódicos pequenos), 613MB de memória RAM, apenas armazenamento EBS e baixo desempenho para entrada e saída de dados [Amazon 2012].

Como resultado, obtivemos o tempo total de execução do conjunto de testes 08:04:53, diminuindo em 14:49:58 o tempo total de execução do Cenário 01. O tempo médio de execução foi de 00:10:46, desvio padrão 00:01:59, coeficiente de variação de 18,23%, melhor e pior tempo de execução 00:03:56 e 00:12:03 consecutivamente. Analisando o valor do desvio padrão observa-se uma variação considerável (conforme ilustra a Figura 4) resultando diretamente no intervalo de confiança. Com um intervalo de 95% obteve-se limite inferior de 00:10:12 e limite superior de 00:11:21 e com um

intervalo de 99% obteve-se limite inferior 00:10:01 e limite superior de 00:11:32.

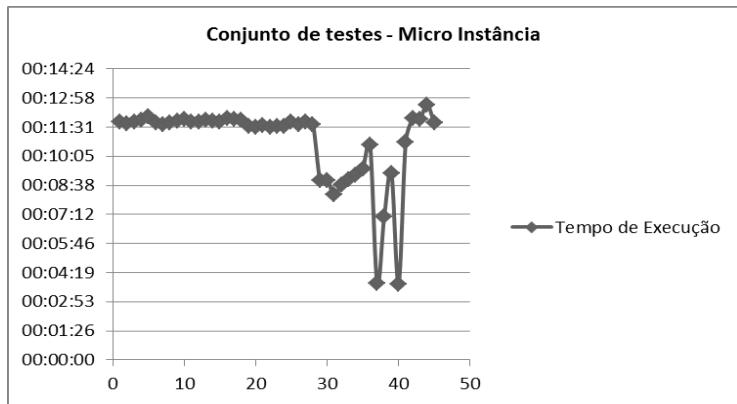


Figura 4. Resultado dos experimentos na Amazon EC2 - *Micro Instance*.

Em relação a grande variação entre o tempo de execução de alguns experimentos, utilizou-se o Critério de Chauvenet para verificar se os mesmos poderiam ser considerados como anomalias. Contudo, nenhum resultado se enquadrou nessa situação. O que leva a inferir que essa diferença se constituiu na condição de que as *Micro instances* são capazes de ultrapassar por um curto espaço de tempo até 2 ECUs, ou seja, o dobro do número de ECUs fornecidas a uma instância do tipo *Small*. Outra hipótese se relaciona com recursos de rede e subsistema de discos compartilhados, possivelmente quando um recurso é subutilizado todas as instâncias virtuais ativas recebem a oportunidade de consumir uma fatia extra desse recurso enquanto o mesmo permanecer disponível [Amazon 2012].

4.4. Cenário 02 - Small Instance

Instâncias do tipo *Small* fazem parte do grupo de instâncias padrão, são recomendadas para a execução da maior parte de aplicativos. Para os testes, as instâncias possuíam Sistema Operacional Linux 32 bits, 1 unidade computacional EC2; 1,7GB de memória RAM e 160GB disponível para armazenamento de dados.

O tempo de execução total do conjunto de testes foi de 04:07:31, diminuindo em 18:47:20 o tempo total de execução do Cenário 01 e em 03:57:22 em comparação com a execução nas *Micro instances*. O tempo de execução médio foi de 00:05:29, desvio padrão de 00:00:08, coeficiente de variação de 2,44%, melhor e pior tempo de execução 00:05:19 e 00:05:52 respectivamente. O tempo de execução individual de cada experimento esteve consistente, como pode ser observado com o desvio padrão estreito. O intervalo de confiança de 95% obteve um limite inferior de 00:05:21 e limite superior de 00:05:37, o intervalo de 99% alcançou limite inferior de 00:05:26 e limite superior de 00:05:32.

4.5. Cenário 02 - Medium Instance

Instâncias do tipo *Medium* também fazem parte do grupo padrão, possuem a mesma finalidade das instâncias do tipo *Small*, ou seja, executar a maior parte de aplicativos. Durante os experimentos, utilizou-se a seguinte configuração: CPU de alto desempenho, 3.75 GB de memória, 2 Unidades de processamento EC1 (1 núcleo virtual com 2

Unidades de processamento EC2 cada), 410 GB de armazenamento de dados e plataforma de 32 bits.

O tempo de execução total do conjunto de testes foi de 02:50:48, uma redução de 20:04:03 se comparado ao Cenário 01, 05:14:05 se comparado a execução nas *Micro instances* e 01:16:43 se comparada a execução nas *Small instances*.

Também obtivemos como resultado, o tempo médio de 00:03:47, desvio padrão de 00:00:09, coeficiente de variação de 4%, melhor e pior tempo de execução 00:03:12 e 00:04:01 respectivamente. O intervalo de confiança de 95% obteve um limite inferior de 00:03:44 e limite superior de 00:03:50, o intervalo de 99% alcançou limite inferior de 00:03:43 e limite superior de 00:03:51.

5. Análise de Speedup

Alguns pontos devem ser considerados antes de avaliarmos um determinado *speedup* de modo positivo ou negativo. Ao idealizarmos o uso de 18 máquinas para efetuar processamento paralelo na nuvem, desejamos um *speedup* linear bastante aproximado ao número de máquinas empregadas.

Contudo, deve ser ponderado que para efetuarmos processamento paralelo na nuvem devemos realizar *upload* do código a ser testado para a mesma. Dependendo do tamanho do projeto, distribuir pacotes pela rede pode ser custoso. Outro fator se relaciona diretamente com o poder de processamento das máquinas e a capacidade de entrada e saída de dados na nuvem. Por último, ainda existe a relação com as instâncias de servidor virtualizadas, ou seja, o desempenho muitas vezes irá depender da quantidade de recursos disponíveis no servidor da nuvem.

Nos experimentos realizados observou-se que *Micro instances* apesar de terem alcançado um *speedup* considerável no seu melhor tempo de execução, não são adequadas para uma grande carga de requisições em um curto espaço de tempo. Para esse contexto se adequam melhor instâncias do tipo *Medium*.

Os experimentos realizados nas *Micro instances* alcançaram um *speedup* de 8.55 e 0.48 de eficiência do algoritmo paralelo para o melhor tempo de execução, *speedup* de 2.89 e 0.16 de eficiência do algoritmo paralelo para o tempo médio e *speedup* 2.61 e 0.14 de eficiência do algoritmo paralelo para o pior tempo de execução em relação aos experimentos locais (Cenário 01). Os testes executados nas *Small instances* alcançaram um *speedup* de 5.70 e 0.32 de eficiência do algoritmo paralelo para o melhor tempo de execução, *speedup* de 5.71 e 0.32 de eficiência do algoritmo paralelo para o tempo médio e *speedup* de 5.70 e 0.32 de eficiência do algoritmo paralelo para o pior tempo de execução em relação aos experimentos locais. Os experimentos realizados em *Medium instances* alcançaram um *speedup* de 9.48 e 0.53 de eficiência do algoritmo paralelo para o melhor tempo de execução, *speedup* de 8.72 e 0.48 de eficiência do algoritmo paralelo para o tempo médio e *speedup* de 7.83 e 0.43 de eficiência do algoritmo paralelo para o pior tempo de execução em relação aos experimentos locais. A Figura 5 apresenta uma comparação de *speedup* entre os tipos de instâncias.

O melhor tempo de execução dos experimentos realizados em *Micro instances* resultou em um *speedup* superior ao resultado do melhor tempo de execução dos testes executados nas *Small instances*. Esse resultado é surpreendente a princípio, embora,

como mencionado anteriormente máquinas do tipo *Micro* podem momentaneamente utilizar até 2 ECUs, obtendo o dobro do poder computacional de um máquina *Small*. Apesar disso, o tempo médio e o pior tempo de execução dos experimentos executados em *Micro instances* são bastante inferiores aos resultados obtidos nas *Small instances*. Os melhores tempos foram obtidos com *Medium instances*, esse resultado já era esperado devido às configurações de *hardware* e pela taxa de entrada e saída de dados superior as outras instâncias testadas.

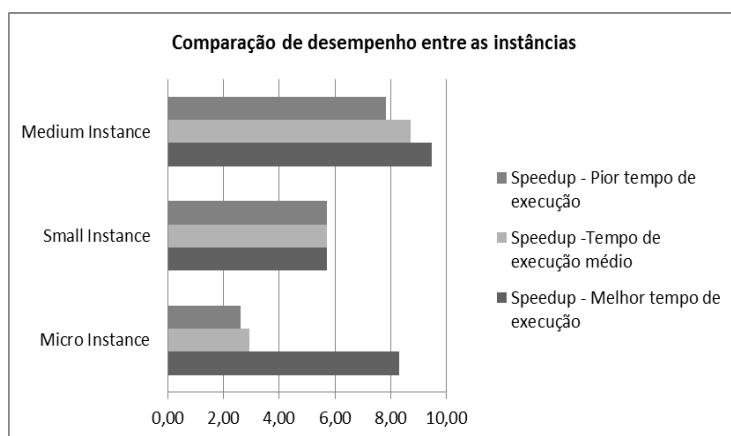


Figura 5. Comparação do speedup entre as instâncias *micro*, *small* e *medium*.

Para realizarmos esses experimentos, tivemos o custo de apenas U\$16,20 para executar as 18 máquinas *Micro*, U\$68,85 para as 18 máquinas do tipo *Small* e U\$137,70 para as 18 máquinas do tipo *Medium*.

6. Considerações Finais

A análise do *speedup* e o custo envolvido no processo é a base para possibilitar que várias outras aplicações avaliem o processamento paralelo na nuvem, visto que, sem números reais sobre a mensuração do *speedup*, não se constituiria os valores relacionados a ganhos de desempenho.

Os resultados dos experimentos realizados indicam ganhos significativos de desempenho da execução dos testes sem aumentar significativamente os custos envolvidos na montagem da infraestrutura de execução, facilitando assim o processo de utilização de plataformas de computação em nuvem como ambientes para a execução de testes automáticos de *software*.

O *CloudTesting* simplifica o método de efetivação de testes automáticos em ambientes distribuídos, obtendo ganhos de desempenho, confiabilidade e simplicidade de configuração por meio da execução paralela de testes automáticos de *software* em ambientes heterogêneos através de uma camada de abstração para os usuários.

Como trabalhos futuros, outros experimentos estão sendo realizados, em relação à utilização de diversos algoritmos de balanceamento no *framework*, para análise de desempenho, além de verificação de testes executados em ambientes heterogêneos. Tais trabalhos embasarão a concepção do desenvolvimento de *software* baseado em testes na nuvem de um modo mais rápido e com resultados mais seguros.

Referências

- Smith, A. e Jones, B. (1999). On the complexity of computing. In *Advances in Computer Science*, pages 555–566. Publishing Press.
- Gamma, E. e Beck, K. (1999). JUnit: A cook's tour. *Java Report*, 4(5):27-38.
- Hughes, D. e Greenwood, G. (2004). A Framework for Testing Distributed Systems, In Proceedings of the 4th IEEE International Conference on Peer-to-Peer computing.
- Kapfhammer, G., M. (2001). Automatically and Transparently Distributing the Execution of Regression Test Suites. In Proceedings of the 18th International Conference on Testing Computer Software.
- Duarte, A. *et al.* (2005). Gridunit: software testing on the grid, Proceedings of the 28th international conference on Software engineering. New York, USA, pp. 779–782.
- Hanawa, T. *et al.* (2010). Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems, Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference.
- Oriol, M. e Ullah, F. (2010). YETI on the Cloud. Software Testing, Verification, and Validation Workshops (ICSTW) Third International Conference 2010.
- Wu, X. e Sun, J. (2010). The Study on an Intelligent General-Purpose Automated Software Testing Suite, Intelligent Computation Technology and Automation (ICICTA) International Conference 2010.
- Banzai, Takayuki e Koizumi, Hitoshi (2010). D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology, Cluster, Cloud and Grid Computing (CCGrid), 10th IEEE/ACM International Conference.
- Ramabhadran, S. e Pasquale, J. (2003). "Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay," Proc. of SIGCOMM '.
- Amazon EC2 (2012). "Amazon Elastic Compute Cloud (Amazon EC2)", <http://aws.amazon.com/pt/ec2/instance-types/>, Junho.
- Dillard, G.M. (1997). "Confidence intervals for power estimates," *Signals, Systems & Computers, 1997. Conference Record of the Thirty-First Asilomar Conference*.
- Pop, S.; Ciascian, I. e Pitica, D. (2010), "Statistical analysis of experimental data obtained from the optical pendulum," *Design and Technology in Electronic Packaging (SIITME), 2010 IEEE 16th International Symposium*.
- Grundy, J. *et al.* (2012), "Guest Editors' Introduction: Software Engineering for the Cloud," *Software, IEEE* , vol.29, no.2, pp.26-29.
- Riungu-Kalliosaari, L.; Taipale, O. e Smolander, K. (2012). "Testing in the Cloud: Exploring the Practice," *Software, IEEE* , vol.29, no.2, pp.46-51.
- Andrade, Nazareno *et al.* (2003). OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp 61-86.

Usando o SilkTest para automatizar testes: um Relato de Experiência

Thiago L. L. Lima^{1,2}, Ayla Dantas¹, Livia M. R. Vasconcelos² e Aluizio N. S. Neto²

¹Departamento de Ciências Exatas – Universidade Federal da Paraíba (UFPB)
Campus IV – Rua da Mangueira, S/N – Companhia de Tecidos Rio Tinto
CEP – 58.297-000 – Rio Tinto – PB – Brazil

²Serviço de Qualidade de Software – Dataprev
João Pessoa, PB – Brazil

{thiago.luna,ayla}@dce.ufpb.br,
{livia.vasconcelos,aluizio.neto}@dataprev.gov.br

Abstract. This work describes the test automation process followed by a public IT company that uses the SilkTest tool. Besides, it also presents an initial evaluation of the acceptance of this process by the developers and analysts from one of the development units of the company. In general, the results obtained showed a good perception about the gains with the test automation process and a good level of satisfaction with the tools adopted by the company.

Resumo. Este trabalho descreve o processo de automação de testes seguido por uma empresa pública de TI e que utiliza a ferramenta SilkTest. Além disso, é apresentada uma avaliação inicial da aceitação desse processo por parte dos desenvolvedores e especificadores de uma das unidades de desenvolvimento da empresa e que tem mostrado uma percepção positiva quanto aos ganhos obtidos com a automação de testes e um bom nível de satisfação com as ferramentas adotadas pela empresa.

1. Introdução

Hoje em dia é crescente a exigência pela qualidade de software, que corresponde à busca pela ausência de defeitos [Juran 1992]. Nesse cenário, os testes tem se tornado uma atividade cada vez mais fundamental para tentar diminuir a quantidade de defeitos de um sistema. Para que possam ser encontrados o quanto antes no desenvolvimento, o que diminui os custos de sua resolução, uma prática que tem se difundido é a automação de testes e a execução constante destes testes durante o desenvolvimento.

