ANAIS PROCEEDINGS

CBSOFT 2015

BRAZILIAN CONFERENCE ON SOFTWARE: THEORY AND PRACTICE

BELO HORIZONTE

SAST 2015

9th BRAZILIAN WORKSHOP ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING

CBSOFT.ORG

Sponsors:

Promotion:

Organizing Institutions:

SAST 2015

9th Brazilian Workshop on Systematic and Automated Software Testing – SAST 2015

September 23rd, 2015
Belo Horizonte – MG, Brazil

ANAIS | *PROCEEDINGS*

COORDENADORES DO COMITÊ DE PROGRAMA DO SAST 2015 | *PROGRAM COMMITTEE CHAIR OF SAST 2015*
Simone do Rocio Senger de Souza (ICMC-USP)
Auri Marcelo Rizzo Vincenzi (DC-UFSCar)

REALIZAÇÃO | *ORGANIZATION*
Universidade Federal de Minas Gerais (UFMG)
Pontíficia Universidade Católica de Minas Gerais (PUC-MG)
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

PROMOÇÃO |*PROMOTION*
Sociedade Brasileira de Computação | Brazilian Computing Society

APOIO | *SPONSORS*
CAPES, CNPq, FAPEMIG, Google, RaroLabs, Take.net,
ThoughtWorks, AvenueCode, AvantiNegócios e Tecnologia.

APRESENTAÇÃO

Teste é a principal atividade de garantia de qualidade de software em vários domínios de aplicação e o seu principal propósito é detectar defeitos no produto ou no processo de desenvolvimento. O panorama atual mostra que estamos cada vez mais dependentes de softwares em atividades cotidianas e em ambientes industriais, emergindo uma necessidade crescente de respostas rápidas e corretas desses sistemas, os quais muitas vezes são críticos, ubíquos, persistentes e adaptativos. Essa necessidade impulsiona o desenvolvimento de técnicas, critérios e ferramentas de apoio à atividade de teste considerando os diferentes domínios de aplicação e paradigmas de programação. Nessa direção, observam-se importantes iniciativas que buscam promover um melhor relacionamento indústria-academia. Abordagens sistemáticas de teste demonstram serem mais efetivas que abordagens *ad-hoc*, permitindo a automatização de atividades repetitivas e melhorando o desempenho e a efetividade da atividade de teste. Além disso, vários estudos de caso e estudos experimentais são reportados, apresentando resultados relevantes que impulsionam pesquisas e transferência de tecnologia para a indústria. Apesar disso, ainda existem desafios para a transferência de tecnologia entre academia e indústria.

O SAST – *Brazilian Workshop on Systematic and Automated Software Testing* (Workshop Brasileiro de Teste de Software Sistemático e Automatizado) é o principal evento científico sobre teste de software no Brasil. Desde 2007 este workshop tem como objetivo promover um fórum anual para discutir questões sobre a sistematização e automação da atividade de teste de software, promovendo a interação entre pesquisadores e indústria de modo a fortalecer cooperações e inovação nessa importante área do desenvolvimento de software.

Nessa 9ª edição, o SAST 2015 ocorre em conjunto com o CBSoft – Congresso Brasileiro de Software: Teoria e Prática, na cidade de Belo Horizonte – MG e é organizado em conjunto pela Universidade de São Paulo (ICMC/USP) – São Carlos e Universidade Federal de São Carlos (UFSCar) – São Carlos.

Neste ano, o programa técnico do SAST compreende a palestra: "***Do we test our mobile applications as well as we use them?",*** proferida pelo ***Prof. Dr.*** **Arilo Claudio Dias-Neto**(UFAM). O tema da palestra aborda um tema relevante e atual. Aplicações móveis estão transformando a sociedade e, portanto, saber como testar essas aplicações é tão importante quando saber usa-las adequadamente. O workshop compreende também três sessões técnicas para que os autores apresentem os oito artigos selecionados, os quais possuem contribuições importantes para a área de teste de software.

O SAST 2015 recebeu a submissão de 20 artigos técnicos. Esses artigos foram cuidadosamente revisados por pelo menos três membros do comitê de programa, contando com a colaboração de revisores nacionais e internacionais. Após a revisão, oito

artigos foram selecionados sendo a taxa de aceitação igual a 40%, mantendo desta forma a média de aceitação das edições anteriores do evento (2007 a 2014).

Este ano os workshops do CBSoft2015 conseguiram efetivar parceria com periódicos qualificados e os melhores artigos dos workshops serão convidados a enviar uma versão estendida de seus artigos para o ***Special Issues of the Workshops of CBSoft'2015***em um dos seguintes periódicos *Journal of the Brazilian Computer Society - JBCS* ou *Journal of Internet Services and Applications – JISA*.

Agradecemos imensamente a todos os membros do comitê de programa e aos eventuais revisores por eles elencados, que colaboraram com as revisões, contribuindo com a melhoria de qualidade dos artigos avaliados. Sabemos que sem a colaboração de vocês não teríamos conseguido concluir o processo de revisão em tão pouco tempo.

Agradecemos ao aluno de doutorado Victor Hugo Santiago Costa Pinto (ICMC/USP) pela criação do novo logo do SAST e pelo apoio com a criação da página do SAST’2015.

Agradecemos aos organizadores do CBSoft 2015 pela oportunidade e infraestrutura disponibilizada e a CAPES pelo auxílio concedido para organização do evento.

Finalmente, gostaríamos de agradecer aos convidados, aos autores dos artigos submetidos e a todos os participantes do Workshop por faze-lo acontecer. Esperamos que o SAST'2015 seja proveitoso e contribua com a ampliação e consolidação da área de Teste de Software no Brasil.

Belo Horizonte, setembro 2015

Simone do Rocio Senger de Souza (ICMC-USP)

Auri Marcelo Rizzo Vincenzi (DC-UFSCar)

Coordenadores do SAST’2015

FOREWORD

Testing is the main activity of software quality assurance and its objective is detecting faults in the product or in the development process. Currently, we are increasingly dependent on software system in daily activities and in industrial environments and therefore, we need quick and correct answers of these systems, which are often critical, ubiquitous, persistent and adaptive. This necessity stimulates the development of techniques, criteria and supporting testing tools considering different domains and programming paradigms.

A notable academic progress has already been achieved in the area. Also, the industry has continuously improved test processes. Systematic approaches have proven to be more effective than ad-hoc testing as well as automation of strenuous and repetitive activities. Moreover, successful cases have been reported with results that foment new research and technology transfer to industry. However, there are still challenges for this technology transfer between academia and industry.

SAST – *Brazilian Workshop on Systematic and Automated Software Testing* is the main software testing event in Brazil. Since 2007 this workshop aims to build an annual meeting to discuss issues and challenges in the testing systematization and automation area fostering the cooperation among research and industry communities.

In this 9th edition, SAST'2015 is co-located with the Brazilian Conference on Software: Theory and Practice (CBSoft 2015), at Belo Horizonte city, and will take benefit from other important joint event. SAST'2015 is organized by the Universidade de São Paulo (ICMC/USP) and the Universidade Federal de São Carlos (UFSCar).

The technical program of SAST'2015 includes an invited talk:"***Do we test our mobile applications as well as we use them?"*** presented by ***Prof. Dr.*** **Arilo Claudio Dias-Neto**(UFAM). The talk subject is very relevant considering that nowadays mobile applications are changing the way society interacts and works. Thus, knowing how to test these applications is as important as knowing how to use them. Also, the technical program has three technical sections for paper presentations, which have important contributions for the state of the art state of software testing.

In this year, 20 papers have been submitted to SAST'2015. These papers were carefully revised for at least three members of the program committee, with the cooperation of national and international reviewers. After the review phase, 8 papers were selected. The paper's acceptance rate of SAST 2015 was 40%, which is in line with the average acceptance rate of papers from all previous editions (2007-2014).

The workshops of CBSoft2015 will select the best paper and these papers will be invited to send an extended version to ***Special Issues of the Workshops of CBSoft'2015***in one of the following journals: *Journal of the Brazilian Computer Society - JBCS* or*Journal of Internet Services and Applications – JISA*.

We thank all of the program committee members and the external reviewers for their interesting comments and suggestions to improving the quality of the submitted papers. We acknowledge that your collaboration was important to conclude the review process in a short time. We thank to the PhD student Victor Hugo Santiago Costa Pinto (ICMC/USP) for the creation of new SAST logo and for the support during the development of SAST’2015 website.

We also thank the organizers of CBSoft 2015 for the opportunity and the infrastructure offered. We thank CAPES for the funding to the event organization.

Finally, we would like to thank the guests, the authors of submitted papers and all the participants of the Workshop for making it happen. We expect the SAST'2015 to be a profitable and a contributing event to the expansion and consolidation of Software Testing in Brazil.

Belo Horizonte, September 2015

Simone do Rocio Senger de Souza (ICMC-USP)

Auri Marcelo Rizzo Vincenzi (DC-UFSCar)

Chairs of SAST’2015

COMITÊ DE ORGANIZAÇÃO | *ORGANIZING COMMITTEE*

CBSOFT 2015 GENERAL CHAIRS

Eduardo Figueiredo (UFMG)
Fernando Quintão (UFMG)
Kecia Ferreira (CEFET-MG)
Maria Augusta Nelson (PUC-MG)

CBSOFT 2015 LOCAL COMMITTEE

Carlos Alberto Pietrobon (PUC-MG)
Glívia Angélica Rodrigues Barbosa (CEFET-MG)
Marcelo Werneck Barbosa (PUC-MG)
Humberto Torres Marques Neto (PUC-MG)
Juliana Amaral Baroni de Carvalho (PUC-MG)

WEBSITE AND SUPPORT

Diego Lima (RaroLabs)
Paulo Meirelles (FGA-UnB/CCSL-USP)
Gustavo do Vale (UFMG)
Johnatan Oliveira (UFMG)

COMITÊ TÉCNICO | *TECHNICAL COMMITTEE*

COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE*

Adenilso Simão (ICMC/USP)
Alexandre Petrenko (CRIM, Canada)
Ana Cavalcanti (University of York, UK)
André T. Endo (UTFPR)
Arilo Claudio Dias Neto (IComp/UFAM)
Auri M. R. Vincenzi (DC/UFSCAR)
Avelino F. Zorzo (PUC/RS)
Cassio Rodrigues (INF/UFG)
Edmundo Spoto (INF/UFG)
Eduardo Guerra (LAC/INPE)
Eliane Martins (IC/Unicamp)
Elisa Y. Nakagawa (ICMC/USP)
Ellen F. Barbosa (ICMC/USP)
Fabiano C. Ferrari (DC/UFSCAR)
Fábio F. Silveira (ICT/UNIFESP)
Fátima Matiello-Francisco (INPE)
Guilherme H. Travassos (COPPE/UFRJ)
Jorge Figueiredo (DSC/UFCG)
José Carlos Maldonado (ICMC/USP)
Juliano Iyoda (CIn/UFPE)
Marcelo d'Amorim (CIn/UFPE)
Márcio E. Delamaro (ICMC/USP)
Marcos Chaim (EACH/USP)
Mário Jino (FEEC/UNICAMP)
Patrícia Machado (DSC/UFCG)
Plínio Leitão-Júnior (UFG)
Ricardo Anido (IC/Unicamp)
Roberta Coelho (DIMAp/UFRN)
Sandra C.P.F. Fabbri (DC/UFSCAR)
Silvia R. Vergílio (DInf/UFPR)
Simone R. S. Souza (ICMC/USP)
Valdivino A. Santiago Júnior (LAC/INPE)
Wilkerson L. Andrade (DSC/UFCG)

REVISORES EXTERNOS | *EXTERNAL REVIEWERS*

Adailton Araujo (UFG)
Ana Paiva (Universidade do Porto, Portugal)
Anderson Belgamo (IFSP-Piracicaba)
Draylson Souza(ICMC/USP)
Frank Affonso (UNESP)
Gilmar Arantes (UFG)
Jacson R. Barbosa (UFG)
Lucas B. Oliveira (ICMC/USP)
Santiago Matalonga (Universidad ORT Uruguay)
Vinícius H. Durelli (ICMC/USP)

PALESTRAS CONVIDADAS | *INVITED TALKS*

Do we test our mobile applications as well as we use them?
Arilo Claudio Dias-Neto (UFAM)

Abstract: Mobile applications have been considered an important technological innovation in our society. However, despite we are great users of mobile applications, are we so good testers? This talk discusses some of the challenges introduced by the mobile platform in software testing activities, presenting the main characteristics and limitations. This talk also discusses how the results already known about software testing across multiple platforms can be applied to mobile applications and some lacks observed as research opportunities being worked in this field.

Dr. Arilo Claudio Dias-Netois an associate professor at Institute of Computing at the Federal University of Amazonas, since 2010. He holds a Doctor degree in Systems Engineering and Computer Science from COPPE/UFRJ in 2009, Master degree in Systems Engineering and Computer Science from COPPE/UFRJ in 2006. He leads the Experimentation and Testing on Software Engineering Group (ExperTS) at the Federal University of Amazonas. Prof. Dias-Neto had previous professional experiences on software testing in some companies (e.g.: Brazilian Navy/Brazil and Siemens Corporate Research/USA) and, currently, he participates in research and development projects with the industry. His research interests include software testing, mobile engineering, search-based software engineering, and empirical software engineering. Additional information can be found at http://www.icomp.ufam.edu.br/arilo.

TRILHA DE ARTIGOS TÉCNICOS | *TECHNICAL RESEARCH TRACK PAPERS*

SESSION 1: Experimental Studies

Uma Análise da Eficácia de Assertivas Executáveis como Indicadoras de Falhas em Software
Fischer Jonatas (PUC-Rio), Arndt von Staa (PUC-Rio), Eduardo Figueiredo (UFMG) — **1**

Um Estudo Exploratório sobre a Aplicação de Operadores de Mutação Java em Aplicações Android
Jonathas dos Santos (UFAM), Auri Marcelo Rizzo Vincenzi (UFSCar), Arilo Dias Neto (UFAM) — **11**

Aplicação de Propriedades de Weyuker, Parrish e Zweben a Critérios de Adequação
Diogo de Freitas (UFG), Plínio Leitão-Júnior (UFG), Auri Marcelo Rizzo Vincenzi (UFSCar) — **21**

SESSION 2: Structural testing and reviews

Uma ferramenta interativa para visualização de código fonte no apoio à construção de casos de teste unitário
Heleno Campos (IF Sudeste MGl), Luís Rogério Martins Filho (IF Sudeste MG), Marco Antônio Araújo (UFJF / IF Sudeste MG) — **31**

Teste Estrutural Aplicado à Linguagem Funcional Erlang
Alexandre P. Oliveira (ICMC/USP), Paulo Sergio Souza (ICMC/USP), Simone Souza (ICMC-USP), Júlio Estrella (ICMC/USP), Sarita Bruschi (ICMC/USP) — **41**

Challenges in Testing Context Aware Software Systems
Santiago Matalonga (Universidad ORT Uruguay), Felyppe Rodrigues (PESC/COPPE UFRJ), Guilherme Travassos (COPPE/UFRJ) — **51**

SESSION 3: Formal testing

An empirical study of test generation with BETA
João Souza Neto (UFRN), Ernesto Matos (UFRN), Anamaria Martins Moreira (UFRJ) — **61**

InRob-UML: an Approach for Model-based Interoperability and Robustness Testing of Embedded Systems
Anderson Weller (IFES), Eliane Martins (UNICAMP), Fatima Mattiello-Francisco (INPE) — **71**

Uma Análise da Eficácia de Assertivas Executáveis como Indicadoras de Falhas em Software

Fischer J. Ferreira[1], Arndt von Staa[1], Eduardo Figueiredo[2]

[1]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC- Rio) Rio de Janeiro – RJ – Brasil.

[2]Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil.

{fferreira,arndt}@inf.puc-rio.br, figueiredo@dcc.ufmg.br

Abstract. *During software debugging, a significant amount of effort is needed so that programmers can identify the root cause of a failure. In an attempt to make the software capable of detecting bugs on runtime, we analyze a mechanism for indicating failures on software systems with systematic use of executable assertions. A quasi-experiment was conducted by inserting bugs in some data structures through mutation testing. Results showed that the executable assertions were effective in killing all mutants created during the experiment.*

Resumo. *Durante a depuração de software, uma quantidade significativa de esforço é necessária para que os programadores possam identificar a raiz de uma falha. Na tentativa de tornar o software capaz de detectar defeitos em tempo de uso, analisa-se um mecanismo para indicar falhas em sistemas de software com emprego sistemático de assertivas executáveis. Um quase-experimento foi realizado inserindo defeitos em algumas estruturas de dados por meio de teste baseado em mutantes. Os resultados mostraram que as assertivas executáveis para um grupo particular de programas foram eficazes em matar todos os mutantes criados no experimento.*

1. Introdução

Dia a dia aumenta a nossa dependência com relação a sistemas informatizados. Porém sistemas de software, mesmo quando confeccionado seguindo regras rígidas de qualidade, não estão livres das ocorrências de falhas durante a sua vida útil [Brown and Patterson 2001]. São utilizados cada vez mais bibliotecas e serviços remotos que muitas vezes possuem qualidade duvidosa. Além disso, independentemente do cuidado dos projetistas, especificações erradas ou incompletas são também frequentes, pois advém de possível falta de conhecimento por parte dos projetistas [Armour 2000]. Durante a depuração de software, e também durante o uso produtivo, uma quantidade significativa de esforço é despendida para identificar a causa raiz de uma falha [Yi et.al. 2015]. Então, faz-se necessário que exista um mecanismo para facilitar o processo de observação de falhas e de identificação dos fragmentos de códigos relacionados com elas [Pullum 2001].

Ao exercitar um defeito é possível que seja gerado um erro, ou seja, um desvio entre o estado do sistema e o estado que havia sido especificado. Erros podem passar um tempo sem que sejam observados. No momento em que são observados passam a ser falhas. Para poder diagnosticar corretamente e com pouco esforço a causa exata (o

defeito) é importante que seja pequena a latência entre o momento da geração do erro e da sua observação [Magalhães et.al. 2009]. Quanto menor a latência, menor será o volume de código analisado, menor será a interferência de dados e fragmentos de código não relacionados ao defeito. Em adição, é necessário saber o porquê da falha, para que mantenedores ou desenvolvedores sejam capazes de remover os correspondentes defeitos de forma eficaz [Khoshnood 2015].

A violação de uma assertiva executável (AE) indica a presença de uma falha [Khoshnood 2015]. Além de reportar a ocorrência da falha, pode-se também reportar a parte relevante do estado do sistema no momento da sua observação. Surgem então as seguintes questões de pesquisa: (i) as AEs seriam um mecanismo eficaz para observar falhas? (ii) dado um número conhecido de falhas, as AEs podem indicá-las em totalidade?

Neste trabalho não discutimos quais políticas de inserção de AEs devem ser utilizadas. Também não foi objetivo avaliar o custo de sua redação. Em Magalhães et.al. [2009], foi mostrado que AEs requerem em torno de 10% de linhas de código a mais. Como elas tornam necessária a formalização, mesmo que parcial, de aspectos implementacionais do programa, o custo do desenvolvimento diminuiu em virtude da redução do retrabalho inútil. Também não foi objetivo verificar a aceitação pelos desenvolvedores do uso sistemático de AEs ao desenvolver programas. Em Araújo et.al. [2012], foi demonstrado que até mesmo desenvolvedores pouco experientes puderam dar manutenção em um programa com apoio de um conjunto de AEs.

Neste trabalho analisamos, por meio de um experimento, o uso de AEs quanto a sua eficácia para observar falhas. Para tal medimos quantos mutantes são mortos por intermédio das AEs. Um mutante é uma versão propositalmente adulterada do programa sob teste, injetando um ou poucos defeitos nele. Um mutante é dito morto, se um ou mais casos de teste da suíte de teste acuse uma falha. A eficácia da suíte de teste pode ser estimada observando-se quantos mutantes são mortos pela suíte de teste. Espera-se, porém, que um programa devidamente instrumentado com assertivas executáveis seria capaz de acusar a ocorrência de uma falha provocada por um mutante, por meio da observação de alguma das assertivas. A eficácia das assertivas é agora estimada pelo número de mutantes que são mortos por falha detectada por alguma assertiva. Os resultados alcançados no experimento realizado demonstraram que AEs criadas puderam matar todos os mutantes para o grupo de estruturas de dados utilizado.

3. Fundamentação teórica

Esta seção apresenta a fundamentação teórica para o entendimento deste trabalho. Ela contempla uma visão geral sobre assertivas executáveis (Subseção 3.1) e conceitos sobre teste baseado em mutantes (Subseção 3.2).

3.1. Assertivas executáveis (AEs)

As falhas que os programas apresentam geralmente são demonstradas como saídas inconsistentes, inesperadas ou resultados indesejáveis. Tais saídas fornecem aos desenvolvedores poucas informações para rastrearem a causa raiz do problema [Clarke and Rosenblum 2006]. Sem um mecanismo que possa informar o fragmento de código associado às falhas, o processo de inspeção no código requer muito esforço, sujeito a muitos erros humanos e tende a ser impreciso [Yi et.al. 2015].

Segundo Duncan e Hölzle [1998], AEs são expressões booleanas que devem ser satisfeitas caso o fragmento de código a que são associadas esteja operando corretamente. Elas são uma das mais úteis técnicas automatizadas disponíveis para detecção de falhas e fornecimentos de informações sobre pontos onde o defeito está localizado no código [Clarke and Rosenblum 2006]. As AEs são essencialmente especificações executáveis e verificáveis, viabilizam assim atuar como oráculos dinâmicos [Staa 2000].

Redigir AEs no desenvolvimento inicial do código contribui para escrever código mais correto, e estimula os desenvolvedores a pensarem no problema a ser resolvido, em vez de começar a codificar, antes que a solução seja suficientemente bem entendida [Araújo et.al. 2012]. Elas obrigam a formalização de especificações. Essa formalização, mesmo que incompleta [Akhtar and Missen 2014], reduz significativamente o volume de defeitos acidentalmente inseridos pelo programador. Observa-se também que o custo do desenvolvimento diminui em virtude da redução significativa do volume de retrabalho inútil [Hall 1990, Bowen and Hinchey 1994].

Segundo Araújo et.al. [2012] o conjunto de AEs representa entre 8 e 12% de acréscimo de linhas de código ao código original e não fazem parte da implementação de um algoritmo ou classe, portanto não participam no processamento dos algoritmos. As AEs podem ser facilmente ativadas e desativadas em sistema em produção ou desenvolvimento.

Uma das vantagens das AEs é que podem ficar ativadas em sistema em produção, podendo observar inconsistências para entradas reais em todo o ciclo de vida do software. Porém, AEs incompletas podem levar a falsos negativos e assertivas incorretas podem levar a falsos positivos [Staa 2015]. Assim, o entendimento do domínio do sistema é fundamental para qualidade das AEs criadas. Os falsos positivos são facilmente sanados. Os falsos negativos podem ser, em grande parte, sanados ainda durante os testes, já desde os testes de unidade.

3.2. Teste baseado em mutantes

Teste baseado em mutantes é uma técnica utilizada para aferir a eficácia da suíte de teste, inicialmente proposto por Demillo et.al. [1978]. Por meio de injeções de defeitos no programa original são criadas várias versões alteradas desse programa cada qual corresponde a um mutante. Os defeitos são injetados pela aplicação sistemática de operadores de mutação. Esses operadores correspondem aos defeitos típicos. Segundo Delamaro et.al. [2007], os operadores de mutação surgiram de estudos que determinavam os erros mais comuns cometidos por programadores considerando linguagens de programação específicas.

Quando um operador de mutação é aplicado ao programa original, e o programa original possui a estrutura sintática que o operador de mutação consegue modificá-lo, então essas modificações são exercitadas e um conjunto de mutantes é gerado. A cada ocorrência encontrada no programa original dessa estrutura sintática é criado um novo mutante. Cada mutante criado é sintaticamente diferente dos demais mutantes para o um mesmo programa.

Segundo a sintaxe e semântica do programa escrito, os mutantes são gerados por meio de pequenas variações do programa original e não no universo de todas as variações possíveis. Essa prática é formulada como base em duas hipóteses. (i) Hipótese

do programador competente que estabelece: um programa criado por um programador competente está correto ou está próximo do correto e (ii) hipótese do efeito do acoplamento que preconiza: defeitos complexos estão ligados a defeitos simples e, por isso, a detecção de um defeito simples pode levar a descoberta de defeitos complexos [Demillo et.al. 1978].

Para cada mutante, é aplicado o conjunto original de testes. Nos casos de testes, se algum falhar, o mutante ao qual foi testado diz-se morto. Diante disso, a suíte de teste foi capaz de observar o desvio sintático correspondente ao mutante. Se os testes passarem o mutante é considerado como vivo, pois a suíte de teste não foi capaz de observar o comportamento incorreto provocado pelo mutante. Contudo, alguns mutantes que permanecem vivos poderão ser equivalentes ao programa original. Para determinar se um mutante é equivalente se faz necessária uma análise feita pelo testador, e definir se o mutante se manteve vivo por ser realmente equivalente ao programa original ou se os testes não foram capazes de perceberem o defeito injetado. Caso os testes detectarem as falhas artificiais, assume-se que detectarão falhas reais [Delamaro et.al. 2007]. Evidentemente isso requer a geração substantiva e sistemática de mutantes.

Como exemplo de criação de um mutante, na Figura 1 é demonstrado um fragmento de código de um programa original e o seu respectivo código modificado. O operador de mutação ROR, que substitui os operadores relacionais, foi escolhido para aplicar a mutação no código original. No caso específico do exemplo o operador relacional igual (==) foi substituído por diferente (!=).

Fragmento de código original

```
private Comparable elementAt(AvlNode t ){
return t == null ? null : t.element; }
```

Mutante para ROR

```
private Comparable elementAt(AvlNode t ){
return t != null ? null : t.element;}
```

Figura 1. Exemplo de mutante criado por meio do operador de mutação ROR

Para entendimento dos termos utilizados no experimento será utilizada a seguinte terminologia: (i) *Mutante:* o programa original modificado; (ii) *Operador de mutação:* o agente que definirá qual será o tipo de transformação a que o programa original será submetido; (iii) *Mutante morto:* quando a suíte de teste consegue distinguir o mutante do seu respectivo programa original, ou seja, neste caso um ou mais testes não passaram. (iv) *Mutante vivo:* quando a suíte de teste não consegue distinguir o mutante do seu respectivo programa original, ou seja, neste caso todos os testes passaram. (v) *Mutante equivalente:* quando por análise de um desenvolvedor experiente o mutante é considerado equivalente ao seu respectivo programa original.

4. Configuração do experimento

Esta seção apresenta detalhes da configuração do experimento. Ela contempla o modelo e demonstração do uso de assertivas executáveis (Subseção 4.1), métrica utilizada (Subseção 4.2) e justificativas do uso de mutantes para inserção de defeitos (Subseção 4.3).

4.1 Modelo e demonstração do uso de assertivas executáveis

No experimento realizado foram utilizadas apenas assertivas executáveis estruturais. Estas envolvem condições e diversos objetos pertencentes às classes que realizam a estrutura de dados. Essas condições devem ser sempre verdadeiras quando a estrutura

não está sendo alterada [Staa 2000]. Para confecção das AEs utilizadas no experimento foram realizados os seguintes passos: (i) estender a classe original que se deseja instrumentar a fim de que sejam herdados todos os seus métodos e atributos, os quais serão utilizados para criação das AEs. (ii) Na classe filha devem ser inseridas as AEs, para elas sejam chamadas por apenas um método, cria-se um método público que internamente invoca as AEs criadas. (iii) Cada AE será codificada em um método privado que retornará *false,* caso o fragmento de código em que ela estiver associada tiver alguma inconsistência ou desvio do estado computacional pretendido. A AE deverá informar qual a inconsistência observada, bem como os dados manipulados e em que ponto do código foi observado. Se nenhum defeito for observado o conjunto de AEs deve retornar *true.*

A Figura 2 demonstra um exemplo de utilização do modelo no qual uma implementação da árvore AVL é instrumentada. Nesse exemplo apenas uma AE é chamada pelo verificador. O exemplo de como a AE é implementada pode ser observado na Figura 4.

```
public class AvlTreeInstrumentada extends AvlTree {
public boolean verificador() {
if (!verificadorArvoreBinaria(root)) {return false;}
return true;
}
```

Figura 2. Exemplo do modelo de criação de AE

Para descrever as assertivas foi usado um misto de linguagens formais e português proposto por Staa [2000]. Um exemplo pode ser observado na Figura 3, no qual uma assertiva é criada para verificar se uma dada árvore mantém a propriedade de árvore binária de pesquisa. Assim, essa assertiva define que cada elemento da árvore em questão que tenha um filho à esquerda, seja menor que seu pai direto. Além disso, existindo o filho à direita, seja maior que seu pai direto. Essa forma de descrever assertiva também poderá ser usada para documentá-las.

$$\forall\, n \in \acute{a}rvore\text{: } (n \longrightarrow left\,!= null) \Rightarrow ((n \longrightarrow elemento) > (n \longrightarrow left \longrightarrow elemento\,))$$

$$\forall n \in \acute{a}rvore\text{: } (n \longrightarrow right != null) \Rightarrow ((n \longrightarrow elemento) < (n \longrightarrow right \longrightarrow elemento))$$

Figura 3. Assertiva para uma árvore binária de pesquisa

A Figura 4 demonstra a implementação na linguagem Java da assertiva descrita na Figura 3. Caso a propriedade de árvore binária de pesquisa seja violada (linhas seis e sete do código exemplo), uma exceção será gerada com informações relativas ao exato elemento em que foi observada a inconsistência (linhas de oito a doze do código exemplo).

```
1.  private boolean verificadorArvoreBinaria(AvlNode t) {
2.  if (t != null) {
3.  verificadorArvoreBinaria(t.left);
4.  if (t.left != null && t.right != null) {
5.  try {
6.  if ((((MyInteger) (t.left.element)).intValue())>(((MyInteger) (t.element)).intValue())
7.  && (((MyInteger) (t.right.element)).intValue())<(((MyInteger) (t.element)).intValue())){
8.  throw new IllegalStructureException();}
9.  }catch (IllegalStructureException e) { e.printStackTrace();
10. System.err.println("@ A árvore não é uma árvore de busca binária :"
11. + "filho da esquerda: " + t.left.element
12. + "filho da direita: "  + t.right.element+ "elemento pai" + t.element);} }
13. verificadorArvoreBinaria(t.right);} return true;}
```

Figura 4. AE implementada

4.2. Métrica utilizada

Segundo Delamaro et.al. [2007] um escore de mutação pode ser calculado com uma análise do número de mutantes que permaneceram vivos e mortos, sendo que esse escore de mutação deve variar entre zero e um. Quanto maior for o valor do escore de mutação, mais adequado será o conjunto de casos de teste para testar o programa. O escore de mutação originalmente criado foi adaptado para aferir a eficácia das AEs. Assim, a Figura 5 demonstra o cálculo do escore de mutação para AEs:

$$EM(P,A) = \frac{DM(P,A)}{M(P) - EQ(P)}$$

Figura 5. Escore de mutação para o conjunto de AEs

Sendo: (i) *EM(P,A):* escore de mutação de um programa P em relação ao conjunto de AEs; (ii) *DM(P,A):* número de mutantes mortos pelas AEs; (iii) *M(P):* número total de mutantes gerados a partir de programa P; (iv) *EQ(P):* número de mutantes considerados equivalentes a P;

4.3. Justificativa do uso de mutantes para inserção de defeitos

O uso de AEs não mede a eficácia da suíte de testes, uma vez que não é possível estimar-se o número de defeitos existentes. Uma das formas de criar uma estimativa é por meio do uso de mutantes. Por meio do emprego de operadores de mutação a um programa original, com apoio de uma ferramenta para testes de análise mutante, torna-se possível conhecer e gerenciar as falhas que correspondam aos mutantes. Quanto maior o número e a variedade de mutantes, maior a confiabilidade assegurada pela suíte de teste para ser capaz de identificar um número grande desses mutantes, mediante aos oráculos contidos na suíte de teste.

Quando as AEs são utilizadas e essas acusam as falhas observadas ao executar a suíte de testes, os oráculos de teste possivelmente não chegam a acusar falhas, pois as AEs já as detectaram. Então se as AEs funcionam a contento é de se esperar que elas observem o defeito antes do programa retornar para o controle do teste. Assim, deixa-se de medir a qualidade da suíte de teste, forma pela qual o teste de análise mutante originalmente foi concebido, e passa-se a medir a eficácia do conjunto de AEs em identificar as falhas provocadas pelos mutantes. Com isso, as AEs agora podem matar os mutantes, quando elas conseguem observar o erro gerado ao executar o mutante. Como, por definição da abordagem de teste baseado em mutantes, cada mutante contém um ou poucos defeitos interdependentes, a falha observada por uma AE corresponde a ter detectado o erro gerado pela alteração, portanto o mutante terá sido morto.

