



Volume 12

WOES'2010

I Workshop Brasileiro de Otimização em Engenharia de Software

**30 de Setembro de 2010
Salvador • Bahia • Brasil**

ANAIIS

Coordenadores do Comitê de Programa

Márcio de Oliveira Barros
Jerffeson Teixeira de Souza

Coordenadores Locais do SBES

Christina von Flach Garcia Chavez
Cláudio Nogueira Sant'Anna

Coordenador Geral do CBSOFT

Manoel Gomes de Mendonça Neto

Realização

LES • Laboratório de Engenharia de Software
DCC • Departamento de Ciência da Computação
UFBA • Universidade Federal da Bahia

Promoção

SBC • Sociedade Brasileira de Computação



Volume 12

WOES'2010

I Brazilian Workshop on Optimization in Software Engineering

**September 30th, 2010
Salvador, Bahia, Brazil**

PROCEEDINGS

Program Committee Chairs

Márcio de Oliveira Barros
Jerffeson Teixeira de Souza

SBES Local Co-Chairs

Christina von Flach Garcia Chavez
Cláudio Nogueira Sant'Anna

CBSOFT General Chair

Manoel Gomes de Mendonça Neto

Organizers

LES • Software Engineering Laboratory
DCC • Department of Computer Science
UFBA • Federal University of Bahia

Promoted by

SBC • Brazilian Computing Society

Apresentação

Algoritmos de busca têm sido aplicados em diversas áreas da Engenharia de Software, como a elicitação de requisitos, planejamento de tempo e custo de projetos, avaliação de qualidade e alocação de recursos humanos. Em algumas destas aplicações são utilizadas técnicas tradicionais de busca, como força bruta ou *branch-and-bound*. No entanto, à medida que os problemas crescem em complexidade (ou seja, no número de variáveis envolvidas e nas relações entre estas variáveis), estas técnicas passam a exigir muitos recursos computacionais para encontrar a solução ótima para estes problemas.

Assim, a utilização destas técnicas tradicionais em situações onde a resposta deve estar rapidamente disponível ou onde existem atualizações freqüentes das informações disponíveis para a tomada de decisão, é limitada por seu longo tempo de execução. Surge então o campo da Otimização em Engenharia de Software, ou Engenharia de Software baseada em Busca (*Search Based Software Engineering*, ou SBSE), onde os problemas de larga escala da Engenharia de Software são descritos como problemas de otimização e técnicas de busca baseadas em heurísticas (como os algoritmos genéticos, colônia de formigas, têmpora simulada, entre outros) são utilizadas para resolvê-los.

O I Workshop Brasileiro de Otimização em Engenharia de Software (WOES) tem como objetivo trazer para âmbito nacional o recente crescimento de interesse nesta área. O workshop pretende ser um fórum para discussão e divulgação de pesquisas sobre o tema, e um evento regular, que fomente a participação de pesquisadores brasileiros na comunidade internacional que trata do tema.

Nessa primeira edição, recebemos 15 submissões, envolvendo todas as regiões do país – 2 artigos da região Sul, 3 do Sudeste, 1 do Centro-Oeste, 7 do Nordeste e 2 do Norte. Desses, 9 foram aceitos para apresentação e publicação nos anais. Os coordenadores tomaram a decisão final acerca dos artigos selecionados a partir de análise dos comentários dos avaliadores e da relevância do tema do artigo para o workshop.

Gostaríamos de agradecer mais uma vez aos autores e revisores. O trabalho cuidadoso dos revisores contribuiu muito para garantirmos a qualidade dos artigos selecionados. Agradecemos, também, o apoio da organização local do CBSOFT, nas pessoas de Christina von Flach Garcia Chavez e Cláudio Nogueira Sant'Anna e suas equipes, assim como ao Coordenador Geral do CBSOFT, Manoel Gomes de Mendonça Neto.

Esperamos que o WOES seja uma experiência positiva para todos.

Salvador, Setembro de 2010.

*Márcio Barros e Jerffeson Souza
Coordenadores do Comitê de Programa do WOES 2010*

Foreword

Search algorithms have been applied in several Software Engineering areas, such as requirements, project planning, software design, quality assessment, and resource allocation. Many of these applications use traditional search techniques, such as brute force or branch-and-bound. However, when the addressed problems grow in complexity (i.e. the number of variables and their relationships), these basic search techniques tend to require too much computational power to find their optimal solutions.