A prática de testes automáticos é muito difundida por processos ágeis como o Extreme Programming (XP), mas também tem se difundido mesmo em organizações que utilizem metodologias mais tradicionais como o RUP. Testes automatizados são programas ou scripts simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas sobre o sistema. A grande vantagem desta abordagem é que a execução de todos os casos de teste pode ser facilmente e rapidamente repetida a qualquer momento e com pouco esforço, estando este concentrado na criação de scripts e com a infraestrutura de execução. Como é relativamente fácil executar todos os testes a qualquer momento, mudanças no sistema podem ser feitas com maior segurança.

Mesmo com tais vantagens, em algumas empresas se percebe um certo nível de resistência para a automação de testes, principalmente devido ao esforço inicial do processo e por demandar às vezes uma mudança de cultura na organização. Neste

trabalho é relatada a experiência com o processo de automação de testes em uma empresa pública de TI que utiliza a ferramenta SilkTest e a percepção dos funcionários de uma de suas unidades de desenvolvimento quanto a esse processo. Percebeu-se que, de maneira geral, os funcionários que participaram do estudo têm visto claramente os ganhos com a automação de testes e que têm se mostrado de maneira geral satisfeitos com a solução adotada pela empresa.

Este artigo está organizado da seguinte forma. Na Seção 2 será apresentada uma visão geral das ferramentas utilizadas na empresa para automação de testes. Na Seção 3 será mostrada uma visão do fluxo de automatização de testes dentro da empresa de TI em que este trabalho se foca. Na Seção 4 é feita uma avaliação do processo de desenvolvimento em relação às atividades de automação de testes e ao uso das ferramentas de apoio à automação. Por fim, na Seção 5 são mostradas as conclusões deste trabalho e as principais lições aprendidas com esse processo de automatização dentro da empresa e de sua avaliação inicial.

2. Visão Geral das Ferramentas SilkTest e SilkCentral

Na automação dos testes, um ponto muito importante e que deve ser levado em consideração são as ferramentas utilizadas. Dependendo da escolha feita, pode-se, por exemplo, gerar uma resistência no processo de automação e que se deve a uma dificuldade maior na utilização inicial da ferramenta do que com o processo de automação em si.

Algumas das ferramentas de automação são as ferramentas de gerenciamento de testes e as ferramentas de geração de scripts. Nas ferramentas de gerenciamento de testes são realizadas diversas atividades tais como: planos de teste, execuções de testes, geração de relatórios entre outras, e nessa categoria temos diversas ferramentas como: Salome-TMF, TestLink e a SilkCentral, na qual se focou este trabalho.

Já nas ferramentas de geração de scripts, temos como principal característica a funcionalidade “*Record and Replay*” que permite ao usuário gravar suas ações interativamente e posteriormente executar essas ações inúmeras vezes através dos scripts criados com a gravação, comparando os resultados obtidos na execução com os resultados esperados. Essa abordagem apresenta vantagens pois requer um menor esforço de desenvolvimento de testes. Porém, também apresenta algumas desvantagens, como o forte acoplamento entre a interface do software e os scripts de teste, o que faz com que se perca os scripts quando há mudanças na interface. Nessa categoria há várias ferramentas no mercado como: Selenium, TestComplete e também o SilkTest, que será mais detalhado no decorrer desta seção.

As ferramentas SilkCentral e SilkTest pertencem à suíte da Micro Focus, para criação e gerenciamento de testes. O SilkCentral é uma ferramenta bastante completa de gerenciamento de testes e apresenta uma gama de funcionalidades que facilitam a criação e o gerenciamento dos testes de um software. Através desse software as execuções dos testes podem ser agendadas escolhendo-se o que testar e com que frequência. Feito o agendamento, os testes são executados automaticamente e os resultados da execução podem ser acompanhados através de relatórios gerados pela ferramenta. A ferramenta se integra com outras ferramentas da suíte podendo assim ser feita a rastreabilidade dos casos de uso e o casamento entre estes e os casos de teste do projeto, permitindo assim que se tenha uma visão da cobertura de testes do projeto.

A ferramenta SilkTest é uma ferramenta de automação de testes funcionais para aplicações. É uma IDE que fornece gravação, execução, debugging e edição dos scripts de testes. SilkTest é capaz realizar a tarefa “*Record and replay*” para testes baseados no uso da interface, onde a ferramenta captura as ações do usuário e gera os scripts que são posteriormente reproduzidos. Os scripts podem ser gerados em diferentes linguagens de programação, sendo uma delas a linguagem Java, sendo esta a linguagem utilizada na Empresa Pública de TI utilizada no estudo de caso descrito neste artigo. Os scripts são gerados através de um plugin utilizado no Eclipse para o SilkTest chamado Silk4j.

3. Fluxo de automatização dos testes dentro da empresa

A Empresa pública de TI de que trata este artigo é uma empresa que fornece soluções de tecnologia da informação e da comunicação para o governo brasileiro e que conta com mais de 3.000 empregados, espalhados por várias cidades brasileiras. Com o crescimento da empresa nos últimos anos, observou-se a necessidade de se ter ferramentas que auxiliasssem no processo de desenvolvimento da empresa como um todo e por se tratar de uma empresa pública, houve a necessidade de realizar contratação através de pregão. Nesse processo, a empresa ganhadora foi a Micro Focus, que oferece uma suíte integrada de desenvolvimento de software, com ferramentas de Gerência de Requisitos (Caliber), Gerência de Configuração e Repositório (Star Team), Análise e Projeto (Together) e Testes (Silk Central, Silk Test e Silk Performer) e tal variedade de ferramentas que se integrassem entre si era um requisito fundamental da empresa. Nesse sentido, os processos da organização tiveram de se adequar às ferramentas que estariam disponíveis para assim melhor usufruir de seus recursos.

Antes da adoção dessas ferramentas, alguns projetos da organização já seguiam a prática de automatização de testes com outras ferramentas, tais como o Selenium e JMeter. No entanto, com a aquisição da suíte, foi mais fortemente incentivada a utilização das novas ferramentas por todos os novos projetos e a prática de automação de testes começou a ser mais disseminada dentro da empresa. Esse processo de disseminação e popularização da utilização das ferramentas e da criação de testes automáticos foi conduzido pela equipe de qualidade de software da empresa.

O processo de desenvolvimento incluindo a automação com as ferramentas da suíte apresenta uma série de atividades. A Figura 1 sintetiza algumas partes do processo de desenvolvimento da empresa relacionadas à automatização de testes funcionais e descritas a seguir. Inicialmente são gerados como artefatos os requisitos do projeto e seus casos de uso com o apoio da ferramenta Caliber. Tais casos de uso são usados como insumos para criação de um documento com os casos de teste do projeto. Esses casos de teste são escritos na ferramenta SilkCentral e o processo demanda que sejam criados casos de teste para cada fluxo do caso de uso. Uma vez criados, esses casos de teste são executados manualmente utilizando-se também como apoio uma ferramenta de execuções manuais vinculadas ao SilkCentral que é a ferramenta Manual Test Client. Devem ser definidos então quais casos de teste serão automatizados. Exige-se que pelo menos para os casos de uso classificados como críticos haja esse processo de automação para cobrir 100% dos fluxos básico, alternativos e de exceção dos casos de uso. Uma vez que se escolhe os casos de uso a automatizar, os scripts de teste relativos a tais casos de uso são criados utilizando a ferramenta SilkTest em sua versão para o Java, o Silk4j. Os scripts criados são então vinculados ao caso de teste referente à funcionalidade que o mesmo está testando. O processo da empresa prevê que para cada teste automatizado ele possa ser disparado tanto manualmente quanto a partir de

agendamentos automáticos. O agendamento de execuções e a análise de resultados dos testes pode ser feita pela ferramenta SilkCentral.

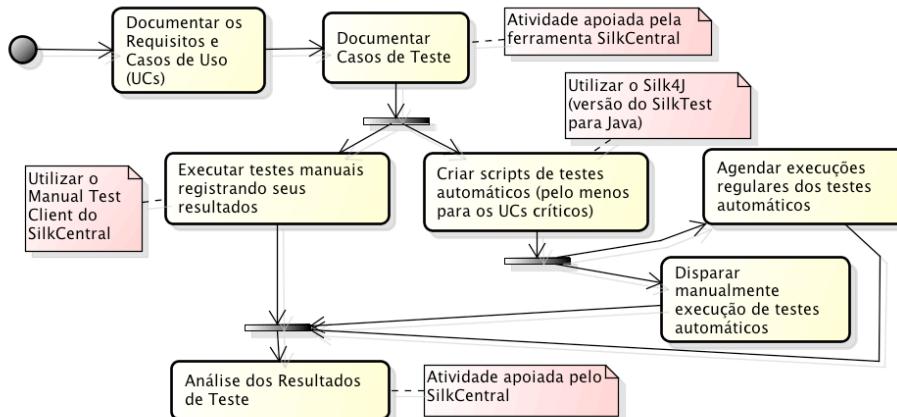


Figura 1. Fluxo de algumas atividades relacionadas aos testes funcionais

4. Avaliação do processo de testes e do uso das ferramentas

Atualmente a empresa dispõe de um processo de desenvolvimento bastante completo que aborda todas as disciplinas presentes no desenvolvimento de um software. Dentro da disciplina de testes, o processo traz bem definidos os artefatos e os procedimentos para criação dos testes automáticos. Em geral, tem-se observado que os projetos têm automatizado bem mais casos de uso que os demandados pelo padrão de aceitação dos testes automáticos, mas é importante investigar quão adequado está sendo o processo e sua aceitação por parte dos responsáveis pelo processo de automação considerando as ferramentas que estão sendo utilizadas.

Para se realizar uma melhor avaliação do uso das ferramentas e do processo de automação no ambiente da empresa pública de TI objeto deste artigo, utilizou-se um levantamento através de questionário online disponibilizado aos indivíduos envolvidos no processo de desenvolvimento dentro da empresa. Neste artigo serão relatados os dados obtidos em uma das unidades de desenvolvimento de software da empresa.

O instrumento questionário foi escolhido por ser uma importante ferramenta para se obter os dados necessários para análise. Parasuraman (1991) afirma que construir questionários não é uma tarefa fácil e que aplicar tempo e esforço adequados para a construção do questionário é uma necessidade, um fator de diferenciação favorável. Neste sentido, o questionário aplicado foi bastante estudado antes de sua aplicação. Ele continha dez questões, sendo nove destas objetivas e uma questão aberta. Seu objetivo era obter dados sobre a satisfação dos codificadores e especificadores com as atividades de automação de testes da empresa e com as ferramentas adotadas.

O questionário foi enviado por e-mail para desenvolvedores e especificadores de uma das unidades de desenvolvimento da empresa e que fazem parte de projetos que são obrigados pelo processo a seguir procedimentos de automação de seus testes e que utilizam o SilkTest. Dentre os vinte e nove (29) funcionários para os quais o questionário foi enviado e que atendiam ao perfil indicado para avaliação, dez (10) o responderam e todas as respostas preservaram o anonimato de quem as repondeu.

Considerando a distribuição da amostra que participou da avaliação, viu-se que dentre os que responderam, 40% eram codificadores, 40% especificadores e 20% acumulavam ambos os papéis dentro do projeto. Dentre os que responderam, observou-se que 70% acredita que os testes automáticos influenciam muito na qualidade dos

sistemas criados e 30% acredita que têm influencia na qualidade. Nenhum dos que responderam achou que não influenciam. Com relação à relevância percebida do processo de automação, 100% da amostra considera importante (2) ou muito importante (8) o processo de automação de testes dentro do projeto e acham compensador o esforço com automação em face dos benefícios trazidos.

Ao comparar o esforço de se projetar testes manuais com o de fazer testes automáticos, com as ferramentas existentes, as opiniões se dividiram. 40% acreditam que é mais trabalhoso se produzir os testes automáticos, 40% acreditam que o esforço é equivalente e apenas 20% acreditam que o esforço para produzir testes automáticos é menor que o esforço para sistematizar testes manuais. É importante, porém, ressaltar, que mesmo com a automação, o processo de desenvolver os casos de teste para testes manuais não deixou de existir e que tais planos são utilizados para a criação dos scripts automáticos.

Dentre os que responderam o questionário, 90% deles estão envolvidos em projetos em que a automação de testes já está de fato efetiva, sendo que 60% declaram que os testes automáticos são criados frequentemente e 30% afirmaram que em seus projetos testes são raramente criados.

E quanto ao processo adotado pela empresa e sua abordagem em relação à automação dos testes, 60% dos que responderam o questionário acreditam que as atividades pedidas pelo processo em relação a automação de testes estão adequadas às necessidades da empresa e 40% apontam que há um excesso de atividades pedidas pelo processo, sendo algumas necessárias e outras não. Nenhum dentre os que respondeu considerou que as atividades de automação requisitadas são desnecessárias. Nos questionários, perguntou-se quais casos de uso do projeto devem ser automatizados (se devem ser de fato apenas os críticos). 60% deles acreditam que todos os casos de uso do projeto devem ser automatizados, 30% deles acreditam que apenas os casos de uso considerados críticos devem ser automáticos e 10% acreditam que os casos de uso a serem automatizados devem ser escolhidos pelo cliente. No tocante à ferramenta utilizada para automação dos testes, 80% dos que responderam o questionário se declaram satisfeitos (7) ou muito satisfeitos (1) com a ferramenta utilizada na empresa. Apenas 20% declararam estar insatisfeitos.

O questionário também investigou através de uma questão aberta os benefícios observados nos projetos com o processo de automação dos testes. Vários citaram como benefício a economia de tempo (maior produtividade da equipe) e melhor segurança no desenvolvimento com a execução rápida dos testes de regressão, que visam garantir que o sistema ainda satisfaz aos seus requisitos [Rothermel 94] mesmo após mudanças. Também foram investigados os pontos negativos trazidos pela automação de testes. Dentre eles, pode-se destacar o esforço inicial de se automatizar e a dificuldade de se criar e de se manter uma massa de dados que possa ser utilizada para os testes. Porém, todos concordam que os benefícios trazidos são bem maiores que as desvantagens apontadas.

5. Conclusões

Considerando os resultados obtidos com o levantamento feito, há indícios de que a abordagem de automação de testes seguida pela empresa utilizando o SilkTest tem sido bem acolhida por seus funcionários. Uma das lições aprendidas foi que a aceitação do processo de automação de testes depende do uso de uma boa política de incentivo e que faça com que todos sintam os ganhos trazidos pela automatização dos testes. Acredita-

se que um possível fator motivador para um alto índice de aceitação das atividades de automação de testes na empresa estudada é o fato da ferramenta utilizada se basear em mecanismos de *record and replay* associada ao fato do processo da empresa já demandar a existência de planos de teste bem estruturados, o que faz com que a fase de *record* corresponda em grande parte à execução do plano já previsto e na gravação dos resultados esperados. Uma outra lição aprendida no estudo feito foi que um outro fator que contribui para a aceitação da automatização de testes é que existam poucas mudanças na interface do sistema sendo testado, pois mudanças assim fazem com que scripts de testes nesse estilo sejam frequentemente perdidos, algo que em outros ambientes gera resistência. Outro fator de impacto claro na aceitação da automação é o ganho percebido no tempo de execução dos testes automáticos comparado com o tempo e esforço de executá-los manualmente.