O uso sistemático de AE torna possível assegurar-se que, existindo algum defeito, as falhas por ele provocadas serão observadas por alguma AE. Evidentemente, isso implica a necessidade de uma política de instrumentação (inserção de assertivas) suficientemente abrangente. Neste trabalho não procuramos discutir que políticas de inserção de assertivas devem ser idealmente utilizadas. Baseamos o estudo no uso de assertivas executáveis estruturais que são capazes de verificar se as propriedades formais, invariantes estruturais, das estruturas de dados são satisfeitas.

5. Resultados obtidos com o experimento

As implementações das estruturas de dados utilizadas no experimento foram extraídas do livro *Data Structures and Algorithm Analysis in Java* [Weiss 2012]. Assim, para essas estruturas de dados foram inseridas AEs como descritas na Seção 4.1. As AEs utilizadas no experimento foram criadas pelo autor principal desse trabalho e revisada pelos demais autores.

Com a finalidade de criar os mutantes para cada estrutura de dados foram utilizados todos os operadores de mutação que a ferramenta MuClipse [Smith and Williams 2007] fornece. Foi configurado na ferramenta MuClipse o tempo de 4500 ms para execução dos mutantes.

Os resultados obtidos com os testes de análise mutante para verificar a eficácia das AEs estão apresentados na Tabela 1. Nessa tabela são descritos os tipos de mutantes criados por meio dos operadores de mutação de métodos ou classe.

Tabela 1: Resultados obtidos com teste mutantes e assertivas executáveis

Estrutura de dados	Método	Classe	Total	Equivalentes	Morto	Vivo	EM
AA Tree	133	56	189	2	189	0	1,0
AVL Tree	139	16	155	6	155	0	1,0
Binary Heap	191	2	193	1	193	0	1,0
Binary Search Tree	50	5	55	1	55	0	1,0
Binomial Queue	225	7	232	0	232	0	1,0
Black Red Tree	88	88	176	5	176	0	1,0
BTree	1582	30	1612	16	1612	0	1,0
Deterministic Skip List	32	40	72	0	72	0	1,0
Fibonacci Heap	167	39	206	0	206	0	1,0
Leftist Heap	32	6	38	0	38	0	1,0
Linked List	173	87	260	12	260	0	1,0
Pair Heap	203	87	290	1	290	0	1,0
Spaly Tree	54	142	196	3	196	0	1,0
Treap	72	23	95	1	95	0	1,0

As colunas da Tabela 1 contêm as seguintes informações: A primeira coluna descreve o nome da estrutura de dados utilizada. Na segunda e terceira colunas são apresentados os mutantes oriundos de operadores de mutação sobre código de método e de classe respectivamente. A próxima coluna descreve o número total de mutantes criados que consiste no somatório dos tipos de mutantes de método e classe. Ainda na quinta coluna é informado o número de mutantes equivalentes. O critério para análise dos mutantes equivalentes utilizado neste trabalho (i) baseou-se na comparação da saída que o mutante produz em relação à saída que o programa original apresenta para uma mesma instância de entrada e (ii) por meio da análise dos autores comparando o código original e seu respectivo mutante equivalente. Quando a saída for a mesma em ambos os casos, e quando o programa original e o mutante equivalente forem sintaticamente equivalentes, considerou-se que o mutante é equivalente ao programa original. Para isso os mutantes vivos e o programa original foram submetidos a uma entrada de cinquenta mil elementos aleatórios.

Nas próximas colunas, foram descritos o número de mutantes não equivalentes mortos, vivos e o escore de mutação respectivamente para as estruturas de dados instrumentados com AEs. Por fim a última coluna da tabela descreve os escores de mutação (EM) para programas instrumentados segundo a métrica proposta neste

trabalho na Seção 3.2. As assertivas tiveram o escore de mutação igual a 1,0. Isso demonstra que elas foram 100% eficaz para detectar as modificações que os mutantes apresentaram em relação ao programa original.

No link: http://labsoft.dcc.ufmg.br/doku.php?id=about:mutants_list podem ser visualizados arquivos utilizados no experimento, separados da seguinte forma: (i) código original da estruturas de dados e implementações das AEs, (ii) código fonte dos mutantes criados e (iii) suíte de teste.

6. Riscos à Validade

A redação de AE reduz o número de defeitos, mas não os elimina. Além disso, o conjunto de AE muitas vezes é incompleto, e pode até ser incorreto, permitindo a ocorrência de falsos negativos**.** Assim se faz necessário o esforço adicional para o entendimento de detalhes dos requisitos, pois se as assertivas não contemplarem os pontos críticos do sistema, falhas de grande monta não serão observadas por elas. No experimento realizado neste trabalho o custo em tomar conhecimento das partes críticas foi nulo, porque as propriedades das estruturas de dados são bem definidas e provadas na literatura relacionada. Portanto, não foi levado em consideração esse fator que é primordial que a criação do conhecimento dos requisitos do sistema. Em outro experimento que se tenha que levantar os requisitos para confecção das AEs, elas podem ficar incompletas e não terem a eficácia de 100% que foi observado no experimento deste trabalho.

Ainda as AEs confeccionadas no experimento foram criadas pelo primeiro autor e validada pelos demais autores, no entanto o resultado seria mais fidedigno se as AEs fossem criadas e validadas por um grupo maior de participantes, a fim de que, detalhes da criação e custos associados fossem analisados. Além do mais, os sistemas alvo do experimento foram apenas estruturas de dados. Faz-se necessário que experimento seja replicado para outros tipos de sistemas, como exemplo sistemas de informação.

No experimento realizado, o custo do entendimento das especificações das estruturas de dados foi nulo, por suas propriedades serem definidas e provadas na literatura. Como a qualidade das AEs está diretamente relacionada ao entendimento dos requisitos do sistema de software. Não se pode afirmar se os resultados alcançados no presente trabalho poderiam ser replicados para outros experimentos, que fosse necessário o levantamento dos requisitos para elaboração das assertivas.

7. Trabalhos relacionados

Existem vários exemplos bem sucedidos na literatura que demonstram os resultados obtidos com a aplicação de AEs na indústria, tais como: projeto de roteamento de mensagens e sistema de alerta de congestionamento de veículos [Larsen et.al. 2006]; software para Nave Espacial da NASA [Feather 1998]; ferramenta para reparação de testes [Yang 2012]; Sistema robótico multi-agente para o transporte de estoque de armazéns [Akhtar and Missen 2014].

Uma análise do uso de AEs em um sistema em uso foi realizada por Magalhães et.al. (2009) no qual demonstrou que vinte e duas falhas puderam ser observadas por assertivas durante testes comparadas com cinco falhas observadas por outros meios durante testes [Magalhães et.al. 2009].

A principal diferença na abordagem utilizada no trabalho proposto, comparada com trabalhos anteriores que também avaliam a eficácia de assertivas, se dá pelo fato do experimento deste trabalho ser feito sobre defeitos conhecidos e gerenciáveis. Os defeitos são conhecidos por meio dos mutantes. Sendo esses gerados por operadores de mutação pré-determinados, possibilitando assim o conhecimento prévio do total dos defeitos injetados. Também podem ser gerenciáveis por meio da funcionalidade oferecida pela ferramenta utilizada a qual foi MuClipse [Smith and Williams 2007], que possibilita a identificação do estado dos mutantes (vivos ou mortos) após serem testados. Ainda, a observação dos erros que cada mutante apresenta comparado com o programa original.

8. Conclusões e trabalhos futuros

Por meio da implementação em forma de AEs das propriedades das estruturas de dados utilizadas, as AEs puderam encontrar falhas dos mutantes antes dos oráculos associados aos casos de teste. Como demonstrado, todos os mutantes criados foram mortos pela intervenção das AEs criadas. Elas foram 100% eficazes na detecção das anomalias que os mutantes apresentaram em relação o programa original. Portanto, as falhas inseridas sistematicamente foram detectadas pelas assertivas executáveis em tempo de execução.

Por fim, as questões de pesquisa levantadas para este quase-experimento podem ser respondidas. (i) As AEs seriam um mecanismo eficaz para observar falhas? Sim, como observado para as estruturas de dados utilizados as AEs foram capazes de observar as falhas. (ii) Dado um número conhecido de falhas as AEs podem indicá-las em totalidade? Sim, todos os mutantes criados foram mortos pelas as AEs.

A partir desses resultados pôde-se replicar o experimento para um número maior de programas a serem instrumentados e mais indivíduos confeccionando as AEs. Assim, pode ser analisados dados relativos à confecção de AEs tais como: (i) custo de sua redação; (ii) capacidade e aceitação de desenvolvedores para criarem AEs e corrigir falhas apontadas por elas e (iii) tempo utilizado para confecção.

Referências

Akhtar, N. and Missen, M. M. S. (2014). “Practical application of a light-weight formal implementation for specifying a multi-agent robotic system” International Journal of Computer Science Issues, Vol. 11, Issue 1, No 2, January 2014.

Araújo, T., Wanderley, C. and von Staa, A. (2012). “An introspection mechanism to debug distributed systems”. In Software Engineering (SBES), 2012 26th Brazilian Symposium on (pp. 21-30). IEEE.

Armour, P. G. (2000). “The five orders of ignorance”. Communications of the ACM, 43(10), 17-20.

Bowen, J. P., and Hinchey, M. G. (1994, January). “Seven more myths of formal methods: Dispelling industrial prejudices”. In FME'94: Industrial Benefit of Formal Methods (pp. 105-117). Springer Berlin Heidelberg.

Brown, A., and Patterson, D. A. (2001). “To err is human”. In Proceedings of the First Workshop on evaluating and architecting system dependability (EASY’01).

Clarke, L. A., and Rosenblum, D. S. (2006). “A historical perspective on runtime assertion checking in software development.” ACM SIGSOFT Software Engineering Notes, 31(3), 25-37.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). “Introdução ao teste de software” Rio de Janeiro, RJ, BR: Editora Campus, 2007. 408 p.

Demillo, R.; Lipton, R.; Sayward, F (1978). “Hints on Test Data Selection: Help for the Practicing Programmer”. Computer, 11(4):34–41.

Duncan, A., and Hölzle, U. (1998). “Adding contracts to Java with Handshake”. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA.

Feather, M. S. (1998). “Rapid application of lightweight formal methods for consistency analyses”. Software Engineering, IEEE Transactions on, 24(11), 949-959.

Hall, A. (1990). “Seven myths of formal methods”. Software, IEEE, 7(5), 11-19.

Khoshnood, S. (2015). “Constraint Solving for Diagnosing Concurrency Bugs“ (Doctoral dissertation, Virginia Tech).

Larsen, P.G., Fitzgerald, J.S., Riddle, S. (2006). “Learning by Doing: Practical Courses in Lightweight Formal Methods using VDM++”. Technical Report CS-TR:992, School of Computing Science, Newcastle University

Pullum, L. L. (2001). “Software fault tolerance techniques and implementation”. Boston London. Artech House Computing Library.

Magalhães, J., Staa, A.v, and de Lucena, C. J. P. (2009). “Evaluating the recovery oriented approach through the systematic development of real complex applications. Software: Practice and Experience”, 39(3), 315-330.

Smith, B. H., and Williams, L. (2007, September). “An empirical evaluation of the MuJava mutation operators”. In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007 (pp. 193-202). IEEE.

Staa, A.v. (2000) “Programação Modular, Desenvolvendo programas complexos de forma organizada e segura”. Rio de Janeiro. Editora Campus/Elsevier.

Staa, A.v. (2015) “Teste Automatizado 2” nota de aula PUC-Rio, 2015. Acessado em 15-06-2015 no endereço: http://www.inf.puc-rio.br/~inf1413/docs/INF1413_Aula23_TesteAutomatizado-2.pdf

Weiss, M,A. (2012) “Data Structures and Algorithm Analysis in Java”. 3rd ed. ISBN-13: 978-0-13-257627-7.

Yi, Q., Yang, Z., Liu, J., Zhao, C., & Wang, C. (2015). “A synergistic analysis method for explaining failed regression tests”. In *International Conference on Software Engineering*.

Um Estudo Exploratório sobre a Aplicação de Operadores de Mutação Java em Aplicações Android

Jonathas S. dos Santos[1], Auri M. R. Vincenzi[2], Arilo C. Dias-Neto[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brasil

[2]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
São Carlos – SP – Brasil

{jss,arilo}@icomp.ufam.edu.br, auri@dc.ufscar.br

Abstract. *Applying efficient testing techniques in mobile applications has become essential due to the growing of complexity and importance of them in our society. One of the most efficient testing criteria is a Mutation Testing. This paper presents an exploratory study that evaluated the use of eleven Java Mutation Operators in Android applications. The results obtained in the study, improve the feasibility of the mutation testing in the context of Android mobile applications considering the use of Java operators. However, the results, although preliminary, raise the possibility of an adequacy to the specific features present on Android codes.*

Resumo. *Aplicar técnicas de teste eficientes em aplicações móveis é essencial devido ao crescimento da complexidade e importância delas na sociedade. Um dos critérios de teste mais eficiente é o Teste de Mutação. Este artigo apresenta um estudo exploratório que avaliou o uso de onze operadores de mutação Java em aplicações Android. Os resultados obtidos endossam o uso do critério de Teste de Mutação para aplicações móveis considerando operadores Java. No entanto os resultados, mesmo preliminares, levantam a possibilidade de uma adequação às características específicas presentes em códigos Android.*

1. Introdução

Aplicações móveis (ou simplesmente *apps*) tornaram-se parte da rotina das pessoas não apenas para entretenimento, como por exemplo jogos ou redes sociais, mas também para tarefas críticas, tais como transações bancárias, segurança residencial ou cuidados médicos (van der Meulen e Rivera, 2014). O crescimento no número de *apps* tem desafiado engenheiros de software a desenvolver tais software com alto nível de qualidade para torna-los atrativos e competitivos neste novo mercado (van der Meulen e Rivera, 2014).

Por outro lado, segundo Muccini *et al.* (2012) a plataforma móvel tem introduzido novas restrições a serem consideradas durante o desenvolvimento de software, como o pequeno tamanho das telas, conectividade limitada, taxas de consumo de energia, capacidade de processamento reduzida e diversas formas de entrada de dados.

Neste contexto, uma das formas de garantir a qualidade nas *apps* é por meio de atividades de teste de software. Assim, tal atividade representa um importante desafio de pesquisa no contexto atual da Engenharia de Software. De acordo com Muccini *et al.* (2012), testar *apps* é mais desafiador que testar aplicações em outras plataformas devido

às restrições citadas anteriormente.

Visando prover testes de software com qualidade, um dos critérios de teste mais efetivos é o Teste de Mutação (TM) (Jia e Harman, 2011). Usando esse critério, defeitos que refletem enganos comuns feitos por programadores são inseridos em um programa de forma a gerar um programa diferente, chamado de **mutante** (Jia e Harman, 2011). Mais detalhes sobre o critério serão apresentados na Seção 2.

TM é aplicado em diferentes tipos de programas, plataformas ou linguagens de programação com resultados positivos (Jia e Harman, 2011). No entanto, a sua utilização em teste em *apps* ainda é pouco explorada. Uma possível razão para este cenário é que TM possui como ponto negativo o alto custo computacional para sua execução, o que poderia limitar sua utilização em *apps*, devido ao limite de recursos computacionais dos dispositivos móveis (ex: memória, bateria e processador). Além disso, é necessário investigar se operadores de mutação propostos para outras plataformas/linguagens podem ser aplicados para *apps* e se outros operadores devem ser criados para cobrir características específicas presente em *apps*.

Neste contexto, este trabalho tem como objetivo investigar a adequação de operadores de mutação da linguagem Java quando aplicado a *apps* desenvolvidas na plataforma Android (que é uma extensão da linguagem Java para *apps*). Para alcançar esse objetivo foi realizado um estudo exploratório utilizando 11 operadores de mutação Java sumarizados em Ahmed *et al*. (2010) e disponíveis na ferramenta PIT[1]. O estudo envolveu duas *apps* desenvolvidas em Android. Os resultados (escore de mutação e cobertura de código) indicam que operadores de mutação Java podem ser usados no contexto de *apps* em Android, no entanto, alguns indícios apontados a partir dos resultados apontam algumas discussões relevantes.

Este artigo está estruturado da seguinte forma: a Seção 2 apresenta o referencial teórico sobre Teste de Mutação e os principais componentes de *apps* Android. A Seção 3 descreve o planejamento e execução do estudo exploratório para avaliar a utilização de operadores de mutação Java em *apps* desenvolvidas para Android. A Seção 4 são analisados os resultados obtidos no estudo e é iniciada uma discussão sobre alguns indícios apontados no estudo. Finalmente, a Seção 5 apresenta as ameaças à validade do estudo e a Seção 6 apresenta conclusões sobre o trabalho e os próximos passos a serem realizados.

2. Referencial Teórico

2.1. Teste de Mutação

O TM provê um processo para medir a efetividade de casos de teste por meio de mutações, representando erros comuns (Demillo *et al*., 1978). Para isso, defeitos que refletem enganos comuns feitos por programadores são inseridos em um programa de forma a gerar um programa diferente, denominado **mutante** (Jia e Harman, 2011). Esses mutantes são gerados usando **operadores de mutação**, que são regras que definem modificações a serem aplicadas ao programa sob teste.

O objetivo deste critério é usar um conjunto de teste já projetado para avaliar o programa original visando encontrar todos os defeitos inseridos nesses programas

[1] PIT Mutation Testing – http://pitest.org

modificados (em TM, esta operação é chamada “matar os mutantes”). Se os mutantes são mortos, assume-se que o conjunto de teste é eficaz e possibilita encontrar eventuais outros defeitos no programa (Jia e Harman, 2011). Em geral, cada linguagem de programação possui um conjunto diferente de operadores de mutação proposto na literatura técnica, por exemplo operadores para a linguagem C (Agrawal *et al,* 1989), para Java (Kim *et al*, 1999), para C# (Derezinska, 2005), inclusive para especificações formais (Budd e Gopal, 1985).

2.2. Plataforma Android

As *apps* desenvolvidas na plataforma Android são compostas por componentes específicos. Esses componentes são os blocos/módulos essenciais para o seu funcionamento dentro da plataforma. Cada componente é um ponto diferente por meio do qual o sistema operacional pode interagir com uma *app*. Existem cinco tipos diferentes de componentes para *apps* em Android, e cada tipo tem uma finalidade distinta e tem um ciclo de vida diferente que define como o componente é criado e destruído. Os componentes são descritos na Tabela 1.

Tabela 1. Componentes de uma *app* em Android

Componentes	Descrição
Activity	Representa um componente responsável pela gestão da interface do usuário (User Interface - UI) de uma *app* Android (Lecheta, 2013).
Service	É um componente que trabalha em segundo plano para executar operações de longa execução ou executar trabalhos para processos remotos (Lecheta, 2013).
Content Provider	Gerencia dados no sistema de arquivos, na web ou qualquer outro local de armazenamento que ele pode acessar (Android Developers, 2014)
Broadcast Receiver	Gerencia as muitas transmissões que se originam do sistema; por exemplo, uma mensagem de *broadcast* anunciando que a tela desligou, a bateria está fraca ou uma imagem foi capturada (Android Developers, 2014)
Intent	Mensagens assíncronas que permitem que os componentes de *apps* solicitem funcionalidade de outros componentes Android (Android Developers, 2014)

2.3. Testes de Mutação em Aplicações Android

Especificamente no contexto da plataforma Android, Garousi *et al.* (2013) integram o critério de TM em um estudo de caso que tem o objetivo de estabelecer uma relação entre a efetividade dos testes e a densidade de defeitos existentes no código fonte. Por conta disso, o estudo utiliza o critério para inserir defeitos em classes de código Android visando medir o quanto as classes com mais densidades de defeitos são cobertas pelos casos de teste propostos.

Como pontos positivos do trabalho de Garousi *et al.* (2013), além dos excelentes resultados do estudo de caso, indício de que o critério pode ser aplicado para verificar a eficiência dos testes. Como pontos negativos, o artigo não descreve os detalhes da execução do critério, citando apenas a ferramenta utilizada (Lava), e também informações sobre o ambiente e os detalhes de configuração não são descritos.

O estudo de caso realizado por Garousi *et al.* (2013) foi o ponto de partida para o estudo exploratório apresentado neste artigo. Este artigo procura contribuir ao trazer o foco apenas para o critério de TM, considerando a sua eficiência na melhoria de conjuntos de teste, a sua eficácia em revelar falhas e a criticidade e importância de *apps.*

3. Estudo sobre Aplicação de Operadores de Mutação Java em Android

3.1. Objetivos e Planejamento do Estudo

O objetivo deste estudo exploratório é analisar o critério de TM, com o propósito de verificar a sua adequação no contexto de *apps* Android.

3.2. Ferramentas de Mutação em Java e Testes Unitários em Android

No estudo de caso realizado por Garousi *et al.* (2013), foi utilizada a ferramenta Lava (Danicic, 2015), porém a mesma já não recebe suporte e no estudo foram utilizados apenas os operadores aritméticos da mesma. Por isso, foi necessário utilizar outra ferramenta, que permitisse vencer os desafios de configuração com a plataforma Android, também apontados por Garousi *et al.* (2013).

A lista inicial de possíveis ferramentas de TM em Java foi obtida em Delahaye e Du Bousquet (2013), onde autores comparam oito ferramentas em relação a diferentes características, tais como eficiência, compatibilidade com plataformas Java e arcabouços de teste unitário. Como resultado, a ferramenta PIT[2] se mostrou uma ferramenta de simples configuração e uso, por isso optou-se pelo uso dela neste estudo. PIT pode ser executada a partir de linha de comando, além de ter um *plug-in* para Maven[3]. Ela realiza a mutação em nível de *bytecode* e durante a execução identifica quais testes unitários são relevantes para mutação em cada parte do código fonte. PIT implementa um conjunto de onze operadores de mutação Java, listados na Tabela 2.

Tabela 2. Lista de operadores de mutação Java implementados na ferramenta PIT.

ID	Operador de Mutação	Descrição
PIT1	*Conditionals Boundary Mutator*	Substitui os operadores relacionais <, <=, >, >= com seu limite (por exemplo, substituir <= por <).
PIT2	*Negate Conditionals Mutator*	Realiza a negação dos operadores relacionais e de igualdade (por exemplo, substituir < por >).
PIT3	*Remove Conditionals Mutator*	Remove comandos referentes à condição.
PIT4	*Math Mutator*	Realiza a troca entre operadores aritméticos.
PIT5	*Increments Mutator*	Realiza incrementos e decrementos em variáveis locais do código. Também realiza a troca entre decremento e incremento e vice-versa.
PIT6	*Invert Negatives Mutator*	Inverte o valor de números inteiros e números de ponto flutuante.
PIT7	*Inline Constant Mutator*	Modifica constantes de linha. Uma constante em linha é um valor literal atribuído a uma variável não-final.
PIT8	*Return Values Mutator*	Modifica os valores de retorno de chamadas de método, dependendo do tipo de retorno do método (por exemplo, em um retorno do tipo *boolean*, troca *true* por *false*, e vice-versa).
PIT9	*Void Method Calls Mutator*	Remove a chamada de um método do tipo *void*.
PIT10	*Non Void Method Calls Mutator*	Remove chamadas de método para métodos não nulos. Seu valor de retorno é substituído pelo valor padrão Java para o tipo específico.
PIT11	*Constructor Calls Mutator*	Substitui as chamadas de um construtor por valores nulos.

[2] PIT Mutation Testing – http://www.pitest.org
[3] Maven Project– http://maven.apache.org

O arcabouço de teste unitário em Android *Robolectric* foi selecionado, pois ele realiza testes na JVM[4] (do inglês, *Java Virtual Machine*), sem a necessidade da utilização de dispositivos ou emuladores nos testes, sendo tal fator um diferencial para os demais arcabouços de teste para *apps* Android. *Robolectric* atua entre o código da *app* e o código do sistema operacional Android interceptando chamadas e redirecionando a objetos ocultos. Assim, é possível executar testes dentro de uma JVM normal, e com isso executar testes em dispositivos *desktops*, o que facilitado ponto de vista de desempenho (*desktops* possuem mais capacidade do que dispositivos móveis).

3.3. *Apps* Selecionadas para o Estudo

Para este estudo exploratório foram selecionadas duas *apps*. Ambas possuem código aberto e estão disponíveis em repositórios na Web. São elas: *Robolectric Sample* e *Bluetooth Chat*. Tais aplicações foram selecionadas por serem construídas utilizando Maven e possuírem um conjunto de casos de teste projetados por seus desenvolvedores/testadores, pois esses requisitos são necessários para se executar testes de mutação utilizando a ferramenta PIT. As principais características dessas aplicações estão exibidas na Tabela 3.

Tabela 3. Características das *apps* utilizadas no experimento.

Características	*Robolectric Sample*	*Bluetooth Chat*
Objetivo da *App*	Ilustrar o uso do arcabouço Robolectric.	Permitir que 2 dispositivos Android conversem (via texto) por Bluetooth
Número de Classes	27 classes	3 classes
LOC	393	532
#Componentes Android	10	20
# Activities	4	2
#Content Providers	1	1
#Services	1	1
#Broadcast Receivers	1	1
#Intents	3	15
#Casos de Teste	86	40
Objetivos dos Casos de Teste	Testar o fluxo de *Activity* e conteúdo das interfaces	Testar as funcionalidades de Bluetooth em Android
Fonte	Github[5]	Site de Desenvolvedor do Android[6]

3.4. Execução do Estudo

Para a execução deste estudo, foram usadas as configurações recomendadas na documentação da ferramenta PIT por meio de uma *task* do Maven. O procedimento de teste está disponível no site da PIT[7].

Para fins de estudo, as *apps* foram executadas usando todos os operadores em conjunto, com uma única execução para cada operador disponível na ferramenta. O ambiente em que o experimento foi executado tem a seguinte configuração: Sistema operacional Windows 8.1 *Single Language*, Processador Intel I7, 4GB de memória RAM,

[4] The Java® Virtual Machine Specification - https://docs.oracle.com/javase/specs/jvms/se8/html/index.html
[5] https://github.com/robolectric/robolectric-samples
[6] http://developer.android.com/index.html
[7] PIT Quickstart - http://pitest.org/quickstart/maven/

JKD 8, ferramenta PIT versão 1.1.3, Android SDK 2.3.6 (API 10), ferramenta *Robolectric* versão 2.3, *Maven* versão 3.2.1.

4. Análise dos Resultados

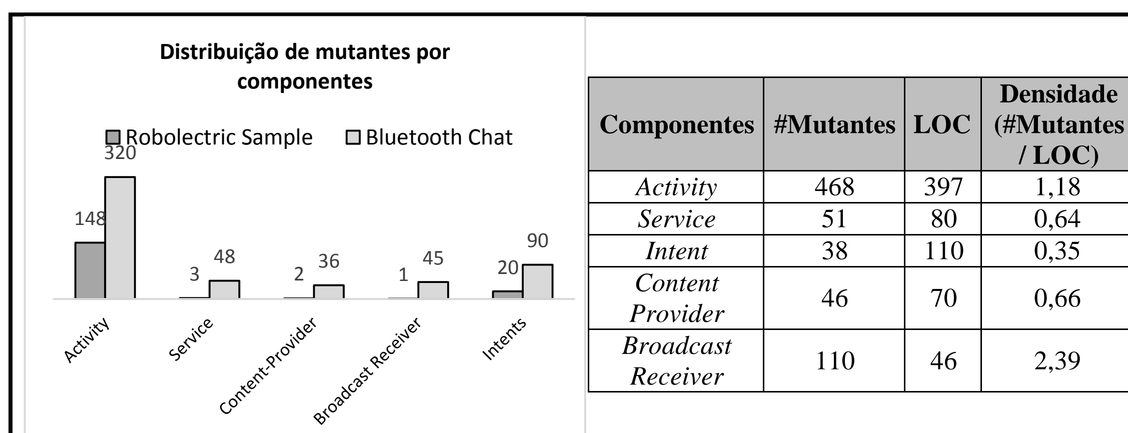
Nesta seção serão apresentados os resultados do estudo e uma discussão com base nos mesmos. O processo de teste de mutação foi executado conforme descrito em Jia e Harman (2011). A Tabela 4 apresenta um resumo dos resultados obtidos.

Tabela 4. Resultados da execução.

Métricas	*Robolectric Sample*	*Bluetooth Chat*
Número de Mutantes Gerados (MG)	614	635
Número de Mutantes Mortos (MM)	595	610
Número de Mutantes Equivalentes (ME)	19	25
Score de mutação ((MM) ÷ MG - ME)	100%	100%
Cobertura de Código (%)	98%	99%
Número de mutantes mortos por *timeout*	522 (88%)	256 (42%)

4.1. Distribuição da Mutação entre os Componentes Android

Um ponto a ser avaliado é a influência dos operadores tradicionais em componentes específicos da plataforma Android (apresentados na Seção 2.3). O resultado dessa avaliação pode ser visualizado na Figura 1, em que na esquerda é visualizada a distribuição dos mutantes por componente/*app* e na direita a análise da densidade dos mutantes por componente para ambas as aplicações (os números de mutantes foram somados para ambas as *apps*).



Componentes	#Mutantes	LOC	Densidade (#Mutantes / LOC)
Activity	468	397	1,18
Service	51	80	0,64
Intent	38	110	0,35
Content Provider	46	70	0,66
Broadcast Receiver	110	46	2,39

Figura 1. Distribuição dos mutantes em componentes Android.

Pode-se observar que *Activity* foi o componente que mais recebeu mutação devido às características das aplicações escolhidas para o estudo, principalmente na *app Robolectric Sample*. Na *app BluetoothChat*, o número de mutantes foi mais diversificado devido à maior utilização dos outros componentes.

Analisando apenas as métricas de escore de mutação e cobertura de código, a execução deste estudo fornece mais um indicativo de que é viável a aplicação do critério de Teste de Mutação no contexto de *apps* Android. Porém, ao analisar os mutantes

gerados no decorrer do processo, foram levantados alguns contrapontos que são passíveis de discussão.

4.2. Adequação dos Operadores Java em Android

Depois de verificados os indicadores, foi realizada uma análise de cada mutante gerado/morto. Verificou-se que muitos mutantes foram mortos por *timeout*. Analisando as possíveis causas, foi verificado que alguns operadores de mutação Java implementados na ferramenta PIT alteram partes trechos de código que influenciam no fluxo de execução em aplicações Android, criando laços infinitos.

Exemplos destas mudanças são: alteração de valor de retorno em funções que são essenciais para o fluxo de navegação em Android, a exclusão de chamadas a construtores de classes essenciais para renderização das interfaces (ex: a chamada *new Activity()*) ou a exclusão de chamada a procedimentos responsáveis pela renderização de conteúdo nas telas das *apps*, como por exemplo o procedimento *setContentView* (Figura 2).

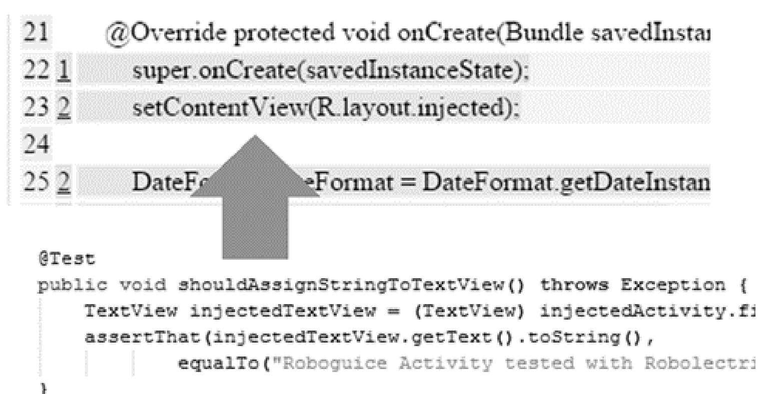


Figura 2. Exemplo de um mutante sendo exercitado.

No exemplo da Figura 2, o caso de teste foi projetado para identificar um elemento gráfico em uma tela da *app*. No entanto, o mutante PIT9 (ver Tabela 2) removeu a chamada para a função ***"setContentView()"*** (linha 23), interferindo diretamente no ciclo de vida da *Activity*. Isso causou um *timeout* quando exercitado pelo caso de teste descrito na parte inferior da Figura 2.