Therefore, the usage of traditional search techniques to support decision making in Software Engineering is limited in situations where the solution is required in a tight time frame or is subjected to information that changes very frequently, possibly requiring the revaluation of the proposed solution. The Search Based Software Engineering (SBSE) research field proposes modeling large scale Software Engineering problems as optimization problems and using heuristic search techniques (such as genetic algorithms, ant colony, simulated annealing, among other) to solve them.

The 1st Brazilian Workshop on Optimization in Software Engineering (WOES) aims to explore the work done by Brazilian researchers on this field, which is experiencing a growing interest from the world-wide research community. The workshop is expected to bring forth discussions on the theme, to identify common research on the field, and to encourage the participation of local researchers in the international community.

In this first edition, we received 15 submissions, from all regions of the country – 2 papers from the South region, 3 from the Southeast, 1 from the Midwest, 7 from the Northeast, and 2 from the North region. Of these, 9 papers were accepted for presentation and publication in the proceedings. The Program Committee Chairs took the final selection decisions by analyzing the comments given by reviewers and the relevance of the topic addressed by the paper for the workshop.

We would like to thank the authors and the reviewers for their effort. The careful evaluation and detailed comments provided by the reviewers were essential to guarantee the quality of the selection process. We also thank the support from the local CBSOFT organization, Prof^a Christina von Flach Garcia Chavez Nogueira, Prof Claudio Sant'Anna and their team, as well as the CBSOFT General Chair, Prof. Manoel Gomes de Mendonça Neto.

We hope you enjoy WOES'2010.

Salvador, September, 2010.

*Márcio Barros e Jerffeson Souza
Program Committee Chairs for WOES 2010*

Biografia dos Coordenadores – Chairs’ Biographies



Márcio de Oliveira Barros é professor adjunto da Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO), onde lidera um grupo de pesquisa que aplica buscas heurísticas e otimização para resolver problemas relacionados com gerência de projetos e projeto de software (*software design*). Suas áreas de interesse também incluem modelagem e simulação aplicados a processos e a gerência de projetos. Márcio obteve seu Doutorado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (COPPE/UFRJ), em 2001. Já orientou mais de 10 dissertações de Mestrado e publicou diversos artigos científicos. Márcio é membro da Sociedade Brasileira de Computação.



Jerffeson Teixeira de Souza recebeu o título de Ph.D. em Ciência da Computação pela *School of Information Technology and Engineering* (SITE) da University of Ottawa, Canadá. Ele é professor adjunto da Universidade Estadual do Ceará (UECE), já tendo trabalhado como Coordenador do Mestrado Acadêmico em Ciência da Computação da UECE (MACC) e Coordenador Geral do Mestrado Integrado Profissional em Computação Aplicada UECE/IFCE (MPCOMP). Atualmente ele é Coordenador dos Grupos de Otimização em Engenharia de Software da UECE (GOES.UECE) e Padrões de Software da UECE (GPS.UECE) e Coordenador da Especialização em Engenharia de Software com Ênfase em Padrões de Software da UECE (EES). Seus interesses de pesquisa são: Otimização em Engenharia de Software, Documentação e Aplicação de Padrões de Software e Estudo de Técnicas e Aplicação de Algoritmos de Mineração de Dados.

Comitê de Programa - Program Committee

Adriana Alvim, PPGI/UNIRIO, Brazil
Alexandre Andreatta, PPGI/UNIRIO, Brazil
Claudia Werner, COPPE/UFRJ, Brazil
Éber Schmitz, NCE/UFRJ, Brazil
Geraldo Robson Mateus, UFMG, Brazil
Gustavo Campos, UECE, Brazil
Juarez Alencar, NCE/UFRJ, Brazil
Mariela Cortés, UECE, Brazil
Pedro Santos Neto, UFPI, Brazil

Revisores Externos – Additional Reviewers

Aloízio Silva, UFMG, Brazil
Andréa Magalhães, COPPE/UFRJ, Brazil
Eduardo Habib, UFMG, Brazil
Ricardo Brito, UFPI, Brazil

Comitê Organizador- Technical Committees

Coordenador Geral do CBSOFT – CBSOFT General Chair

Manoel Mendonça (DCC/IM/UFBA)

Coordenadora de Organização do CBSOFT – CBSOFT Organizing Chair

Vaninha Vieira (DCC/IM/UFBA)

Coordenação Local SBES - SBES Local Co-Chairs

Christina Chavez (DCC/IM/UFBA)

Claudio Sant'Anna (DCC/IM/UFBA)