Pretende-se no futuro estender a avaliação para um maior número de funcionários da empresa, utilizando também entrevistas, para identificar melhorias no processo de automatização de testes e no uso das ferramentas adotadas. Um outro trabalho futuro planejado é a avaliação do processo de automação da empresa utilizando métricas mais objetivas como cobertura de código e quantidade de defeitos encontrados.

Referências

- Juran, J. M. (1992) *Juran on quality by design: the new steps for planning quality into goods and services*. Free Press.
- Parasuraman, A. (1991) *Marketing research*. New York: Addison-Wesley.
- Rothermel, G.; Harrold, M. J. (2010) “A framework for Evaluating Regression Test Selection Techniques”. Proc. Of the 16 th Int’l. Conf. On Softw. Eng., Sorrento, Italy, Mai/1994.
- Salome-TMF. Disponível em <http://forge.ow2.org/projects/salome-tmf>. Acesso em: 16 jun. 2012.
- TestLink. Disponível em <http://teamst.org/>. Acesso em: 16 jun. 2012.
- Silk Central. Disponível em <http://www.borland.com/products/SilkCentral/>. Acesso em: 16 jun. 2012.
- Selenium. Disponível em <http://seleniumhq.org/>. Acesso em: 16 jun 2012.
- TestComplete. Disponível em <http://smartbear.com/products/qa-tools/automated-testing-tools>. Acesso em 16 jun. 2012.
- SilkTest. Disponível em <http://www.microfocus.com/products/silk/silktest/>. Acesso em 16 jun. 2012.
- JMeter. Disponivel em <http://jmeter.apache.org/> Acesso em 16 jun. 2012
- Caliber. Disponivel em <http://www.borland.com/products/caliber/> Acesso em 16 jun. 2012.
- Silk Performer. Disponivel em <http://www.borland.com/products/silkperformer/>. Acesso em 16 jun. 2012.
- Together. Disponivel em <http://www.microfocus.com/products/micro-focus-developer/together/index.aspx>. Acesso em 16 jun. 2012.

Automating Test Case Creation and Execution for Embedded Real-time Systems

Joeffison S. Andrade¹, Lucas R. Andrade¹, Augusto Q. Macedo¹,
Wilkerson L. Andrade¹, Patrícia D. L. Machado¹

¹ Federal University of Campina Grande (UFCG), Campina Grande, Brazil

{joeffison.andrade, lucas.andrade, augusto.macedo}@ccc.ufcg.edu.br
{wilkerson, patricia}@computacao.ufcg.edu.br

Abstract. *Embedded Real-Time Systems (ERTS) usually operate under severe hardware constraints, making it difficult to test the software to be embedded, since this activity often requires instrumentation that may demand exclusive resources and also this instrumentation may interfere with observation of results in a given time slot (probe effect). The RealTimePCO tool has been developed to address these problems by supporting test case definition and execution on a real ERTS platform. The main goal of this paper is to present a strategy for defining and executing ERTS test cases using RealTimePCO. The expressive power and effectiveness of the strategy along with barriers yet to be faced are observed in a case study.*

1. Introduction

Embedded Real-Time Systems (ERTS) have been extensively demanded in practice to perform specialized monitoring and controlling tasks with time constraints that run on embedded devices [Li and Yao 2003]. As they often deal with critical tasks such as in the avionics and automotive domain [Efkemann and Peleska 2011, Peleska et al. 2011], verification and validation activities are needed to assess and control their safety and reliability. Particularly, software behaviour need to be checked on the embedded device that is usually limited and therefore poses a number of barriers to current techniques and tools. In this sense, software testing is a promising technique, since, in principle, it can be applied at lower costs.

However, even for plain testing, a number of challenges need to be faced. For instance, ERTS are usually concurrent and reactive systems with both synchronous and asynchronous events that are developed to run on different devices with different constraints such as memory, processors and so on. Often testing execution needs to control and observe execution without interfering with time measurement – the probe effect problem. While a number of tools have already been proposed in the literature to cope with testing of real-time systems [Larsen et al. 2005], they usually focus on abstract test case generation and online-testing with active evaluation, not addressing the probe effect problem.

RealTimePCO [Macedo et al. 2011] is a tool for ERTS testing that supports both creation and execution of test cases, particularly system and integration, in an embedded platform – the FreeRTOS [The FreeRTOS.org Project]. The tool copes with device constraints, the probe effect problem and is independent of test case generation techniques. Test cases are defined based on points of control and observation that are logged during test case execution for passive evaluation. The main architectural aspects and technical

details of the tool along with a preliminary definition of a testing process is presented in [Macedo et al. 2011]. In this paper, a detailed strategy for following this process is presented together with a case study that illustrates its applicability and observes on the effort required. Moreover, we discuss how complex test cases such as the ones that require observation of linear progression of values can be created and executed. The strategy has been developed to optimizing the time and effort spent when using the tool.

The paper is structured as follows. Section 2 describes the RealTimePCO tool and introduces fundamental concepts. Section 3 presents the testing strategy. Section 4 describes and discusses a case study in the avionics domain and Section 5 illustrates the particular case of test vectors. Section 6 comments on related work. Finally, Section 7 presents concluding remarks along with points for further research.

2. Background

For test case generation, two general approaches can be followed: *online* and *offline* [Hessel et al. 2008]. The former generates the test case during its execution: after giving an input to the *system under test* (SUT), the next input is not known until the output is received and the result analyzed. The second considers that the execution step receives as input the test cases with test data and oracles completely generated, so that the process is linear and its steps are more independent and adaptable. Regarding results evaluation, *passive* testing techniques perform test case evaluation only after test execution is completed [Lee et al. 2002]. On the other hand, *active* testing techniques may influence on the SUT behaviour since they evaluate the results during test execution.

RealTimePCO¹ [Macedo et al. 2010, Macedo et al. 2011] is a tool developed to deal with different challenges of running real-time test cases such as avoiding interferences, making control and observation possible, and allowing results to be analysed in the development platform. Therefore, this tool is applied in a testing process that considers an offline test case generation approach with a passive test case evaluation by SUT instrumentation. Interferences are reduced by adopting an offline testing strategy. In this case, all test cases are extracted from the specification, after that, they are executed against the implementation to obtain a verdict [Hessel et al. 2008]. Thus, the process is clearly divided into two steps making these activities more independent and adaptable.

In order to make control and observation possible, a test case is usually defined in terms of Points of Control and Observation (PCOs) [Naik and Tripathy 2008]. PCOs are well-designated points of interaction called ports, which is accessible to a *Test Driver*. A PCO represents a point where the SUT receives input data from the test driver (Point of Control) or produces output for the test driver (Point of Observation). In summary, a test case is defined by a sequence of input data (Points of Control) and expected results (Points of Observation).

Analysis of test results in development platforms (not in execution platforms) requires passive testing techniques. This kind of testing technique is widely used for testing networks, protocols [Lee et al. 2002] and systems that requires reduced time interference such as ERTS. The idea is to determine whether a SUT is faulty by observing its input/output behaviour without interfering with its normal operations. In this case, verdicts are only emitted through log evaluation after a complete execution of the SUT.

¹<https://sites.google.com/site/realtimepco>

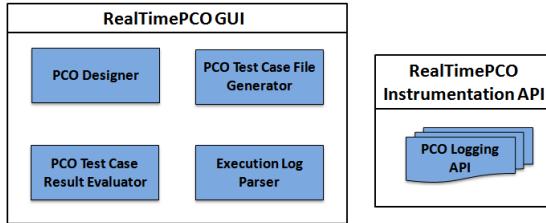


Figure 1. RealTimePCO Main Modules.

The main modules of RealTimePCO are shown in Figure 1. The PCOs are designed in a form-based GUI, then test cases are defined and automatically generated in the C language along with a *Test Driver* for each one. The PCO Logging API is used to instrument the SUT. The *Test Driver* controls execution of the test case on the instrumented SUT. From test execution, logs are generated to be analysed offline by the evaluator module that provides test case verdicts.

3. Testing Strategy

The goal of this section is to present a strategy for using RealTimePCO for ERTS testing by following the process presented in Figure 2. The focus is on functional testing that is based on testing control and behaviour observations. The main issues considered for each step are described and discussed in the sequel.

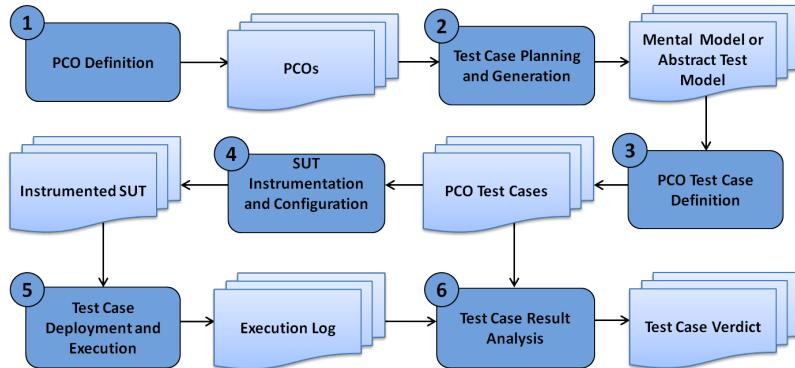


Figure 2. Stages for ERTS testing using RealTimePCO.

Step 1: PCOs Definition. When defining observations, the tester should interact with the developer to define the testing goals. Moreover, the developer should give the tester the complete list of all the input points to the system (at this step consider the system as a black box). Regarding the outputs, there are often two fundamental questions: (i) what are the most important output variables to read?; (ii) when should they be read? The amount of intrusiveness in the SUT code depends completely on these decisions. For instance, consider a reactive system where each input data is followed by an output data and a test case of just one input given and an output expected. In this case, the instrumentation inside the SUT is almost nonexistent, because it can be completely done by the *TestDriver* routine just before the end of the test case before the logging call; On the other hand, if the tester wants to test a quiescent SUT, the instrumentation should be done inside of it, possibly in a highly intrusive way.

During test case design, the tester should know exactly how the SUT is instrumented and what are the PCOs available. Since a number of test cases can share a single configuration of PCOs, it is more cost effective to define them for each testing objective, prior to test case definition. Therefore, in this step, the SUT is analysed together with the testing goals to identify the variables that need to be observed and also controlled – the PCOs. For this, it is required from the tester to have good knowledge on the implementation, since it is key for the effectiveness of the testing activity that the code is observed/controlled at the right places. At first, the tester organizes how inputs are going to be set and how output data are going to be read. Additionally, for observations, it might be necessary to manually instrument the code, for instance, by creating variables that will record partial results that are important for defining a testing verdict.

Step 2: Test Case Planning and Generation. Based on a testing goal and the completed study and definition of PCOs, a realistic test plan can be defined according to what exactly is feasible to be tested. From the plan and test criteria to be met, test cases can be created, for instance, by automatic generation from state-based machines that are often used for modelling real-time systems. Those tasks are not handled by the tool. However, RealTimePCO is independent of any particular test case generation technique and notation. As output, this step must produce test sequences defined in terms of the PCOs previously identified that can be provided as input for the tool in the next step.

Step 3: PCO Test Case Definition. The possibly abstract test cases generated in the last step are then defined in terms of PCOs using the RealTimePCO GUI. From them, the initial test files are generated: a *TestContext* with the *TestDriver* task only; a *TestContextConfig* with the configuration variables and functions of the *TestDriver* task; a *TestAdapter* with the functions needed to interface the SUT and the *TestContext*. Now the generated files contain the *TestDriver* task and the test case sequences prepared to control and observe the instrumented SUT.

The test case is a sequence of steps, which will be executed by the *TestDriver*, with two types of commands: (i) The input data command: every input data is sent by the control points defined before and it occurs while the SUT is not running (the *TestDriver* has the highest priority, so it should execute alone and every time it needs); and (ii) the block command: the *TestDriver* blocks for an amount of time – this command is used to let the SUT execute during a predefined amount of time. This time value is defined by the test case (it should be larger than the largest interval of the test case) and by the SUT specification, that could have some extra constraints, like an initial load time, etc.

Next, the test oracle is defined for passive evaluation as a test tree building activity diagram with assertions. For each block command of the test case sequence, it is allowed to create a sequence of success assertions and several sequences of inconclusive assertions, where each sequence of inconclusive assertions is an inconclusive branch. If there is more than one block command, the assertions are sequentially executed, following the block sequence. It is important to remark that, if the tester has any difficult to write the assertions, it can postpone this step, jump to the test execution phase and, at the end, take a look at the log file then come back to create the assertions.

Step 4: SUT Instrumentation and Configuration. All defined PCOs are added to the SUT code (the instrumentation characteristics dictate the max number of observations each point needs). The initial test files generated in the last step are added to the SUT along with the PCO Logging API. Additionally, for observations, it might be necessary to manually instrument the code with calls to the PCO Logging API. This is the last step of the instrumentation, where the files are added and the needed changes in the code are done. At this time, the code should compile nicely. The result of this step is a completely instrumented SUT prepared to test. Differently from common black-box testing techniques in which test cases are defined by choosing input points, the RealTimePCO testing tool allows to choose input points along with different points of instrumentation, increasing the quantity of possible test cases.

Step 5: Test Case Deployment and Execution. The test case sequences previously defined are inserted into the *TestContext* file that is deployed in the embedded device. The SUT is executed with total control of the *TestDriver* task. Logs are collected during the execution. This step may require assistance from the development team, since test execution (in embedded systems mainly) may require extra environment configurations.

Step 6: Test Case Result Analysis. The log file for each test case is parsed in the RealTimePCO GUI and the oracle assertions are executed, from the root to the leaves following the success path. If it did not pass, the execution backtracks visiting the previous inconclusive paths.

4. Case Study

The system considered in this study is a simulator of an actuation system from an airplane inspired by the specification presented in [de Jesus Júnior 2009]. It was implemented by the authors and it has no more than 500 lines of code. The Actuation System receives a command from the pilot control column to take off with an amount of inclination, the system accumulates pressure linearly then sends this pressure into the raw actuator that push the wing of the airplane in some direction. This system is formed by 5 tasks that communicate through queues in a synchronous manner, so one task sends a message and it cannot continue while the other does not read it. The system is called a simulator because the time constraints are not fixed, they can be set before the execution or during it, changing the behaviour of the system accordingly.

Figure 3 shows the communication between the 5 tasks (*Solenoid Valve*, *EHSV*, *Actuation Raw*, *LVDT* and *ACE*) and their behaviour. Basically, *ACE*, which is the highest priority task, is designed to receive commands from the pilot, a new pressure request, which are sent to *EHSV*. This task controls *Solenoid Valve*, which periodically sends pressure to *EHSV*, when engaged, until it reaches the required value by *ACE*. After the value is reached, this value is sent to *Actuation Raw* that calculates the displacement value and the force from a simple simulation of these functions. After the value has been computed, it is sent to an *LDVT* which also makes a simple simulation of a linear variable differential transformation. The value is then returned to *ACE* that checks if the command was carried out and displays the verdict before returning to standby for a new command.