Uma reflexão possível é de que isso pode ser um indício de que os operadores Java necessitam de uma adaptação na modificação sintática que realizam para serem utilizados na plataforma Android. Assim, isto possibilitaria a geração de mutantes mais consistentes, que possam refletir melhor os erros inseridos na plataforma Android. Contudo, antes são necessários estudos futuros para verificar se tal problema é referente apenas ao componente *Activity*, por exemplo, ou se propaga para os demais componentes.

4.3. Adequação de Ferramenta de Teste de Mutação Java para Android

A partir do que foi descrito anteriormente, levantou-se outra limitação por conta da ferramenta escolhida para o estudo, pois ela realiza a mutação em *bytecode*. Muitas vezes, as alterações são idealizadas para o código fonte, mas quando são executadas no *bytecode* acabam não refletindo a mutação projetada. Isso pode representar um indício de que o nível no qual a mutação é aplicada pode não ser ideal para a plataforma Android.

Apesar de ser possível criar e executar mutantes com a ferramenta PIT em *apps*

Android, o grande número de mutantes que morrem por *timeout* chama a atenção. Uma contribuição seria executar um estudo similar utilizando uma ferramenta de mutação em nível de código fonte, afim de verificar se existem diferenças na execução do critério, ou até mesmo em outra ferramenta de mutação em *bytecode* a fim de confirmar os pontos citados acima.

Por fim, outro ponto de discussão do trabalho é a adequabilidade no uso de um ambiente de testes de mutação em Java para a plataforma Android, em relação ao processo de compilação/construção de um projeto Android e execução dos testes em um arcabouço de testes unitários. Neste estudo, foi utilizado o arcabouço *Robolectric*, pois ele executa testes unitários de *apps* em Android por meio da máquina virtual Java. O fato do arcabouço realizar os testes a partir da JVM pode não permitir que, ao testar os mutantes, os mesmos possuam acesso a características específicas providas pelo dispositivo.

Isso seria um indício de que, além de operadores Java adaptados, faz-se necessário também um apoio ferramental específico para a plataforma Android que possibilite não apenas a implementação dos operadores novos/adaptados, mas ainda a compilação e construção de um projeto e execução dos casos de teste em arcabouços de teste unitário para Android. Além disso, outra possível contribuição a partir deste estudo seria investigar formas de realização de testes de mutação que executem em dispositivos reais, possibilitando o acesso a recursos específicos providos por eles.

5. Ameaças à validade

- **Validade interna**: validade interna está relacionada com o controle do processo de experimentação para recolher dados analisados. Neste estudo foi utilizada a ferramenta PIT para a geração de mutantes. Obviamente, a baixa quantidade de operadores de mutação implementado pela ferramenta com respeito aos descritos em Ahmed *et al.* (2010) não permite resultados mais conclusivos.
- **Validade externa**: validade externa refere-se ao risco de generalizar os resultados obtidos a partir do estudo apresentado. Neste estudo, foram utilizados apenas duas *apps* com um total de 925 linhas de código. Outra limitação é devido à relevância das *apps* que foram usados no estudo. Como são *apps* com um propósito didático, não se pode generalizar os resultados para situações reais ou industriais, por exemplo. Outra limitação dos programas escolhidos foi que, para alguns operadores, nenhum mutante foi gerado. Isto ocorre quando o programa não contém a estrutura sintática exigida pelo operador para gerar mutantes. Finalmente, o conjunto de teste utilizado no experimento já compunha ambas as *apps*, o que poderia representar uma limitação na investigação de critério de TM.
- **Validade de *Constructo***: o risco de construção deste trabalho reside no processo utilizado no estudo. Para isso, foi investido tempo no estudo do critério, na tentativa de minimizar os impactos no estudo.

6. Conclusões e Trabalhos Futuros

Neste artigo, foi descrito um estudo exploratório que avaliou a utilização de 11 operadores de mutação Java em *apps* Android. Os dados foram discutidos a fim de tentar elucidar a possibilidade da utilização de operadores de mutação Java, consequentemente do critério de TM no contexto de testes para *apps* Android.

Assim, os resultados indicam, que utilizando os operadores implementados pela

ferramenta PIT, foi possível aplicar teste de mutação em aplicações Android, visto que mutantes foram gerados em ambas as aplicações e alcançando-se uma alta cobertura de código para ambas as *apps*. No entanto, a maioria dos mutantes foram mortos por *timeout*, porque as mudanças no código afetado o fluxo de execução em *apps* Android.

Em relação à diversidade dos mutantes gerados entre os componentes de Android, em termos absolutos, a maioria deles concentra-se no componente de *Activity*, visto que é o principal componente utilizado no desenvolvimento de aplicações Android. Uma execução com um número maior de *apps* é necessária para obter resultados mais conclusivos.

Como conclusão, os indícios apontados no estudo evidenciam a necessidade de uma investigação mais apurada nos operadores de mutação da linguagem Java, tendo em vista uma possível adaptação dos mesmos, visando uma adequação ainda maior para o contexto Android, a fim de reduzir, principalmente, o número de mutantes mortos por *timeout*. Uma solução possível pode ser vista em Salva e Zafimiharisoa (2013), no qual os autores trabalham com Semeadura de Erros em componentes *Intent*.

Com isso, trabalhos futuros podem ser projetados a fim de alcançar resultados mais conclusivos. Em primeiro lugar, existe uma necessidade de obter dados de mutação de outros projetos Android. Para isso, pretende-se executar um outro estudo utilizando um número maior de *apps* e uma ferramenta de teste de mutação mais robusta, como por exemplo a MuJava (Ma *et al.*, 2006).

Como trabalho em andamento, a ferramenta MuJava está sendo adaptada para a execução do TM em aplicações Android, tendo em vista as questões e limitações apresentadas inicialmente por Garousi *et al.* (2013) e confirmada neste artigo. Além disso, como sugerido acima, a manutenção dos operadores Java, ou até mesmo a criação de novos operadores para o contexto de *apps* Android torna-se um desafio interessante. Acredita-se que isso pode ampliar a qualidade e consistência dos mutantes gerados e consequentemente, prover uma melhor qualidade para os casos de teste para *apps*.

7. Agradecimentos

Agradecemos ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e ao Instituto de Desenvolvimento Tecnológico (INDT) pelo apoio financeiro fornecido a essa pesquisa.

8. Referências

Agrawal, H. and DeMillo, R.A. and Hathaway, B. and Hsu, W. and, Krauser, E.W. and Martin, R.J. and Mathur, A.P. and Spafford, E. (1989) "Design of Mutant Operators for the C Programming Language," Technical Report SERC-TR-41-P, Purdue University.

Ahmed, Z. and Zahoor, M. and Younas, I. (2010) "Mutation operators for object-oriented systems: A survey", International Conference on Computer and Automated Engineering (ICCAE 2010), vol. 2, pp. 614–618.

Android Developers Documentation (2014). Disponível em http://developer.android.com/guide/index.html. Acessado em 22/04/2015.

Avancini, A. and Ceccato, M. (2013) "Security testing of the communication among Android applications", 8th International Workshop on Automated Software Test,

AST, pp. 57–63.

Budd, T. A. and Gopal, A. S. (1985) “Program Testing by Specification Mutation,” Computer Languages, vol. 10, no. 1, pp. 63-73.

Danicic, S. (2007) “Lava: a system for mutation testing of Java programs”, <http://www.doc.gold.ac.uk/~mas01sd/mutants/> (acessado em Junho de 2015).

Delahaye, M. and Du Bousquet, L. (2013) “A comparison of mutation analysis tools for java”, Proceedings of the International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA), pp. 187–195.

Demillo, R. A. and Lipton, R. J. and Sayward, F. G. (1978) “Hints on Test Data Selection”, Computer (Long. Beach. Calif). vol. 11, pp. 34–41.

Derezinska, A. (2006) “Quality Assessment of Mutation Operators Dedicated for C# Programs,” Proc. Sixth International Conference on Quality Software.

Garousi, V. and Kotchorek, R. and Smith, M. (2013) “Test Cost-Effectiveness and Defect Density: A Case Study on the Android Platform”. Advances in Computers, vol. 89, pp 163–206.

Jia, Y. and Harman, M. (2011) “An analysis and survey of the development of mutation testing”, IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678.

Kim, S.W. and Clark, J. and McDermid, J. (1999) “The Rigorous Generation of Java Mutation Operators Using HAZOP”, Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA’99), pp. 1–20.

Lecheta, R. R. (2011) Google Android: Aprenda a criar aplicações para dispositivos moveis com o Adroid SDK, 2a ed. Santos - SP.

Lu, L. and Hong, Y. and Huang, Y. and Su, K. and Yan, Y. (2012) “Activity page based functional test automation for android application”, 3rd World Congress on Software Engineering (WCSE 2012), pp. 37–40.

Ma, Y. and Offutt, J. and Kwon, Y. "Mujava: a mutation system for java". Proceedings of the 28th International Conference on Software Engineering (ICSE’06), 2006, pp. 827-830.

Muccini, H. and Francesco, A. and Esposito, P. (2012) “Software testing of mobile applications: Challenges and future research directions”, International Workshop on Automated Software Test (AST), pp. 29–35.

Salva, S. and Zafimiharisoa, S. R. (2013) “Data vulnerability detection by security testing for Android applications”, Conference on Information Security for South Africa (ISSA 2013), pp. 1–8.

Van der Meulen, R. and Rivera, J. (2014) “Gartner Says by 2017, Mobile Users Will Provide Personalized Data Streams to More Than 100 Apps and Services Every Day”, Gartner Newsroom.

Aplicação de Propriedades de Weyuker, Parrish e Zweben a Critérios de Adequação

Diogo M. de Freitas[1], **Plínio S. Leitão-Júnior**[2], **Auri M. R. Vincenzi**[3]

[1]Escola de Engenharia Elétrica, Mecânica e de Computação
Universidade Federal de Goiás (UFG) – Goiânia, GO – Brasil

[2]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia, GO – Brasil

[3]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
São Carlos, SP – Brasil

diogom42@gmail.com, plinio@inf.ufg.br, auri@dc.ufscar.br

Abstract. *In 1986 Weyuker introduced a set of properties seeking to synthesize adequacy criteria's abstract and intuitive features. In 1991 this set was redefined and had dependencies identified by Parrish and Zweben. This paper applies the properties sets to structural, functional and falt-based criteria, aiming the its evaluation and comparison. The results were the verification of the dependencies between the properties and the analysis of some criteria according with the properties. Finally, it was observed that the combined use of the two sets of properties gives greater abilities to the comparison between adequacy criteria.*

Resumo. *Em 1986 Weyuker introduziu um conjunto de propriedades buscando sintetizar características abstratas e intuitivas de critérios de adequação. Em 1991 este conjunto foi redefinido e teve dependências apontadas por Parrish e Zweben. Este artigo aplica os conjuntos de propriedades a critérios de teste estruturais, funcionais e baseados em defeitos, objetivando-se sua avaliação e comparação. Os resultados obtidos foram a verificação das dependências entre as propriedades e a análise de alguns critérios conforme as propriedades. Por fim, constatou-se que a utilização conjunta dos dois conjuntos de propriedades apresenta maior utilidade para comparação de critérios.*

1. Introdução

Teste de software é uma prática que envolve a execução do software para dados de entrada e representa uma disciplina importante para a sua qualidade. Pesquisas mostram que pelo menos 50% do custo total do software deve-se a atividade de teste [Sharma et al. 2014].

Critérios de teste estabelecem requisitos para o teste, sendo usados para avaliar o cumprimento desses requisitos (critério de adequação) ou para guiar a geração de dados de teste (critério de seleção). Num período em que o teste de software recebia os primeiros esforços para sua formalização, [Goodenough e Gerhart 1975] apresentou uma teoria de

Diogo M. de Freitas é bolsista PIBIC CNPq - Brasil

critérios de adequação e provou um teorema que afirma que um conjunto de teste é capaz de indicar a ausência de defeitos em um programa. Essa teoria teve papel fundamental na definição posterior de novos critérios de seleção e foi composta de conceitos como *critério confiável* (todos os conjuntos de testes adequados revelam algum defeito ou nenhum conjunto revela qualquer defeito) e *critério válido* (para cada defeito no programa existe algum conjunto de teste que o revela).

Nesse cenário, Weyuker introduziu um conjunto de "axiomas" (propriedades), que busca sistematizar características abstratas e intuitivas sobre critérios de adequação. Foram apresentadas oito propriedades [Weyuker 1986] e posteriormente três novas propriedades foram adicionadas [Weyuker 1988].

[Parrish e Zweben 1991] estendeu os estudos de Weyuker, generalizando as propriedades para qualquer tipo de critério, além de incluir novas formalizações e uma análise teórica de dependências entre as propriedades. Os autores também apresentaram um conjunto reduzido de propriedades sem dependência mútua (derivado do conjunto proposto por Weyuker), tornando-o mais próximo de uma teoria axiomática.

O presente artigo lida com os conceitos desses autores, para responder às questões: (Q1) qual a análise de critérios de testes funcionais, estruturais e baseadas em defeitos, com respeito às propriedades de Weyuker, Parrish e Zweben? e (Q2) que propriedades distinguem os critérios de teste em relação às suas características?

O objetivo primário deste trabalho é responder às questões de pesquisa acima. Outros objetivos, com respeito às propriedades de Weyuker e de Parrish e Zweben, são: abstrair e apresentar os principais conceitos pertinentes às propriedades; e avaliar um conjunto de critérios de teste com respeito às propriedades.

O texto organiza-se da seguinte forma. A Seção 2 apresenta conceitos e definições fundamentais à Seção 3, que traz as propriedades apresentadas em [Weyuker 1986] e [Weyuker 1988] com modificações feitas em [Parrish e Zweben 1991] e o conjunto reduzido de propriedades apresentado em [Parrish e Zweben 1991]. As demais seções apresentam as contribuições do presente artigo.

Para a aplicação dos dois conjuntos de propriedades, foram selecionados seis critérios de diversas naturezas. Na Seção 4 os critérios são apresentados e avaliados a luz de cada propriedade. A Seção 5 traz uma análise dos resultados da aplicação das propriedades aos critérios e a Seção 6 conclui o artigo.

2. Conceitos e Definições

A presente seção introduz conceitos e definições utilizados neste texto. *Dado de teste* denota o dado usado para executar o software durante o teste. Uma *especificação* é uma função que define o que deve ser calculado por um programa, ou seja, quais devem ser suas saídas. O *domínio de uma especificação* é o conjunto de valores para os quais a especificação é definida. O *domínio de um programa* é o conjunto de valores para os quais o programa é definido, isto é, é o conjunto de dados de entrada para os quais o programa termina normalmente. O conjunto dos pontos que podem ser representados por uma linguagem L é simbolizado por *rep(L)*, enquanto *rep(S)* simboliza o conjunto dos pontos que podem ser representados por uma especificação S.

Qualquer segmento contínuo de comandos de um programa é dito um *componente*

deste. *Composição de programas* é obtida substituindo os comandos de saída do primeiro programa por uma versão do segundo sem comandos de entrada. Dois programas são ditos *equivalentes* se e somente se ambos apresentam a mesmas saídas para todas as entradas possíveis. Dois programas tem a *mesma estrutura* se um pode ser transformado no outro aplicando-se qualquer uma das seguintes regras, quantas vezes for necessário: (i) substituir um operador relacional em um predicado; (ii) substituir uma constante em um predicado; (iii) substituir uma constante em um comando de atribuição; e (iv) substituir um operador aritmético em um comando de atribuição. Se dois programas são equivalentes e de mesma estrutura ao mesmo tempo, apenas com possíveis diferenças em nomes de variáveis, eles são *renomeações* um do outro.

Cobertura de comandos ocorre quando cada comando acessível de um programa é executado pelo menos uma vez por algum dado do conjunto de teste. *Cobertura de definições* ocorre quando para cada definição de uma variável de um programa (o ponto em que esta tem seu valor alterado por uma atribuição), o caminho entre esta definição e algum uso posterior da variável (em operações ou predicados) é executado. Este caminho deve ser livre de definições, isto é, não deve redefinir o valor da variável, e alcançável.

Um conjunto de teste T é dito *exaustivo* quando este abrange todos os pontos representáveis do domínio da especificação S do programa, $rep(S) \subseteq T$. Programas que requerem teste exaustivo em um domínio finito rep(L) para a cobertura de comandos são chamados *programas exaustivos* (notação E). Os programas exaustivos para a cobertura de definições são referidos pela notação E'.

Um critério costuma relacionar conjuntos de teste a programas e/ou especificações. Um critério é uma função que para um programa e uma especificação produz conjuntos de teste que o satisfaz. *Critérios baseados em programas* dependem da estrutura do programa e nem sempre têm os mesmos conjuntos de teste adequados para todos os programas. *Critérios independentes de programas* têm sempre os mesmos conjuntos de testes adequados para todos os programas. *Critérios baseados em especificações* dependem da especificação do programa e nem sempre têm os mesmos conjuntos de teste adequados para todas as especificações. *Critérios independentes de especificações* têm sempre os mesmos conjuntos de testes adequados para todas as especificações de programas.

3. Propriedades Sobre Critérios de Adequação

As propriedades introduzidas em [Weyuker 1986] e [Weyuker 1988] foram reformuladas e formalizadas em [Parrish e Zweben 1991] de modo a torná-las generalizadas. Tais mudanças tornou-as aplicáveis a qualquer um dos quatro tipos de critérios classificados anteriormente quanto às dependências de especificações e de programas, visto que o conjunto original aplica-se somente a critérios independentes de especificações. Este conjunto é chamado de *Conjunto de Propriedades Generalizadas* e contém onze propriedades.

Propondo-se ser uma teoria axiomática, um conjunto pequeno de propriedades independentes com definições simples é desejável. Portanto, [Parrish e Zweben 1991] redefiniu e eliminou algumas das Propriedades Generalizadas resultando num conjunto reduzido livre de dependências. Este é o *Conjunto Reduzido de Propriedades* e é composto por cinco das onze propriedades de Weyuker e duas propriedades redefinidas.

3.1. Propriedades Generalizadas

A seguir estão definidos os enunciados das Propriedades Generalizadas bem como suas formalizações, onde (P, S) refere-se ao programa P e/ou especificação S apropriada, dependendo das dependências de programas e especificações do critério.

Aplicabilidade (APP): Para cada (P, S), existe um conjunto de teste adequado.

Aplicabilidade Não Exaustiva (NA): Existe um (P, S) e um conjunto de teste T tal que (P, S) é adequadamente testado por T, e T não é um conjunto de teste exaustivo.

Monotonicidade (MON): Para todo (P, S) e conjuntos de teste T e T', se T é adequado para (P, S), e $T \subseteq T'$ então T' é adequado para (P, S).

Conjunto Vazio Inadequado (IES): O conjunto vazio não é adequado para qualquer (P, S).

Antiextensionalidade (AE): Existem programas P e Q tais que P $\equiv$ Q, e existe um conjunto de teste T e especificação S, tal que T é adequado para (P, S), mas não é adequado para (Q, S).

Mudança Múltipla Geral (GMC): Existem programas P e Q, que são do mesmo formato (same shape), e um conjunto de teste T e especificação S tal que T é adequado para (P, S), mas não é adequado para (Q, S).

Antidecomposição (AD): Existe um programa P com componente Q, especificações S e S' e conjunto de teste T tal que T é adequado para (P, S), T' é o conjunto de vetores de valores que as variáveis podem assumir na entrada de Q para algum t em T, e T' não é adequado para (Q, S').

Anticomposição (AC): Existem programas P e Q, especificações S e S', e conjunto de teste T tais que T é adequado para (P, S) e o conjunto das saídas de P para T é adequado para (Q, S'), mas T não é adequado para a (P; Q, S).

Renomeações (REN): Sejam P e Q renomeações, para qualquer conjunto de teste T, T é adequado para (P, S) se e somente se T é adequado para (Q, S).

Complexidade (COMP): Existe uma especificação S tal que, para cada n, $\|rep(L)\| \geq n > 0$, existe um programa P tal que (P, S) é adequadamente testado por um conjunto de teste de tamanho n, mas não é adequadamente testado por um conjunto de teste de tamanho n – 1.

Cobertura de Comandos (SC): Para todo (P, S) e conjunto de teste T, se T é adequado para (P, S), então T cobre todos os comandos de P.

Observa-se que as propriedades apresentadas podem ser universais ou existenciais, as *propriedades universais* representam características que devem ser satisfeitas para todos os programas ou especificações enquanto as *propriedades existenciais* representam características que devem ser satisfeitas por pelo menos um programa e/ou especificação. As propriedades Aplicabilidade Não Exaustiva, Antiextensionalidade, Mudança Múltipla Geral, Antidecomposição e Anticomposição são existenciais e, portanto, representam uma restrição mais fraca em relação às demais propriedades, que são universais.

[Parrish e Zweben 1991] demonstra ainda teoremas com as seguintes dependências entre as propriedades generalizadas: (1) $APP \wedge IES \rightarrow AD$; (2) $SC \rightarrow IES$; (3)

$SC \wedge NA \rightarrow AE$; (4) $SC \wedge APP \rightarrow AD$; (5) $SC \wedge COMP \rightarrow AE$;

Das propriedades apresentadas, apenas as quatro primeiras são apontadas como interessantes para quaisquer tipos de critérios, enquanto as demais são especializadas para critérios baseados em programas [Weyuker 1986]. Para as demais propriedades existem implicações que partem das seguintes propriedades:

Não Inaplicabilidade (NI): Algum programa pode ser adequadamente testado com respeito a alguma especificação.

Não Exaustividade com Respeito à Linguagem de Programação (NAL): Algum programa pode ser adequadamente testado com respeito a alguma especificação sem exigir que todos os pontos representáveis da linguagem sejam selecionados.

Observa-se que quase todos os critérios independentes de programas satisfazem NI e NAL. As implicações são: (1) Se um critério C é independente de programas, então C não satisfaz AE, GMC, AC e COMP e satisfaz a REN; (2) Se um critério C é independente de programas e satisfaz NI e IES, então C satisfaz a AD; e (3) Se um critério C é independente de programas e satisfaz NAL, então C não satisfaz a SC.

3.2. Conjunto Reduzido de Propriedades

O Conjunto Reduzido de Propriedades surge das Propriedades Generalizadas, [Parrish e Zweben 1991] buscava um conjunto livre de dependências, reduzido e de apenas propriedades universais. Assim, foi apresentada uma nova propriedade de cobertura de definições, que substitui SC, e a revisão da propriedade Aplicabilidade Não Exaustiva conforme programas exaustivos E'.

Cobertura de Definições (DC): Para qualquer programa, especificação e conjunto de teste, se T é adequado à (P, S), então T causa a cobertura de um caminho limpo de definições para algum uso alcançável para toda definição alcançável de P.

Aplicabilidade Não Exaustiva Revisada (RNA'): Para todo programa não contido em E', existe uma especificação para qual o programa é testado adequadamente e não exaustivamente.

Partindo da revisão de Aplicabilidade Não Exaustiva (RNA') e da substituição de Cobertura de Comandos (por DC) tem-se as seguintes dependências: (1) $DC \wedge RNA' \rightarrow AE$; (2) $DC \wedge RNA' \rightarrow GMC$; (3) $DC \wedge RNA' \rightarrow AD$; e (4) $DC \wedge RNA' \rightarrow AC$.

Das 11 propriedades, Antiextensionalidade, Mudança múltipla Geral, antidecomposição e Anticomposição podem ser eliminadas, já que são derivações de RNA' e DC. O conjunto resultante ficou reduzido a sete propriedades (APP, RNA', MON, IES, REN, COMP, DC), todas universais e sem dependências entre si.

4. Estudos de Caso: Análise de Critérios de Teste

Esta seção enriquece os conceitos dos artigos, agregando valor pela aplicação das propriedades tratadas na Seção 3 às técnicas de teste: funcional, estrutural e baseada em defeitos. Para tal, foram selecionados os critérios Todos os Comandos [Myers et al. 2004], Todos os Caminhos [Myers et al. 2004], Todas as Definições [Rapps e Weyuker 1982], Classes de Equivalência [Myers et al. 2004], Análise do Valor Limite [Myers et al. 2004] e Teste de Mutação [DeMillo et al. 1978].

Um conjunto de teste satisfaz *Todos os Comandos* com respeito a um programa se este provoca a execução de todos os comandos do programa. Para que um conjunto de teste seja adequado com respeito a *Todos os Caminhos*, todos os caminhos possíveis do programa devem ser percorridos. Se um conjunto de teste cobre todas as definições de um programa, alcançáveis ou não, então *Todas as Definições* é satisfeito. Estes critérios são independentes de especificação e dependentes de programas.

O objetivo das classes de equivalência é dividir o domínio da especificação em partições de modo que dados para os quais o programa se comporta da mesma forma possam ser testados conjuntamente. Um conjunto de teste satisfaz *Classes de Equivalência* com respeito a algum programa e especificação se cada classe de equivalência contém pelo menos um dado do conjunto. Se o conjunto contém dados das fronteiras de cada classe de equivalência, então *Análise do Valor Limite* também é satisfeita. Ambos os critérios são dependentes de especificação e independentes de programas.

A um programa são aplicadas diversas modificações pontuais (mutações) e cada modificação deriva uma versão modificada do programa original (mutante). Um conjunto de teste é adequado ao *Teste de Mutação*, se para cada mutante não equivalente do programa contém um dado cujas saídas do mutante e do programa original são distintas (mata o mutante). Teste de Mutação é um critério dependente de programas e de especificações.

O restante desta seção aplica os conjuntos de propriedades estudados na Seção 3 aos seis critérios apresentados. Reitera-se que [Parrish e Zweben 1991] não contempla a aplicação de propriedades a critérios de teste, sendo esta, uma contribuição deste trabalho.

4.1. Aplicação das Propriedades Generalizadas

Os critérios Todos os Comandos, Todos os Caminhos ou Todas as Definições não produzem conjuntos de testes adequados para programas com comandos ou definições não acessíveis. Para os critérios Classes de Equivalência, Análise do Valor Limite ou Teste de Mutação é sempre possível compor um conjunto de testes adequado, já que o número de classes de equivalências e mutantes é sempre finito, não são criadas classes de equivalência vazias ou que não podem ser testadas e mutantes que não podem ser mortos são considerados equivalentes. Assim sendo, *Aplicabilidade* é satisfeita por Classes de Equivalência, Análise do Valor Limite e Teste de Mutação e falha para os demais critérios.

Dentre os critérios sob análise, nenhum requer sempre teste exaustivo para ser adequado, isto é, para todos os critérios existe algum programa e especificação que podem ser testados adequadamente por um conjunto não exaustivo. Assim sendo, *Aplicabilidade Não Exaustiva* satisfaz aos seis critérios.

Qualquer um dos critérios aceita adição de dados extras ao conjunto de teste, sem perder as características que o torna adequado. *Monotonicidade* é, portanto, satisfeita pelos seis critérios.

Assumindo que todo programa tem pelo menos uma variável de entrada, o conjunto vazio não é adequado a qualquer um dos critérios analisados. Tal conjunto não executaria qualquer programa ou incluiria dados de qualquer classe de equivalência. *Conjunto Vazio Inadequado* satisfaz os seis critérios.

Como os critérios Todos os Comandos, Todos os Caminhos, Todas as Definições e Teste de Mutação consideram o funcionamento interno dos programas, programas equi-

valentes ou de mesma estrutura que realizam operações ou desvios diferenciados para um mesmo dado podem não ser testados adequadamente com respeito a algum destes critérios por um mesmo conjunto de teste. Já os critérios Classes de Equivalência e Análise do Valor Limite, que independem da estrutura interna do programa, tratam dois programas equivalentes com uma mesma especificação da mesma forma e geram os mesmos conjuntos de teste. *Antiextensionalidade* e *Mudança Múltipla Geral* são, portanto, satisfeitas pelos critérios Todos os Comandos, Todos os Caminhos, Todas as Definições e Teste de Mutação e falham para os critérios Classes de Equivalência e Análise do Valor Limite.

Todos os Comandos, Todos os Caminhos e Todas as Definições testam seus componentes com os mesmos parâmetros, diferentemente de Classes de Equivalência e Análise do Valor Limite que, como programa e seu componente possuem especificações diferentes, derivam conjuntos de testes que podem não ser ao mesmo tempo adequados ao programa e componente. Para Teste de Mutação, programas com componentes inexecutáveis podem não compartilhar testes adequados com estes componentes. *Antidecomposição* é satisfeita por Classes de Equivalência, Análise do Valor Limite e Teste de Mutação e falha para critérios Todos os Comandos, Todos os Caminhos e Todas as Definições.

Se um conjunto de teste satisfaz Todos os Comandos e Todas as Definições com respeito a um programa e a saída deste satisfaz os mesmos critérios com respeito a outro programa, então este conjunto é adequado à composição dos programas. O mesmo não acontece para o critério Todos os Caminhos, pois o conjunto de teste pode não percorrer todas as combinações de caminhos criadas com a composição dos dois programas. Como a propriedade considera que a composição de dois programas é combinada com a especificação do primeiro programa, um conjunto de teste adequado ao primeiro programa para Classes de Equivalência e Análise do Valor Limite é adequado à composição. Para Teste de Mutação, consideram-se dois programas P e Q, um conjunto T adequado à P de modo que suas saídas sejam adequadas à Q e um mutante P' não equivalente de P que é morto por T e tem saída adequada à Q. A composição de P' e Q é um mutante não equivalente da composição de P e Q que não é morto por T, pois Q não é capaz de diferenciar as saídas de P e P'. Dessa forma, *Anticomposição* é satisfeita por Todos os Caminhos e Teste de Mutação e falha para os demais critérios.

Qualquer um dos critérios trata renomeações de programas da mesma forma, ou seja, não comprometem as características que o tornam o teste adequado. *Renomeações* é, portanto, satisfeita pelos seis critérios.

Através de incrementos de complexidade é possível criar programas para cada n no intervalo $\|rep(L)\| \geq n > 0$ que, combinados a uma especificação fixa, sejam adequadamente testados com respeito aos critérios Todos os Comandos, Todos os Caminhos, Todas as Definições ou Teste de Mutação por conjuntos de teste de tamanho mínimo igual a n. Entretanto, o mesmo não pode ser feito para os critérios Classes de Equivalência e Análise do Valor Limite, pois a mesma especificação faz com que qualquer programa produza os mesmos conjuntos de teste. Assim, *Complexidade* é satisfeita por Todos os Comandos, Todos os Caminhos, Todas as Definições ou Teste de Mutação e falha para Classes de Equivalência e Análise do Valor Limite.

Todos os Comandos, Todos os Caminhos e Teste de Mutação garantem a cobertura de comandos acessíveis ao contrário de Todas as Definições, Classes de Equivalência e

Análise do Valor Limite. Portanto a propriedade *Cobertura de Comandos* é satisfeita por Todos os Comandos, Todos os Caminhos e Teste de Mutação e falha para Todas as Definições, Classes de Equivalência e Análise do Valor Limite.

4.2. Aplicação do Conjunto Reduzido de Propriedades

Das propriedades do conjunto reduzido resta avaliar as propriedades Aplicabilidade Não Exaustiva Revisada e Cobertura de Definições, pois as demais propriedades também fazem parte do Conjunto de Propriedades Generalizadas, que já foi analisado.

Os critérios independentes de especificação que não satisfazem a propriedade Aplicabilidade não podem satisfazer Aplicabilidade Não Exaustiva Revisada, pois esta também exige que exista um teste adequado para qualquer programa. Os demais critérios, Classes de Equivalência, Análise do Valor Limite e Teste de Mutação, permitem o teste não exaustivo e adequado para qualquer programa não contido em E' e alguma especificação. Assim, *Aplicabilidade Não Exaustiva Revisada* falha para os critérios Todos os Comandos, Todos os Caminhos e Todas as Definições e é satisfeita pelos critérios Classes de Equivalência, Análise do Valor Limite e Teste de Mutação.

Cobertura de definições é obtida naturalmente por Todas as Definições, Todos os Caminhos e Teste de Mutação (pela morte de todos os mutantes com alterações em definições). Os demais critérios não garantem cobertura de definições. Logo, *Cobertura de Definições* é satisfeita por Todas as Definições, Todos os Caminhos e Teste de Mutação e falha para Todos os Comandos, Classes de Equivalência e Análise do Valor Limite.