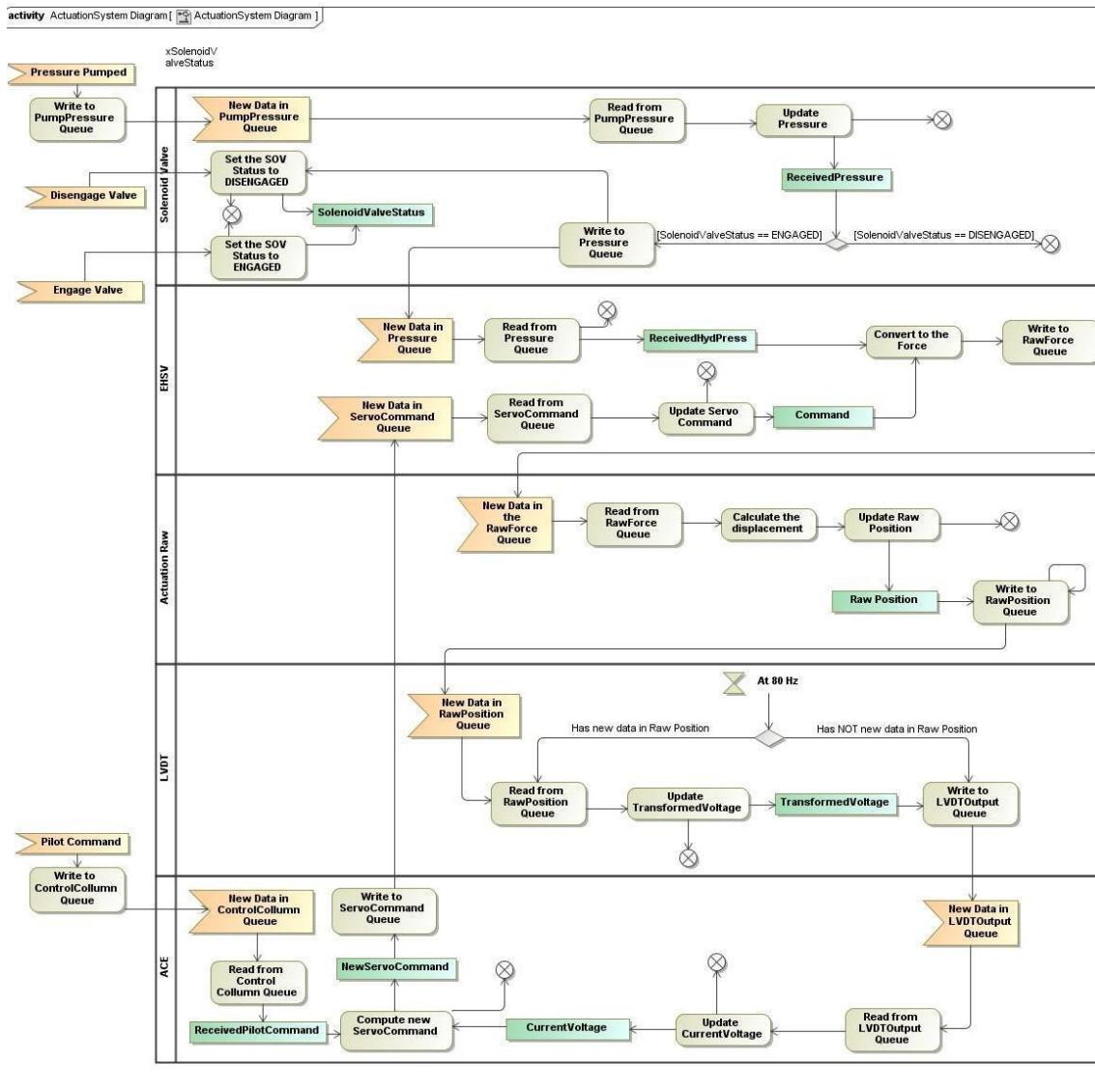


Figure 3. Actuation System Behaviour.

The PCOs were extracted from the activity diagram. The goal was to identify all system variables that needed to be set or have values changed. A total of 4 points of control and 10 points of observation were identified. During this activity, the tester lasted 1 minute and 29 seconds interacting with the tool.

For this system, 20 test cases were chosen to explore situations where: (i) no time constraints are imposed; (ii) stability is kept by certain uses; (iii) soft time constraints are set; (iv) hard time constraints are set; (v) preconditions are not met. Using the RealtimePCO GUI, all 20 test cases were defined in 31 minutes and 17 seconds, an average of 1 minute and a half per test case.

At the SUT instrumentation and configuration step, the Logging API and all files generated by RealtimePCO (TestAdapter.c, TestAdapter.h, TestContext.c, TestContext.h, TestContextConfig.c e TestContextConfig.h) were added to the SUT context. For each observation, a call to xReadObservation("⟨⟨observation_point_id⟩⟩") was manually added

to the SUT code. For including all observation points, 11 lines of code were added to the SUT. After that, the main module of the SUT was changed in order to import TestContextConfig.h and its first line became a call to vStartTestDriver(), which is responsible for starting the execution of test cases. Finally, some methods of the TextContextConfig.c file need to be filled out in order to connect the SUT with the test architecture. All this step was performed in 12 minutes and 39 seconds.

The instrumented SUT was deployed into the FreeRTOS Cortex-M3 port using the LPCXpresso² development platform and its execution lasted 3 minutes and 57 seconds ininterruptely. After execution, all logs were automatically analysed for emitting test verdicts in less than 1 second.

The test suite and all test files can be downloaded in the RealTimePCO site¹. This case study was completely executed in less than 50 minutes. It allows to realise that the knowledge of the SUT is very important during the testing process. However, the adopted process overcomes important barriers of ERTS testing such as definition of test cases and analysis of results in the development platform, while the execution of tests is performed in the execution (target) platform. Furthermore, RealtimePCO takes the probe effect problem into account.

5. Observing Linear Progression of Values

As mentioned before, ERTS testing often requires a complex test design due to the fact that multiple processes are required to interact and synchronize avoiding the probe effect. Additionally, for controlling systems, it is often required to observe on the progressive change of values of a given variable. For instance, consider the actuation system. If a new increment command is established by the pilot, internal variables must be progressively and precisely updated according to this increment otherwise the aircraft may loose balance. Therefore, test cases must observe whether or not this value will be correctly and progressively changed. In the sequel, we describe a test case that can be used to test this kind of scenario and what are the steps to implement it in RealTimePCO.

The scenario to be tested is the one where the variable *lTransformedVoltage* grows linearly with a growth factor equals to the value of the variable *lCommand*, while the system is working as specified.

The first step is to create the points of control and observation in RealTimePCO. Therefore, an observation point for *lTransformedVoltage* and a control point for *lCommand* is created. After that, the auxiliary control points (*lPressure*, *lNewCommandInterval*, and *lNewPressureInterval*) are created in order to have a precise control of the SUT.

The second and third steps are the creation of the test case and the definition according to the PCOs. The test case sequence is created paying attention to these aspects: the *lTransformedVoltage* variable has a factor that is another variable, *lCommand*, in such way that the eventual change of the value of *lCommand* should affect the linear growth of *lTransformedVoltage*; the system must remain stable and working as specified. Steps are added in the test case, where, in each step, the value of *lCommand* is changed; some auxiliary inputs can be added in order to keep the system stable. Then, the assertions are created using the Valued Linear Creation functionality of RealTimePCO to automatially

²<http://www.lpcxpresso.code-red-tech.com>

Input Vector	Expected Outputs Assertions		
lCommand:20; lPressure:4; lNewCommandInterval:4000; lNewPressureInterval:400	lTransformedVoltage==020; lTransformedVoltage==060; lTransformedVoltage==100	lTransformedVoltage==040; lTransformedVoltage==080;	
lCommand:40; lPressure:8	lTransformedVoltage==140; lTransformedVoltage==220; lTransformedVoltage==300	lTransformedVoltage==180; lTransformedVoltage==260;	
iCommand:60; lPressure:10	lTransformedVoltage==360; lTransformedVoltage==480; lTransformedVoltage==600	lTransformedVoltage==420; lTransformedVoltage==540;	

Table 1. Test case that checks whether lTransformedVoltage grows linearly, with or without changes to the command and pressure steps.

add, in every step of the generated test case that has a linear growth factor, a sequence of assertions to check whether a value equal to the current value of *lCommand* was considered in the increment. An instance of this test case with data can be seen in Table 1 where 5 changes of the variable are scheduled to be observed.

The fourth step is the instrumentation of the SUT. At this step, the files TestContext, TextContextConfig, and TestAdapter are added and properly configured in the SUT. The points of observation are specifically added in the SUT, exactly before the output of the observed variables. Finally, the fifth step is the execution of the test case. After that, the log file is generated. The sixth step is the evaluation, where the log file is loaded and analysed by RealTimePCO. This is illustrated in Figure 4, where Step 1 is the setup and the following steps correspond to the sequences from Table 1.

6. Related Work

Di Guglielmo *et al* [Di Guglielmo et al. 2011] proposes a framework for designing and validation of embedded software where test cases for testing PSL properties can be automatically generated. However, test execution is performed by simulation, not running on a real platform as proposed by our work. Also, the RT-Tester³ tool fully supports management, generation and execution of test cases for embedded software, but the scope is on active evaluation and simulation. However, the analysis of test results in development platforms through passive evaluation is more appropriate for avoiding probe effect.

UPPAAL TRON is a conformance testing tool for RTSs based on the Uppaal Timed Automata. The testing approach is online through adapters written in C++ or Java [Larsen et al. 2005]. This tool was designed to run software integration tests, so it is not prepared to run in a real-time operating system. *Rational Test RealTime*⁴ is a proprietary tool of IBM that executes offline test cases defined in a proprietary language and evaluate actively the test results, interfering on the time. Another proprietary tool is *VectorCast*⁵, restricted to software testing at the unit level. On the other hand, our goal is to support test execution in the device (execution platform) with passive evaluation in the development platform.

³<http://www.verified.de/en/products/rt-tester>

⁴<http://www.ibm.com/software/awdtools/test/realtime>

⁵<http://www.vectorcast.com>

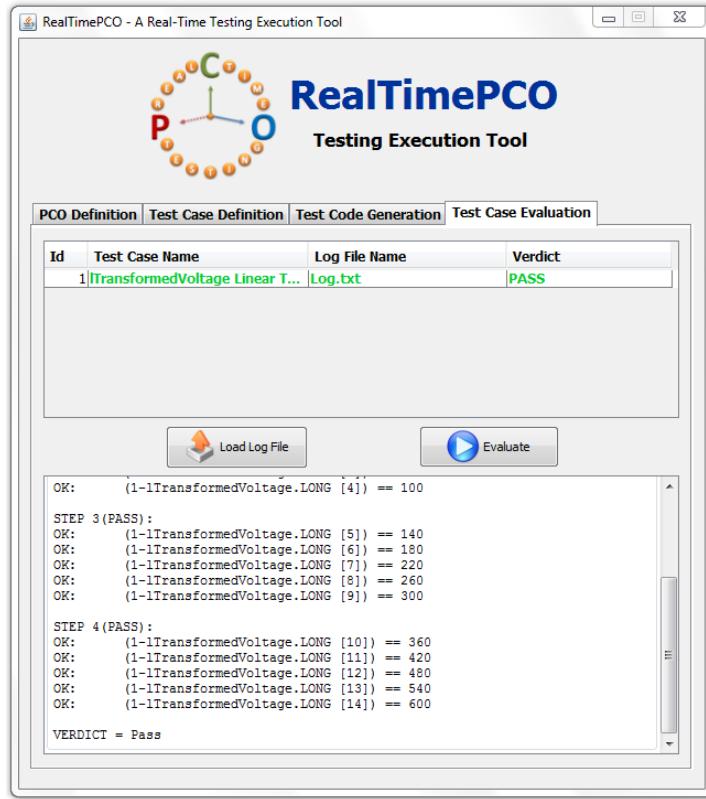


Figure 4. RealTimePCO Results Evaluation.

TTCN-3 is an international test standard consolidated by the industry. Its main characteristic is modularization with the adapters, between the test case (specified in an abstract language with the same name) and the system at code level. The standard was updated to aggregate the ERTS testing needs [Serbanescu and Schieferdecker 2010]. However, the actual time is not guaranteed if the test is based on message transfer between different systems. To the best of our knowledge, there is no tool that implements these updates on the standard and, even so, the proposed solution focus on active evaluation.

7. Concluding Remarks

This paper presents a test case definition and execution strategy based on the use of the RealTimePCO tool for functional software testing of ERTS. The strategy is based on the definition of PCOs that guide test case definition and code instrumentation/generation. Test case evaluation is passive in order to address the probe effect as defined by RealTimePCO. The strategy is independent of any particular test case generation notation and technique. By using RealTimePCO, test cases can be defined to be executed in a real embedded platform. A case study illustrates the effort needed to follow the strategy. Moreover, definition and execution of a test case that observes on linear progression of values is discussed. As further work, the tool will be integrated with different test case generation tools and also embedded platforms in to provide a complete automated solution. Also, scalability of the strategy needs to be evaluated by extensive case studies.

Acknowledgement This work was supported by CNPq grants 484643/2011-8 and 560014/2010-4. Also, this work was partially supported by the National Institute of Science and Technology for Software Engineering (INES⁶), funded by CNPq/Brasil, grant 573964/2008-4.

References

- de Jesus Júnior, J. B. (2009). Designing and formal verification of fly-by-wire flight control systems. Master's thesis, Federal University of Pernambuco.
- Di Guglielmo, G., Fujita, M., Di Guglielmo, L., Fummi, F., Pravadelli, G., Marconcini, C., and Foltinek, A. (2011). Model-driven design and validation of embedded software. In *AST'11*, pages 98–104, New York, NY, USA. ACM.
- Efkemann, C. and Peleska, J. (2011). Model-based testing for the second generation of integrated modular avionics. In *2011 IEEE ICSTW/AMOST*, pages 55–62, Washington, DC, USA. IEEE Computer Society.
- Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P., and Skou, A. (2008). Testing real-time systems using UPPAAL. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer.
- Larsen, K., Mikucionis, M., and Nielsen, B. (2005). Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer.
- Lee, D., Chen, D., Hao, R., Miller, R. E., Wu, J., and Yin, X. (2002). A formal approach for passive testing of protocol data portions. In *ICNP'02: Proc. of the 10th IEEE Int. Conf. on Network Protocols*, pages 122–131. IEEE Computer Society.
- Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. CMP Books.
- Macedo, A. Q., Andrade, W. L., Almeida, D. R., and Machado, P. D. L. (2010). Automating test case execution for real-time embedded systems. In *ICTSS'10*, pages 37–42. Short Paper.
- Macedo, A. Q., Andrade, W. L., and Machado, P. D. L. (2011). RealTimePCO - A Tool for Real-Time Embedded Systems Testing Execution. In *18th Tools Session of the 2nd Brazilian Conference on Software: Theory and Practice (CBSOFT 2011)*, pages 72–78.
- Naik, K. and Tripathy, P. (2008). *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Inc.
- Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., and Zahlten, C. (2011). A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *ICTSS'11*, pages 146–161, Berlin, Heidelberg. Springer-Verlag.
- Serbanescu, D. A. and Schieferdecker, I. (2010). Testing environment for real-time communications intensive systems. In *ICNS'10: Proc. of the Sixth Int. Conf. on Networking and Services*, pages 368–374. IEEE Computer Society.
- The FreeRTOS.org Project. FreeRTOS. <http://www.freertos.org>.

⁶www.ines.org.br

Controlando a Diversidade e a Quantidade de Casos de Teste na Geração Automática a partir de Modelos com *Loop*

Jeremias D. S. de Araújo¹, Emanuela G. Cartaxo¹,
Francisco G. de Oliveira Neto¹, Patrícia D. L. Machado¹

¹SPLab-UFCG, Av. Aprígio Veloso, 882, Campina Grande, Brasil

jeremias.araujo@ccc.ufcg.edu.br,

{emanuela,netojin}@copin.ufcg.edu.br, patricia@computacao.ufcg.edu.br

Abstract. Depending on the test criteria and selection strategy considered, a huge or even infinite number of test cases, possibly with a high degree of redundancy, can be generated from a model, particularly due to the presence of loops. This paper presents a new technique for the automatic generation of test cases from models that parameterizes the number of times loops should traversed, maximizing the exploration of different sequences. The technique follows all-one-loop-paths coverage as stop criteria and uses a basic strategy to convert a model into an oriented tree expanded with loop replications. A case study illustrates the diversity of test cases that can be generated.

Resumo. Dependendo do critério de teste e da estratégia de seleção adotados, um número infinito ou muito grande de casos de teste com considerável redundância pode ser gerado devido, principalmente, à presença de loops. Este artigo propõe uma técnica para a geração automática de casos de teste a partir de modelos que parametriza a quantidade de vezes em que loops são percorridos e maximiza a exploração de diferentes sequências. A técnica adota o critério all-one-loop-paths para parada e uma estratégia de conversão do modelo em uma árvore orientada expandida com replicação dos loops. Um estudo de caso mostra a diversidade de casos de teste que podem ser gerados.

1. Introdução

Geração automática de casos de teste a partir de modelos pode tomar como base algum tipo específico de máquina de estados [Pretschner 2005]. Dada uma máquina de estados que representa um determinado sistema, é aplicado um algoritmo de busca – seja ele em profundidade ou largura – e dessa forma, os casos de teste são obtidos.

O problema abordado neste artigo vem a tona quando – em algum dos modelos usados para geração – aparecem um ou mais *loops*. Diante disto, o algoritmo usado para geração precisa ser capaz de lidar com esses *loops*; caso contrário, ao aplicar apenas um algoritmo de busca tradicional é possível enfrentar o problema de explosão de estados e, consequentemente, gerar um conjunto de casos de teste infinito. Por outro lado, pode-se usar um critério de cobertura, como, por exemplo, o “all-one-loop path” que sugere a geração de casos de teste que cubram todos os caminhos passando pelo menos uma vez nos *loops*. O fato de passar uma ou mais vezes por um *loop* (repetição) é interessante pelo fato de testar exaustivamente funções que poderão ser causas de exceções (quando

exercitadas repetidamente) e, consequentemente, revelar defeitos. Porém, esta repetição pode também ser considerada uma fonte de redundância no conjunto gerado, de forma que é fundamental a análise apropriada do contexto no qual essa geração será aplicada.

Neste artigo, apresentamos uma técnica para geração automática de casos de teste, considerando como modelo base os Sistemas de Transições Rotuladas (*Labelled Transition Systems* – LTS). Essa técnica recebe como entrada um modelo LTS e o número de replicações desejada, caso exista *loops* no modelo. A idéia é transformar o LTS em uma “árvore orientada”. Esta árvore gerada apresenta marcações que indicam onde as repetições podem ocorrer. Paralelamente são geradas subárvores que representam a repetição da sequência que exercita o *loop* no modelo. A cada replicação (referenciada neste artigo como “expansão da árvore”) desejada, os vértices dessa subárvore são inseridos no vértice correspondente. Este processo garante o controle da quantidade de vezes que o *loop* será exercitado pelas sequências (i.e. caminhos) geradas.

Este artigo é estruturado da seguinte forma. Na Seção 2, são apresentados conceitos básicos, seguidos por considerações acerca de trabalhos relacionados (Seção 3). A descrição da técnica proposta é feita na Seção 4 por meio de um exemplo de sua execução. Por sua vez, na Seção 5, é apresentado um estudo de caso que ilustra a diversidade de casos de teste que podem ser gerados quando comparada a outras técnicas.

2. Fundamentação Teórica

Grafos e Árvores Um grafo $G = (V, E)$ é composto por um conjunto de vértices (V) e um conjunto de arestas (E). Cada aresta é composta por dois elementos de V . Em outras palavras, uma aresta relaciona dois vértices u, v . Quando a ordem dos vértices é levada em consideração, o grafo é dito direcionado, caso contrário, é não-direcionado. Uma árvore é um tipo especial de grafo: não-direcionado e acíclico. É comum designar um vértice como *raiz*, e consequentemente, considerar a ordem dos demais vértices que são alcançáveis a partir da raiz. As árvores que tem um vértice raiz e possuem arestas direcionadas são conhecidas como *árvores orientadas*. Sob uma perspectiva simplificada, uma árvore orientada é um grafo dirigido e acíclico.

Labelled Transition Systems (LTS) Um LTS é uma quádrupla $S = (Q, A, T, q_0)$, onde Q é um conjunto finito não vazio de estados; A é um conjunto finito não vazio de rótulos (denotando ações); T , a relação de transição, é um subconjunto de $Q \times A \times Q$ e q_0 é o estado inicial.

Critérios de Cobertura O critério de cobertura é um conjunto de regras que impõe requisitos de teste em um conjunto de casos de teste. O critério de cobertura especifica itens do sistema que devem ser exercitados durante o teste. Aquele pode ser usado para medir a adequação do conjunto de casos de teste (se o nível de cobertura de um dado critério é atendido) ou decidir quando parar de testar (testes são executados até se atingir um determinado nível de cobertura) [Ammann and Offutt 2008]. Usualmente, a cobertura é baseada em elementos do modelo utilizado para geração (e.g. estados, saídas, entradas, dentre outros). Considerando uma máquina de estados, por exemplo, podemos citar alguns critérios de cobertura, dentre eles [Uutting and Legeard 2006]: (a) (*all-states*) - todo estado deve ser visitado pelo menos uma vez; (b) (*all-transitions*) - toda transição deve ser visitada pelo menos uma vez; (c) (*all-transition-pairs*) - todo par de transições adjacentes deve ser visitado pelo menos uma vez; (d) (*all-loop-free-paths*) - todos os caminhos sem

loops são visitados pelo menos uma vez (um caminho é considerado sem *loops* quando não tem repetição); (e) (*all-one-loop-paths*) - o *loop* é exercitado no máximo uma vez; (f) (*all-round-trips*) - requer um teste para cada *loop* no modelo, mas não requer que todos os caminhos que precedem ou seguem um *loop* sejam testados.

3. Trabalhos Relacionados

Na literatura, há diversos trabalhos referentes à geração automática de casos de teste a partir de modelos [Hartman and Nagin 2004, Santiago et al. 2006, Machado and Sampaio 2010]. A geração, na maioria destes trabalhos, ocorre a partir da cobertura de elementos do modelo utilizado para representar o software (e.g. estados, transições, condições, dentre outros). Diante disto, de acordo com o critério utilizado, diferentes conjuntos de casos de teste podem ser obtidos. Portanto, esse critério de cobertura utilizado é um dos fatores que deve ser investigado durante o desenvolvimento do algoritmo de geração, procurando atingir o objetivo do teste a ser executado.

Simão *et al.* apresentaram uma técnica denominada “SPY-method”, capaz de gerar conjuntos de casos de teste menores por meio da cobertura de diferentes ramos de uma Máquina de Estados Finita (MEF). Outro aspecto explorado pela técnica é a geração de um conjunto mínimo de casos de teste capaz cobrir os “m” estados em uma MEF referente à implementação de uma MEF da especificação do software [Simão et al. 2009]. Diante disto, o conjunto gerado tem como um dos objetivos cobrir os estados da MEF, e os autores não discutem a perspectiva acerca da cobertura de *loops* na MEF.

Cartaxo *et al.* apresentam uma estratégia para a geração automática de casos de teste a partir de diagramas de sequência. Esta estratégia observa as interações entre o usuário e o sistema representadas por meio de um diagrama de sequências, e a partir destas interações, um LTS é utilizado para a geração automática dos casos de teste [Cartaxo et al. 2007]. Porém, como o algoritmo de geração é baseado em uma busca em profundidade, um *loop* é executado no máximo uma vez.

Já Sapna e Mohanty [Sapna and Mohanty 2009] apresentaram um algoritmo para a geração automática de casos de teste a partir de digarams de atividades. Este algoritmo transforma um diagrama de atividade em uma árvore, aplicando uma DFS e restringe que o loop seja executado no máximo duas vezes para evitar o problema da explosão de caminhos, desta forma, uma atividade é visitada no máximo duas vezes.

Um dos principais aspectos referentes à técnica proposta neste trabalho é a possibilidade de parametrizar a cobertura de *loops* num conjunto de casos de teste gerados automaticamente a partir de um modelo. Junto a esta cobertura de *loops*, também é atingida a cobertura de transições e estados do modelo, permitindo a geração de um conjunto de casos de teste capaz de explorar os diversos cenários do software. Detalhes acerca do objetivo e execução desta técnica são apresentados na seção seguinte.

4. Descrição da Técnica

A técnica consiste em transformar de forma automática um modelo LTS com *loops* em uma Árvore de casos de teste, onde cada caminho entre o nó raiz e suas respectivas folhas representa um caso de teste, eliminando dessa forma os *loops* presentes no LTS.

A principal característica da técnica de geração é a parametrização, ou seja, apesar de não existir *loops* em árvores, a estrutura carrega consigo a possibilidade de expansão,

sendo assim possível controlar a quantidade de vezes que se deseja exercitar um *loop*. Esse controle de passagem no *loop* é feito por inserção de subárvores de repetição, que são obtidas observando as estruturas de repetição no LTS.

A técnica é subdividida em três etapas: (1) Geração da árvore a patir de um modelo LTS; (2) Geração das subárvores de repetição; e (3) Expansão, que é processo de colagem das subárvores de repetição, que serão detalhados nas próximas seções.

4.1. Problema na Geração com *Loops*

Considerando o modelo mostrado na Figura 1 (a) e aplicando-se a DFS (*Depth First Search*) tradicional (da raiz até a folha), será obtido o seguinte conjunto de casos de testes que pode ser visto na Figura 1 (b). Suponha que fosse necessário exercitar o loop do modelo (aresta *g*) mais de uma vez; como exemplo temos o uso do cartão de crédito. Sabe-se que o cartão é bloqueado caso a senha seja informada 3 vezes consecutivas de forma incorreta, então é necessário testar a senha informada ao menos 3 vezes. A técnica aqui apresentada torna possível esse tipo de verificação de forma simples e automática.

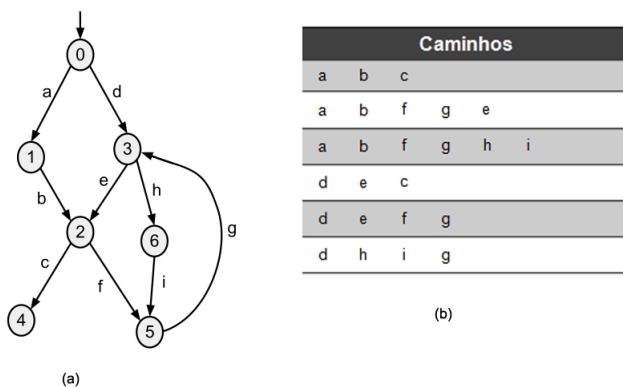


Figura 1. Exemplo para descrição do problema com *loops*.

4.2. Geração da Árvore

A árvore é então gerada a partir do LTS. À medida que as transições do grafo são exercitadas por meio de uma DFS, os respectivos nós e ligações da árvore são criados. Na Figura 2 é apresentado o pseudocódigo relativo a esta etapa.

```

01: #Pseudocódigo da criação da árvore
02:
03: DFSGERAÇÃO(v) {
04:   para cada aresta saindo de v:
05:     se v não foi visitado
06:       marca v como visitado
07:       criamos os nós pai e filho e uma ligação entre eles
08:
09:       DfsGeração(vTo)
10:       remove v dos visitados
11:       se v já foi visitado
12:         adicionamos o nó a uma lista de marcados
13: }

```

Figura 2. Pseudocódigo - Geração da Árvore.

pass

Como pode ser visto na Figura 2, uma DFS tradicional é executada no modelo LTS e, a cada aresta exercitada do grafo, são criados e adicionados nós e ligações correspondentes a árvore. Se o vértice já foi visitado então o vértice correspondente será adicionado a uma lista de nós marcados. É através desses nós que faremos a expansão da árvore. Veja a árvore gerada na Figura 3 (a) a partir do grafo da Figura 1 (a), note que os nós “2”, “5” , “3”, e “3” na Figura 3 (a) estão destacados, pois estão na lista de nós marcados.

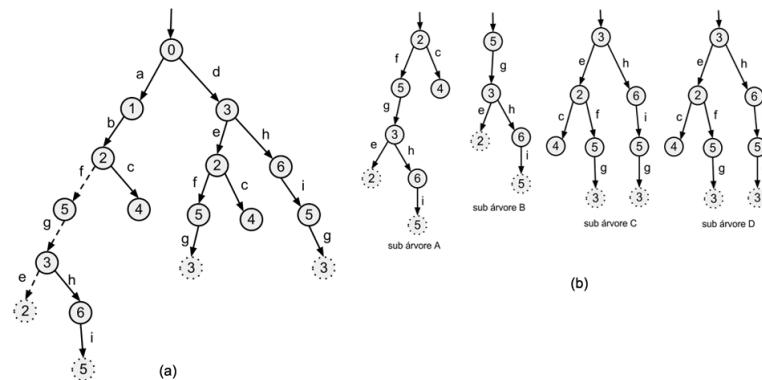


Figura 3. Árvore e as respectivas subárvores.

Acompanhe a sequência “0 – 1 – 2 – 5 – 3 – 2” no grafo Figura 1 (a). É possível observar que esse mesmo caminho foi criado na árvore - Figura 3 (a) - e o último no “2” está destacado, pois no grafo ele já tinha sido visitado na sequência.

4.3. Geração de Subárvores de Repetição

As subárvores de repetição representam uma passagem no *loop*. Elas são criadas a partir da árvore, aplicando o algoritmo de DFS e são guardadas em uma lista estática acessada apenas no processo de expansão onde são inseridas na árvore. Esse processo de gerar as subárvores é feito uma única vez. Veja o pseudocódigo na Figura 4.

```

01: #Pseudocódigo da criação das subárvores
02:
03: GERARSUBARVORE () {
04:     para cada nó marcado
05:         soba na árvore até encontrar um nó com mesma identificação
06:         dfs (nó) {
07:             ...cria subárvore
08:         }
09:         guarda a subárvore em uma lista
10: }
```

Figura 4. Pseudocódigo - Geração de subárvores de repetição.