5. Análise de Resultados

Partindo dos resultados obtidos da aplicação das propriedades aos critérios (Tabela 1), é possível apurar as dependências expostas em [Parrish e Zweben 1991] para Propriedades Generalizadas e as dependências que resultam no Conjunto Reduzido de Propriedades. As dependências são representadas utilizando-se o conectivo implica, da lógica proposicional. Para verificá-las, portanto, são desejadas ocorrências de dependências do tipo $Verdadeiro \rightarrow Verdadeiro$, que validam a implicação, ou do tipo $Verdadeiro \rightarrow Falso$, que invalidam a implicação. As demais combinações de resultados são inconclusivas, pois $Falso \rightarrow Verdadeiro$ e $Verdadeiro \rightarrow Verdadeiro$ são verdadeiras.

Das dependências do Conjunto das Propriedades Generalizadas, a Dependência 1 ($APP \wedge IES \rightarrow AD$) é atendida para Classes de Equivalência, Análise do Valor Limite e Teste de Mutação, em que APP, IES e, consequentemente, AD são satisfeitas. A aplicação das propriedades aos critérios Todos os Comandos, Todos os Caminhos e Teste de Mutação permite a verificação da Dependência 2 ($SC \rightarrow IES$), da Dependência 3 ($SC \wedge NA \rightarrow AE$) e da Dependência 5 ($SC \wedge COMP \rightarrow AE$), em que SC, NA, COMP, IES e AE são satisfeitas. A Dependência 4 ($SC \wedge APP \rightarrow AD$) somente é observada para Teste de Mutação, que satisfaz todas as propriedades.

Para os critérios independentes de programas é observado que, conforme a Implicação 1, AE, GMC, AC e COMP não são satisfeitas enquanto REN é satisfeita. Ainda, como as propriedades NAL e NI são satisfeitas por ambos os critérios, nota-se que AD é satisfeita, validando a Implicação 2, e SC não é satisfeita, validando a Implicação 3. Ou seja, pelas implicações para critérios independentes de programas, a aplicação destas propriedades é dispensável, pois os resultados já são conhecidos.

Tabela 1. Resumo da aplicação das propriedades aos critérios

	Todos os Comandos	Todos os Caminhos	Todas as Definições	Classes de Equivalência	Análise do Valor Limite	Teste de Mutação
APP	Não	Não	Não	Sim	Sim	Sim
NA	Sim	Sim	Sim	Sim	Sim	Sim
MON	Sim	Sim	Sim	Sim	Sim	Sim
IES	Sim	Sim	Sim	Sim	Sim	Sim
AE	Sim	Sim	Sim	Não	Não	Sim
GMC	Sim	Sim	Sim	Não	Não	Sim
AD	Não	Não	Não	Sim	Sim	Sim
AC	Não	Sim	Não	Não	Não	Sim
REN	Sim	Sim	Sim	Sim	Sim	Sim
COMP	Sim	Sim	Sim	Não	Não	Sim
SC	Sim	Sim	Não	Não	Não	Sim
RNA'	Não	Não	Não	Sim	Sim	Sim
DC	Não	Sim	Sim	Não	Não	Sim

As dependências do Conjunto Reduzido de Propriedades puderam ser verificadas somente pelo Teste de Mutação, que é o único que satisfaz DC e RNA' simultaneamente.

Observa-se que nenhum dos critérios contraria qualquer dependência e que os critérios que não atendem a alguma dependência não são impedidos de satisfazerem as propriedades resultantes da mesma. Entretanto, é constatado que critérios independentes de especificações que não satisfazem APP não podem satisfazer RNA'. Esta é uma dependência do Conjunto Reduzido de Propriedades que não foi apontada pelos autores.

Comparar os critérios apenas pelo Conjunto Reduzido de Propriedades para Todos os Caminhos e Todas as Definições, não permite distinção entre os critérios ou alguma conclusão de que um critério é melhor que o outro. Entretanto, ao aplicar o Conjunto de Propriedades Generalizadas observa-se que Todas as Definições não satisfaz duas propriedades que Todos os Caminhos satisfaz (AC e SC), evidenciando que, Todos os Caminhos é um critério mais abrangente que Todas as Definições.

A exclusão das propriedades AE, GMC, AD e AC no Conjunto Reduzido de Propriedades se mostra interessante para critérios que satisfazem RNA' e DC. Entretanto, quando alguma destas duas propriedades não é satisfeita não é possível fazer afirmação alguma quanto aos resultados da aplicação de suas quatro propriedades resultantes. Assim, nota-se que algumas características deixam de ser mensuradas pelo Conjunto Reduzido de Propriedades para critérios que não atendem as propriedades RNA' ou DC. Por esta razão as Propriedades Generalizadas persistem como um conjunto relevante para a avaliação destes critérios apesar de suas dependências.

Uma alternativa seria adicionar, como propriedades secundárias, AE, GMC, AD e AC ao Conjunto Reduzido de Propriedades ao aplica-lo a algum critério em que RNA' ou DC não são satisfeitas. Dessa forma, continuar-se-ia com um conjunto forte e sem dependências, mas que dispõe de propriedades derivadas dos teoremas de dependências quando estes não são aplicáveis.

Julgando o Conjunto Reduzido de Propriedades apto a constituir uma teoria axi-

omática, suas propriedades podem ser classificadas como axiomas com as demais propriedades do conjunto das Propriedades Generalizadas (AE, GMC, AD, AC e SC) as complementando, mas com o status de propriedade secundária. Dessa forma, ter-se-ia um conjunto com sete axiomas e cinco propriedades secundárias que mantém a essência dos axiomas propostos em [Weyuker 1986] e [Weyuker 1988], mas é forte e reduzido.

6. Conclusões

Este trabalho se propôs a entender e avaliar o comportamento e significado dos conjuntos de propriedades apresentadas em [Weyuker 1986], [Weyuker 1988] e [Parrish e Zweben 1991] aplicados a critérios conhecidos e os conceitos associados.

Da aplicação das propriedades aos critérios selecionados percebe-se que os critérios mais fortes satisfazem mais propriedades em relação aos critérios menos eficientes. O critério mais robusto, Teste de Mutação, atende a todas as propriedades, enquanto os critérios funcionais, mais limitados, satisfazem o menor número de propriedades.

Observou-se ainda que o Conjunto Reduzido de Propriedades rende os mesmos resultados para os critérios Todos os Comandos e Todas as Definições, entretanto o primeiro satisfaz duas propriedades do Conjunto de Propriedades Generalizadas a mais que o segundo, significando que, em certos casos, o Conjunto Reduzido de Propriedades deixa de avaliar as propriedades eliminadas pelo mesmo.

Para trabalhos futuros, é prevista a publicação de um texto rico em exemplos que demonstrem a aplicação das propriedades aos critérios selecionados, bem como uma revisão das propriedades envolvidas na dependência apontada na seção 5 para o Conjunto Reduzido de Propriedades. Ainda, produzir-se-á um estudo apurado da significância das características representadas pelas propriedades em ambientes reais de teste.

Referências

DeMillo, R. A., Lipton, R. J., e Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41.

Goodenough, J. B. e Gerhart, S. L. (1975). Toward a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510.

Myers, G. J., Sandler, C., Badgett, T., e Thomas, T. M. (2004). *The Art of Software Testing*. John Wiley & Sons, 2a edition.

Parrish, A. e Zweben, S. H. (1991). Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering*, 17:565–581.

Rapps, S. e Weyuker, E. J. (1982). Data flow analysis techniques for test data selection. *In: VI International Conference on Software Engineering*, pages 272–278.

Sharma, I., Kaur, J., e Sahni, M. (2014). A test case prioritization approach in regression testing. *International Journal of Computer Science and Mobile Computing*, 3:607–614.

Weyuker, E. J. (1986). Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12:1128–1138.

Weyuker, E. J. (1988). The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 35:668–675.

UMA FERRAMENTA INTERATIVA PARA VISUALIZAÇÃO DE CÓDIGO FONTE NO APOIO À CONSTRUÇÃO DE CASOS DE TESTE DE UNIDADE

Heleno de S. Campos Junior, Luís Rogério V. Martins Filho, Marco A. P. Araújo
Instituto Federal do Sudeste de Minas Gerais
Campus de Juiz de Fora (IF Sudeste MG)
Juiz de Fora – MG – Brasil

heleno_scj@hotmail.com, luisrogeriojf@gmail.com,
marco.araujo@ifsudestemg.edu.br

Abstract. *This paper presents a tool capable of generating Control Flow Graphs aiming to help the unit test case development for Java language and contributing to increase software's quality and reliability, making possible more interactivity between the developer and software products. The generated graph shows independent paths of the parsed source code. Besides calculating the Cyclomatic Complexity metric and showing the maximum amount of necessary test cases to achieve a better unit test's coverage level.*

Resumo. *Este trabalho apresenta uma ferramenta capaz de gerar Grafos de Fluxo de Controle com o intuito de apoiar o desenvolvimento de casos de teste de unidade na linguagem Java, visando contribuir para uma maior qualidade e confiabilidade de produtos de software, possibilitando uma maior interação com o desenvolvedor através de técnicas de visualização. O grafo gerado mostra os caminhos independentes do código fonte analisado, através do cálculo do valor da Complexidade Ciclomática para determinar a quantidade máxima de testes a serem desenvolvidos, possibilitando incrementar a cobertura do código fonte atingido pelos testes de unidade.*

1. Introdução

Com a crescente demanda por sistemas mais complexos, incluindo sistemas distribuídos, computação em nuvem, aplicativos para dispositivos móveis, entre outros, surge a necessidade de se desenvolver sistemas de maior qualidade e cada vez mais confiáveis e manuteníveis. Para solucionar esse problema é necessário que, durante o ciclo de vida do software, haja uma maior preocupação com essas características, pois isso irá garantir que o software gerado seja mais confiável, com menos defeitos para o usuário final e seja mais facilmente mantido em futuras versões.

Teste de software é uma técnica de apoio à qualidade de software que serve para revelar defeitos durante o desenvolvimento de um sistema [Younessi 2002]. De forma geral, pode ser classificado em teste estrutural, que tem por objetivo apoiar a construção de casos de teste a partir da estrutura do código fonte, caracterizando uma técnica de caixa branca (*white box*); em teste funcional, que tem por objetivo analisar o comportamento externo do software na perspectiva do usuário, normalmente baseado em requisitos e modelos de software e, ao contrário dos testes estruturais, trata-se de uma técnica de caixa preta (*black box)* [Nidhra e Dondeti 2012]; e em testes baseados em defeitos, utilizando dados históricos para identificar defeitos presentes em um sistema em desenvolvimento [Morell 1990]. Visando apoiar os testes,

principalmente seu planejamento, diferentes métricas estão disponíveis para apoiar tais atividades [Ramos e Valente 2014].

No contexto de testes estruturais, objetivo deste trabalho, a métrica Complexidade Ciclomática de McCabe [McCabe 1976] é amplamente utilizada para apuração de características como a predição da manutenibilidade do software e a facilidade de se compreender um módulo de software [Michura e Capretz 2005]. A métrica determina o número de caminhos independentes que um algoritmo possui, a partir de um Grafo de Fluxo de Controle, que representa o fluxo de execução de um determinado código fonte, podendo ser usada para apoiar a construção de casos de teste de unidade, que é o objetivo desse trabalho. Nessa técnica, o valor da Complexidade Ciclomática equivale ao número máximo de casos de testes necessários para a cobertura de todas as condições do algoritmo, ao contrário de outras técnicas de teste que utilizam combinações lineares dos caminhos do grafo para cobertura de todos os caminhos possíveis, o que pode representar redundância nos testes [Watson 1996].

A literatura técnica apresenta uma carência de ferramentas interativas que apoiem a visualização de testes estruturais através do uso da Complexidade Ciclomática, como mostrado na revisão sistemática deste trabalho, não evidenciando os caminhos independentes no Grafo de Fluxo de Controle ou não exibindo qual a condição que um nó predicado possui.

O trabalho realizado pretende suprir essas características com uma ferramenta que pode apoiar, interativamente, o desenvolvimento de casos de teste de unidade e capacitação do desenvolvedor na construção de testes estruturais e compreensão do código fonte. Além disso, técnicas de visualização são importantes formas de se obter compreensão, e são fundamentais para se criar um modelo mental sobre as informações, podendo auxiliar na obtenção de um conhecimento mais profundo sobre o objeto analisado [Silva et al 2012].

Este artigo é apresentado em mais quatro seções, além dessa introdução. A seção 2 mostra conceitos e trabalhos relacionados à proposta deste trabalho. A seção 3 apresenta a ferramenta construída e seu funcionamento. Por sua vez, a seção 4 apresenta um estudo experimental realizado no sentido de avaliar a ferramenta construída. Por fim, a seção 5 apresenta a conclusão e trabalhos futuros.

2. Conceituação e trabalhos relacionados

A seguir são revisados conceitos essenciais para que se possa entender a utilização e aplicação da ferramenta, além do detalhamento da revisão sistemática realizada.

2.1 Complexidade Ciclomática

A complexidade ciclomática é uma métrica cujo objetivo é medir a complexidade de determinado código fonte. Quanto maior seu o valor, maior a dificuldade de se entender, modificar e, consequentemente, testar o código fonte [McCabe 1976].

De forma a facilitar a visualização do fluxo de um código fonte, McCabe utiliza o conceito de Grafo de Fluxo de Controle como forma de representação, composto por vértices que fazem referência a blocos ou linhas de código fonte, e arestas que conectam os vértices formando caminhos do fluxo de execução [Allen 1970], identificando quais são os possíveis caminhos de execução de um algoritmo.

Tomando como exemplo a Figura 1, tem-se um método escrito na linguagem Java para calcular a aprovação de um aluno, e o grafo da Complexidade Ciclomática que corresponde ao código fonte. Cada vértice foi marcado no código fonte com seu respectivo número em forma de comentário (precedido de "//"), para fins de entendimento. Como exemplo, o vértice 1 corresponde à condição de que a porcentagem de presença do aluno seja maior ou igual a 75. Caso esta condição seja verdadeira, o fluxo segue para o vértice 2, caso contrário, para o vértice 6, e assim sucessivamente. Como pode ser observado, o grafo pode ajudar a entender quais são os possíveis caminhos do fluxo de execução do algoritmo, apoiando a construção dos casos de teste.

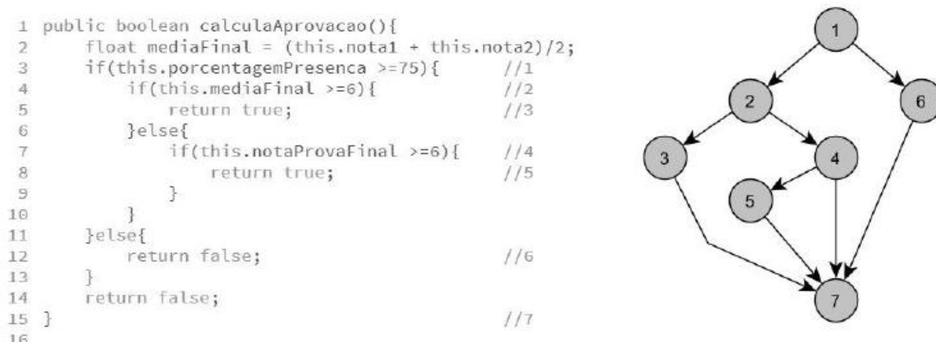


Figura 1: Grafo da complexidade ciclomática correspondente ao código fonte

A seguir, observam-se formas para calcular a Complexidade Ciclomática:

- através da fórmula V(G) = E – N + 2P, onde V(G) é a Complexidade Ciclomática, E é o número de arestas presentes no grafo, N é a quantidade de vértices e P a quantidade de métodos conectados. Como o objetivo neste trabalho é calcular a complexidade de um único método por vez, P será sempre 1. Para o exemplo apresentado, V(G) = 9 – 7 + 2 = 4;
- através do número de nós predicados, acrescido de 1. Um nó predicado representa uma condição no algoritmo, como os nós 1, 2 e 4 da figura. Acrescido de um, tem-se novamente a Complexidade Ciclomática igual a 4;
- contando-se a quantidade de áreas fechadas presentes no grafo, incluindo a área externa, correspondendo à área total do grafo;
- ou através do número de caminhos independentes do grafo, como exemplificado na Figura 2. Caminhos indendepentes são caminhos que o fluxo de execução deve seguir para que todas as instruções em um método sejam executadas pelo menos uma vez. Cada caminho independente inclui pelo menos uma aresta que não tenha sido percorrida antes do caminho ser definido [McCabe e Watson 1996].

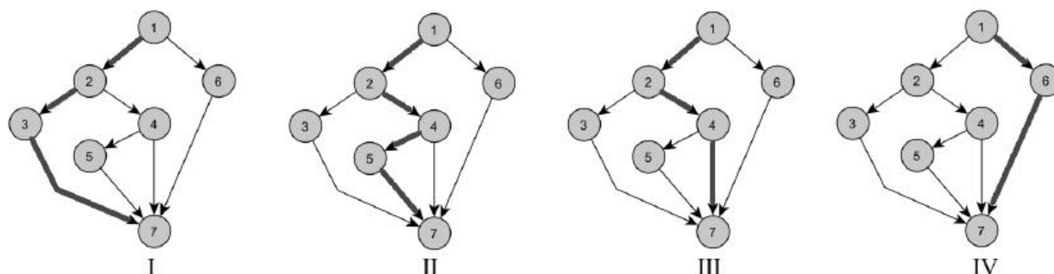


Figura 2: Caminhos independentes do grafo do método *calculaAprovacao()*

A partir do conhecimento dos caminhos independentes, pode-se determinar os casos de teste de unidade que devem ser executados para obter a cobertura das condições do método a ser testado, tendo como base a seguinte inequação [Boghdady et al 2011]:

Cobertura das condições do Código Fonte ≤ Complexidade Ciclomática V(G) ≤ Todos os Caminhos Possíveis

2.2 Teste de Unidade

Esse tipo de teste tem como foco testar o funcionamento de unidades de código fonte, ou seja, métodos ou módulos [IEEE 1990]. Trata-se de uma técnica de caixa branca, pois o desenvolvedor que realiza o teste deve ter conhecimento do código fonte e seu funcionamento. O objetivo desse teste é verificar se determinado método retorna o resultado esperado para determinada entrada de dados. Geralmente, utiliza-se uma tabela para descrever os valores que as variáveis devem assumir para atingir determinado resultado. Cada entrada da tabela é chamada de um caso de teste. Para cobrir as saídas e condições de um algoritmo, devem haver tantos casos de teste quantos caminhos independentes no grafo da Complexidade Ciclomática, exceto quando algum desses caminhos não é atingível. A Tabela 1 apresenta os casos de teste para o método *calculaAprovacao()*, anteriormente apresentado, utilizando critérios de Análise do Valor Limite [Clarke et al 1982], que tem como premissa utilizar valores próximos à fronteira dos valores estabelecidos pelas condições em que são testados.

Tabela 1: Casos de teste referentes ao método *calculaAprovacao()*

Caso de Teste	porcentagemPresenca	Nota1	Nota2	notaProvaFinal	Resultado esperado
1	75.0	6.0	6.0	-	True
2	75.0	6.0	5.9	6	True
3	75.0	5.9	5.9	5.9	False
4	74.0	-	-	-	False

2.3 Revisão Sistemática e trabalhos relacionados

No contexto deste trabalho, foi realizada uma revisão sistemática da literatura [Kitchenham 2004] na busca de trabalhos que abordam assuntos como a geração de grafos de Complexidade Ciclomática, identificação de caminhos independentes e apoio no planejamento de casos de teste. Segue um resumo do protocolo de pesquisa.

Questão de Pesquisa: Quais são as ferramentas e *plugins* existentes na literatura técnica que abordam Complexidade Ciclomática e representação de grafos?

Buscando responder à questão de pesquisa, utilizou-se a seguinte *string* de busca: *("Cyclomatic Complexity") AND ("Tool*" OR "Plugin*" OR "Extension*") AND ("Graph*" OR "Graph Theory").*

A busca foi realizada em 6 bases distintas (*Scopus*, *Science Direct*, *ISI Web of Science*, *IEEE Digital Library*, *ACM Digital Library* e *EI Compendex*) resultando em um total de 37 artigos recuperados. Esses foram refinados, aplicando os critérios de inclusão e exclusão definidos no protocolo e retirando-se os artigos duplicados. Ao final da revisão restaram 19 artigos, sendo os principais descritos a seguir.

[Shukla e Ranjan 2012] propõem uma métrica de complexidade baseada na métrica de McCabe e, para fins de avaliação, foi desenvolvida uma ferramenta que calcula os valores para diferentes métricas como LoC (*Lines of Code*), *Halstead*,

Complexidade Ciclomática, e a própria métrica abordada sendo que, além do cálculo da Complexidade Ciclomática, é gerado o Grafo de Fluxo de Controle. A ferramenta é capaz de analisar módulos na linguagem C.

[Boghdady et al 2011] desenvolveram uma técnica de geração de testes através de Diagramas de Atividade UML (*Unified Modeling Language*), gerando um grafo chamado ADG (Grafo de Diagrama de Atividades) a partir de uma ADT (Tabela de Diagrama de Atividades) e então, com o uso da Complexidade Ciclomática, trata-se valores máximos e mínimos de casos de testes, sendo gerados os testes dos caminhos independentes.

[Lertphumpanya e Senivongse 2008] desenvolveram uma ferramenta para a geração de testes para Serviços Compostos WS-BPEL. Através de um arquivo WS-BPEL, a ferramenta extrai informações como variáveis e funções, separa URLs para um XML, cria um Grafo de Fluxo de Controle do código fonte, de onde tira os caminhos independentes para a geração de testes.

Os artigos revisados mostram técnicas e ferramentas para geração de Grafos de Fluxo de Controle, cálculo da Complexidade Ciclomática e apoio à construção de casos de teste. A Tabela 2 apresenta um comparativo dessas ferramentas. Ao contrário da ferramenta desenvolvida neste trabalho, nenhuma possui suporte para navegação pelos caminhos do grafo, exibição das condições de cada nó predicado em relação a cada caminho, ou marcação do código fonte ao selecionar um determinado nó. Ainda, a ferramenta aqui apresentada possui código fonte aberto e suporte à linguagem Java.

Tabela 2: Comparativo das ferramentas e técnicas encontradas

Autor	Ferramenta	Suporte à Linguagem	Construção de Caminhos Independentes	Grafo de Fluxo de Controle	Exibição das Condições	Marcação do Código
[Shukla e Ranjan 2012]	Software Metric Tool (SMT)	C	Não	Sim	Não	Não
[Boghdady et al 2011]	-	Diagrama de Atividades (UML)	Sim	-	-	-
[Lertphumpanya e Senivongse 2008]	Basis Path Testing Tool	WS-BPEL	Sim	Sim	Sim	Não

3. Uma ferramenta interativa para apoio à construção de testes de unidade

Com objetivo de apoiar a criação de casos de teste de unidade para métodos escritos na linguagem Java até sua versão 1.7, foi construída uma ferramenta de apoio ao desenvolvedor, atendendo aos seguintes requisitos:

- a ferramenta deve analisar um arquivo de classe da linguagem Java;
- a ferramenta deve gerar um grafo da Complexidade Ciclomática para cada método da classe analisada;
- a ferramenta deve permitir ao desenvolvedor navegar pelos caminhos independentes dos grafos gerados;
- a ferramenta deve fornecer uma análise sobre os grafos gerados contendo caminhos independentes, condições para que esses caminhos sejam satisfeitos e a Complexidade Ciclomática do método analisado;

- a ferramenta deve facilitar o entendimento do grafo através da indicação do trecho de código correspondente a cada vértice.

A utilização da ferramenta segue as fases apresentadas através de um Diagrama de Atividades da UML [Rumbaugh 1999] (Figura 3). São elas: (i) *Scanner*, onde a ferramenta deve ler o código fonte de uma classe na linguagem Java; (ii) *Parser,* deve processar a estrutura da classe lida gerando uma estrutura de grafo para cada método da classe alvo; (iii) *Renderer*, a partir da estrutura de grafos criada, deve-se renderizar os grafos na tela; e (iv) *Analysis*, a partir do grafo criado, realizar análises sobre os dados gerados.

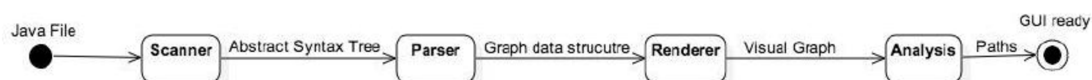


Figura 3: Diagrama de atividades da ferramenta

3.1. *Scanner*

Para essa atividade, é utilizada uma biblioteca de código fonte aberto [JavaParser 2015]. A biblioteca é capaz de ler arquivos de código fonte na linguagem Java e disponibilizar uma *Abstract Syntax Tree* (AST), representando a estrutura do código fonte lido e possibilitando a coleta de informações sobre o mesmo.

3.2 *Parser*

Nessa atividade é necessário processar a AST para gerar uma estrutura de grafos que represente o código fonte. Para isso, é utilizado um *Recursive Descent Parser* [Burge 1975] que permite processar a AST a partir de regras para blocos de comando. Foram criadas regras para os seguintes blocos de comando: *if / else, switch / case, for-each, for, while, do-while, return, break.*

3.3 *Renderer*

A partir do momento que se tem a estrutura do grafo da Complexidade Ciclomática, deve-se renderizá-lo na tela para o usuário. Para isso, utiliza-se a biblioteca JUNG (*Java Universal Network/Graph Framework*) [Jung 2015], também de código aberto.

3.4 *Analysis*

Possuindo o grafo renderizado, é possível realizar análises estáticas sobre o mesmo. Exemplo de análise é a geração dos caminhos independentes e cálculo da Complexidade Ciclomática. Para a geração dos caminhos independentes, é usado um método conhecido como *baseline method* [McCabe e Watson 1996]:

- partindo do nó inicial até o nó final do grafo, calcula-se o caminho mais à esquerda (*leftmost path),* chamado de caminho base;
- a partir do caminho base, para cada nó predicado, cria-se um novo caminho invertendo a condição do nó. Repete-se esse passo enquanto houverem nós predicados que não foram invertidos. Ao final, tem-se um caminho base somado à quantidade de caminhos independentes derivados (de acordo com a existência de nós predicados no grafo).

Após a conclusão dessas atividades, é apresentada uma interface gráfica (Figura 4) que disponibiliza as funcionalidades da ferramenta.

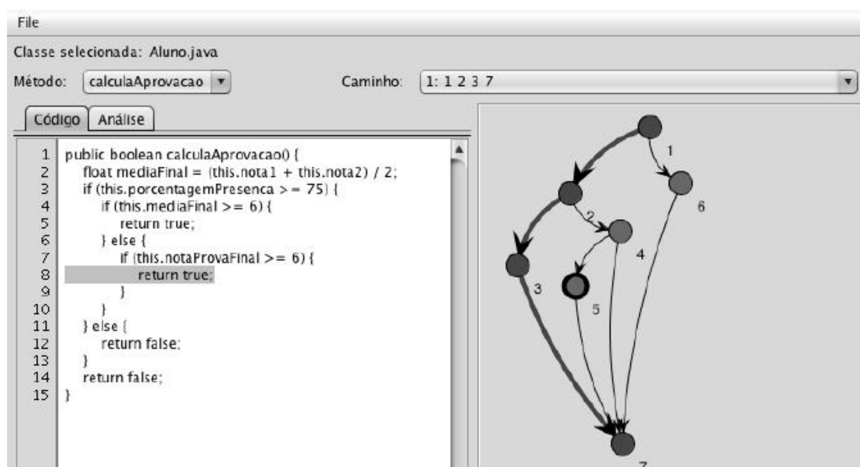
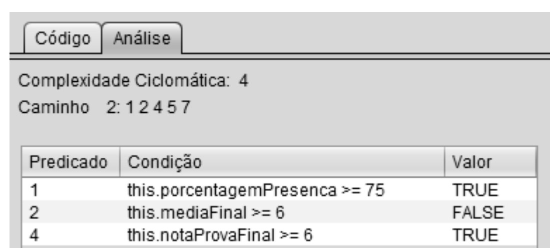


Figura 4: Interface gráfica da ferramenta

À esquerda da figura é exibido o código fonte do método analisado, enquanto à direita é renderizado o grafo correspondente. Os caminhos independentes calculados ficam disponíveis para seleção em uma caixa de combinação acima do grafo. Quando um caminho é escolhido, o mesmo é destacado no grafo. Caso o usuário queira saber a que bloco de comando no código fonte um vértice equivale, o mesmo pode selecionar o vértice desejado, e então o bloco de comando é destacado à esquerda.

Existe também a aba análise (Figura 5), cujo objetivo é fornecer ao usuário informações sobre o grafo, contendo o valor da Complexidade Ciclomática, o caminho selecionado e a indicação dos valores das condições a serem satisfeitas para a execução daquele caminho. Essa análise possibilita ao programador planejar o caso de teste para cada caminho.



Predicado	Condição	Valor
1	this.porcentagemPresenca >= 75	TRUE
2	this.mediaFinal >= 6	FALSE
4	this.notaProvaFinal >= 6	TRUE

Figura 5: Aba análise da ferramenta

4. Condução de um estudo experimental

Com o objetivo de avaliar o apoio que a ferramenta pode proporcionar ao desenvolvedor no planejamento de casos de teste de unidade, foi elaborado um estudo experimental com acadêmicos do curso de Bacharelado em Sistemas de Informação do IF Sudeste MG – Campus Juiz de Fora. Os participantes estavam no sétimo período do curso e possuíam conhecimentos na linguagem de programação Java, e já haviam estudado sobre Complexidade Ciclomática e testes de unidade. O estudo contou com um total de 12 participantes que foram divididos aleatoriamente em dois grupos, e apenas a um dos grupos foi permitida a utilização da ferramenta como apoio à atividade.

Foi disponibilizado um método que calcula a aprovação de um aluno a partir de dados de algumas variáveis (porcentagemPresenca, nota1, nota2 e notaProvaFinal), conforme mostra a Figura 6, para que cada aluno desenvolvesse os casos de teste de unidade de acordo com sua experiência.

```
1  public boolean obterAprovacao() {
2      if(porcentagemPresenca!=null && porcentagemPresenca < 75){
3          return false;
4      }
5      float mediaFinal = 0;
6      if(this.nota1 != null && this.nota2 != null){
7          mediaFinal = (this.nota1 + this.nota2)/2;
8      }else{
9          return false;
10     }
11     if(mediaFinal >= 6){
12         return true;
13     }else if(this.notaProvaFinal!=null && this.notaProvaFinal >=6){
14         return true;
15     }else{
16         return false;
17     }
18
19 }
```

Figura 6: Código referente ao método utilizado no estudo experimental

A complexidade do método em questão é igual a 8, ou seja, são necessários no máximo 8 casos de teste para se obter a cobertura total das condições do algoritmo. Os dados referentes ao desempenho alcançado pelos alunos estão representados na Tabela 3, tendo sido obtidos através de uma ferramenta para o ambiente de desenvolvimento Eclipse capaz de medir a cobertura de desvios nos testes desenvolvidos por cada aluno.

Tabela 3: Resultado da aplicação do estudo experimental

Aluno	Cobertura Grupo 1 (utilizando a ferramenta)	Cobertura Grupo 2 (não utilizando a ferramenta)
1	78,57%	63,29%
2	92,86%	13,29%
3	78,57%	70,43%
4	100,00%	77,57%
5	92,86%	84,71%
6	78,57%	63,29%

O resultado foi analisado estatísticamente considerando o seguinte teste de hipóteses, a um nível de significância de 5%:

H_0 (hipótese nula): as médias dos grupos são iguais.

H_1 (hipótese alternativa): as médias dos grupos são diferentes.

Utilizando o método de Shapiro-Wilk para verificar a normalidade das amostras, foi constatado que as mesmas possuem distribuições normais, pois ambas apresentaram um *p-value* > 0,100, ficando acima do nível de significância estabelecido. Em relação à homocedasticidade (variância constante) também foi constatado que as amostras são homocedásticas pois, pelo teste de Levene, o *p-value* encontrado foi de 0,203, também maior que o nível de significância utilizado.

Com os pressupostos de normalidade e homocedasticidade validados, pode-se utilizar um método estatístico paramétrico para comparação das médias, no caso, foi utilizado o método Test T, cujo resultado é apresentado na Figura 7.

```
Two-sample T for Cobertura

Tipo        N   Mean  StDev  SE Mean
Ferramenta  6  86,91   9,50      3,9
Manual      6   62,1  25,32     10,3


Difference = μ (Ferramenta) - μ (Manual)
Estimate for difference:  24,8
95% CI for difference:  (0,21; 49,4)
T-Test of difference = 0 (vs ≠): T-Value = 2,25  P-Value = 0,0484  DF = 10
```

Figura 7: Resultado do método Test T

Como o *p-value* encontrado (0,0484) é menor que o nível de significância estabelecido, rejeita-se a hipótese nula e aceita-se a hipótese alternativa de que as médias são diferentes. No caso, a média do grupo que utilizou a ferramenta é de 86,91%, enquanto a média para o outro grupo é de 62,1%, evidenciando que a utilização da ferramenta foi positiva no apoio à criação de casos de teste para o método fornecido.

Em complemento a esse resultado, foram coletadas opiniões dos participantes após a realização do estudo. Dentre as dificuldades encontradas por quem não utilizou a ferramenta, estão o entendimento do código fonte, a identificação dos caminhos, e a determinação dos valores limites corretos. Somente um participante construiu manualmente o grafo da Complexidade Ciclomática como apoio. Para o grupo que utilizou a ferramenta, a maior dificuldade foi a localização no código fonte de nós que representam condições compostas (que utilizam conectores lógicos). Foi constatado por todos os alunos que utilizaram a ferramenta que a mesma auxiliou no entendimento do código fonte e identificação dos casos de teste, facilitando a construção dos mesmos.

5. Conclusão e Trabalhos Futuros

O trabalho realizado apresenta uma ferramenta interativa de geração de grafos de Complexidade Ciclomática que realiza uma análise da estrutura de um código fonte e apresenta nós para exibição. Esses nós são renderizados na tela em conjunto com informações coletadas na análise da Complexidade Ciclomática e dos nós predicados, exibindo também os caminhos independentes, apoiando a construção de casos de teste de unidade para o desenvolvimento de produtos de software de maior qualidade.

Através da análise estatística sobre os dados do estudo experimental realizado, ficou evidenciado que o grupo que utilizou a ferramenta como apoio à criação de casos de teste teve melhor desempenho que o grupo que não utilizou, para o caso abordado. Entretanto, serão necessários novos estudos experimentais para confirmar os resultados obtidos. Apesar do número pequeno de participantes, a análise estatística considerou métodos que podem ser utilizados em amostras com poucos elementos. Não é possível ainda generalizar os dados, em função do tamanho da amostra.

Como trabalhos futuros pretende-se estender a ferramenta para um *plugin* integrado a ambientes de desenvolvimento de software, como NetBeans e Eclipse, assim como aplicação de novos algoritmos de exibição do grafo, visto que o algoritmo utilizado atualmente é limitado nessa tarefa. É também interesse o apoio para geração automática de casos de teste através dos caminhos independentes gerados, eliminando a redundância de casos de teste inatingíveis.

Referências

[Allen 1970] Allen, F. E. (1970). Control flow analysis, Proceedings of a symposium on Compiler optimization, p.1-19, Urbana-Champaign, Illinois.

[Boghdady et al 2011] Boghdady, P. N., Badr, N. L., Hashim, M. A., & Tolba, M. F. (2011). An enhanced test case generation technique based on activity diagrams. Em *Computer Engineering & Systems (ICCES), 2011 International Conference* (p. 289-294).

[Burge 1975] Burge, W. H. (1975), Recursive Programming Techniques (The Systems Programming Series), Addison Wesley.

[Clarke et al 1982] Clarke, L. A., Hassell, J., & Richardson, D. J. 1982. A close look at domain testing. IEEE Trans. Softw. Eng. (Vol. 8, No. 4, p. 380–390).

[IEEE 1990] IEEE (1990). "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology".

[JavaParser 2015] JavaParser, disponível em <http://javaparser.github.io/javaparser/>, acesso em Junho, 2015.

[Jung 2015] Java Universal Network/Graph Framework, disponível em <http://jung.sourceforge.net/>, acesso em Junho, 2015.

[Kitchenham 2004] Kitchenham, B. (2004). *Procedures for performing systematic reviews*. Keele, UK, Keele University, 33(2004), p. 1-26.

[Lertphumpanya e Senivongse 2008] Lertphumpanya, T., & Senivongse, T. (2008). A basis path testing framework for WS-BPEL composite services. *Proceedings of the 7th WSEAS*, p. 107-112.

[McCabe 1976] McCabe, T. J. (1976). A complexity measure. Em: *IEEE Trans. Software Eng*. 1976, (Vol. SE-2, N. 4. p. 308-320).

[McCabe e Watson 1996] McCabe, Thomas, J., Watson, Arthur, H. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication* 500-235.

[Michura e Capretz 2005] Michura, J., & Capretz, M. A. (2005). Metrics suite for class complexity. *In Information Technology: Coding and Computing*, 2005. ITCC 2005. International Conference on (Vol. 2, p. 404-409).

[Morell 1990] Morell, L. J. (1990). A theory of fault-based testing. IEEE Trans. on Soft. Eng., 16(8):844–857

[Nidhra e Dondeti 2012] Nidhra, S., & Dondeti, J. (2012). Black Box And White Box Testing Techniques–A Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2012, (Vol.*2* , No. 2, p. 29-50).

[Ramos e Valente 2014] Ramos, M. E., & Valente, M. T (2014). Análise de Métricas Estáticas para Sistemas JavaScript. Em: *II Brazilian Workshop on Software Visualization, Evolution, and Maintenance (VEM 2014),* 2014, (p. 30-37).

[Rumbaugh et al 1999] Rumbaugh, J., Jacobson, I., Booch, G. (1999). The Unified Modeling Language Reference Manual. Addison-Wesley.

[Shukla e Ranjan 2012] Shukla, A., & Ranjan, P. (2012). A generalized approach for control structure based complexity measure. Em *Recent Advances in Information Technology (RAIT), 2012 1st International Conference* (p. 916-921).

[Silva et al 2012] Silva, A. N., Carneiro, G. F., Zanin, R. B., Dal Poz, A. P., Martins, E. F. O. (2012). Propondo uma Arquitetura para Ambientes Interativos baseados em Multiplas Visões. *II Workshop Brasileiro de Visualização de Software*, Natal, RN (p. 1 – 8).

[Watson 1996] Watson, A. H. (1996). *Structured testing: Analysis and extensions*. Princeton University.

[Younessi 2002] Younessi, H. (2002). *Object Oriented Defect Management of Software*. Prentice Hall PTR.

Teste Estrutural Aplicado à Linguagem Funcional Erlang

Alexandre P. Oliveira[1], Paulo S. L. Souza[1], Simone R. S.Souza[1], Júlio C. Estrella[1], Sarita M. Bruschi[1]

[1]Instituto de Ciências Matemáticas e de Computação – ICMC
Universidade de São Paulo – São Carlos, SP – Brasil

{alexandreponce,pssouza,srocio,jcezar,sarita}@icmc.usp.br

Abstract. *Erlang functional language allows the development of real-time and fault-tolerant parallel programs used in critical systems such as circuit-switched telephone networks. Therefore, validation, verification and testing are fundamental and contribute to improving the quality of programs. This paper, which describes an ongoing research, proposes a set of structural testing criteria for Erlang programs, which considers features such as, communication between functions and between processes, synchronization, concurrency and recursion. The criteria were applied in an Erlang program and the results indicate that it is possible to use structural testing in this context.*

Resumo. *A linguagem funcional Erlang permite o desenvolvimento de programas paralelos tolerantes a falhas e de tempo real, usados em sistemas críticos, tais como rede de telefonia por comutação de circuitos. Assim, a validação, verificação e teste são fundamentais e contribuem para melhorar a qualidade dos programas. Este artigo, relacionado a um trabalho em andamento, propõe um conjunto de critérios de teste estruturais para programas Erlang os quais consideram características como, comunicação entre funções e entre processos, sincronização, concorrência e recursividade. Os critérios foram aplicados em um programa Erlang e os resultados indicam que é possível utilizar o teste estrutural nesse contexto.*

1. Introdução

O paradigma de programação funcional se baseia em funções matemáticas, apresentando características não presentes no paradigma imperativo, como por exemplo: dados imutáveis, *higher-order functions* (funções de ordem superior), *lazy evaluation* e *pattern matching* (correspondência de padrão) (Sebesta, 2009). Alguns exemplos de linguagens funcionais são: Haskell, Scala (Tate, 2010) e Erlang (Armstrong et al., 2003).

Em linguagens imperativas, como C, C++ e Java, a computação geralmente é especificada com variáveis e comandos de atribuição. Um programa escrito em uma linguagem funcional, geralmente usa como meio de computação funções que recebem determinados parâmetros, os quais são avaliados através de correspondência de padrão.

Em geral, aplicações desenvolvidas em linguagens funcionais são usadas em sistemas críticos, tais como redes de comutação telefônica e deve fornecer alta qualidade, confiabilidade e eficiência. Neste contexto, validação, verificação e

atividades de teste são necessárias e contribuem para melhorar a qualidade dos programas funcionais (Balakrishnan e Anand, 2009). Na literatura são encontrados poucos trabalhos ((Widera, 2004), (Widera, 2005), (Guo et al., 2009), (Christakis e Sagonas, 2011), (Gotovos et al., 2011)) que exploram o teste estrutural para linguagens funcionais. Além disso, os mesmos não exploram adequadamente os aspectos de concorrência e comunicação existentes nessas linguagens.

Este artigo apresenta um conjunto de critérios de teste estruturais para programas em Erlang, que incluem suas principais características, tais como, comunicação entre funções, comunicação entre processos, sincronização, concorrência e recursividade. Os critérios de teste foram definidos para explorar o fluxo de dados e de controle desses programas, considerando seus aspectos sequenciais e concorrentes.

Duas são as principais contribuições deste artigo. A primeira é o estabelecimento das principais características de uma linguagem no paradigma funcional, de modo a permitir a aplicação do teste estrutural. Isto é necessário porque a estrutura de codificação de linguagens funcionais difere do paradigma imperativo (não existe uma função principal, por exemplo), onde gerar rastro de execução (*trace)* é um desafio, dificultando a instrumentação do programa necessária à avaliação dos testes. A segunda é a definição de critérios estruturais voltados ao teste das principais características da linguagem Erlang, tais como, chamadas de função, recursão e passagem de mensagem.

Este artigo está estruturado da seguinte forma. A Seção 2 apresenta as principais características da linguagem funcional Erlang. Os conceitos básicos, o modelo de teste adotado para a definição dos critérios de teste e os resultados da aplicação dos critérios de teste são apresentados nas Seções 3 e 4. Na Seção 5 são apresentados os trabalhos relacionados e na Seção 6 as conclusões e trabalhos futuros são apresentados.

2. A linguagem Erlang

Erlang é uma linguagem de programação declarativa com núcleo funcional, projetada para programação concorrente, em tempo real e sistemas distribuídos tolerantes a falhas (Armstrong et al., 2003). O objetivo principal de Erlang foi melhorar a programação das aplicações telefônicas que eram atualizadas em tempo de execução, e consequentemente, não permitiam a perda do serviço durante a atualização do código (Cesarini e Thompson, 2009). Em 1996, foi lançado o *framework Open Telecom Platform* (OTP) para a programação de *switches* e outros produtos da Ericsson. OTP inclui todo ambiente para desenvolvimento Erlang e um conjunto de bibliotecas e ferramentas (Armstrong et al., 2003).

As principais características de Erlang que são de interesse para o contexto deste trabalho são (Cesarini e Thompson, 2009):

1. Correspondência de Padrão: é um mecanismo básico para atribuir valores às variáveis. Em Erlang, atribuir um valor para uma variável é chamado de ligação (*bound*), uma variável *bound* nunca pode ter seu valor alterado (Armstrong et al., 2003). Correspondência de padrão também é usada para controlar o fluxo de execução de programas e extrair valores de tipos de dados compostos (Logan et al., 2010).

2. *Guards* e Recursão: são muito utilizados em Erlang e considerados como uma extensão de correspondência de padrão. *Guards* são condições avaliadas e que precisam

ser satisfeitas para continuar a executar o conteúdo de uma cláusula (Logan et al., 2010). Como os dados em Erlang são imutáveis, uma forma de substituir as iterações das linguagens imperativas é com o uso de recursão. A recursão controla a ordem em que as expressões de correspondência são avaliadas (Sebesta, 2009).

3. Passagem de Mensagem: Processos Erlang não compartilham dados e os mesmos trocam dados por passagem de mensagens.

A Figura 1 apresenta uma versão de um programa para cálculo do *Greatest Common Divisor* (GCD) em Erlang. Esse programa utiliza até quatro processos concorrentes e funciona da seguinte forma: o programa principal recebe três números e envia os dois primeiros para o primeiro processo escravo (linha 5) e os dois últimos para o segundo processo escravo (linha 6). Os processos escravos ficam bloqueados esperando os valores do processo mestre (linha 6). Após o recebimento, os escravos calculam o valor, enviam para o programa principal (linha 12) e finalizam sua execução. O cálculo pode envolver os três processos ou apenas dois - isso depende dos valores de entrada. Os processos escravos utilizam recursão (linhas 10 e 11) e dependendo dos valores recebidos a função calc() pode ser instanciada várias vezes. O resultado do cálculo do programa GCD é mostrado pelo processo mestre (linhas 18 e 24).

```
1          -module(gcdm).
2   Nó     -export([start/3]).
3   /*0*/ start(NumX, NumY, NumZ) ->
4   /*0*/     global:register_name(pid0, self()),
5   /*1*/     global:whereis_name(pid1) ! {NumX, NumY, Pid0},
6   /*2*/     global:whereis_name(pid2) ! {NumY, NumZ, Pid0},
7   /*3*/     receive
8   /*4*/       {X, Pid} ->
9   /*5*/ /*6*/   waity(X)
10  /*7*/     end.
11  /*8*/ waity(A) ->
12  /*9*/      receive
13  /*10*/       {B, Pid} ->
14  /*12*/         if (A > 1) and (B > 1) ->
15  /*13*/          global:whereis_name(pid3) ! {A, B, self()},
16  /*15*/ /*16*/   fim();
17  /*11*/        true ->
18  /*14*/          io:format("GCD = ~w. Fim!~n", [A])
19  /*17*/        end
20  /*17*/     end.
21  /*18*/ fim() ->
22  /*19*/     receive
23  /*20*/       {C, Pid} ->
24  /*21*/         io:format("GCD = ~w. Fim!~n", [C])
25  /*21*/       end.
```

```
1   -module(gcds1).
2   -export([start/0,wait/0]).
3       start() ->
4   Nó      global:register_name(pid2, spawn(gcds2, wait, [])).
5   /*0*/ wait() ->
6   /*1*/      receive
7   /*2*/          {X, Y, Mpid} ->
8   /*3*/ /*4*/        calc(X, Y, Mpid);
9   /*5*/      end.
                              /*7*/          /*10*/
10  /*6*/ calc(A, B, Pidm) when A < B -> calc (A, B - A, Pidm);
                              /*8*/
11        calc(A, B, Pidm) when B < A -> calc (A - B, B, Pidm);
                          /*9*/
12        calc(A, B, Pidm) -> Pidm ! {A, self()}. /*12*/
                                   /*11*/
```

Figura 1. Programa GCD Principal e Programa GCD Escravo.

3. Teste estrutural para programas Erlang

A definição do teste estrutural para programas Erlang é baseada no modelo e critérios de teste apresentados em Souza et al. (2013) e em Sarmanho et al. (2008), os quais foram definidos para o teste estrutural de programas concorrentes com linguagens imperativas. Maiores detalhes sobre o teste estrutural de programas concorrentes podem ser obtidos em Souza et al. (2007). O foco é no teste de unidade desses programas, em geral, caracterizando cada processo como uma unidade e explorando também o teste da comunicação entre os processos. Para considerar este novo contexto Erlang, adaptações significativas foram feitas. Todas as adaptações feitas são descritas nesta seção e exemplos das mesmas podem ser vistos na Figura 2 e na Tabela 1.

Como um programa Erlang é composto por várias funções, o teste de unidade pode não ser eficiente. Isso torna necessário o uso do teste de integração para se obter resultados relevantes em relação ao módulo testado, ou seja, o teste interfunção. O

objetivo de teste interfunção é testar as interações entre as diversas funções que compõem um módulo. Considerando que o conjunto das possíveis chamadas de funções pode ser extenso (por exemplo, chamadas recursivas) e custoso para o teste, pode se utilizar um subconjunto desse conjunto. Desse modo, o modelo de teste para Erlang considera um número *n* fixo e conhecido de processos concorrentes criados durante a inicialização da aplicação concorrente. Cada processo *p* pode executar diferentes programas. Entretanto, cada processo possui seu próprio espaço de memória. O programa paralelo é composto por um conjunto de *n* processos paralelos $Prog = \{p^0, p^1, ..., p^{n-1}\}$. Cada processo *p* tem seu próprio Grafo de Fluxo de Controle (GFC^p), que é composto por um conjunto de nós N^p e um conjunto de arestas E^p. Cada processo p^i é composto por um número f^i fixo e conhecido de funções. Cada função $f^{i,j}$ possui seu Grafo de Fluxo de Controle ($GFC^{p,f}$), que é composto por um conjunto de nós $N^{p,f}$ e um conjunto de arestas $E^{p,f}$ e pertencem a um GFC^p. O conjunto de arestas de sincronização entre as funções de *p* é representação pela notação E^p_{sync}. A notação $n^{p,f}$ é utilizada para representar os nós do $GFC^{p,f}$, sendo que *n* corresponde ao nó, *p* indica o processo e f a função. Cada nó corresponde a um conjunto de comandos que são executados sequencialmente em cada função.

Para que seja possível testar a comunicação e sincronização entre os processos e com base nas definições e grafos anteriores, é definido o Grafo de Fluxo de Controle Paralelo (GFCP) do programa *Prog*. O GFCP é composto por todos os GFC^p (para p = 0..*np*-1) e pelas arestas de comunicação e sincronização entre os processos concorrentes denominadas E_{sync}. Assim, as arestas de um *GFCP* são classificadas da seguinte forma: **conjunto de arestas intrafunções (**$E^{p,f}$**)** que contém as arestas internas de uma função *f*; **conjunto de arestas interfunções (**E^p_{sync}**)** que contém as arestas que representam a comunicação entre duas funções de um processo *p*; e **conjunto de arestas interprocessos (**E_{sync}**)** que contém as arestas de sincronização que representam a comunicação entre os processos, ou seja, ocorrem entre dois processos concorrentes.

O envio de dados (*send*) usado na comunicação entre processos Erlang é assíncrono e o recebimento (*receive*) é bloqueante com *buffer* (Armstrong et al., 2003). A comunicação entre os processos utiliza o mecanismo ponto a ponto.

Assim, N_{sync} representa um subconjunto de nós que contém primitivas de comunicação e sincronização. Essas primitivas são agrupadas e classificadas de acordo com a sua semântica. Para isso é considerado um conjunto $C = \{c_0, c_1,...c_{nc-1}\}$, onde *nc* indica a quantidade de grupos de primitivas existentes em *Prog* e cada c_i indica um desses diferentes grupos. Para agrupar as primitivas é utilizado um conjunto *B = {´BS´, ´BR´}*, representando as seguintes primitivas de comunicação/sincronização: *BS* = send bloqueante (*Blocking Send*) e BR = receive bloqueante (*Blocking Receive*). Assim, um nó N_{sync} é uma tripla ($n_i^{p,f}$, c_j, b_k), onde o nó $n_i^{p,f} \in N^{p,f}$ possui uma primitiva de comunicação/sincronização $c_j \in C$ e $b_k \in B$.

Para representar as chamadas de funções e as funções recursivas, o modelo de teste controla o empilhamento e desempilhamento das chamadas de uma função. Para isso, os seguintes conjuntos são propostos: **Conjuntos** N_{call} **e** N_{return} representam nós com, respectivamente, chamadas de função e retorno de uma função chamada. Esses nós são interligados por meio de arestas E^p_{sync} e o **Conjunto** $N_{recursive}$ representa nós que controlam o empilhamento e desempilhamento das chamadas recursivas.

Para interligar o conjunto de $N_{recursive}$, o modelo define novos tipos de arestas: **conjunto de arestas de chamadas recursivas (E_{stack})** com as arestas representando as chamadas recursivas (empilhamento) de uma mesma função *f*; e **conjunto de arestas de retorno da recursão ($E_{unstack}$)** com arestas representando o retorno das chamadas recursivas (desempilhamento) de uma mesma função *f*;

Para identificar os nós N_{call} e N_{return} e o número de chamadas recursivas de uma função *f* é necessário especificar um conjunto $L = \{‘C’, ‘R’, 0, 1, 2, ..., n\}$ onde *C* indica uma chamada de função, *R* indica retorno de uma função e os números indicam a quantidade de chamadas recursivas de uma função. Com isso é possível controlar o empilhamento e desempilhamento dessas chamadas recursivas. Assim, o nó $N_{recursive}$ é uma tripla $(n^{p,f}, f_i, l_j)$, onde o nó $n^{p,f}$ possui uma chamada recursiva da função f_i e possui um número representado por $l_j \in L$.

Os nós N_{call} e N_{return} também são uma tripla $(n^{p,f}, f_i, l_j)$, onde o nó $n^{p,f}$ possui uma chamada de função ou retorno de uma função f_i e possui a identificação *C* ou *R* de $l_j \in L$.

Um caminho intrafunção $\pi^{p,f}$ em $GFC^{p,f}$ de uma função *f*, contém somente arestas $E^{p,f}$. O caminho $\pi^{p,f}$ é representado por $\pi^{p,f} = (n_0^{p,f}, n_1^{p,f}, \ldots, n_m^{p,f})$, onde $n_i^{p,f}$, $n_{i+1}^{p,f} \in E^{p,f}$. Um caminho interfunção Π^p no GFC^p é aquele que contém pelo menos uma aresta interfunção e é representado por $\Pi^p = (\pi^{p,f1}, \pi^{p,f2}, \pi^{p,f3}, \ldots, \pi^{p,fn}, S^p)$, onde S^p é o conjunto de pares de sincronização executado, de forma que toda aresta $(n_x^{p,fk}, n_y^{p,fl}) \in S^p$ e faz parte do conjunto E^p_{sync}. Um caminho interprocesso Π no *GFCP* tem pelo menos uma aresta E_{sync} e é representado por $\Pi = (\Pi^0, \Pi^1, \ldots, \Pi^i, S)$, onde *S* representa os pares de sincronização entre os processos que foram executados no caminho Π, tal que $S \in E_{sync}$.

Uma variável *x* é definida quando um valor é armazenado na sua posição de memória correspondente. As variáveis em linguagens funcionais são imutáveis, ou seja, uma vez definida não é possível atribuir um novo valor as mesmas. Neste caso, tem-se uma única definição para uma variável *x*. Se necessário mudar o valor de uma variável é necessário fazer uma cópia para outra variável. Assim, tem-se D^p que é o conjunto de todas as variáveis do processo *p* e um subconjunto $D^p_{copy} \subseteq D^p$ que é composto por todas as variáveis definidas por cópia de outra variável em *p*. O modelo adota dois tipos de definições: **f-def (definição inicial)** é representada pela tripla $(n^{p,f}, d, t)$, onde $d \in D^p$ e é definida em $n^{p,f}$ e possui uma identificação *t*; **c-def (definição por cópia)** é representada pela tripla $(n^{p,f}, d’, t)$, onde existe uma definição por cópia em $n^{p,f}$ da variável *d’* com identificação *t*.

O modelo utiliza os tipos de uso de variáveis apresentados em Souza et al. (2013), uso computacional (c-uso), uso predicativo (p-uso) e uso em mensagens (m-uso) e estabelece dois novos tipos de uso: **uso funcional (f-uso)** onde ocorre em chamada de função, relacionado a um nó N_{call} ou N_{return} em arestas interfunções $(n_i^{p,f1}, n_j^{p,f2}) \in E^p_{sync}$ e **uso recursivo (r-uso)** onde ocorre em uma chamada recursiva, relacionado a um nó $N_{recursive}$ as arestas de chamadas recursivas $(n_i^{p,f}, n_j^{p,f}) \in E_{stack}$ ou $E_{unstack}$;

As associações estabelecem a composição de pares de definições e usos de variáveis para serem testadas. As associações c-uso e p-uso capturam aspectos sequenciais do programa, pois são associações intrafunções, ou seja, a definição e uso da variável ocorrem na mesma função *f*. No caso da associação m-uso considera-se a

existência de um segundo processo p_2 que irá receber a mensagem, caracterizando a comunicação/sincronização interprocessos. Isso é similar ao definido em Souza et al (2013). Considerando o uso de funções e recursão, este modelo estabelece dois novos tipos de associações. A associação f-uso é dada pela tripla ($n_i^{p,f1}$, ($n_i^{p,f1}$, $n_i^{p,f2}$), (d, *t*)), tal que ($n_i^{p,f1}$, (d, *t*)) é uma *f-def* ou *c-def* de *d* com identificação *t*, ($n_i^{p,f1}$, $n_i^{p,f2}$) $\in E^p_{sync}$, sendo um f-uso de *d*. e existe um caminho livre de definição (Souza et al., 2007) com relação a *d* de ($n_i^{p,f1}$) para ($n_i^{p,f1}$, $n_i^{p,f2}$). Essa associação captura a comunicação interfunções de um processo. A associação r-uso é dada pela tripla ($n_i^{p,f1}$, $n_j^{p,f1}$), ($n_i^{p,f1}$, $n_j^{p,f1}$), (d, *t*)), tal que ($n_i^{p,f1}$, $n_j^{p,f1}$), (d, *t*)) tem uma *f-def* ou *c-def* e um c-uso ou p-uso de *d* com identificação *t*, ($n_i^{p,f1}$, $n_j^{p,f1}$) $\in E_{stack}$, sendo um r-uso de *d* e existe um caminho livre de definição com relação a *d* de ($n_i^{p,f1}$) para ($n_i^{p,f1}$, $n_j^{p,f1}$). O foco dessa associação é o empilhamento da recursão.

Com base nessas definições é possível estabelecer uma família de critérios estruturais para programas Erlang. O modelo considera os seguintes critérios estabelecidos em (Souza et al., 2013): Todos-Nós, Todas-Arestas, Todos-Nós-sync, Todas-Arestas-sync, Todos-c-Usos, Todos-p-Usos e Todos-m-Usos.

Para testar a comunicação entre funções e também as chamadas recursivas de funções foram definidos os seguintes critérios de fluxo de controle e de comunicação: **Todos-Nós-Call** requer que todos os nós do conjunto N_{call} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste; **Todos-Nós-Return** requer que todos os nós do conjunto N_{return} sejam exercitados pelo menos uma vez pelo conjunto de casos de teste; **Todos-Nós-Recursive** requer que todos os nós do conjunto $N_{recursive}$ sejam exercitados pelo menos uma vez pelo conjunto de casos de teste; **Todas-Arestas-stack** requer que todas as arestas dos conjuntos E_{stack} sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste; **Todas-Arestas-unstack** requer que todas as arestas dos conjuntos $E_{unstack}$ sejam exercitadas pelo menos uma vez pelo conjunto de casos de teste.

Novos critérios baseados em fluxo de dados foram definidos para testar as associações descritas na seção anterior: **Todos-r-Usos** requer que todas as associações r-uso sejam exercitadas e **Todos-f-Usos** requer exercitar todas as associações f-uso.

4. Exemplo de aplicação

Esta seção ilustra como os dados definidos anteriormente são coletados de modo a gerar o GFCP e aplicar os critérios de teste, considerando o programa exemplo da Figura 1. Apesar de simples, esta aplicação possui as características necessárias para a aplicação dos critérios de teste, sendo concorrência, chamadas de função e recursão. O processo p^m representa o processo principal e p^0, p^1 e p^2 são os processos escravos.

A Figura 2 apresenta o GFCP para o programa GCD da Figura 1, pode ser observado que as arestas foram identificadas e ilustradas com cores distintas.

A Tabela 1 contém alguns dos conjuntos obtidos a partir do modelo de teste proposto e que serão utilizados para geração dos requisitos de teste. A Tabela 2 mostra alguns dos elementos requeridos para os critérios propostos.

Cada caminho pode ser executado por um conjunto de dados de teste (entrada) para o programa. Por exemplo, considerando os dados de teste a DT1 = {NumX = 1, NumY = 2, NumZ = 1}, os caminhos executados são: π_{p^m} = {0, 1, 2, 3, 4, 5, 8, 9, 10,

11, 14, 17, 6, 7}; πp^0 = {0, 1, 2, 3, 6, 7, 10, 6, 9, 11, 12, 10, 7, 12, 4, 5}; πp^1 = {0, 1, 2, 3, 6, 8, 10, 6, 9, 11, 12, 10, 8, 12, 4, 5}. O processo p^2 não executa porque o resultado já foi produzido pelos processos p^0 e p^1. Para esta execução, considerando os critérios Todos-m-Usos, Todos-f-Usos e Todos-r-Usos. O percentual de cobertura foi de 66%, 70% e 66%, respectivamente.

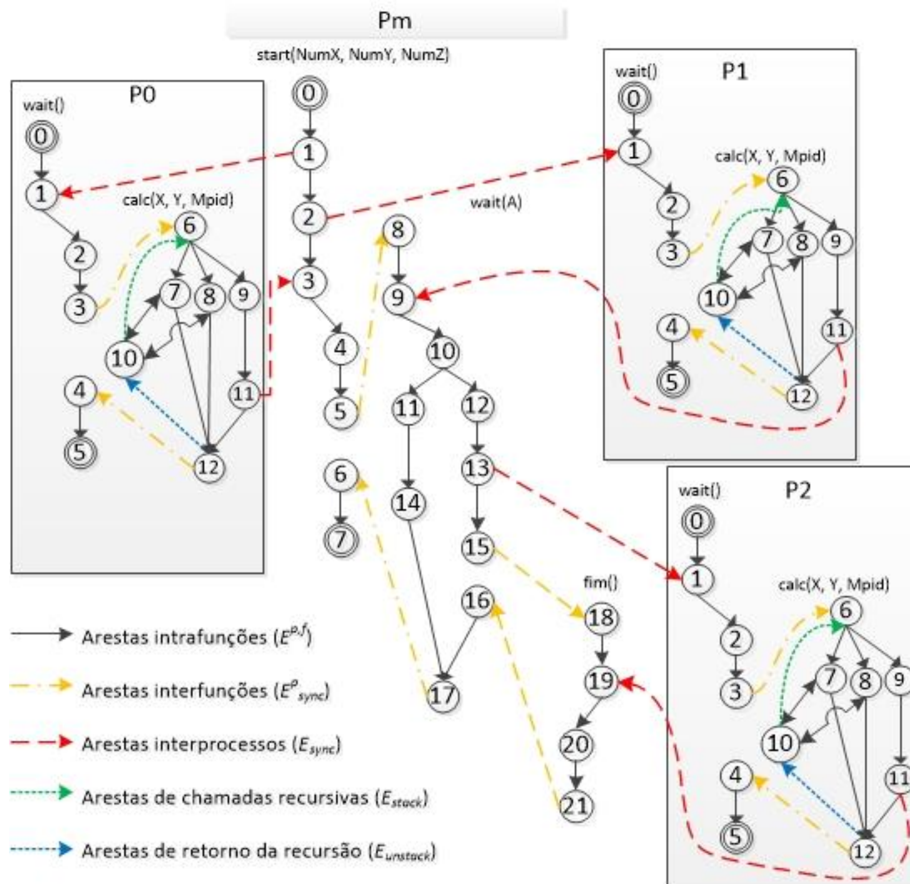


Figura 2. Programa GCD Principal e Programa GCD Escravo.

5. Trabalhos Relacionados

Em Oliveira et al. (2015) são apresentados os resultados de um mapeamento sistemático sobre teste aplicado a linguagens funcionais. Uma síntese dos trabalhos desse mapeamento sistemático que mais se relacionam com este artigo é descrita a seguir.

Existem trabalhos que definem modelos e critérios para o teste estrutural de programas Erlang, como Widera (2004) que descreve um modelo de teste que abrange um subconjunto de funções Erlang para programas sequenciais. Em Widera (2005) o modelo é estendido para suportar *higher-order functions* e complementa o modelo incluindo aspectos de concorrência. Guo et al. (2009) definem um modelo e a partir deste extraem informações para representar predicados e medir suas coberturas.

Considerando o contexto de programas concorrentes, Christakis e Sagonas (2011) apresentam uma técnica para detectar erros relativos à passagem de mensagem em Erlang. Gotovos et al. (2011) desenvolveram uma ferramenta de teste chamada *Concuerror* para auxiliar no desenvolvimento guiado por teste (TDD). Essa ferramenta

utiliza casos de teste para detectar erros relativos à concorrência, tais como, *deadlocks* e condições de disputa em programas Erlang.