Assim, como descrito acima, a obtenção das subárvores de repetição é feita com uma DFS tradicional. Acompanhe a criação da subárvore A na Figura 3 (b), através do nó “2” que esta destacado na Figura 3 (a), sobe-se na árvore (caminho tracejado na árvore 2-3-5-2) procurando o nó com mesmo rótulo, nesse caso o nó de rótulo “2”. Quando encontrado, aplica-se uma DFS criando a subárvore A. Perceba que as subárvores de repetição mantêm uma lista de nós marcados assim como a árvore. No exemplo da subárvore A, os nós “2” e “5” compõem a lista de nós marcados. Ao fim da DFS, guarda-se a subárvore para o próximo passo, a expansão.

4.4. Expansão da Árvore

Na expansão da árvore, adiciona-se cada subárvore de repetição a seu respectivo nó de origem, ou seja, faz-se a inserção das subárvores na árvore. Na Figura 5, é apresentado o pseudocódigo que descreve a expansão da árvore.

```

01: #Pseudocódigo da expansão
02:
03: EXPANSÃO(nóMarcado[], subárvores[], N) {
04:     while N->0{
05:         para cada nó marcado
06:             encontre sua subárvore(faça uma cópia)
07:             COLAGEM(nó, subárvore)
08:             ... insere a subárvore na árvore
09:             atualiza a lista de nós marcados da árvore
10:     }
11: }
```

Figura 5. Pseudocódigo - Expansão da Árvore.

Aplicando o algoritmo acima na árvore da Figura 3 (a) com as subárvores na Figura 3 (b), obtém-se a árvore da Figura 6 (a). Perceba que a nova árvore pode ser expandida várias vezes e cada expansão significa que o loop foi exercitado uma vez, assim - se necessário - podem ser criados casos de testes que passem 3 vezes pelo(s) loop(s) basta fazer o parâmetro $N = 3$.

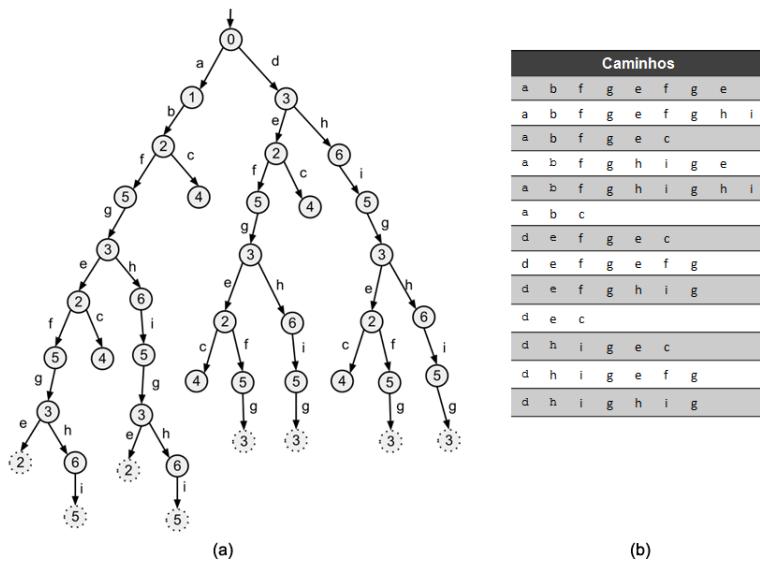


Figura 6. Árvore com Expansão (a) e Casos de Testes gerados (b)

Para esse exemplo, assim como para outros, é possível prever a quantidade de casos de teste obtidos com N expansões especificadas. Note o padrão obtido para o exemplo acima na Tabela 1. A previsão da quantidade de casos de teste não pode ser generalizada, sendo dependente da estrutura do modelo.

Na Figura 6 (b) temos o conjunto de casos de teste para uma expansão da árvore da Figura 3 (a). Veja que na Figura 1 (b) havia 6 casos de teste, enquanto que na Figura 6 (b) há 13 casos de teste. Dessa forma, a cada expansão aumentamos a cobertura de caminhos e de testes do modelo.

Tabela 1. Descrição acerca da previsão na quantidade de casos de teste para este exemplo.

N (repetições)	Exemplo	Nós /Folhas Marcadas	Folhas não marcadas	Casos de teste
N = 0	1 ^a árvore (Figura 2)	2-5-3-3 (4 elementos)	2	$4 + 2 = 6$
N = 1	2 ^a árvore (Figura 4)	2-5-2-5-3-3-3-3 (8 elementos)	5	$8 + 5 = 13$
N = 2	3 ^a árvore	2-5-2-5-2-5-2-5-3- 3-3-3-3-3-3-3 (16 elementos)	11	$16 + 11 = 27$
...				
N = n	n-ésima árvore	$2^{(n+1)}$	$3 \times 2^{(n-1)} - 1$	$2^{(n+1)} + 3 \times 2^{(n-1)} - 1$

5. Estudo de Caso

Para observar o desempenho da técnica, executamos um estudo de caso, em que três técnicas de geração automática de casos de teste a partir de modelos são executadas e os respectivos resultados são então comparados e analisados com relação à cobertura de sequências (i.e. caminhos) no modelo. Uma vez que cada algoritmo apresenta uma estratégia de geração diferente, os conjuntos gerados serão diferentes a partir das sequências de transições que eles cobrem.

Partindo de um LTS, os casos de teste foram gerados utilizando um algoritmo de busca em profundidade tradicional (DFS), um algoritmo de geração proposto por Sapna e Mohanty [Sapna and Mohanty 2009], e a técnica proposta neste trabalho. O modelo utilizado como estudo de caso é apresentado na Figura 7.

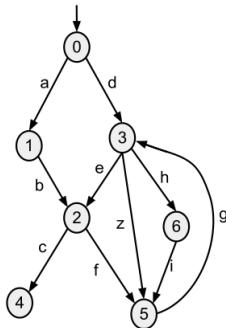


Figura 7. Modelo utilizado como estudo de caso

Para a execução do estudo de caso, foi utilizada apenas 1 expansão da árvore gerada pela técnica proposta neste trabalho (Subseção 4.4). Esta quantidade foi definida com o objetivo de manter uma igualdade com a quantidade de vezes em que o *loop* é executado pelo algoritmo de geração proposto por Sapna e Mohanty. Dessa forma, evitamos um viés na comparação referente à cobertura, uma vez que, na técnica proposta aqui, mais expansões caracterizam uma cobertura maior dos *loops* do modelo.

A partir do modelo da Figura 7, foram gerados os conjuntos de casos de teste apresentados na Figura 8 (a), (b) e (c) (respectivamente, os conjuntos referentes ao algo-

ritmo DFS – D, a técnica proposta neste trabalho – T, e o algoritmo proposto por Sapna e Mohanty – S). Para cada conjunto de casos de teste, foram observadas: a quantidade de casos de teste gerados; e as transições cobertas pelos diferentes casos de teste.

(a)			(b)		
D	Caminho		T	Caminho	
D1	a	b	D1	a	b
D2	a	b	D2	a	b
D3	a	b	D3	a	b
D4	a	b	D4	a	b
D5	d	e	D5	a	b
D6	d	e	D6	a	b
D7	d	z	D7	a	b
D8	d	h	D8	a	b
S	Caminho		T	Caminho	
S1	a	b	T1	a	b
S2	a	b	T2	a	b
S3	a	b	T3	a	b
S4	a	b	T4	a	b
S5	a	b	T5	a	b
S6	a	b	T6	a	b
S7	a	b	T7	a	b
S8	d	e	T8	a	b
S9	d	e	T9	a	b
S10	d	e	T10	a	b
S11	d	e	T11	a	b
S12	d	e	T12	d	h
S13	d	z	T13	d	h
S14	d	z	T14	d	h
S15	d	z	T15	d	h
S16	d	z	T16	d	e
S17	d	h	T17	d	e
S18	d	h	T18	d	e
S19	d	h	T19	d	e
S20	d	h	T20	d	e
			T21	d	z
			T22	d	z
			T23	d	z
			T24	d	z

(c)

Figura 8. Conjuntos gerados por cada técnica

É possível observar que o algoritmo com a menor quantidade de sequências cobertas foi o DFS (Figura 8 (a)). Com ele foram gerados apenas 8 casos de teste, enquanto que os demais geraram muito mais casos de teste - 24 e 20 casos de teste, para as Figuras 8 (b) e (c), respectivamente. Outra característica do conjunto gerado pelo algoritmo DFS é que o *loop* é exercitado apenas 1 vez, e que a cobertura de alguns caminhos envolvendo este *loop*, não é realizado (e.g. o caminho *d,e,f,g,h,i*).

Por sua vez, o algoritmo proposto por Sapna e Mohanty gerou um conjunto com 20 casos de teste. Em alguns casos de teste deste conjunto, o *loop* é exercitado de 1 a 2 vezes, sendo capaz de explorar mais cenários quando comparado ao algoritmo DFS tradicional, fornecendo, portanto, uma cobertura maior dos caminhos e, consequentemente, das transições presentes no modelo.

Com relação aos resultados obtidos pela técnica proposta neste trabalho, é possível observar um aumento na cobertura obtida por Sapna e Mohanty. A quantidade de casos de teste no conjunto aumentou para 24, e analisando cada conjunto é possível observar que, além de cobrir 4 novos caminhos, estes 24 casos de teste cobrem vários cenários cobertos por Sapna e Mohanty, assim como aumenta a cobertura de transições em alguns destes caminhos gerados.

Mais detalhes acerca da comparação entre os conjuntos gerados pela nossa técnica e pelo algoritmo de Sapna e Mohanty pode ser observado na Figura 9. A Figura 9 (a) mos-

tra que as duas técnicas cobrem caminhos semelhantes, porém, em 12 destes caminhos nossa técnica cobre uma quantidade maior de transições que a técnica proposta por Sapna e Mohanty.

Por meio da Figura 9 (b), verificamos que ambas as técnicas cobrem 7 caminhos iguais. Portanto, através do estudo de caso, é possível concluir que o conjunto gerado pela nossa técnica apresenta uma cobertura maior das transições e caminhos do modelo quando comparada com os algoritmos analisados.

(a)

S	T	Caminho Comum	Transição Coberta por S	Transição Coberta por T
S ₁₅	T ₂₄	d z g z		g
S ₁₆	T ₂₁	d z g h i		g
S ₁₄	T ₂₃	d z g e f		g
S ₁₁	T ₂₀	d e f g z		g
S ₁₂	T ₁₇	d e f g h i		g
S ₁₀	T ₁₉	d e f g e f		g
S ₁₈	T ₁₄	d h i g e f		g
S ₁₅	T ₁₉	d h i g		z
S ₂₀	T ₁₂	d h i g h i		g
S ₃	T ₆	a b f g e f g h		i
S ₇	T ₂	a b f g h i g h		i
S ₅	T ₉	a b f g z g h		i
S ₄	T ₁₀	a b f g z g e	c	

(b)

S	T	Caminho Comum
S ₁	T ₁	a b c
S ₈	T ₁₆	d e c
S ₂	T ₅	a b f g e c
S ₉	T ₁₈	d e f g e c
S ₁₃	T ₂₂	d z g e c
S ₁₇	T ₁₃	d h i g e c
S ₆	T ₃	a b f g h i g e

(c)

T	Caminho
T ₄	a b f g h i g z
T ₇	a b f g e f g e
T ₈	a b f g e f g z
T ₁₁	a b f g z g z

Figura 9. Semelhanças entre os conjuntos gerados pelas técnicas.

6. Conclusões e Trabalhos Futuros

Este trabalho apresenta uma técnica para a geração automática de casos de teste a partir de um modelo. A principal característica desta técnica é a capacidade de parametrizar a cobertura de *loops* presentes no modelo. A partir do modelo, é gerada uma árvore capaz de expandir seus ramos por meio de subárvores que representam os caminhos repetidos no modelo (i.e. os *loops* no modelo).

Considerando que a partir do modelo é gerada uma árvore, diversos benefícios são obtidos, uma vez que diversos trabalhos na literatura já investigaram propriedades das árvores, como por exemplo, o custo logarítmico de busca em árvores binárias. Diante disto, essas propriedades podem ser utilizadas para analisar o modelo de software por meio da árvore obtida pela técnica e obter conclusões acerca de cobertura, estrutura, dentre outros aspectos do modelo.

Um dos problemas observados é a quantidade de casos de teste redundantes que podem ser gerados através da técnica. À medida que a árvore vai sendo expandida (por meio da união de várias subárvores), a quantidade de casos de teste tende a crescer bastante, e, como estas subárvores representam a repetição dos caminhos, tanto a cobertura

como a redundância no conjunto gerado também tende a crescer. Apesar disto, podem ser utilizadas técnicas de seleção automática de casos de teste (e.g. estratégias baseadas em similaridade entre os casos de teste) para automaticamente reduzir esta redundância e manter a cobertura parametrizada dos *loops*.

Dentre os trabalhos futuros, está a análise das subárvores geradas pela técnica. É possível que, por meio das subárvores, conjuntos de casos de testes possam ser gerados. Dessa forma, o conjunto final de casos de teste seria composto pela união destes subconjuntos, permitindo um controle maior durante a expansão da árvore, evitando uma explosão na quantidade de casos de teste gerados.

Agradecimentos. Este trabalho é apoiado pelo CNPq, processos 484643/2011-8 e 560014/2010-4, pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES¹), financiado pelo CNPq, processo 573964/2008-4. A segunda autora é beneficiária de auxílio financeiro da CAPES – Brasil.

Referências

- Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.
- Cartaxo, E. G., Neto, F. G. O., and Machado, P. D. L. (2007). Test case generation by means of uml sequence diagrams and labeled transition systems. In *SMC*, pages 1292–1297.
- Hartman, A. and Nagin, K. (2004). The agedis tools for model based testing. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA ’04*, pages 129–132, New York, NY, USA. ACM.
- Machado, P. and Sampaio, A. (2010). Automatic test-case generation. In Borba, P., Cavalcanti, A., Sampaio, A., and Woodcock, J., editors, *Testing Techniques in Software Engineering*, volume 6153 of *LNCS*, pages 59–103. Springer.
- Pretschner, A. (2005). Model-based testing. In *Proceedings of International Conference on Software Engineering - ICSE*, pages 722–723.
- Santiago, V., do Amaral, A. S. M., Vijaykumar, N. L., Mattiello-Francisco, M. d. F., Martins, E., and Lopes, O. C. (2006). A practical approach for automated test case generation using statecharts. In *Proceedings of COMPSAC ’06*, pages 183–188, Washington, DC, USA. IEEE Computer Society.
- Sapna, P. G. and Mohanty, H. (2009). Prioritization of scenarios based on uml activity diagrams. In *CICSyN*, pages 271–276.
- Simão, A., Petrenko, A., and Yevtushenko, N. (2009). Generating reduced tests for fsm's with extra states. In *Proceedings of the 21st International Conference on Testing of Software and Communication Systems, TESTCOM ’09/FATES ’09*, pages 129–145. Springer-Verlag.
- Uutting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

¹www.ines.org.br

Geração Aleatória de Dados para Programas Orientados a Objetos

Fernando H. Ferreira¹, Márcio E. Delamaro², Marcos L. Chaim¹,
Fátima N. Marques¹, Auri M. R. Vincenzi³

¹Escola de Artes, Ciências e Humanidades (EACH)
Universidade de São Paulo (USP) – São Paulo, SP – Brazil

²Instituto de Ciência Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – São Carlos, SP – Brazil

³Instituto de Informática (INF)
Universidade Federal de Goiás (UFG) – Goiânia, GO – Brazil

{fer.henrique, chaim, fatima.nunes}@usp.br, delamaro@icmc.usp.br, auri@inf.ufg.br

Abstract. *The random generation of test individuals is a widely used for test data generation techniques such as Simulated Annealing, Hill Climbing and Evolutionary Algorithms. This article describes a new technique for generating random test individuals for programs written in object-oriented languages. Differently of the previous ones, the proposed technique allows the generation of random values for primitive types and objects, generation of values for arrays, and targeted random generation of values. Initial validations show that the technique can generate test data at limited cost. The technique presented is part of a framework for test data generation currently in development.*

Resumo. *A geração aleatória de indivíduos de teste é muito utilizada por técnicas de geração de dados de teste como Têmpera Simulada, Subida de Encosta e Algoritmos Evolucionários. Este artigo descreve uma nova técnica de geração aleatória de indivíduos de teste para programas escritos em linguagens orientadas a objetos. Diferentemente das anteriores, a técnica proposta permite a geração aleatória de valores para tipos primitivos e objetos, geração de valores para vetores e uso de geração aleatória dirigida de valores. Validações iniciais mostram que os custos para gerar dados de teste são limitados. A técnica descrita é parte de um arcabouço para geração de dados de teste.*

1. Introdução

Uma das maneiras de aumentar a qualidade do software é por meio da atividade de teste. Porém, essa atividade por ser custosa e consumir muito tempo. A geração automática de dados de teste é uma tarefa essencial para o teste de software, pois a automatização permite a redução do custo de desenvolvimento e o aumento da qualidade do software (Sagarna et al. 2007; Silva e van Someren 2010). Existem diferentes técnicas de geração de dados de teste. Muitas dessas soluções para geração de dados de teste utilizam a geração aleatória de dados como um passo essencial. Os Algoritmos Evolucionários (Tonella 2004) utilizam a geração aleatória de dados como passo na geração de uma população inicial de elementos. Os algoritmos de Subida de Encosta (McMinn 2004) e

Têmpera Simulada (Bertsimas e Tsitsiklis 1993; Barros e Tedesco 2008; Pinheiro 2010) também fazem uso da geração aleatória para criação de suas populações de indivíduos durante suas execuções.

Normalmente, a geração aleatória de dados de teste é feita para parâmetros primitivos (i.e., int, double, float, char, boolean). No entanto, para que a geração automática seja possível de ser utilizada em programas produzidos pelas empresas e indústrias de software, é necessário que sejam gerados dados não somente para tipos primitivos, mas para estruturas mais complexas, em especial no contexto de orientação a objetos.

A geração automática de objetos no teste de programas orientados a objetos é difícil, pois os objetos são alocados dinamicamente. Desta forma, a utilização de conceitos como herança, polimorfismo e classes abstratas torna impossível determinar em tempo de compilação qual código será executado (Edvardsson 1999). Ao contrário do software procedural, o software orientado a objetos nem sempre se comporta da mesma maneira quando o mesmo método é executado com os mesmos valores de entrada, pois uma das maiores peculiaridades do software orientado a objetos é o controle de estado dos objetos. O estado de um objeto caracteriza sua composição em um dado momento de tempo e pode modificar o comportamento de seus métodos (Tonella 2004).

Uma solução proposta para esta questão é a seleção aleatória de métodos para serem executados entre a criação da instância da classe sob teste e a chamada ao método sob teste. A utilização desses métodos intermediários objetiva estimular mudanças no estado do objeto para que diferentes estados, que possam interferir na execução do método sob teste, sejam descobertos (Tonella 2004; Silva e van Someren 2010).

Esse artigo apresenta uma técnica de geração de dados aleatórios para programas escritos em linguagens orientadas a objetos. Essa técnica permite gerar dados para: vetores, instâncias de objetos e tipos primitivos. Além disso, sua implementação, descrita neste trabalho, gera os dados de teste em dois formatos: classes de teste no formato JUnit (JUnit 2012) e representação concisa de indivíduos definida por Tonella (2004). A técnica apresentada é parte integrante de um arcabouço de geração de dados de teste para programas orientados a objetos. Este arcabouço provê diferentes técnicas de geração de dados de teste, além de recursos que permitem a sua extensão para o desenvolvimento de novas técnicas de geração de dados de teste.

Na próxima seção são discutidos conceitos básicos sobre a geração aleatória de dados de teste. Na Seção 3 é apresentada a descrição da técnica explicando seu funcionamento. Em seguida, a Seção 4 apresenta os métodos de validação do arcabouço. Os trabalhos relacionados são discutidos na Seção 5. Por fim, a Seção 6 contém as conclusões.

2. Conceitos Básicos

Nessa seção são apresentados conceitos sobre a geração aleatória de indivíduos de teste e a abordagem adotada para representar os indivíduos de teste.

2.1. Geração aleatória de indivíduos de teste

Os indivíduos de teste representam as entradas, isto é, os dados de teste, para testar um dado método. A técnica de geração aleatória de indivíduos de teste é a técnica

mais simples, pois sua utilização não exige a análise de representações do sistema (e.g. código-fonte). Em sistemas muito complexos ou programas que possuam um conjunto de condições complexas, este método pode ser uma má escolha, pois a probabilidade de selecionar uma entrada adequada dentro de um conjunto gerado de forma aleatória é baixa. Outro problema é que ao longo de sua execução, conjuntos de valores que exercitam o mesmo comportamento são gerados. Este cenário não é adequado, pois torna boa parte dos resultados redundantes (Edvardsson 1999; Sen et al. 2005; Burnim e Sen 2008; Delamaro et al. 2010).

Segundo Pacheco et al. (2008), a eficiência do teste aleatório é uma questão não resolvida dentro da comunidade de teste, pois alguns estudos sugerem que o teste aleatório não é tão efetivo quanto as demais técnicas de geração de dados de teste, mas outros trabalhos afirmam que o teste aleatório, devido a sua velocidade e escalabilidade, é uma técnica eficiente.

2.2. Representação de indivíduos de teste

Tonella (2004) propõe uma representação de indivíduos de teste para o teste evolucionário de software orientado a objetos. Essa representação especifica uma estrutura que agrupa sequências de comandos, criação de objetos, mudanças de estados e chamadas a métodos. Na representação de Tonella um indivíduo é dividido em duas partes, separadas pelo caractere “@”(arroba). A primeira parte contém uma sequência de ações (i.e., construtores e métodos), separadas pelo caractere “:”(dois pontos). Cada ação pode conter um novo objeto, atribuído a uma variável do indivíduo, indicada como “\$id”. A segunda parte contém os valores de entrada dos métodos para serem usados nas suas invocações. Essa segunda parte é exclusiva para representação de valores de tipos primitivos ou vetores de tipos primitivos, separados pelo caracter “,”(vírgula).

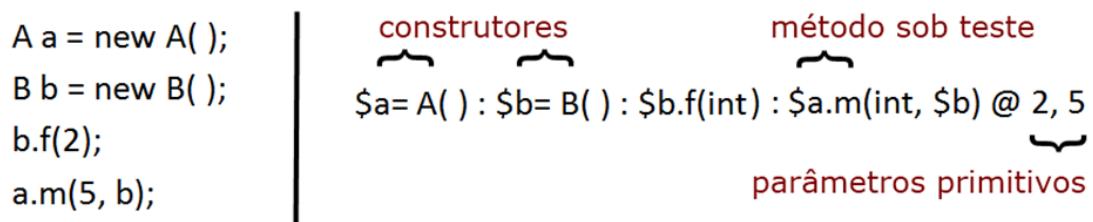


Figure 1. Representação de um indivíduo de teste.

A Figura 1 apresenta o modo como a representação de Tonella é aplicada. Do lado esquerdo pode-se observar um bloco de código e do lado direito sua representação utilizando a abordagem de Tonella. Pode-se notar que os valores inteiros utilizados como parâmetros para os métodos são posicionados do lado direito do símbolo de “@”(arroba), enquanto que as chamadas a métodos são posicionadas do lado esquerdo. Vale ressaltar a sintaxe utilizada para representar a construção de instâncias de objetos e as chamadas a métodos. No caso, a instrução “A a = new A();” foi escrita na representação de Tonella com a sintaxe “\$a = A()”, assim como a sintaxe de chamada de método “b.f(2);” foi representada com a sintaxe “\$b.f(int)”.

3. Descrição da técnica

A técnica proposta analisa *bytecodes* Java, de forma que nenhum código fonte é necessário para que sejam executadas suas funções. O arquivo *bytecode* é uma representação binária que contém informações sobre uma classe, tais como: seu nome, o nome de sua superclasse, informações sobre os métodos, variáveis e constantes utilizadas, além das instruções de cada um de seus métodos. A análise funciona desta forma, pois o presente trabalho visa à geração de dados de teste para a ferramenta de teste estrutural JaBUTi, que também trabalha analisando *bytecodes* Java (Vincenzi et al. 2003; Delamaro et al. 2007). Esta seção descreve todos os passos contidos nessa técnica de geração aleatória de dados de teste.

3.1. Análise do código do programa

Para geração de dados para um método é preciso fazer a leitura das assinaturas dos métodos e das dependências da classe sob teste. Os demais métodos são importantes, pois eles serão utilizados para gerar diferentes estados do objeto sob teste, e os tipos dependentes são importantes para que seja possível saber de quais tipos primitivos ou objetos deve-se criar instâncias durante o processo de geração de dados de teste. Este processo de identificação de métodos e tipos de dados dependentes é recursivo, pois sempre que identificada a dependência a um tipo de dados e este tipo de dados é um objeto, então, é preciso identificar seus métodos e suas dependências a outros tipos de dados. Desta forma, ganha-se desempenho durante a geração aleatória de dados, pois já se tem todos os tipos de dados dependentes mapeados e salvos em memória, não exigindo a releitura do *bytecode*, agregando desempenho à execução da técnica.

3.2. Geração aleatória de indivíduos

A geração aleatória de indivíduos é dividida em dois passos, sendo eles: criação aleatória de um indivíduo de teste e geração aleatória de valores.

Segundo Tonella (2004), um indivíduo de teste corresponde a uma sequência de um construtor e invocações de métodos, seguidos dos valores de seus parâmetros. A criação aleatória de um indivíduo parte da criação de uma instância da classe sob teste, por meio da seleção aleatória de um de seus construtores. Dado um conjunto de construtores identificados durante a leitura do *bytecode*, um construtor é selecionado aleatoriamente e atribuído ao objeto. Após este passo é preciso selecionar métodos intermediários que estimulem um estado para o indivíduo de teste. A escolha de métodos intermediários é feita de forma parecida com a seleção de construtores. Dado um conjunto de métodos identificados durante a leitura do *bytecode*, é escolhida uma quantidade aleatória N de métodos intermediários que o indivíduo deve conter, então são selecionados N métodos aleatoriamente e adicionados ao indivíduo sob teste. E como parte final da construção da estrutura, é adicionado ao indivíduo o método sob teste.

Sempre que um construtor ou método é adicionado à estrutura do indivíduo é feita a leitura dos parâmetros esperados por estes elementos. Tal leitura procura identificar o tipo de dados e a multiplicidade do parâmetro. Caso o tipo de dados do parâmetro seja um objeto, então o mesmo processo de seleção de construtor e de métodos intermediários é executado para o tipo de dados do parâmetro lido. Este passo é importante para que

o parâmetro que depende de um objeto também tenha uma instância construída a partir de um construtor válido e seu estado construído por meio da execução de métodos intermediários.

O segundo passo da geração aleatória de indivíduos corresponde à geração aleatória de valores. Depois de construído o indivíduo de teste, sua estrutura (construtor e chamadas a métodos) já é conhecida, além de suas dependências a outros objetos. Desta forma, a geração aleatória tem apenas de percorrer a estrutura do indivíduo (e de suas respectivas dependências) identificando parâmetros primitivos, e gerando valores aleatórios para eles. A geração aleatória de valores primitivos é mais simples se comparada à geração aleatória de objetos (discutida no passo anterior), pois os tipos primitivos pertencem a grupo limitado de tipos, sendo esses tipos: boolean, char, short, byte, int, long, float e double.

A geração aleatória de instâncias de objetos é mais complexa, pois existem desafios como: geração de valores para tipos de dados dependentes, dependência a outros objetos, criação de instâncias de classes abstratas e recursividade durante a geração de tipos aleatórios. A geração de valores para tipos de dados dependentes corresponde à geração aleatória de instância de objetos para dependências da classe sob teste. Essa geração é trabalhosa e pode acarretar na seleção recursiva de tipos de dados, que ocorre quando um tipo de dados depende de uma instância de objeto do mesmo tipo da sua, fazendo com que a técnica gere recursivamente inúmeras instâncias de objeto do mesmo tipo, caindo em um laço infinito.

Este problema, assim como a dependência entre as classes, pode acarretar em um alto grau de profundidade, isto é, acaba-se criando uma árvore de dependência de objetos muito grande, tornando o indivíduo de teste muito complexo. Para solucionar essa questão foi utilizado um limiar de profundidade, que quando ultrapassado limita a criação de novas instâncias de objetos. Esse controle de limitações é feito durante a seleção de métodos intermediários dos objetos, no qual são filtrados apenas os métodos que dependem apenas de tipos primitivos. Desta maneira, limita-se a geração de objetos, já que novos métodos intermediários que dependem apenas de tipos primitivos não irão exigir a criação de novas instâncias de objetos.

3.3. Geração dirigida de valores aleatórios

A técnica permite a passagem de parâmetros que direcionem a geração aleatória de valores dos tipos primitivos. Tais parâmetros modificam o domínio de geração dos indivíduos e direcionam o domínio de valores a um intervalo específico de valores. Inicialmente, sem a passagem de parâmetros, a geração aleatória de valores segue o mesmo domínio proposto por Tonella (2004), conforme Tabela 1.

Table 1. Domínio padrão de valores para geração aleatória de indivíduos.

Tipo de Dados	Domínio
Inteiros e números reais	De 0 a 100
Boleanos	Valores boleanos verdadeiro e falso
Strings	Caracteres alfanuméricos ([a-zA-Z0-9])

A geração aleatória dirigida é ideal em cenários nos quais se tem conhecimento

dos possíveis domínios de entrada esperados pelo método sob teste. Desse modo, a geração de valores torna-se mais precisa, pois define um domínio de possíveis valores, tornando a geração aleatória de valores menos abrangente e mais precisa.