Tabela 1. Informações extraídas pelo modelo de teste para o programa gcd.

n	4	Prog	p^m, p^0, p^1, p^2		
p^m	$\{f^{m,0}, f^{m,1}, f^{m,2}\}$	p^0	$\{f^{0,0}, f^{0,1}\}$		
p^1	$\{f^{1,0}, f^{1,1}\}$	p^2	$\{f^{2,0}, f^{2,1}\}$		
N^m	$\{n_0^{m,0} \ldots, n_7^{m,0}\}, \{n_8^{m,1} \ldots, n_{17}^{m,1}\}$ e $\{n_{18}^{m,2} \ldots, n_{21}^{m,2}\}$				
N^0	$\{n_0^{0,0}, \ldots, n_5^{0,0}\}$ e $\{n_6^{0,1} \ldots, n_{12}^{0,1}\}$				
N^1	$\{n_0^{1,0}, \ldots, n_5^{1,0}\}$ e $\{n_6^{1,1} \ldots, n_{12}^{1,1}\}$				
N^2	$\{n_0^{2,0}, \ldots, n_5^{2,0}\}$ e $\{n_6^{2,1} \ldots, n_{12}^{2,1}\}$				
D^m	{(d0, ´NumX´), (d1, ´NumY´), (d2, ´NumZ´), ….				
D^m_{copy}	{(d4, ´X´), (d5, ´Pid´), (d6, ´A´), (d7, ´B´), ….				
C	$\{c_0, c_1\}$, onde c_0 = ´send´ e c_1 = ´receive´				
N_{sync}	$\{(n_1^{m,0}, c_0, \text{´BS´}), (n_2^{m,0}, c_0, \text{´BS´}), (n_3^{m,0}, c_1, \text{´BR´}), (n_9^{m,1}, c_1, \text{´BR´}), (n_{13}^{m,1}, c_0, \text{´BS´}), (n_{19}^{m,2}, c_1, \text{´BR´}), (n_1^{0,0}, c_1, \text{´BR´}), (n_{11}^{0,1}, c_0, \text{´BS´}), (n_1^{1,0}, c_1, \text{´BR´}), (n_{11}^{1,1}, c_0, \text{´BS´}), (n_1^{2,0}, c_1, BR), (n_{11}^{2,1}, c_0, \text{´BS´})\}$				
N_{call}	$\{(n_5^{m,0}, f^{m,0}, \text{´C´}), (n_{15}^{m,1}, f^{m,1}, \text{´C´}), (n_3^{0,0}, f^{0,0}, \text{´C´}), (n_3^{1,0}, f^{1,0}, \text{´C´}), (n_3^{2,0}, f^{2,0}, \text{´C´})\}$				
N_{return}	$\{(n_6^{m,0}, f^{m,0}, \text{´R´}), (n_{16}^{m,1}, f^{m,1}, \text{´R´}), (n_4^{0,0}, f^{0,0}, \text{´R´}), (n_4^{1,0}, f^{1,0}, \text{´R´}), (n_4^{2,0}, f^{2,0}, \text{´R´})\}$				
$N_{recursive}$	$\{(n_{10}^{0,1}, f^{0,1}, \text{´*´}), (n_{10}^{1,1}, f^{1,1}, \text{´*´}), (n_{10}^{2,1}, f^{2,1}, \text{´*´})\}$ * não é possível obter esta informação estaticamente, pois a recursão ocorre durante a execução - é um aspecto dinâmico.				
E^m	$\{(n_0^{m,0}, n_1^{m,0}), (n_1^{m,0}, n_2^{m,0}), (n_2^{m,0}, n_3^{m,0}), \ldots\}$				
E^0	$\{(n_0^{0,0}, n_1^{0,0}), (n_1^{0,0}, n_2^{0,0}), (n_2^{0,0}, n_3^{0,0}), \ldots\}$				
E^1	$\{(n_0^{1,0}, n_1^{1,0}), (n_1^{1,0}, n_2^{1,0}), (n_2^{1,0}, n_3^{1,0}), \ldots\}$				
E^2	$\{(n_0^{2,0}, n_1^{2,0}), (n_1^{2,0}, n_2^{2,0}), (n_2^{2,0}, n_3^{2,0}), \ldots\}$				
E_{sync}	$\{(n_1^{m,0}, n_1^{0,0}), (n_1^{m,0}, n_1^{1,0}), (n_1^{m,0}, n_1^{2,0}), (n_2^{m,0}, n_1^{0,0}), (n_2^{m,0}, n_1^{1,0}), (n_2^{m,0}, n_1^{2,0}), (n_{13}^{m,1}, n_1^{0,0}), (n_{13}^{m,1}, n_1^{1,0}), (n_{13}^{m,1}, n_1^{2,0}), (n_{11}^{0,1}, n_3^{m,0}), (n_{11}^{0,1}, n_9^{m,1}), (n_{11}^{0,1}, n_{19}^{m,2}), (n_{11}^{1,1}, n_3^{m,0}), (n_{11}^{1,1}, n_9^{m,1}), (n_{11}^{1,1}, n_{19}^{m,2}), (n_{11}^{2,1}, n_3^{m,0}), (n_{11}^{2,1}, n_9^{m,1}), (n_{11}^{2,1}, n_{19}^{m,2})\}$				
E^m_{sync}	$\{(n_5^{m,0}, n_8^{m,1}), (n_{17}^{m,1}, n_6^{m,0}), (n_{15}^{m,1}, n_{18}^{m,2}), (n_{21}^{m,2}, n_{16}^{m,1})\}$				
E^0_{sync}	$\{(n_3^{0,0}, n_6^{0,1}), (n_{12}^{0,1}, n_4^{0,0})\}$				
E^1_{sync}	$\{(n_3^{1,0}, n_6^{1,1}), (n_{12}^{1,1}, n_4^{1,0})\}$				
E^2_{sync}	$\{(n_3^{2,0}, n_6^{2,1}), (n_{12}^{2,1}, n_4^{2,0})\}$				
E_{stack}^0	$\{(n_{10}^{0,1}, n_6^{0,1})\}$	E_{stack}^1	$\{(n_{10}^{1,1}, n_6^{1,1})\}$	E_{stack}^2	$\{(n_{10}^{2,1}, n_6^{2,1})\}$
$E_{unstack}^0$	$\{(n_{12}^{0,1}, n_{10}^{0,1})\}$	$E_{unstack}^1$	$\{(n_{12}^{1,1}, n_{10}^{1,1})\}$	$E_{unstack}^2$	$\{(n_{12}^{2,1}, n_{10}^{2,1})\}$

Widera (2004) propõe um conjunto de critérios de teste baseados no fluxo de dados para Erlang. Dando continuidade, em Widera (2005) o autor introduz o conceito de *du-chain*, que é uma sequência de pares de definição e uso. Isso se faz necessário devido aos dados em Erlang serem imutáveis, quando é necessário alterar o valor de uma variável uma cópia é feita para outra variável, desencadeando uma sequencia de definições e usos de variáveis. Baseado neste conceito, Widera (2005) estabeleceu critérios de cobertura para essas sequencias de definições e usos. O autor considera o *trace* no fluxo de dados de códigos de Erlang e descreve as propriedades de um protótipo de interpretador para GFC. O interpretador instrumenta o código-fonte para avaliar partes do GFC que são cobertos pelos casos de teste.

No geral, os trabalhos não aplicam corretamente as técnicas de teste. Não exploram o processo de testes, tais como, geração de casos de teste e avaliação de atividade de teste. Os trabalhos não tiveram como foco testar a qualidade dos programas e sim, objetivos mais específicos como, refatoração para identificar códigos semelhantes, teste de regressão para diminuição de casos de teste e detecção de *deadlocks*.

Tabela 2. Alguns elementos requeridos pelos critérios de teste para o programa GCD.

Critérios	**Elementos requeridos**		
Todos-Nós	$n_0^{m,0}, \ldots, n_{21}^{m,2}, n_0^{0,0}, \ldots$	Todas-Arestas	$(n_0^{m,0}, n_1^{m,0}), \ldots, (n_3^{m,0}, n_4^{m,0}), (n_0^{0,0}, n_1^{0,0}), \ldots,$
Todos-Nós-sync	$(n_1^{m,0}, c_0, \text{´BS´}), (n_2^{m,0}, c_0, \text{´BS´}), (n_3^{m,0}, c_1, \text{´BR´}), \ldots.$		
Todas-Arestas-sync	$(n_1^{m,0}, n_1^{0,0}), (n_1^{m,0}, n_1^{1,0}), (n_1^{m,0}, n_1^{2,0}), (n_2^{m,0}, n_1^{0,0}), \ldots$		
Todos-Nós-Call	$(n_5^{m,0}, f^{m,0}, \text{´C´}), (n_{15}^{m,1}, f^{m,1}, \text{´C´}), (n_3^{0,0}, f^{0,0}, \text{´C´}), (n_3^{1,0}, f^{1,0}, \text{´C´}), (n_3^{2,0}, f^{2,0}, \text{´C´})$		
Todos-Nós-Return	$(n_6^{m,0}, f^{m,0}, \text{´R´}), (n_{16}^{m,1}, f^{m,1}, \text{´R´}), (n_4^{0,0}, f^{0,0}, \text{´R´}), (n_4^{1,0}, f^{1,0}, \text{´R´}), (n_4^{2,0}, f^{2,0}, \text{´R´})$		
Todos-Nós-Recursive	$(n_{10}^{0,1}, f^{0,1}, \text{´*´}), (n_{10}^{1,1}, f^{1,1}, \text{´*´}), (n_{10}^{2,1}, f^{2,1}, \text{´*´})$		
Todas-Arestas-stack	$(n_{10}^{0,1}, n_6^{0,1}), (n_{10}^{1,1}, n_6^{1,1}), (n_{10}^{2,1}, n_6^{2,1})$		
Todas-Arestas-unstack	$(n_{12}^{0,1}, n_{10}^{2,1}), (n_{12}^{0,1}, n_{10}^{2,1}), (n_{12}^{2,1}, n_{10}^{2,1})$		
Todos-c-Usos	$(n_0^{m,0}, n_1^{m,0}, d0), (n_0^{m,0}, n_1^{m,0}, d1), (n_0^{m,0}, n_1^{m,0}, d3), \ldots$		
Todos-p-Usos	$(n_8^{m,0}, (n_{10}^{m,0}, n_{11}^{m,0}), d6), (n_9^{m,0}, (n_{10}^{m,0}, n_{12}^{m,0}), d7)\ldots$		
Todos-m-Usos	$(n_0^{m,0}, (n_1^{m,0}, n_1^{0,0}), d0), (n_0^{m,0}, (n_1^{m,0}, n_1^{0,0}), d1), \ldots$		
Todos-f-Usos	$(n_3^{m,0} , (n_5^{m,0}, n_8^{m,1}), (X, d4)), (n_1^{0,0} , (n_3^{0,0}, n_6^{0,1}), (X, d12)), \ldots$		
Todos-r-Usos	$((n_6^{0,1}, n_7^{0,1}), (n_{10}^{0,1}, n_6^{0,1}), (A, d15)), ((n_6^{0,1}, n_8^{0,1}), (n_{10}^{0,1}, n_6^{0,1}), (A, d15)),\ldots$		

Alguns trabalhos tiveram como foco a especificação de critérios de teste estrutural. Entretanto, faltam pesquisas sobre como derivar os testes de programas funcionais e como extrair informações relevantes desses programas, a fim de orientar a atividade de teste. Faltam estudos que analisem ferramentas e critérios de teste.

6. Conclusão

O teste de programas Erlang não é uma tarefa trivial, em razão de seu paradigma funcional. Este artigo contribui neste sentido, propondo um conjunto de critérios de teste para apoiar o teste desses programas, focando na comunicação entre funções, comunicação entre processos, sincronização, concorrência e recursividade.

O modelo de teste proposto captura as características citadas anteriormente para a linguagem funcional Erlang. O modelo de teste foi baseado no modelo proposto por Souza et al. (2013), que originalmente foi proposto para o paradigma imperativo e necessitou de adaptações para o paradigma funcional. Os critérios propostos são os primeiros passos no sentido de contribuir para o teste estrutural de programas Erlang.

Para o avanço da pesquisa destacam-se alguns desafios: adaptação das características da linguagem Erlang para especificar critérios de teste estrutural, realização da instrumentação do código para gerar arquivo de rastro para uma melhor análise dos critérios e a necessidade de ferramentas que auxiliem o processo de análise.

Este artigo possui algumas limitações. O modelo não considera tolerância a falhas e não especifica critérios para controlar a sequência de pares de definições e usos

de variáveis. O artigo não explora a eficácia dos critérios propostos e também não avalia a relação de inclusão dos mesmos. Faz-se necessária uma análise melhor dos critérios propostos, por meio de experimentos que considerem aplicações mais complexas. Espera-se que tais aplicações simulem mais profundamente as principais características da linguagem Erlang, bem como os critérios de teste em relação à capacidade de revelar defeitos. Tais limitações serão consideradas durante a evolução desta pesquisa.

Agradecimentos

Os autores agradecem o apoio financeiro da FAPESP sob o processo nº 2013/01818-7.

Referências

Armstrong, J. Concurrency Oriented Programming in Erlang. Invited talk, FFG. 2003.

Balakrishnan, A; Anand, N. Development of an automated testing software for real time systems. In Int. Conf. on Industrial and Inf. Systems (ICIIS), 2009, pp. 193 - 198.

Cesarini, F. and Thompson, S. Erlang Programming - A Concurrent Approach to Software Development. O'Reilly Media, 2009. 496p.

Christakis, M.; Sagonas, K. Detection of asynchronous message passing errors using static analysis. In: 13th Int. Conf. on Practical Aspects of Declarative Languages, PADL'11, p.5-18, USA, January 24-25, 2011.

Gotovos, A.; Christakis, M.; Sagonas, K. Test-driven development of concurrent programs using concuerror. In 10th ACM SIGPLAN workshop on Erlang (Erlang '11). ACM, New York, 2011.

Guo, Q.; Derrick, J.; Walkinshaw, N. Applying Testability Transformations to Achieve Structural Coverage of Erlang Programs. International Conference on Testing of Software and Communication Systems a, Netherlands, November 2-4, 2009.

Logan, M.; Merritt, E.; Carlsson, R. Erlang and OTP in Action. Manning Pub. 2010.

Myers, G. J. The Art of Software Testing. 2 ed. John Wiley & Sons, 2004.

Oliveira, A. P., Souza, P. S. L., Souza, S. R. S., Estrella, J. C., Bruschi, S. M. A Systematic Mapping about Testing of Functional Programs. In: 13th Int. Conf. on Software Engineering Research and Practice (SERP). Las Vegas, July 27-30, 2015.

Sarmanho, F.; Souza, P. S. L.; Souza, S. R.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: ICCS'08 - 8th Int. Conf. on Computational Science, Part I, Berlin, Springer-Verlag, 2008, pp. 337–346.

Sebesta, R. W. Concepts of programming languages. 10th ed. 2009

Souza, S. R. S.; Souza, P. S. L.; Vergilio, S. R. Teste de programas concorrentes. In: M. E. Delamaro; J. C. Maldonado; M. Jino. (Org.). Introdução ao Teste de Software. 1ª ed. São Paulo: Elsevier Ed Ltda, 2007, v. 1, p. 231-250.

Souza, P. S. L.; Souza, S. R. S.; Rocha, M. G.; Prado, R. R.; Batista, R. N. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In: ICCS - Int. Conf. on Computational Science, Barcelona, 2013, pp. 149–158.

Tate, Bruce A. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. Pragmatic Bookshelf, 2010.

Widera, M. Flow graphs for testing sequential Erlang programs. In Proceedings of the 3rd ACM SIGPLAN Erlang Workshop. ACM Press, 2004.

Widera, M. Data flow coverage for testing Erlang programs. In Marko van Eekelen, In: TFP'05 - Sixth Symposium on Trends in Functional Programming, Sep, 2005.

Challenges in Testing Context Aware Software Systems

Santiago Matalonga[1], Felyppe Rodrigues[2], Guilherme H. Travassos[2]

[1]Universidad ORT Uruguay. Cuareim 1471, 11100, Montevideo, Uruguay

[2] Universidade Federal do Rio de Janeiro/COPPE/PESC. Rua Horácio Macedo, 2030
Cid. Universitária Rio de Janeiro–RJ 21941-450

[1]smatalonga@uni.ort.edu.uy, [2]{felyppers,ght}@cos.ufrj.br

Abstract. *Context aware software systems (CASS) are becoming pervasive in our lives. Nevertheless, it is not clear whether traditional (no- context aware) software testing techniques are adequate for testing CASS. Therefore, a quasi systematic literature review was used to identify 11 relevant sources that mentioned 15 problems and 4 proposed solutions, which were analyzed and classified into 3 groups of challenges and strategies for dealing with CASS testing. Additionally, some recommendations for testing such software applications with the currently available testing technologies are presented. However, we argue that new context aware testing techniques need to be developed in order to assure the quality of CASS.*

1. Introduction

Context aware systems are becoming even more pervasive in our lives, from recommendation software using our information to select songs to wearables, monitoring our biometric information and acting in response to it. At this work, *context* is*"any piece of information that may be used to characterize the situation of an entity (logical and physical objects present in the system's environment)*", and *context aware* is the *"dynamic property representing a piece of information that can evolutionarily affect the overall behavior of the software system in the interaction between the actor and computer"* [Mota 2013]. Therefore, when one or more behaviors of a software application are affected by information it can sense from the context where it is being executed, such application is defined to be context aware software application (CASS).

Testing allows the observation and validation of software application's behaviors. However, one of the issues when designing software tests is always concerned with the tradeoff decision between the testing set completeness and the resources available to execute the software testing process. When context awareness features come into play, and at the very least, the available test space from which test cases are selected can significantly increase, consequently increasing the testing partiality and in some sense making unfeasible the using of traditional testing technologies (those ones used to test no context aware software).

In order to identify the available knowledge on testing CASS, a *quasi*-systematic literature review (qSLR) had been undertaken under the umbrella of CNPq - CAcTUS project. The qSLR protocol execution revealed 11 technical papers that provided

evidence regarding challenges and solutions on testing CASS. Analyzing these selected sources, a set of 15 issues and four proposed solutions were identified. Furthermore, by means of applying qualitative analysis techniques, 15 problems were abstracted intending to represent two distinct challenges: (1) How to deal with CASS testing within "*Device and hardware constraints*" and (2) How to deal with all possible combinations of "*context variations*" when designing test cases. These two abstractions enabled us to observe that three out of the five proposed solutions have been experimentally evaluated deal with the challenge of mitigating the "context variation" problem. No experimental results dealt with other aspects of context aware like "Device and Hardware constrains".

Therefore, upon analysis of these results, we claim that new alternatives for testing CASS need to be developed in order to properly evaluate the quality of such software systems. Current identified proposals tend to fix the value of context variables during test case design. We argue that these strategies are limited in light of the identified challenges (especially the problems associated with "Context Variation"), since the tester will always be faced with trade off decisions between coverage and test effort/cost. Therefore, we envision the emergence of Test Design and Execution techniques especially targeted at CASS, where the evaluation of the unit under test is performed under the unconstrained variation of the context.

This paper is organized as follows. Section 2 presents the research method, with an initial analysis of the selected technical literature in section 2.1. A summary of the selected sources is provided in section 3. Section 4 presents issues practitioners have to take into account when applying the identified solutions to their environment. In section 5 we present recommendations for testing CASS. Section 6, presents our vision for context aware software testing. Finally, in section 7 we conclude this paper and call for new directions of research for context aware software testing.

2. Research method

In order to acquire the state of the art in challenges on testing CASS, a *quasi* Systematic Literature Review (qSLR) has been undertaken. A qSLR follows the same formalism of a Systematic Literature Review, but due to the maturity of the field under study it is still not possible to aggregate data through comparative meta analysis [Travassos et al. 2008]. We make this distinction when the technical literature of the phenomenon under study cannot provide a comparison baseline, yet the rigor and completeness of the secondary study are not affected. A key strength of this research strategy is its capacity to acquire evidence-based knowledge. Even though, for the sake of brevity and simplicity, we do not go into the qSLR protocol details [Rodrigues et al. 2014][1], this section describes the highlights.

---

[1] See the full protocol available at CACTUS portal http://lens.cos.ufrj.br/cactus

The main research question is "*What are the existing methods for testing context aware software systems?*". The search string used to retrieve the primary sources from the search engines was structured using the PICO strategy [Pai et al. 2004]. ***Population*** **was defined to be "Sensibility to context".** ***Intervention*** **was defined to be "Software testing". No** ***Comparison*** **is available.** ***Outcome*** **was defined to be "Methodology".** The resulting set of keywords was:

"context aware" OR "event driven" OR "context driven" OR "context sensitivity" OR "context sensitive" OR "pervasive" OR "ubiquitous" OR "usability" OR "event based" OR "self adaptive" OR "self adapt" AND "software test design" OR "software test suite" OR "software test" OR "software testing" OR "system test design" OR "system test suite" OR "system test" OR "system testing" OR "middleware test" OR "middleware testing" OR "property based software test" OR "property based software testing" OR "fault detection" OR "failure detection" OR "GUI test" OR "Graphical User Interfaces test" AND" model" OR "metric" OR "guideline" OR "checklist" OR "template" OR "approach" OR "strategy" OR "method" OR "methodology" OR "tool" OR "technique" OR "heuristics"

The search engines were selected due their coverage and previous experience of the researchers. These were IEEEXplore, Scopus and Web of Science. Google Scholar and ACM Digital library where discarded because the implementation of their search algorithm does not favor the repeatability of the results [Gehanno et al. 2013]. A total of 1820 potential sources were retrieved using the aforementioned search string in the search engine. The research method application (filtering through applying the inclusion and exclusion criteria) narrowed them down to the 11 technical sources described in this paper (see section 3). The raw numbers of the filtering process is explained in the fact that a qSLR looks for evidence based in the application of the scientific method. Many relevant and interesting research has been carried out in the topic of testing context aware software systems, yet, our interpretation is that only those 11 primary sources unequivocally report that their research method applies the scientific method.

2.1 Data Analysis

From each of the 11 technical primary sources (Column "Source" in Table 1) we extracted the Problems and Solutions that the authors mentioned context awareness brings to testing. We further distinguished them among those that were part of the experimental setting described in the source from those that were mentioned or cited.

Table 1 - Problems and solutions for Testing CAS

Source	Mentioned Problems	Proposed Solutions	Experimentally Observed Problems	Experimentally evaluated Solutions
[Alsos and Dahl 2008]	-	-	-	-
[Amalfitano et al. 2013]	Resource scarceness Variety of running conditions Integration of devices	Extensive platform testing	Variety of running conditions	Exploit defined scenarios with mutation
[Jiang et al. 2007]	Devices resources constraints Many usage scenarios which cannot be tested due to the high interactivity between devices and the environment	-	-	-
[Canfora et al. 2013]	-	-	-	-
[Merdes et al. 2006]	-	-	-	-
[Ryan and Gonsalves 2005]	Diversity of hardware Difficulties in collecting user reactions	-	-	-
Satoh [Satoh 2003]	-	-	-	-

[Tse et al. 2004]	Prohibitive number of possible situations Behavior changes when context changes	-	Prohibitive number of possible situations	Automatically generate variations of know situations "metamorphic testing"
[Wang et al. 2007]	Determining and controlling when and how to feed a series of changing context Integration with thick middleware Anticipation of all relevant context changes, when they could impact behavior Asynchronous communication of context information	-	asynchronous communication of location information	Proposed Process[Identify points in the program on which it can be affected by the context (CAPPS), Test for CAPPS, Mutate for the rest]
[Wang and Chan 2009].	Massive volume of context provide unprecedented details to process	-	-	-
[Wang et al. 2014]	-	-	-	-

In Table 1, the reader can observe that only three proposals (column "Experimentally Observed Problems") deal with one of the identified problems (column "Mentioned Problems"). In order to elaborate further on this result, we applied content analysis [Weber 1990]. Through this technique, the extracted data presented in Table 1 was analyzed for proximity and sorted in chunks in order to look for and derive abstractions. The results of this analysis are presented in graph format in Figure 1.

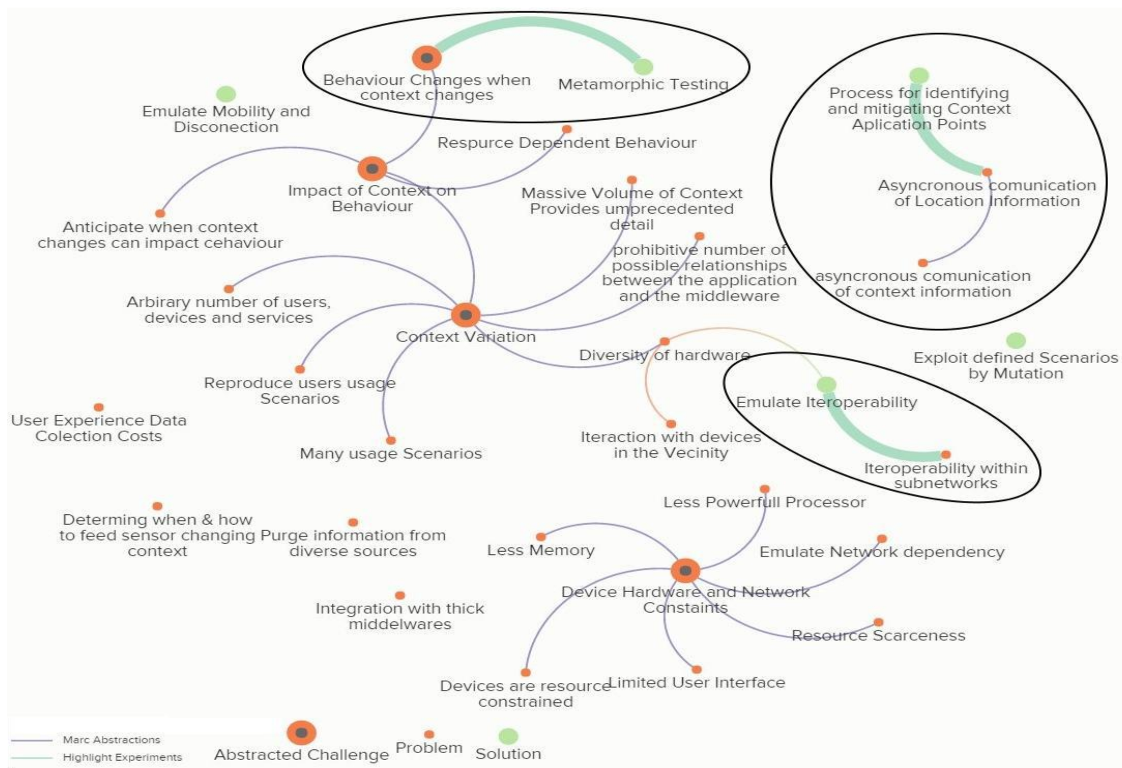


Figure 1 Relationship between challenges and solutions

Hubs with dark bull's eyes represent challenges resulting from the analysis. Thick connector lines link the problems that have been experimentally studied against proposed solutions. Thin lines present the associations only mentioned in the technical literature. The two main hubs in Figure 1 represent the highest abstraction levels of challenges identified in the selected technical literature (reported in Table 1). These are:

- Device hardware and network constraints. The star-like layout of connected nodes show that no proposed solution has been experimented for this challenge.

- Context variation; which is the abstraction standing for several identified problems (for instance 'Many usage scenarios' or "diversity of hardware") that deal with the problems associated with the impact of variation of one or more context variables (for instance, associations from the problem nodes to the solutions show how the researchers have dealt with the problem).

This analysis also gave way to the possibility of classifying the selected technical literature in one of three categories: *Usability*, for those sources dealing with testing problems or evaluating CASS usability; *Tool validation*, for those sources which aims at evaluating tools for testing CASS and; *Solve Challenge*, for those sources providing experimental data in solving one of the identified problems. The following section presents the selected technical sources description grouped by this categorization.

3. Summary of Findings

3.1 Evaluating usability of CASS

Ryan and Gonsalves (2005) present an evaluation on the effects of device diversity and mobility on the usability of CASS. Their experimental design involves the evaluation of the usability of the same functionality in four different platforms. The technology used for this experiment has already been phased out; therefore, generalization is no longer possible. Their results show that application type was not a factor that influenced user satisfaction, and that location is a factor for the preferences of users regarding platform selection.

Alsos and Dahl (2008) present the design and construction of a simulated environment for usability testing in the Healthcare Information Technology (HCIT) solutions. They argue that generating controlled environments for testing usability of HCIT solutions is expensive and their experiment is oriented towards identifying usability problems in the laboratory or in the intended deployment environment. The only context aware variable concerned with the experiment is the indoor location within the healthcare facility.

Canfora et al. (2013) present the design and development of a platform designed to automate the usability evaluation of devices in the field. Their area of application is a proprietary brand of embedded software, and their study aims at identifying and understanding the variation of usage scenarios for these proprietary devices running in the field. Their results show how their solution can identify more user experience problems than using survey-based tools for data collection.

3.2 Tools for testing CASS

Merdes et al. (2006) present the design of a similar framework for testing CASS. This proposal is designed around the testing of resource utilization and constraints of mobile devices. They define test cases by constraining the access to shared resources of CASS running in smart devices.

Jiang et al. (2007) present the design and evaluation of a black box testing tool for the automation of software testing for context aware applications in smart devices. The

implementation is provided by the Symbian platform−which is no longer relevant in the smart phone market. Nonetheless, their results show that there is benefit in developing such a tool to automate black box testing.

Wang and Chan (2009) propose the metric of Context Diversity as a proxy for the coverage of CASS under test. This metric was evaluated against a controlled set of Test Suites designed by the authors. In Wang et al. (2014), their hypothesis is further evaluated with a huge test suite generated by applying mutation algorithms. The results of both papers sustain the hypothesis that Test Suites ranking higher in context diversity also attain more coverage.

3.3 Proposed solutions for the challenges on testing CASS

Satoh (2003) is concerned with the location awareness of devices. To solve this issue, a sandbox environment (emulator) had been created. Satoh presents a case study of the proposed emulator by describing a scenario where the main use case is the acquisition of remote printing capabilities.

Tse et al. (2004) is concerned with the context variation problem. Their strategy is to dynamically generate test cases via mutation. In addition to this, Tse et al. (2004) provide a mechanism for defining the expectation of the generated test cases output by abstracting relationships between inputs and outputs regarding a predefined set of test cases.

Wang et al. (2007) present another framework for testing CASS. This framework is based on the idea of CAPPs. By identifying these CAPPs the authors suggest the definition of test cases by using structural analysis testing. These CAPPs are identified in the application code, so that the tool can exploit several pre-defined strategies to generate new test cases by navigating the call structure. This approach is similar to [Tse et al. 2004] and [Amalfitano et al. 2013]) in that it first relies on human-generated test cases, and then use computing power to dynamically generate more test cases in order to mitigate the challenge of "Context Diversity".

Amalfitano et al.(2013) design test cases taking the context into account. They propose a two-step process on which the tester first designs test cases using context variables as input, and then applies mutation testing techniques to dynamically generate more test cases. They present a case study where they implement this approach to deal with an android device with four apps downloaded from the android store market.

4. Threats to the generalization of the identified approaches.

According to the results presented in the previous section, a practitioner with the need of testing CASS is faced with a limited set of empirically validated options, even then presenting limited coverage.

First of all, the identified strategies for evaluating usability of CASS all rely on the investment in infrastructure. These investments do not appear to be reusable for a third party. For instance, the results by Ryan and Gonsalves (2005) need to be reviewed since

the technology used is no longer relevant. Merdes et al.(2006)'s solution was implemented for proprietary hardware, and Alsos and Dahl (2008)'s approach requires extensive investment in physical infrastructure.

The proposal by Wang and Chan (2009) has been evaluated with the aim of helping practitioners to determine the expected test suite coverage. They do not provide guidance to generate test suites, however the practitioner can use their proposed metric to evaluate whether new test cases enhance the system under test coverage.

Finally, the identified technical literature has proposed and empirically evaluated solutions to the challenge of dealing with the "context diversity" ([Satoh 2003], [Tse et al. 2004], and [Amalfitano et al. 2013]). Despite the importance of these initial evidence, some limitations can be observed. For instance, the approach by Satoh (2003) had been empirically evaluated only for the context variable "location". The other approaches suffer from the following two limitations. Firstly, they rely on software platforms, which can become outdated due to technology shifts; Secondly, they rely on mutation to produce test suites, which represents a software testing technique that might be useful to evaluate robustness, but does not guarantee functional correctness.

5. Recommendations for testing CASS.

This section presents recommendations for testing CASS that can be abstracted from the identified technical literature aiming at to partially mitigate the lack of specialized software technologies to support such software applications testing.

Build quality in. This recommendation can be enacted with: *Assure the functionality of the software system.* Do it by testing the functional requirements using traditional test cases. This is a pre-requisite in some of the proposals presented in the papers ([Satoh 2003], [Tse et al. 2004], [Amalfitano et al. 2013] . *Incorporate context aware variables in traditional test case techniques*. For instance, subject to the limitations discussed in the following section, context aware variables can be used in Branch Testing design technique [Wang and Chan 2009]. *If possible, automate the test cases definition and execution*. The automation of test cases and their execution, when feasible, will reduce the overhead in regression testing and enable the execution of dynamic techniques like random or mutation testing [Tse et al. 2004].

Identify context variables in the requirements to evaluate where things could go wrong. It is highly likely that defects in CASS will come with unforeseen values feed to the system by the context [Wang et al. 2007], therefore identifying these CAPPS could be complemented by the use of defensive coding techniques.

Use automatic dynamic execution tools. Tools that automate the generation of execution scenarios–like those that provide Random Testing capabilities- (in spite that they cannot assure the correctness of the application) are useful to evaluate the robustness of the application, and has been observed useful for support the testing of CASS [Amalfitano et al. 2013].

6. Limitations of current approaches and CASS testing perspective

Even though the previous recommendations are applicable at the moment, they do not solve all the issues available in CASS. As seen in Table 1, the experimentally validated solutions only deal with the challenge of context variation. We feel this only partially addresses the reality of testing CASS. Although the technical literature has clearly identified other problems and challenges, we argue that the available identified solutions offer limited relief to the practitioners. From our list of recommendations: the first one (build quality in) only go so far as to assure functional correctness. It does not evaluate how different context variables can interact or vary during CASS execution. The second recommendation, offers some degree of mitigation to the limitation of the first one, but it is limited by the ad-hoc capacity on identifying CAPPS. And finally, automatic testing tools cannot warrant functional correctness, they can only address reliability.

Therefore, we envision the possibility of *context aware software testing* for CASS. It represents the evolution of test design techniques that do not design test cases to mitigate the challenges context awareness brings to software testing, but in contrast take advantage of the context variation to better cover the CASS under test.

7. Conclusions and on going works

Context aware software systems are mainstream. This means that software organizations are developing and deploying CASS. In order to identify the current available challenges for testing such systems, a *q*SLR was undertaken with the aim of finding empirical evidence on how CASS can be tested. The results of this review show that practitioners can count with a limited set of options to face the challenges the testing of CASS can bring. This paper has grouped the proposed solutions in three different sets.

First of all, in order to evaluate the *usability* of CASS, researchers have relied on the construction of expensive infrastructures to simulate selected aspects of the context in which the application is expected to run. Secondly, in order to *manage an expected big set of test cases*, researchers have built frameworks and infrastructure enabling practitioners to manage and generate massive amount of test cases in order to deal with the huge number of input combinations that context awareness brings into a software application. These strategies have been useful in the context of the presented case studies, but need constant update to keep up with the changes in technology and more research to mitigate the current threats to the generalization of the results. Finally, according to our findings, the variation in context values is the only challenge that has been explicitly tackled by research, although they do not take into account the context variation when performing the test cases. The common strategy from the identified research is to build a convenient set – in size and coverage - of test cases, and take advantage of a computer processing power to dynamically generate more test cases. But the drawback of these automation tools is that they do not necessarily increase coverage, and even so, the increased coverage does not necessary guarantee correctness in functionality.

Based on these challenges, we have abstracted a set of three recommendations that are readily available for practitioners who need to test CASS. First off all, practitioners should not rely only on testing to assure CASS quality (*build quality in*). Then, they should design the test cases by taking special consideration into the context variables. Finally, if possible, we recommend practitioners to take advantage of automatic dynamic execution tools to evaluate the robustness of the CASS under test.

Nevertheless, we believe that these results are not enough to evaluate the quality of the resulting software. The random addition of test cases, without a competent Test Oracle to verify the results, cannot guarantee neither quality nor correctness of functionality. Therefore, we claim that new software testing techniques need to be developed embracing the challenges concerned with context aware software testing. These techniques should not try to mitigate the challenges of context variation, but to accept the context changes as a reality of such software systems. These context aware software testing techniques should take advantage of this variation in order to ensure the quality of CASS.

8. Acknowledgements

This work was supported by "CAcTUS - Context-Awareness Testing for Ubiquitous Systems" project partially financed by CNPq – Universal 14/2013 Number 484380/2013-3. Prof. Travassos is a CNPq Researcher. Prof. Matalonga is supported by Banco Santander Uruguay grant for young researchers.

9. References

Alsos, O. A. and Dahl, Y. (2008). Toward a best practice for laboratory-based usability evaluations of mobile ICT for hospitals. *Proceedings of the 5th Nordic conference on Human-computer interaction building bridges - NordiCHI '08*, p. 3.

Amalfitano, D., Fasolino, A. R., Tramontana, P. and Amatucci, N. (2013). Considering Context Events in Event-Based Testing of Mobile Applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*.

Canfora, G., Mercaldo, F., Visaggio, C. A., et al. (2013). A case study of automating user experience-oriented performance testing on smartphones. In *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*.

Gehanno, J.-F., Rollin, L. and Darmoni, S. (2013). Is the coverage of Google Scholar enough to be used alone for systematic reviews. *BMC medical informatics and decision making*, v. 13, p. 7.

Jiang, B., Long, X. and Gao, X. (2007). MobileTest: A tool supporting automatic black box test for software on smart mobile devices. In *Proceedings - International Conference on Software Engineering*.

Merdes, M., Malaka, R., Suliman, D., et al. (2006). Ubiquitous RATs: How Resource-Aware Run-Time Tests Can Improve Ubiquitous Software System. In *6th International Workshop on Software Engineering and Middleware, SEM 2006*.

Mota, L. S. (2013). Uma Abordagem para Especificação de Requisitos de Ubiquidade em Projetos de Software.

Pai, M., Mcculloch, M., Gorman, J. D., et al. (2004). Systematic reviews and meta-analyses: an illustrated, step-by-step guide. *Natl.Med J India*, v. 17, p. 86–95.

Rodrigues, F., Matalonga, S. and Travassos, G. H. (2014). Report: Systematic literature review protocol: Investigating context aware software testing strategies. . www.cos.ufrj.br/~ght/cactus_pr012014.pdf.

Ryan, C. and Gonsalves, A. (2005). The effect of context and application type on mobile usability: An empirical study. In *Conferences in Research and Practice in Information Technology Series*.

Satoh, I. (2003). Software testing for mobile and ubiquitous computing. In *The Sixth International Symposium on Autonomous Decentralized Systems, 2003. ISADS 2003.*

Travassos, G. H., Santos, P. S. M. Dos, Mian, P. G., Neto, A. C. D. and Biolchini, J. (mar 2008). An Environment to Support Large Scale Experimentation in Software Engineering. In *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008).*

Tse, T. H., Yau, S. S., Chan, W. K., Lu, H. and Chen, T. Y. (2004). Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*

Wang, H. and Chan, W. K. (2009). Weaving Context Sensitivity into Test Suite Construction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*.

Wang, H., Chan, W. K. and Tse, T. H. (2014). Improving the Effectiveness of Testing Pervasive Software via Context Diversity. *ACM Trans. Auton. Adapt. Syst.*, v. 9, n. 2, p. 9:1–9:28.

Wang, Z. W. Z., Elbaum, S. and Rosenblum, D. S. (2007). Automated Generation of Context-Aware Tests. In *29th International Conference on Software Engineering (ICSE'07).*

Weber, R. (1990). *Basic Content Analysis*. Newbury Park, California: Sage Publications.

An empirical study of test generation with BETA*

Anamaria M. Moreira[1] **, Ernesto C. B. de Matos**[2] **, João B. Souza Neto**[2]

[1] Departamento de Ciência da Computação – DCC/UFRJ

[2]Departamento de Informática e Matemática Aplicada – DIMAp/UFRN

anamaria@dcc.ufrj.br, {ernestocid,jbsneto}@ppgsc.ufrn.br

Abstract. *Software Testing and Formal Methods are two techniques that focus on software quality and that can complement each other. Many researchers in both fields have tried to combine them in different ways. BETA is one of the efforts in this line of work. It is a tool-supported approach to generate test cases based on B-Method specifications. In this paper, we present an empirical study that was made to evaluate BETA. We performed two case studies with different objectives and that used different techniques to measure, not only the quantity, but the quality of the test cases generated by the approach. With the results of these case studies, we were able to identify some flaws and propose improvements to enhance both the approach and the tool.*

1. Introduction

The industry needs reliable and robust software. This fact increases the demand for methods and techniques that focus on software quality. *Formal Methods* and *Software Testing* are two techniques that have this purpose. Many researchers have tried to combine these two techniques to take advantage of their most interesting aspects. This combination can bring many benefits, such as the reduction of development costs through the application of verification techniques in the initial development phases, when faults are cheaper to be fixed, and the automatic generation of tests from formal specifications [6]. Generating test cases from formal specifications is very convenient. Since these specifications usually state the software requirements in a complete and unambiguous way, they can be a good source to generate test cases. This approach for test case generation can be particularly useful in scenarios where formal methods are not strictly followed, and the code is not formally derived from the model [4]. These tests can help to verify the conformance between the abstract model and the actual implementation.

In [11, 12], the authors presented a tool-supported approach called BETA (*B Based Testing Approach*). BETA generates unit tests from formal specifications written using B-Method [1] notation. Using input space partitioning techniques [2], BETA is capable of generating test cases that verify the conformance between the code implemented and the model that originated it. Initially, BETA was evaluated through two case studies. These case studies showed the feasibility of the approach and tool and were important for an initial evaluation. However, it was not a complete evaluation because the final stages of the testing process – the execution of the tests and the evaluation of its results – were not

*This work is partly supported by CAPES and CNPq grants 2057/14-0 (PDSE), 237049/2013-9, 573964/2008-4, (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

performed, being limited just to the test case design. Because of this, it was necessary to submit BETA to new case studies with the intention of performing a complete evaluation, performing all stages of the testing process and evaluating the quality of the test cases.

Other model-based testing tools share similarities with BETA. [9] presents BZ-TT (*BZ-Testing-Tools*), a tool-supported approach for boundary value test generation from B and Z specifications. BZ-TT was evaluated through comparison with a well-established test set created by specialists. The authors conclude that BZ-TT generated a good set of tests, covering most of the previously defined test set, but they have not provided information on the practical use of the tool. [14] presents ProTest, an automatic test environment for B specifications. It does not present an assessment of ProTest, only discussing differences with respect to other approaches. [15] presents TTF (*Test Template Framework*), which generates unit tests from Z specifications. TTF also uses input space partitioning techniques to generate test cases. TTF was evaluated in several case studies, but they mainly focused on the evaluation of abstract test cases and comparisons with other approaches, not providing information on the quality of the tests generated.

In this context, this work aims to perform an empirical study of BETA to analyze the approach and tool in new situations and in a complete manner, focusing not only on quantitative aspects but also on the quality of the test cases. To do this, BETA was evaluated through two new case studies with different complexity and objectives. For both case studies, the entire testing process was performed, from test case design to the test results evaluation. The results were evaluated quantitatively and qualitatively, using metrics such as statement and branch coverage, and mutation analysis [3].

The remainder of this paper is organized as follows: Section 2 introduces the B-Method and presents the BETA approach and tool; Section 3 presents the methodology of the study; Section 4 presents the two case studies we performed, giving an overview of the process and results for both; Section 5 discusses the final results; ultimately, Section 6 concludes with final discussions and future work.

2. The B-Method and the BETA tool

The B-Method is a formal method that uses concepts of first-order logic, set theory and integer arithmetic to specify abstract state machines that represent a software behavior. The consistency of these specifications can be guaranteed by proving that some automatically generated verification conditions are valid. The method also provides a refinement mechanism in which machines may pass through a series of refinements until they reach an algorithmic implementation, called B0, which can be automatically converted into code. Such refinements are also subject to a posteriori analysis through proofs.

A B machine usually has a set of *variables* that represent the software state and a set of *operations* to modify the state. Restrictions on possible values that variables can assume are formulated in the so-called machine *invariant*. The method also has a *precondition* mechanism for operations of a machine. To ensure that an operation behaves as expected, it is necessary to ensure its precondition is satisfied.

2.1. BETA

BETA is a model-based testing approach to generate unit tests from B-Method abstract state machines. The tool [12] receives an abstract B machine as input and produces test

case specifications. BETA is capable of defining positive and negative test cases for a software implementation. Positive test cases use input data that are valid according to the source specification and negative test cases use input data that are not predicted by the specification. It uses input space partitioning strategies, *Equivalent Classes* (ECS) and *Boundary Value Analysis* (BVS), and combinatorial criteria, *Each Choice* (EC), *Pairwise* (PW), and *All Combinations* (AC), to generate test cases [2]. The main objective of the test cases generated by BETA is to verify the accordance between the code implemented – that could be manually implemented or generated by a code generation tool – and the abstract model that originated it.

In summary, the approach starts with an abstract B machine, and since it generates tests for each unit of the model individually, the process is repeated for each one of its operations. The process goes as follows:

1. *Identify Input Space:* once the operation to be tested is chosen, the approach identifies the variables that compose the operation's input space and, as such, may influence its behavior. The input space for a B operation is composed of its formal parameters and the state variables that are used by the operation;
2. *Find Operation's Characteristics:* the approach then proceeds exploring the model to find interesting characteristics about the operation. These characteristics are constraints applied to the operation under test. These constraints can be found in invariants, preconditions, and conditional statements, for example;
3. *Partition Input Space and Define Test Case Scenarios:* after the characteristics are enumerated; they are used to create test partitions for the input space of the operation under test. Then, combinatorial criteria are used to select and combine these partitions into test cases. A test case is expressed by a logical formula that describes the test scenario;
4. *Generate Test Case Data:* to obtain input data for these test scenarios a constraint solver is used to solve the logical formulas obtained in the previous step;
5. *Generate Test Case Specifications:* once the test data is obtained, the tool generates test case specifications that guide the developer in the implementation of the concrete test cases;
6. *Obtain Oracle Values:* to obtain oracle data for the test cases the original model has to be animated using the generated test data to verify what is the expected system behaviour for the test case according to the original model. This step is performed manually and can be done using an animation tool for B models;
7. *Concretize Test Data:* there are some situations where the test data generated by the approach does not match the data structures used in the actual implementation of the system. This happens because the test data is generated based on data structures used by the abstract models that use abstract data structures. Because of this, some adaptations in the generated test data might be necessary during the implementation of the concrete test cases.

The BETA tool is free and open-source and can be downloaded on `http://www.beta-tool.info`.

3. Methodology

The work presented in this paper had the objective to perform an empirical study to evaluate BETA, detecting possible limitations and analyzing the quality of the test cases it

generates. To achieve this goal and provide a foundation to improve BETA, we addressed the following questions:

- **Q1** What are the difficulties encountered during the application of the BETA approach in its entirety, and what are the limitations of the BETA tool?
- **Q2** How do the BETA implementation of the partitioning strategies and the combinatorial criteria differ in terms of the amount of generated tests (also taking into consideration feasible and infeasible test case scenarios)?
- **Q3** How do the BETA implementation of the partitioning strategies and the combinatorial criteria differ in terms of system coverage and ability to detect faults?

The first question addresses practical aspects of the use of BETA while the two other questions take into consideration the analysis of the its results. To answer these questions, we submitted BETA to two case studies. In both case studies, the difficulties and limitations found were reported and the results were quantitatively and qualitatively analyzed, using metrics such as statement and branch coverage (baseline coverage criteria), and mutation analysis (often used as reference to evaluate other criteria due to the high quality of its results in spite of its higher costs [2]).

4. Case Studies

4.1. Lua API

In this case study, BETA was used to generate tests for the C API (Application Program Interface) of the Lua programming language [7]. The case study was performed using a partial model of the API specified using the B-Method notation. The model was developed by [13] and was based on the documentation of the API. This model has a total of 23 abstract machines that were divided into three categories: *Lua's types and values definition*, *API's state definition*, and *API's operations definition*. The model has a level of complexity that had not been explored before by BETA, such as compound structures, complex types and values (Lua allows flexible types that do not have a simple representation in B-Method) and a high number of operations (71 API operations).

A. Test Case Generation

To generate the test cases, the machines that defined API operations were submitted to BETA. Initially, each combination of partitioning strategy (ECS and BVS) and combinatorial criteria (EC, PW and AC) was tried, but the tool was not capable of generating test case specifications. After investigation, we found out that the problem was related to the constraint solver used by the tool. The complexity of the Lua API model was just too much for the constraint solver. To solve this problem, we tried to change the constraint solver settings, such as increasing the computation timeout, but it was not enough.

Given this limitation of the constraint solver used by BETA, we decided to use a simplified version of the Lua API model to continue the case study. The simplified version is a subset of the original model, where only a few of the Lua types and their related state and operations were considered. This version has a total of 11 abstract machines, instead of 23 from the original model and specifies 25 API operations, instead of 71 from the original model. For this version, BETA was capable to generate the test cases using all

partitioning strategies and combinatorial criteria. The model for the Lua API does not use numerical ranges in the definitions, so, when this happens, the partitioning strategies ECS and BVS generate the same results. Considering the total of test cases generated with the strategy ECS (or BVS), Figure 1 presents the amount of infeasible test cases, and positive and negative feasible test cases generated by BETA for each combinatorial criteria. The criterion AC generated the largest amount of test cases, followed by PW and EC.

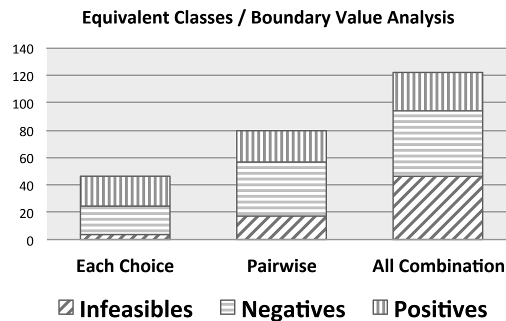


Figure 1. Test cases generated in Lua API case study plot

B. Test Case Implementation

Before implementing the test cases, expected results were obtained using the method proposed by the BETA approach, to be used as test oracles. After the oracles definition, the tests for each one of the 25 operations were implemented. In this step of the study, the negative tests were not considered because it was not our goal to evaluate robustness of the Lua API, besides that, more information is necessary to perform this type of test, such as what is the expected behaviour of the software for a scenario that was not foreseen.

The tests implementation had challenges related to the gap between the Lua API B model and the Lua API standard implementation. The Lua API has a complex implementation. To implement the test cases it was necessary to map the variables of the API's B model into variables of the API's standard implementation. Moreover, it was necessary to create a function to adapt the test data generated by BETA into test data for the actual implementation. After this, all positive tests were implemented and executed. The tests results and its analysis are presented in the next section.

C. Results and Analysis

Since only positive tests were implemented and executed, it was expected that all tests passed. However, some tests for three (3) operations of the API failed. These tests failed because the obtained results with the standard implementation were different from the expected results derived from the B model. Thus, the B model of these operations did not match their standard implementation, that was the main reference for the model. Through reverse engineering, the tests generated by BETA found problems in the Lua API B model.

To analyze the tests generated by BETA in this case study, we used statement and branch coverage metrics. Considering only the main file of the API implementation, and

only the operations tested in this study, the coverage results are summarized in Table 1. The coverage results showed that the tests generated with the criterion AC obtained better results, followed by PW and EC. However, differences in the resulting coverage are much less significant than the differences in the underlying quantity of tests.

	Equivalent Classes / Boundary Value Analysis		
Coverage	**EC**	**PW**	**AC**
Statement	70.6%	76.5%	85.9%
Branch	38.2%	44.1%	58.8%

Table 1. Lua API Case Study Coverage Results

4.2. b2llvm and C4B

In this case study, BETA was used to contribute with the validation of two code generation tools for the B-Method, b2llvm [5] and C4B[1]. Both code generators receive as input a B implementation, in the B0 notation, and produce code (in LLVM, for b2llvm, and C, for C4B) as output. A set of B modules (B abstract machines and their B implementations) was used with both code generators and BETA. The tests generated by BETA were used as oracles to verify the compliance of the code generated by b2llvm and C4B with the original B abstract machine. This case study was presented in [10], mainly focusing on the validation of b2llvm and C4B.

A. Test Case Generation

A set of 13 B modules (*ATM, Sort, Calculator, Calendar, Counter, Division, Fifo, Prime, Swap, Team, TicTacToe, Timetracer* and *Wd*) was selected to be used in the code generators and BETA. The B modules selected use a reasonable range of structures and resources of the B-Method to exercise b2llvm and C4B. Besides, they present different scenarios to exercise BETA. To generate the test cases, the abstract machines of the 13 modules were submitted to the BETA tool. The tool was capable of generating test cases using all partitioning strategies and all combinatorial criteria. But, for two modules, *Division* and *Prime*, the BETA tool was not able to generate the test cases with BVS strategy. This issue was reported to be fixed in the next release of the tool. As some B modules use numerical ranges in their specifications, the results obtained with ECS and BVS partitioning strategies were different.

Considering the total of test cases generated for all 13 modules, Figure 2 presents a plot graph of the amount of infeasible test cases, and positive and negative feasible test cases generated by BETA using ECS (first plot graph) and BVS (second plot graph) partitioning strategies, for each combinatorial criteria. The plots show that the BVS strategy generates the largest amount of test cases. As seen in the first case study, the combinatorial criterion AC generated the largest amount of test cases, followed by PW and EC.

B. Test Case Implementation

The expected results were obtained, using the method proposed by the BETA approach, to be used as test oracles. As with the first case study, the negative test cases were not

[1]C4B is a C code generator integrated with Atelier B, which is an IDE for the B-Method.

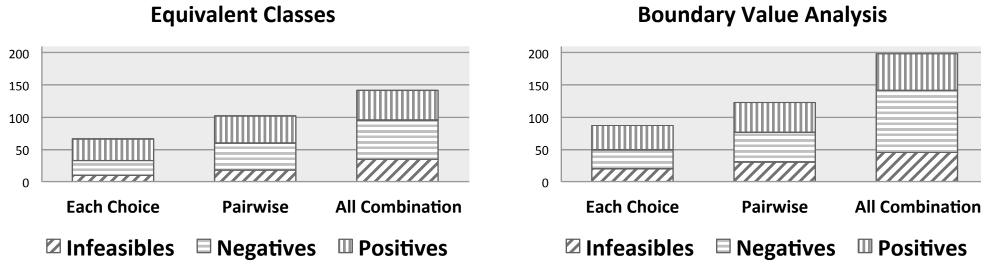


Figure 2. Test cases generated in b2llvm and C4B case study plot

implemented. The reason for this decision is that the code generators (b2llvm and C4B) directly translate the B implementations into executable code, and we can not expect them to behave properly in a situation not foreseen in the specification, and that is exactly what happens in negative tests. The test implementation in this case study did not present the same challenges encountered in the first case study, but still had difficulties related with the differences between the abstract test data, generated by BETA, and the concrete test data. The refinement of the test data (transformation of abstract data to concrete data) was made following the B implementation of each module. With the concrete test data, the test implementation was made without further challenges.

C. Results and Analysis

For each B module, the tests generated by BETA were executed in the LLVM code generated by b2llvm and the C code generated by C4B. The b2llvm tool is still under development, because of that, the LLVM code for some modules could no be generated and the tests were not performed. Nevertheless, the tests generated by BETA can help the b2llvm development team and guide them through the development. The tests for the B modules that b2llvm was capable of generating code did not reveal any problems.

C4B was capable of generating C code for all B modules, because of that, all positive tests generated by BETA were executed. The test results revealed problems in the C code generated by C4B for the *Timetracer* module. This problem was reported to the C4B development team. The tests also found problems related with the refinement process of two modules (*Prime* and *Wd*). This result, that was not related to the code generators, shows that BETA can be used as an alternative to complement the validation in a formal development with the B-Method.

To evaluate the tests generated by BETA, we used statement and branch coverage, and mutation analysis as metrics. In this evaluation, only the test results for the C code generated by C4B were considered (without the *Timetracer* module). The results are summarized in Table 2. The table presents an average of the coverage obtained with ECS and BVS partitioning strategies, and AC, PW and EC combinatorial criteria. To evaluate the capability of the generated test cases to detect faults we performed a mutation analysis, that works with fault simulation by injecting syntactic changes in the program under test [2]. The program with a syntactic change is called *mutant*. A mutant is said to be *killed* when a test can differentiate it from the original. If a mutant can not be distin-

guished from the original, it is called *equivalent*. The ratio between the number of killed mutants and the number of non-equivalent mutants is called *mutation score* and it is used to measure the quality of a test set. Since mutation analysis works with faults simulation, their results provide reliable information about the test effectiveness [3]. The tool *Milu* [8] was used to generate the mutants. The equivalent mutants were detected manually, and the execution and analysis were performed automatically by scripts. Table 3 presents the mutation analysis results. The table shows information about the modules, such as the number of operations and the number of non-equivalent mutants, and the mutation score by the tests generated with each partitioning strategy and each combinatorial criterion. The last row of Table 3 presents an average of the mutation scores. This average does not include the modules *Division* and *Prime* because the BETA tool did not generate tests with the strategy BVS for these two modules.

	Equivalent Classes			Boundary Value Analysis		
Coverage	**EC**	**PW**	**AC**	**EC**	**PW**	**AC**
Statement	72.8%	86.2%	87.6%	78.4%	90.8%	93.3%
Branch	17.7%	32%	45.6%	22.4%	36.7%	50.3%

Table 2. b2llvm and C4B Case Study Coverage Results

Modules			Percentage of mutants killed - Mutation Score %					
			Equivalent Classes			Boundary Value Analysis		
Name	**Op.**	**Mutants**	**EC**	**PW**	**AC**	**EC**	**PW**	**AC**
ATM	3	11	81.8	81.8	81.8	81.8	81.8	81.8
Sort	1	123	89.4	89.4	89.4	89.4	89.4	89.4
Calculator	6	120	47.5	47.5	47.5	74.2	74.2	74.2
Calendar	1	67	9	19.4	19.4	25.4	35.8	35.8
Counter	4	87	41.4	85.1	85.1	41.4	94.2	94.2
*Division**	1	29	31	31	31	-	-	-
Fifo	2	40	90	90	90	90	90	90
*Prime**	1	66	34.8	34.8	53	-	-	-
Swap	3	8	100	100	100	100	100	100
Team	2	89	68.5	68.5	68.5	68.5	68.5	68.5
TicTacToe	3	764	0	21.9	40.3	0	20.4	40.3
Wd	3	68	91.2	91.2	91.2	91.2	91.2	91.2
Average	**30**	**1,472**	**61.9**	**69.5**	**71.3**	**66.2**	**74.5**	**76.5**

Table 3. Mutation Analysis Results

5. Discussions

In both case studies, the last stages of the BETA approach (test data refinement and test implementation), presented more challenges. The reason for this is that these last steps were not supported by the tool and had to be performed manually. In the other hand, the use of the tool did not present difficulties, requiring the user only to know how to select the parameters (partitioning strategy and the combinatorial criterion) to generate the test cases. Another advantage of the tool when compared to other tools with the same objective is that it requires no adaptations in the model to generate the test cases. It receives the input models as they originally are. The test case specifications are also helpful to guide the implementation of the concrete test cases. About limitations, the first case study revealed a particular limitation related to the constraint solver used by BETA. Because of this, the case study was not executed entirely, a simplified version of the B model had to be used and only a few operations were tested. This result showed that the

constraint solver limitations are propagated to BETA tool and, because of that, the testing process may be affected. These results provide answers to the first question (**Q1**).

The second question (**Q2**) addresses the quantitative aspects of the tests generated by BETA. The results correspond to what should theoretically be expected: BVS generates more tests than ECS when numerical ranges are used in the B module; the combinatorial criterion AC generates substantially more tests than PW, which in turn generates more tests than EC. This pattern was followed by the amount of infeasible tests and feasible positive and negative tests. These results were not different from those obtained in the first case studies performed in [11]. The infeasible test cases are related, generally, to contradictions on the formulas that represent the test cases. Since the criterion AC combines all partitions, it is normal that it generates more infeasible cases.

The third question (**Q3**) addresses the quality of the tests generated by BETA. With the tests generated by BETA we were able to identify errors in both case studies. In the first case study we found errors in a B model through reverse engineering using the tests generated from itself, and in the second case study we found errors in one of the code generators and provided tests to be used in the development of the other. These results showed that BETA is useful to the verification and validation process. To evaluate the tests generated by BETA we used statement and branch coverage, and mutation analysis. The resulting coverage followed the quantity patterns obtained for **Q2**, i.e., more tests lead to greater coverage. However, the results did not show significant variations between the partitioning strategies and the combinatorial criteria as observed in the quantities. These results indicate that, considering only the positive tests, the ECS partitioning strategy obtained almost the same results obtained with BVS. Besides, the results showed that with the combinatorial criterion PW it is possible to obtain very close results from those obtained with AC and with less costs, since it generates less tests. The results obtained for BETA, although derived from a restricted set of models, are consistent with common knowledge that advocates PW combination as providing a good cost/benefit relation [2].

As seen in the coverage analysis, the obtained results reveals that the tests generated by BETA, even in the best cases, could not achieve all statements and decisions of the system under test, which can affect the testing process. The mutation analysis reinforced the coverage results. Even in the best case, the mutation score was not much higher than 75% on average. These results indicate that it is necessary to improve BETA to generate more effective tests, but also provide guidelines on how to do it. For instance, one observation we got from coverage analysis is a small variety of concrete test data provided by the constraint solver which leads to a lower coverage. Requiring the constraint solver to provide more randomly generated data or to provide sets of data for each test case are possible solutions currently in study for integration in BETA.

6. Conclusions and Future Work

This work performed an empirical study to evaluate a tool-supported test case generation approach called BETA. For the evaluation presented in this paper, BETA was submitted to two case studies that presented different scenarios and objectives that were not explored before. As a result, we established parameters on the qualities and limitations of BETA and, because of that, we have a basis to propose and implement improvements in the approach and the tool. Differently from other work in the field ([15, 14, 9]), the evaluation

presented here stands out for doing a complete assessment of the test generation process, from generation of abstract test cases to the implementation of the concrete tests, and also for doing an analysis focusing on the quality of the test cases using mutation testing.

As a consequence of this work, we developed a test script generator to partially automate the last step of the BETA approach, the implementation of concrete tests, and we are adding new testing techniques based on *Logical Coverage* [2] to improve the quality of the test cases. Also, as ongoing and future work we plan to: further explore the limitations of BETA when it comes to complexity of the models, possibly experimenting with different constraint solvers as data generation support; automate the test data refinement; improve the current partitioning and data selection strategies implemented by BETA.

References

[1] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge U. Press, 2005.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge U. Press, 2010.

[3] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of the 27th Intl. Conference on Software Engineering*, 2005.

[4] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop*, Workshops in Computing. Springer, 1994.

[5] D. Déharbe and V. Medeiros Jr. Proposal: Translation of B Implementations to LLVM-IR. *Brazilian Symposium on Formal Methods*, 2013.

[6] R. M. Hierons et al. Using Formal Specifications to Support Testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, 2009.

[7] R. Ierusalimschy. *Programming in Lua*. Lua.Org, 3rd edition, 2013.

[8] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Practice and Research Techniques. TAIC PART '08. Testing: Academic Industrial Conference*, 2008.

[9] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *LNCS*. 2002.

[10] E. C. B. Matos, D. Déharbe, A. M. Moreira, C. Hentz, V. Medeiros Jr, and J. B. Souza Neto. Verifying Code Generation Tools for the B-Method Using Tests: a Case Study. In *9th International Conference on Tests & Proofs*, 2015.

[11] E. C. B. Matos and A. M. Moreira. BETA: A B Based Testing Approach. In *Formal Methods: Foundations and Applications*, volume 7498 of *LNCS*. Springer, 2012.

[12] E. C. B. Matos and A. M. Moreira. BETA: a tool for test case generation based on B specifications. In *Proc. of CBSoft Tools*, 2013.

[13] A. M. Moreira and R. Ierusalimschy. Modeling the Lua API in B. Draft, 2013.

[14] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theoretical Computer Science*, 111, 2005.

[15] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. *SIGSOFT Softw. Eng. Notes*, 18(3):11–18, 1993.