Um exemplo relevante, para geração dirigida de valores aleatórios, é a necessidade de testar um método que receba como parâmetro um caminho para um arquivo. A geração aleatória provê apenas a geração de valores aleatórios, cujo conteúdo não faz parte do esperado. Neste caso, a geração aleatória não fornece valores correspondentes ao domínio esperado, desta maneira é indicada a utilização da geração aleatória dirigida, fornecendo valores que façam parte do domínio esperado, no caso do exemplo, um conjunto de caminhos válidos para arquivos.

3.4. Geração de valores para vetores e matrizes

A geração de valores para vetores e matrizes é sempre ativada quando for identificado um parâmetro que é um vetor ou uma matriz. Existem dois tipos de vetores: vetores para tipos primitivos e vetores para objetos. A geração de dados para vetores segue a mesma lógica, sendo ele para tipos primitivos ou objetos. A geração dos valores aleatórios para vetores se inicia pela identificação do tipo de dados base do vetor. Após essa identificação, um valor inteiro aleatório é gerado para definir o tamanho do vetor a ser gerado. Com isso, criam-se valores para preencher o vetor seguindo as mesmas regras propostas para geração de parâmetros primitivos ou objetos.

3.5. Formatos para exportação dos dados gerados

Para exportação dos dados gerados pode-se escolher entre dois formatos de saída, pela representação de indivíduos de teste proposta por Tonella ou por meio da geração de classes de teste no formato JUnit.

4. Validação da Técnica

Para exemplificar a utilização da técnica foi construída uma estrutura de classes, ilustrada na Figura 2, que fornece construtores e métodos que exigem a passagem de parâmetros. Tais métodos fazem uso de parâmetros de tipos primitivos e estruturados.

Esta estrutura possui quatro classes (Carro, Motor, Radiador e Cambio), estas classes dependem umas das outras e são utilizadas como parâmetros dos métodos, o que obriga a criação dinâmica de instâncias de objetos para atender a passagem de parâmetros.

A geração aleatória de indivíduos mostrou-se eficaz, gerando valores para os tipos primitivos, tipos complexos (objetos) e vetores. A seguir, um indivíduo gerado pela técnica:

```
$carro=Carro(String,int):$carro.acelerar(float)@"AAA-5555",2008,31.197704F
```

Pode-se observar que este indivíduo é simples, não exige a criação dinâmica de nenhum objeto ou utilização de vetores, possui apenas a criação da classe sob teste, a chamada do método sob teste (no caso o método “acelerar(float)” da classe “Carro”) e a passagem de parâmetros primitivos. De forma complementar, a Figura 3 apresenta o mesmo indivíduo de teste representado no formato JUnit.

No exemplo a seguir, pode-se observar um indivíduo mais complexo do que o anterior, exigindo a construção dinâmica de objetos e utilização de vetores:

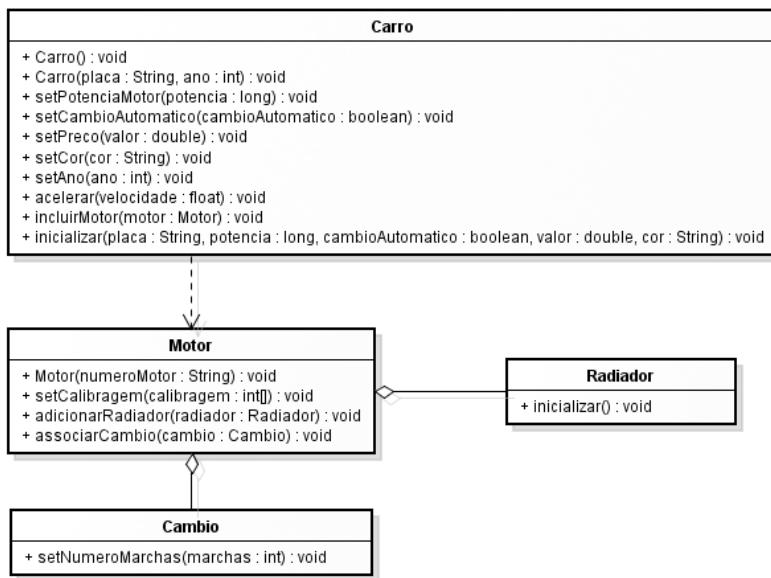


Figure 2. Modelo de entidades utilizado pela técnica.

```

@Test
public void testMethod(){
    Carro carro= new Carro("AAA-5555",2008);
    carro.acelerar(31.197704F);
}

```

Figure 3. Teste unitário de um indivíduo simples gerados pela técnica proposta.

```

$carro=Carro():$carro.toString():$motor=Motor(String):$motor.setCalibragem(int[])
$radiador=Radiador():$radiador.inicializar():$motor.adicionarRadiador($radiador):
$carro.incluirMotor($motor):$carro.setCambioAutomatico(boolean):$carro.hashCode():
$carro.acelerar(float)@”AAA-5555”,[2011,2010,2008,2006,2006],false,151.0165F

```

Ao lado esquerdo do símbolo de arroba se vê o construtor utilizado para criação do método sob teste, a sequência de métodos intermediários e a chamada do método sob teste, e, do lado direito, os valores utilizados nos parâmetros primitivos.

Na Figura 4 pode-se observar o método de teste JUnit criado a partir do indivíduo gerado. Nota-se que a sequência de chamadas aos métodos intermediários continua sendo a mesma, a única mudança é a forma de representação do indivíduo de teste.

A técnica apresentou custo reduzido para geração dos indivíduos de teste. A Tabela 2 apresenta a quantidade de indivíduos gerados e o tempo necessário para criá-los.

Para realização dos experimentos foi utilizado um computador com sistema operacional Microsoft Windows 7/64 bits, processador Intel i7 2.50GHz, 4Gb RAM e Java 1.6.

5. Trabalhos Relacionados

Muitas ferramentas são criadas para geração aleatória de dados de teste. A maioria destas ferramentas apenas geram dados para tipos primitivos, suportando apenas aplicações es-

```

@Test
public void testMethod(){
    Carro carro= new Carro();
    carro.toString();
    Motor motor= new Motor("AAA-5555");
    motor.setCalibragem(new int[]{2011,2010,2008,2006});
    Radiador radiador= new Radiador();
    radiador.inicializar();
    motor.adicionarRadiador(radiador);
    carro.incluirMotor(motor);
    carro.setCambioAutomatico(false);
    carro.hashCode();
    carro.acelerar(151.01656F);
}

```

Figure 4. Teste unitário de um dos indivíduos gerados pela técnica proposta.

Table 2. Quantidade de indivíduos gerados e o tempo de execução necessário para criá-los.

Indivíduos	Tempo (segundos)
100	0,109
1.000	0,235
10.000	0,766
100.000	5,86

critas em linguagens procedimentais e parte das aplicações que utilizam linguagens orientadas a objetos.

Uma ferramenta relevante de teste aleatório de software é o Randoop (Pacheco e Ernst 2007; Pacheco et al. 2008). Randoop (*Random Tester for Object Oriented Programs*) utiliza *Feedback-Directed Random Testing*, técnica de geração aleatória de dados de teste que gera um conjunto de teste para descoberta de falhas em programas orientados a objetos. Seu algoritmo cria sequências de chamadas a métodos e, com base no resultado de suas execuções, identifica as entradas reveladoras de falhas. A técnica proposta também cria sequências de chamadas a métodos para estimular diferentes estados nos objetos, mas ao invés de utilizar Feedback-Directed Random Testing utiliza-se geração aleatória dirigida de valores.

TestFul (Miraz et al. 2009; Baresi e Miraz 2010) é uma outra proposta de geração de dados de teste que adota a ideia de que testes devem colocar os objetos em seus devidos estados e fornecer os valores para os parâmetros de entradas das funções sob teste. TestFul faz uma pré-análise da classe sob teste, com o objetivo de identificar todas as classes que devem ser envolvidas no teste. Isso é feito considerando todos os parâmetros de todos os construtores e métodos públicos contidos na classe sob teste. Após esta pré-análise, o algoritmo de TestFul gera aleatoriamente uma população inicial para então utilizar um algoritmo de Subida de Encosta para realizar pequenas modificações nos indivíduos, a fim de melhorar a aptidão de sua população. A técnica proposta baseia-se na ideia de TestFul em realizar a pré-análise da classe sob teste, a fim de identificar todas as dependências envolvidas no teste e tornar-se mais eficiente.

Silva e Someren (2010) apresentam uma abordagem que combina a utilização

de algoritmos genéticos com análise estática, para o teste automatizado de classes, na linguagem *Eiffel*. A análise estática utilizada procura, dentro da classe sob teste, a ocorrência de valores primitivos, sendo que tais valores são aproveitados para dirigir a geração aleatória da população inicial utilizada pelo algoritmo evolucionário. O trabalho de Silva e Someren também faz uso da geração aleatória dirigida, com uma diferença: seus valores não são parametrizáveis, eles são definidos por meio de valores primitivos encontrados dentro da classe sob teste.

Esse trabalho compartilha características com os trabalhos relacionados e provê recursos que demais trabalhos da literatura não fornecem. A geração dinâmica de instâncias de objetos não é um recurso compartilhado por todas as técnicas de geração de dados de teste para programas escritos em linguagens orientadas a objetos. A técnica de geração de dados de teste proposta neste trabalho não somente cria instâncias dinâmicas de objetos como também estimula a construção de diferentes estados para os objetos por meio da chamada de métodos intermediários. Essa técnica de geração aleatória de dados de teste também difere das demais ao gerar dados aleatórios para vetores, recurso que não foi encontrado nas demais técnicas. Mais um recurso que sobressai das demais técnicas é a possibilidade de parametrizar valores para geração aleatória dirigida, que ao contrário de trabalhos como o de Silva e van Someren (2010), não é obrigatório e permite a interferência do testador. Outra diferença aos demais trabalhos é a possibilidade de exportar os indivíduos gerados em dois formatos distintos, sendo eles: a representação proposta por Tonella (2004) e classes de teste no formato JUnit.

6. Conclusões

O uso de técnicas de geração aleatória de indivíduos é comum. Outras técnicas de geração de dados de teste - como Têmpera Simulada, Subida de Encosta e Algoritmos Evolucionários - também fazem uso desta técnica durante o seu processamento.

A técnica proposta neste documento provê recursos para geração aleatória de indivíduos para programas escritos em linguagens orientadas a objetos, apoiando a geração de valores para tipos primitivos, objetos e vetores. Diferente das outras abordagens, a técnica aqui proposta também permite exportar os resultados em dois formatos, sendo eles: uma representação de indivíduos descrita na literatura e classes de teste no formato JUnit.

A técnica apresentou custo reduzido na geração de indivíduos de teste, gerando mil indivíduos em 0,235 segundo (235 milésimos de segundo) e 100.000 (cem mil) indivíduos em 5,85 segundos. Essa técnica de geração aleatória de dados de teste faz parte de um arcabouço de geração de dados de teste e funciona como base para geração de valores para as demais técnicas presentes no arcabouço. Esse arcabouço fornece diferentes técnicas de geração de dados de teste, além de uma estrutura extensível, própria para criação de novas técnicas de geração de dados de teste.

References

- [Baresi e Miraz 2010] Baresi, L. e Miraz, M. (2010). Testful: Automatic unit-test generation for java classes. volume 2, pages 281–284. ACM - Association for Computing Machinery.

- [Barros e Tedesco 2008] Barros, F. e Tedesco, P. (2008). Introdução aos agentes inteligentes - algoritmos de melhorias iterativas (otimização). *Universidade Federal de Pernambuco, Centro de Informática*.
- [Bertsimas e Tsitsiklis 1993] Bertsimas, D. e Tsitsiklis, J. (1993). Simulated annealing. *Statistical Science*, 8(1):10–15.
- [Burnim e Sen 2008] Burnim, J. e Sen, K. (2008). Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, California, Berkeley.
- [Delamaro et al. 2010] Delamaro, M., Chaim, M. L., e Vincenzi, A. M. R. (2010). *Atualizações em Informática 2010*, volume 1, chapter Técnicas e Ferramentas de Teste de Software, pages 51–110. Editora PUC-Rio, Rio de Janeiro.
- [Delamaro et al. 2007] Delamaro, M., Maldonado, J., e Jino, M. (2007). *Introdução ao Teste de Software*, volume 1. Editora Campus.
- [Edvardsson 1999] Edvardsson, J. (1999). A survey on automatic test data generation.
- [JUnit 2012] JUnit (2012). Junit.
- [McMinn 2004] McMinn, P. (2004). *Search-based Software test Data Generation - A survey*, volume 14. Software Testing, Verification and Reliability.
- [Miraz et al. 2009] Miraz, M., Lanzi, P. L., e Baresi, L. (2009). Testful: Using a hybrid evolutionary algorithm for testing stateful systems. *Genetic and Evolutionary Computation Conference*, pages 1947–1948.
- [Pacheco e Ernst 2007] Pacheco, C. e Ernst, M. (2007). *Randoop: feedback-directed random testing for Java*. Object-Oriented Programming, Systems, Languages & Applications, Montreal, Canada.
- [Pacheco et al. 2008] Pacheco, C., Lahiri, S., e Ball, T. (2008). *Finding Errors in .Net with Feedback-Directed Random Testing*. International Symposium on Software Testing and Analysis, Seattle, WA, USA.
- [Pinheiro 2010] Pinheiro, A. C. (2010). Geração automática de dados de teste: Visão geral. Apresentado na disciplina "SSC5877 Verificação, Validação e Teste de Software".
- [Sagarna et al. 2007] Sagarna, R., Arcuri, A., e Yao, X. (2007). Estimation of distribution algorithms for testing object oriented software. pages 438 – 444, Birmingham, Reino Unido. Centre of Excellence for Research in Computational Intelligence and Applications, School of Computer Science, University of Birmingham, IEEE Congress on Evolutionary Computation.
- [Sen et al. 2005] Sen, K., Marinov, D., e Agha, G. (2005). Cute: A concolic unit testing engine for c. Technical Report 5, Association for Computing Machinery - Software Engineering Notes, New York, NY, USA.
- [Silva e van Someren 2010] Silva, L. S. e van Someren, M. (2010). Evolutionary testing of object-oriented software. *Association for Computing Machinery - Symposium on Applied Computing*, pages 1126–1130.
- [Tonella 2004] Tonella, P. (2004). *Evolutionary Testing of Classes*. ITC-irst Centro per la Ricerca Scientifica e Tecnologica, Italia; Povo; Trento.
- [Vincenzi et al. 2003] Vincenzi, A. M. R., Delamaro, M., e Maldonado, C. (2003). *Java Bytecode - Understanding and Testing - User's Guide*. Universidade de São Paulo - Instituto de Ciências Matemáticas e de Computação.



PROMOÇÃO:



Sociedade Brasileira
de Computação

REALIZAÇÃO:



PATROCINADORES:



Conselho Nacional de Desenvolvimento
Científico e Tecnológico



EMPRESA DE TECNOLOGIA E INFORMAÇÕES
DA PREVIDÊNCIA SOCIAL - DATAPREV

APOIO:



INFORMAÇÕES:

<http://cbsoft.dimap.ufrn.br/>

ISSN 2178-6097