InRob-UML: uma Abordagem para Testes de Interoperabilidade e Robustez baseados em Modelos

Anderson C. Weller[1], **Eliane Martins**[1], **Fátima Mattiello-Francisco**[2]

[1]Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einstein 1251 – 13081-970 – Campinas – SP – Brasil

[2]Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos – SP – Brasil

acweller@gmail.com, eliane@ic.unicamp.br, fatima.mattiello@inpe.br

Resumo. *Neste artigo apresentamos InRob-UML, um método para a geração automática de casos de testes de interoperabilidade e robustez a partir de modelos UML (Unified Modeling Language). O objetivo dos testes é determinar se duas implementações em teste são capazes de interoperar em presença de falhas temporais e de comunicação. O método proposto foi utilizado em um estudo de caso, um sistema de controle de passagem de nível em uma ferrovia, amplamente utilizado na literatura. Com o uso de um gerador de casos de teste baseado em meta-heurística, guiada por propósitos de teste, o InRob-UML evita problemas de explosão combinatória na geração de casos de teste, procurando a sequência de teste mais completa para um dado propósito de teste.*

Abstract. *This article presents InRob-UML, a method for automatic test case generation for interoperability and robustness testing from UML (Unified Modeling Language) models. The goal is to determine whether two implementations are able to interoperate in the presence of temporal and communication faults. The proposed method is applied in a case study, the Generalised Railroad Crossing problem, largely used in the literature. Using a metaheuristic based test case generator, guided by test purposes, the InRob-UML avoid the combinatorial explosion problems in test cases generation searching the most complete test sequence for given test purpose.*

1. Introdução

Contexto. Os testes de interoperabilidade visam determinar [Desmoulin and Viho 2008]: se duas ou mais implementações interagem corretamente e se estas implementações fornecem os serviços especificados durante a interação. É preciso verificar se elas interoperam na presença de falhas, portanto, utilizamos testes de robustez para determinar o grau em que esse sistema pode funcionar corretamente em presença de entradas inválidas ou sob condições ambientais estressantes [IEEE 1990]. A técnica de testes baseados em modelos é utilizada para a geração automática dos casos de teste.

Motivação prática: O tempo cada vez mais curto para colocar o *software* no mercado e a evolução dos requisitos do cliente ao longo do desenvolvimento são alguns dos fatores que têm levado a mudanças na forma como o *software* é elaborado. O reúso de componentes e o desenvolvimento incremental fazem com que os testes de interoperabilidade sejam cada vez mais necessários para determinar se os novos elementos são

capazes de interagir adequadamente com os demais módulos existentes. Mesmo que um componente funcione em um contexto, ele pode falhar em outro devido a problemas no ambiente ou em outros elementos, tornando necessária a realização de testes de robustez. Em trabalho prévio foi proposta a estratégia *InRob* (*INteroperability and ROBustness testing*) [Mattiello-Francisco et al. 2012], utilizada nos testes de interoperabilidade de aplicações espaciais. *InRob* utiliza TIOA (*Timed Input and Output Automata*) para a modelagem do comportamento das implementações em teste (IUT - *Implementation Under Test*), enquanto que nosso interesse atual é utilizar modelos de estado da UML, que são amplamente utilizados na academia e na indústria.

Trabalhos exitentes: Combinar testes de interoperabilidade e de robustez apresenta alguns desafios para a modelagem e para a geração automática de casos de teste. Na modelagem dos testes de interoperabilidade, o modelo deve representar a interação entre duas ou mais IUTs. [Desmoulin and Viho 2008]. Para os testes de robustez, devemos levar em conta as falhas na interação com o ambiente, quer dizer, precisamos de um *workload*, que são entradas que ativam as funcionalidades do sistema, e de um *faultload*, contendo as falhas do ambiente. Em geral, os dois são definidos de forma independente e as falhas são inseridas aleatoriamente, sem preocupação com o estado do sistema. Isso pode levar a situações onde diferentes falhas são injetadas num mesmo cenário de execução, ou nem sejam ativadas [Tsai et al. 1997, Cotroneo et al. 2013]. Já os testes de robustez baseados em modelos (MBRT - *Model-Based Robustness Testing*) [Fernandez et al. 2005, Ali et al. 2012] visam obter casos de teste a partir de um modelo do comportamento da IUT em presença de falhas. Uma forma de representar o comportamento robusto consiste em aumentar o modelo da IUT com as falhas a serem controladas durante os testes [Khorchef et al. 2010, Batth et al. 2007, Mattiello-Francisco et al. 2012]; porém, há a dificuldade em representar as falhas no modelo. Pode-se criar mutações do modelo original da IUT [Fernandez et al. 2005, Cotroneo et al. 2013]; mas, a dificuldade está em gerar casos de teste correspondendo a entradas inoportunas, i.e., entradas que existem no modelo, mas que não são esperados em um dado estado. Outra possibilidade consiste em aplicar mutações aos casos de teste gerados [Rollet 2003]; mas a dificuldade está em determinar a conformidade em relação ao comportamento modelado. Pode-se utilizar modelagem baseada em aspectos para representar o comportamento robusto do sistema [Ali et al. 2012]. Neste caso, o modelo original não precisa ser aumentado, e o comportamento robusto pode ser reutilizado. O modelo de robustez pode ser criado por um especialista, porém, é necessário ordenar os modelos aspectuais para alinhavá-los ao modelo original, para construir um modelo completo e correto a partir do qual os casos de teste sejam gerados. Além da dificuldade na ordenação, também há o risco de explosão de estados caso todos os aspectos sejam combinados de uma só vez.

Objetivo: Esse trabalho propõe uma estratégia para testar a interoperabilidade entre componentes de um sistema em presença de falhas, com o intuito de determinar a robustez na interação entre os componentes. Para atingir esse objetivo, será dada ênfase a problemas de interoperabilidade em presença de falhas de comunicação e de sincronização (restrições de tempo real) entre as IUTs. Outro objetivo é o uso de modelos de estados da UML para representar os componentes em teste. A partir desse modelo serão gerados o *workload* e *faultload* que serão utilizados durante os testes.

Solução: Na *InRob*, as falhas são atribuídas às IUTs interoperantes e introduzidas

no modelo de cada IUT. Apesar do canal de comunicação ser utilizado para mimicar os desvios de tempo na execução dos testes, assume-se que o ambiente não contenha falhas, e que estas são causadas por mau funcionamento de cada IUT. São consideradas apenas as falhas temporais, nas quais o tempo de ocorrência ou a duração da informação fornecida pelas interfaces desvia do que foi especificado [Avizienis et al. 2004]. Na *InRob-UML*, o injetor é modelado como parte do ambiente de comunicação entre os sistemas em teste, o que permite também representar um ambiente com falhas. O injetor é capaz de emular não só falhas temporais, mas também, falhas de conteúdo. Tanto o injetor quanto as IUTs são descritos por modelos independentes de plataforma. Além disso, é possível criar casos de teste para entradas inoportunas, sem que para isso seja necessário criar um modelo completo. Com essa estratégia é possível desenvolver o modelo funcional do injetor que reflita o comportamento de falha do ambiente de forma independente do modelo funcional das IUTs interoperantes, o que possibilita sua reutilização nos testes de outras aplicações, sem apresentar a dificuldade do uso de aspectos. E, como consideramos que as falhas são introduzidas por ambiente hostil, é possível ter um modelo de falhas mais completo.

Resultados: O método é aplicado a um estudo de caso, um sistema de controle de passagem de nível, utilizado na literatura para representar modelagem de sistemas de tempo real [Lavazza et al. 2001, Knapp et al. 2002, Hänsel et al. 2004]. O estudo de caso permitiu analisar se os casos de testes gerados correspondem às expectativas, e permitiu também determinar o que o gerador de casos de teste utilizado pode oferecer aos usuários.

Outline: Seção 2, O estudo de caso para ilustrar o método; Seção 3, O método *InRob-UML*; Seção 4, Geração de casos de teste; Seção 5, Conclusões e trabalhos futuros.

2. O estudo de caso

O GRC (*Generalized Railroad Crossing*) é um sistema que gerencia uma cancela de controle de acesso a um cruzamento de uma estrada de ferro. Os trens seguem em uma única direção, e devem passar por esse trecho com uma velocidade padrão. O controle é feito a partir de informações enviadas por dois sensores, instalados na estrada de ferro, que informam a chegada (*Arriving*) e a saída (*Leaving*) dos trens na região crítica.

Quando um trem passa pelo primeiro sensor, este envia uma mensagem para o sistema de controle (*Crossing*) informando o fato. Em resposta, o *Crossing* envia um comando para a cancela (*Gate*) solicitando o fechamento do cruzamento, e aguarda uma resposta em até $g + \Delta$ segundos, onde g é o tempo máximo para abertura ou fechamento da cancela e Δ é a tolerância máxima para possíveis *delays* na comunicação.

Quando o *Gate* recebe um comando para mudar seu estado atual, ele retorna uma mensagem de confirmação ao final do procedimento de abrir ou fechar (em g segundos). Caso ele já esteja no estado solicitado, a resposta é retornada sem aguardar tempo. Se o *Crossing* não receber uma confirmação de fechamento do *Gate* dentro do prazo máximo, então ele muda seu estado para *Failure* sinalizando que deve ser realizada uma intervenção externa ao sistema, dando tempo suficiente para o trem (t_p) parar antes do cruzamento (C), evitando potenciais perigos e possibilitando o reparo dos equipamentos e a reinicialização do sistema, conforme apresentado por [Hänsel et al. 2004].

Quando o trem sai da região crítica, detectado pelo segundo sensor, o *Crossing* envia comando de abrir para o *Gate*. A espera pela confirmação é temporizada, e após duas tentativas, o *Crossing* sinaliza a falha (*failure*) para a central de operações.

3. *InRob-UML*

Os passos do método *InRob-UML* são descritos brevemente a seguir:

1. Modelagem do serviço e do ambiente com falha: o primeiro é baseado na especificação, e o segundo no modelo de falhas do sistema.
2. Validação do modelo: o modelo gerado é transformado em um modelo executável, independente de plataforma, e sua execução permite validar o modelo e determinar se o sistema reage da maneira esperada aos parâmetros fornecidos.
3. Geração de CTA (Casos de Teste Abstratos): após ser considerado adequado, o modelo é utilizado na geração automática de CTA, descritos em formato independente da implementação. Usamos propósitos de teste (TP - *test purposes*), que descrevem propriedades ou cenários que se deseja testar [Grabowski et al. 1993], para evitar o risco de explosão combinatória. Os propósitos são especificados na forma de diagramas de sequência e a escolha é feita em conjunto com o cliente.
4. Concretização dos CTA: Visa transformar os CTA em *scripts* de teste em formato adequado para a realização na plataforma de execução. Também é feita a implementação do injetor de falhas, com base no modelo do ambiente com falha.
5. Execução dos testes e análise dos resultados: Os *scripts* de teste são aplicados à implementação do serviço em teste, e os resultados observados são comparados aos valores esperados (indicados pelo modelo).

3.1. Modelagem estática

O diagrama de classes na Figura 3.1 representa a estrutura estática, mostrando cada componente e cada tipo de dado utilizado na arquitetura de testes. O componente FEM (*Failure Emulator Mechanism*) emula tanto falhas de comunicação quanto falhas no *Gate*. Ele é parte do ambiente, e representa um canal de comunicação falho entre os subsistemas. As falhas a serem aplicadas (baseadas em [Han et al. 1995]) são: corrupção, duplicação ou perda de pacotes, bem como falhas temporais, atraso e avanço na entrega de pacotes.

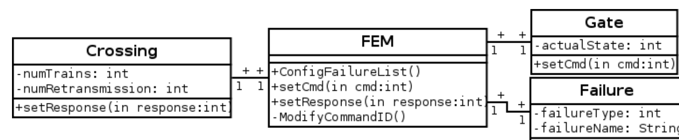


Figura 3.1. Diagrama de classes da arquitetura de testes para o estudo de caso.

3.2. Modelagem dinâmica

Cada componente representado no diagrama de classes tem um diagrama de estados associado, especificando o seu comportamento. Os atributos são variáveis utilizadas em ações e condições de guarda do modelo de estados. Já as operações são codificadas na linguagem Java e representam ações ou condições de guarda.

Uma transição inclui os seguintes elementos: i) nome do evento e seus eventuais parâmetros; ii) condição de guarda, uma expressão lógica que deve ser satisfeita para a transição ser disparada; iii) ações, realizadas quando é disparada uma transição. Além disso, ações podem ser disparadas na entrada ou na saída de um estado.

Para evitar problemas de explosão combinatória, representamos a comunicação (interoperabilidade) entre apenas duas implementações em teste (*one-to-one context*), que

no nosso caso é a interação entre o *Crossing* e o *Gate*. Para efeitos do teste de robustez, a comunicação entre esses dois componentes se dá através do FEM, como mostra a Figura 3.1, que representa a ação do injetor de falhas nessa interação.

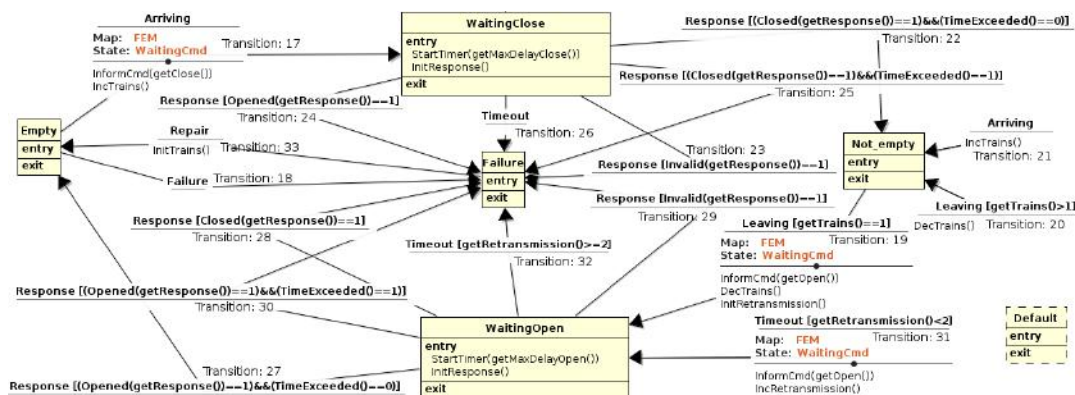


Figura 3.2. Diagrama de estados do componente *Crossing*.

O modelo é criado a partir das especificações do compilador de modelos de estado SMC (*State Machine Compiler*) [Rapp 2015], usando a ferramenta StateMutest (em desenvolvimento pelo grupo), onde cada modelo de estado é chamado de mapa. Apesar de não ser possível representar concorrência, pode-se realizar troca de mensagens entre os mapas através de transições *push*, que passam o controle da execução para outro mapa, e *pop*, que retornam a execução para o mapa chamador.

A Figura 3.2 mostra o diagrama de estados da classe *Crossing*, que inicia no estado *Empty*. Assim que ocorre um *Arriving* é disparado um *push* (transição 17), que envia o comando *Close* para *Gate* - via canal de comunicação (FEM) - e altera seu estado para *WaitingClose*. Assim que o mapa chamado terminar o processamento, ele retorna o fluxo de controle para *Crossing* com uma transição *pop* e dispara um evento neste (transição 8, Figura 3.4). Com isso, dependendo da mensagem recebida e do tempo decorrido, o estado é alterado para *Not_empty* ou *Failure*.

Na implementação há um objeto Relógio para auxiliar as restrições temporais. Ele é iniciado sempre que uma transição *push* é disparada no modelo inicial (ver *StartTimer*, Figura 3.2), e incrementado com o tempo previsto para cada mudança de estado (*IncreaseClock*, Figura 3.3); esses valores são definidos a partir da documentação do sistema. Com isso, é possível executar vários casos de teste sem ficar limitado a um relógio real.

A UML2 define *after(período)* para modelar a ocorrência de um evento após um certo período de tempo [OMG 2011]. Devido a limitações do gerador de casos de teste, os eventos precisam ter nomes distintos para podermos utilizar tempos diferentes em cada um deles. Portanto, definimos que o nome desses eventos devem iniciar com o prefixo "after_", seguido por um complemento para identificar sua função (ver transições 34 e 35, Figura 3.3).

O diagrama do *Gate*, Figura 3.3, especifica que a cancela sempre aguarda a chegada de uma solicitação (estado *WaintingCmd*), e retorna o controle da execução através de um *pop(Response)*, após enviar mensagem de confirmação através de um *InfResponse*. Dessa forma, é necessário armazenar a informação do estado final da última execução para que as condições de guarda selecionem as transições de estado corretas.

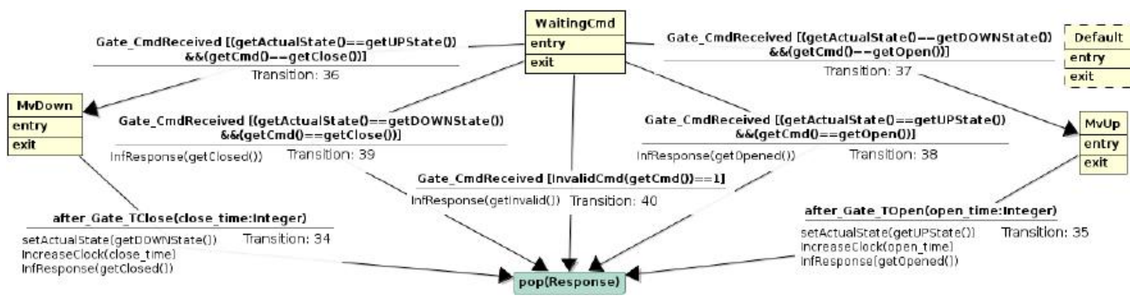


Figura 3.3. Diagrama de estados do *Gate*.

O diagrama do FEM, Figura 3.4, é sempre o ponto inicial da execução do modelo. A transição *push FEM_Setup* configura a sequência de falhas que será aplicada durante a execução do modelo (pode ser uma lista fixa ou escolhida pelo gerador), altera seu estado para *WaitingCmd* (para aguardar as mensagens enviadas pelo canal de comunicação) e direciona o controle da execução para o mapa cliente (*Crossing*).

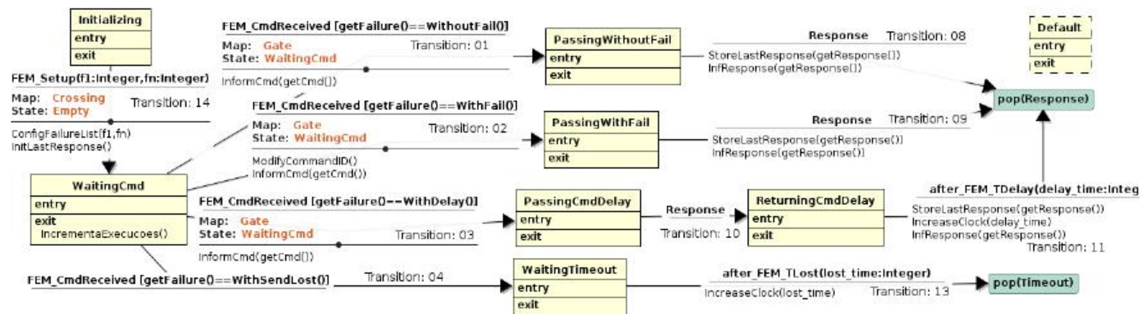


Figura 3.4. Diagrama de estados simplificado do FEM.

A Figura 3.4 apresenta apenas 4 das 7 possibilidades de falhas de comunicação modelada no FEM: na transição 1, a mensagem é repassada sem interferências; na 2 é alterado o conteúdo da mensagem enviada; na 3, a resposta de retorno é retida por tempo suficiente para a ocorrência de *Timeout*; e na 4, a mensagem enviada para o *Gate* é retida, ocasionando um *Timeout* no sistema solicitante.

3.3. Validação do Modelo

A StateMutest permite a animação do modelo. Sendo assim, para validar o modelo construído, criamos manualmente alguns casos de teste (contendo sequências de eventos) e os executamos no modelo dinâmico. A validação é feita através da conferência das saídas produzidas, bem como os estados visitados durante o processo.

4. Geração de casos de teste

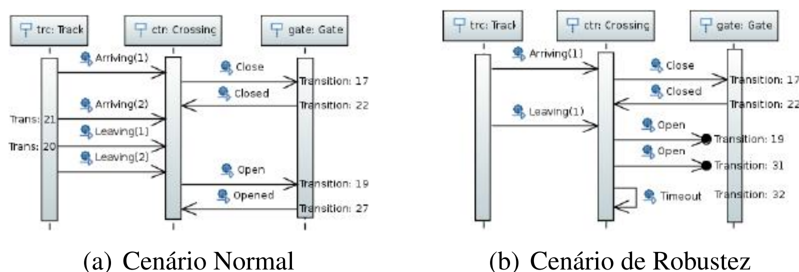
Em um trabalho prévio foi proposto o método MOST (*Multi-Objective Search-based Testing approach from EFSM*), para gerar casos de teste a partir de Máquina Finita de Estados Estendida (MEFE) [Yano et al. 2011]. A MOST usa algoritmo de otimização multiobjetivo para a busca por sequências de entradas que cubram um dado propósito de teste, portanto, não é baseada na enumeração exaustiva de caminhos, mas busca por alguns que atendam a dois objetivos: devem cobrir o propósito de teste e não devem ser muito longos.

Os caminhos são obtidos através da execução do modelo e selecionados através desses objetivos. Os dados são gerados durante a execução, para reduzir a geração de caminhos infactíveis, i.e, sintaticamente possíveis, mas semanticamente inviáveis, devido a conflitos nas condições de guarda. É possível obter vários caminhos que atendam a um propósito de teste, mas, ao contrário dos métodos exaustivos, que buscam todos os caminhos possíveis, MOST seleciona aqueles que melhor atendem aos objetivos da busca.

Para a execução são necessários: i) o modelo executável; ii) domínio dos parâmetros de entrada dos eventos; iii) propósito de teste (uma transição alvo e de um conjunto de cobertura de transições); iv) tamanho máximo da sequência de eventos nos casos de teste; v) número máximo de iterações durante a busca por uma solução; vi) parâmetro de ajuste do algoritmo multiobjetivo ($\tau \geq 0$), que define o grau de determinismo da busca, variando de uma caminhada aleatória ($\tau = 0$), a uma busca determinística local ($\tau \rightarrow \infty$).

4.1. Definição dos *test purposes*

Selecionamos dois TPs em nosso estudo de caso. O primeiro, Figura 4.1(a) representa um comportamento seguro do sistema, baseado em [Knapp et al. 2002], e especifica que, quando *Crossing* recebe o sinal de chegada do primeiro trem, *Gate* recebe comando para fechar (*Close*) a cancela. No entanto, o comando para abertura (*Open*) só é enviado após a confirmação de saída do último trem (*Leaving*). O segundo, Figura 4.1(b), representa um comportamento robusto em caso de falha de comportamento da cancela: se *Gate* não retornar confirmação a duas solicitações de *Crossing* para levantar a cancela (*Open*), então *Crossing* deve informar que há uma falha na passagem de nível e entrar em modo de falha.



(a) Cenário Normal (b) Cenário de Robustez

Figura 4.1. Diagramas de sequência representando dois possíveis TPs.

Dado que MOST aceita como propósitos de teste apenas identificadores de transições, os cenários devem ser mapeados através de uma transição alvo (t_{alvo}) e um conjunto de transições cobertura (t_{cob}). Como exemplo para os cenários especificados na Figura 4.1 temos: para (a) $t_{alvo} = 27$ e $t_{cob} = \{20\}$; e para (b) $t_{alvo} = 32$ e $t_{cob} = \{19, 31\}$.

4.2. Obtenção dos casos de teste

A MOST produz sequências de eventos contendo total ou parcialmente o TP (dependendo se cobrem todas ou apenas algumas transições do conjunto de cobertura). Caso não produza nenhuma sequência, isso pode ser uma indicação de que o TP é infactível.

MOST gera dois tipos de sequência: nominal (SN), que contém somente eventos especificados para os estados, e furtiva (SF), que contém eventos não especificados para os estados. A SF é gerada quando a máquina não é completa em relação às entradas (*input-complete*). A suposição de completude adotada pela UML é de que, ao receber um evento

não especificado para o estado, a máquina permanece inalterada [OMG 2011]. Dessa forma, é possível estender o modelo de falhas considerado para os testes de robustez, de modo a considerar também o comportamento em presença de entradas inoportunas.

Para obtermos um caso de teste é necessário determinar ainda as saídas esperadas e o veredito. Para obter as saídas esperadas, as sequências de entrada são aplicadas ao modelo executável, e as saídas produzidas são armazenadas. Se o caso de teste atende ou não ao propósito de teste, podemos definir os vereditos [Grabowski et al. 1993]:

- Passou: o caso de teste contém integralmente o TP, as saídas produzidas pelo IUT são iguais às saídas esperadas e as restrições temporais são satisfeitas;
- Inconclusivo: o caso de teste contém parcialmente o TP, as saídas produzidas pelo IUT são iguais às saídas esperadas e as restrições temporais são satisfeitas;
- Falhou: corresponde à situação em que as saídas produzidas pelo IUT não são iguais às saídas esperadas ou as restrições temporais não são satisfeitas.

Na Tabela 4.1 é apresentado o caminho de transições da SN (*path*) de dois casos de teste com veredito “Passou” - gerados a partir dos parâmetros informados na Seção 4.1 - seguida do seu tamanho (*pathSize*) e do número de iterações necessárias para obter o caso de teste (*numEval*). A lista contém o código das transições que compõem caso de teste (ver Figura 3.2, 3.3 e 3.4). Os números em negrito destacam t_{alvo} e t_{cob}.

Tabela 4.1. Caminho de transições para os TP’s da Figura 4.1.

Cenário Normal	path 14 18 33 17 1 36 34 8 22 21 **20** 21 **20** 19 1 37 35 8 **27** 17 1 pathSize 21 numEval 1382688
Cenário Robustez	path 14 18 33 17 1 36 34 8 22 21 20 **19** 5 37 35 15 13 **31** 4 13 **31** 4 13 **32** pathSize 24 numEval 92140

Escolhemos os seguintes parâmetros para a busca dos casos de teste: Número máximo de avaliações = 5.000.000; Número de execuções independentes = 100; tamanho máximo da sequência de eventos = 500; τ = 3.75. Os demais parâmetros para configuração dos tempos foram: tempo de atraso = 61s, tempo de avanço = 1s e tempo de perda = 61s no FEM; tempo para fechar = 15s e tempo para abrir = 10s no *Gate*. No cenário normal utilizou-se apenas transmissão sem falha (tipo 1), e no cenário de robustez a escolha das falhas ficou a cargo da MOST (podendo ser da falha 1 ao 7).

4.3. Resultados

Realizamos buscas nos dois cenários escolhidos, e incluímos variações no número de transições de cobertura (T_{Cob}) para verificar se haveriam diferenças nos resultados obtidos. A Tabela 4.2 apresenta um sumário desses valores, e inclui as seguintes informações: Número de vezes que o mesmo cenário foi executado (Exc); Somatório total de casos de teste obtidos ao final das execuções (NCT); Média de casos de teste obtidos por execução do cenário (CTE); Total de casos de teste Válidos/Passou (CTV); Total de Casos de teste Inconclusivos (NCI); e a média de avaliações/iterações por caso de teste válido (MDA).

Utilizamos a ferramenta StateMutest em um Notebook HP Pavilion dv5 com Debian 7.7 (64bits), Intel core i3 M350 (2,27Mhz x 4) com 7,6GiB de memória RAM, e HD SATA de 465 GiB (com 30% de espaço livre), e o tempo médio de execução de cada um dos experimentos foi de 1h 35m 08s e a média geral para a obtenção de um caso de teste que atenda ao propósito de teste foi de 2.324.500 avaliações.

Tabela 4.2. Resultados dos Cenários Normal e de Robustez

		Cenário Normal					Cenário de Robustez				
Exc	T_{Cob}	NCT	CTE	CTV	NCI	MDA	NCT	CTE	CTV	NCI	MDA
8	1	12	1,50	8	4	1.836.867	5	0,63	5	0	1.692.522
8	2	16	2,00	8	8	2.702.582	2	0,25	2	0	1.405.316
8	3	14	1,75	8	6	1.908.298	2	0,25	2	0	2.053.126
8	Todos	16	2,00	8	8	3.008.021	2	0,25	2	0	4.463.931

Após os resultados, não percebemos diferença que justifique a utilização de um conjunto de cobertura maior na busca dos casos de testes nesses dois cenários analisados. Além disso, observamos que o cenário de robustez, por requerer caminhos mais complexos no modelo, obteve menos casos de teste do que o cenário normal.

5. Conclusões e Trabalhos Futuros

O presente trabalho demonstra a estratégia *InRob-UML*, baseada na *InRob*, que utiliza modelos UML para a geração de casos de teste de interoperabilidade e robustez entre dois subsistemas na presença de falhas. Outra diferença é a inclusão do modelo de um injetor de falhas (FEM) no modelo das IUTs interoperantes, elaborado para facilitar a sua reutilização em outras aplicações, com isso diminuindo o tempo de criação de novos projetos. A modelagem do injetor também oferece flexibilidade para modelar falhas temporais e de comunicação, além de permitir sincronizar o *faultload* e *workload* nos testes de robustez.

Os resultados obtidos indicam que o método é viável, mas foram necessárias algumas adaptações ao padrão da UML, devido a características das ferramentas utilizadas na geração dinâmica dos casos de teste.

Alguns parâmetros de entrada da ferramenta precisam ser avaliados de acordo com o modelo e o propósito de teste, portanto é necessário realizar experimentos que permitam calibrar os parâmetros de configuração para otimizar o processo de busca e aumentar a média de soluções por execução. Outro ponto que merece atenção futura é a inclusão de uma solução inicial aos parâmetros da ferramenta utilizada, possibilitando ao algoritmo melhorar uma solução previamente obtida. Também será feito um estudo comparando o uso das sequências nominais e uso das sequências inoportunas para revelar defeitos.

Referências

Ali, S., Briand, L., and Hemmati, H. (2012). Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. *Software & Systems Modeling*, 11(4):633–670.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33.

Batth, S. S., Vieira, E. R., Cavalli, A., and Uyar, M. Ü. (2007). Specification of timed efsm fault models in sdl. In *Formal Techniques for Networked and Distributed Systems–FORTE 2007*, pages 50–65. Springer.

Cotroneo, D., Di Leo, D., Fucci, F., and Natella, R. (2013). Sabrine: State-based robustness testing of operating systems. In *Automated Software Engineering (ASE), 2013 IEEEACM 28th International Conference on*, pages 125–135. IEEE.

Desmoulin, A. and Viho, C. (2008). Automatic interoperability test case generation based on formal definitions. In *Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 234–250. Springer Berlin Heidelberg.

Fernandez, J.-C., Mounier, L., and Pachon, C. (2005). A model-based approach for robustness testing. In *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin Heidelberg.

Grabowski, J., Hogrefe, D., and Nahm, R. (1993). Test case generation with test purpose specification by mscs. *SDL*, 93:253–266.

Han, S., Rosenberg, H. A., and Shin, K. G. (1995). Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213.

Hänsel, F., Poliak, J., Slovák, R., and Schnieder, E. (2004). Reference case study "traffic control systems". In *Integration of Software Specification Techniques for Applications in Engineering*, pages 96–118. Springer Berlin Heidelberg.

IEEE (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

Khorchef, F. S., Berrada, I., Rollet, A., and Castanet, R. (2010). Automated robustness testing for reactive systems: application to communicating protocols. In *Gesellschaft für Informatik (GI)*, page 409. Citeseer.

Knapp, A., Merz, S., and Rauh, C. (2002). Model checking timed uml state machines and collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–414. Springer.

Lavazza, L., Quaroni, G., and Venturelli, M. (2001). Combining uml and formal notations for modelling real-time systems. In *Proceedings of the 8th European Software Engineering Conference*, ESEC/FSE-9, pages 196–206, New York, NY, USA. ACM.

Mattiello-Francisco, F., Martins, E., Cavalli, A. R., and Yano, E. T. (2012). Inrob: An approach for testing interoperability and robustness of real-time embedded software. *Journal of Systems and Software*, 85(1):3–15. Dynamic Analysis and Testing of Embedded Software.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1. Technical report, Object Management Group.

Rapp, C. W. (2015). The state machine compiler. `http://smc.sourceforge.net`.

Rollet, A. (2003). Testing robustness of real-time embedded systems. In *Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of Formal Methods (FM) 2003 Symposium*. Citeseer.

Tsai, T. K., Zhao, H., Hsueh, M.-C., and Iyer, R. K. (1997). Path-based fault injection. In *Proc. 3rd ISSAT Conf. on R&Q in Design*, volume 51, pages 121–125.

Yano, T., Martins, E., and de Sousa, F. (2011). Most: A multi-objective search-based testing from efsm. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 164–173.