Coordenação Local SBCARS - SBCARS Local Co-Chairs

Eduardo Almeida (DCC/IM/UFBA)

Adolfo Almeida Duran (CPD/UFBA)

Coordenação Local SBLP - SBLP Local Co-Chairs

Lais N. Salvador (DCC/IM/UFBA)

Rita Suzana P. Maciel (DCC/IM/UFBA)

Time de Organização – Organizing Team

Antonio Terceiro (DMCC/UFBA)

Antonio Oliveira (DMCC/UFBA)

Bruno Carreiro (DMCC/UFBA)

Glauco Carneiro (DMCC/UFBA)

Ivan Machado (DMCC/UFBA)

Leandro Andrade (CC/UFBA)

Raphael Oliveira (DMCC/UFBA)

Renato Novais (DMCC/UFBA e IFBA)

Rodrigo Rocha (DMCC/UFBA)

Yguaratã Cavalcanti (CIn/UFPE)

Apoio Executivo – Execution

DAGAZ Eventos

Índice

Otimizando a Seleção Combinada de Técnicas de Teste Baseado em Modelos através da Determinação do Menor Conjunto Dominante Multiobjetivo	1
<i>Arielo Claudio Dias Neto (UFAM), Rosiane de Freitas Rodrigues (UFAM)</i>	
Método de seleção de Casos de Teste para Alterações Emergenciais	9
<i>Fábio de Almeida Farzat (PPGI/UNIRIO), Márcio de Oliveira Barros (PPGI/UNIRIO)</i>	
Gerando Dados de Teste para Programas Orientados a Objeto com um Algoritmo Genético Multi-Objetivo	18
<i>Gustavo Henrique de Lima Pinto (UFPR), Silvia Regina Vergilio (UFPR)</i>	
Applying Search-Based Techniques for Requirements-Based Test Case Prioritization	24
<i>Camila Loiola Brito Maia (UECE), Fabrício Gomes de Freitas (UECE), Jerffeson Teixeira de Souza (UECE)</i>	
Uma Abordagem de Otimização Multiobjetiva para o Problema da Priorização da Correção de Defeitos	32
<i>Tarciane de Castro Andrade (UECE), Fabrício Gomes de Freitas (UECE), Daniel Pinto Coutinho (UECE), Jerffeson Teixeira de Souza (UECE)</i>	
Uma Nova Abordagem de Otimização Multiobjetiva para o Planejamento de Releases em Desenvolvimento Iterativo e Incremental de Software	40
<i>Márcia Maria Albuquerque Brasil (UECE), Fabrício Gomes de Freitas (UECE), Thiago Gomes Nepomuceno da Silva (UECE), Jerffeson Teixeira de Souza (UECE), Mariela Inés Cortés (UECE)</i>	
Explorando Técnicas de Agrupamento de Dados na Indexação de Repositórios de Componentes de Software	48
<i>Marcos Paulo Paixão (UFS), Tales Brito (UFPB), Leila Silva (UFS), Gledson Elias (UFPB)</i>	
Formulando a Adaptação de Processos de Desenvolvimento de Software como um Problema de Otimização	56
<i>Andréa M. Magdaleno (COPPE/UFRJ), Marcio Barros (PPGI/UNIRIO), Cláudia Werner (COPPE/UFRJ), Renata Araujo (PPGI/UNIRIO)</i>	
Uma Estratégia de Otimização para Agrupamento de Componentes de Software Baseada em DSM	65
<i>Thaís Alves Burity Pereira (UFPB), Gledson Elias (UFPB)</i>	

Otimizando a Seleção Combinada de Técnicas de Teste Baseado em Modelos através da Determinação do Menor Conjunto Dominante Multiobjetivo

Arilo Claudio Dias Neto, Rosiane de Freitas Rodrigues

Programa de Pós-Graduação em Informática – PPGI

Departamento de Ciência da Computação – DCC

Universidade Federal do Amazonas – UFAM

Manaus – Amazonas, Brasil

{arilo, rosiane}@dcc.ufam.edu.br

Abstract. *The combination of testing techniques is considered an effective strategy to evaluate a software product. However, the selection of which techniques to combine in a software project has been an interesting challenge in the Software Engineering field. This paper presents a proposal extending an approach developed to support the combined selection of model-based testing techniques, named Porantim, applying Multiobjective Combinatorial Optimization strategies, by determining the smallest dominating set in a bipartite and bi-weighted graph. Thus, greedy and local search strategy algorithms are proposed generating solutions aiming at maximizing the coverage of software project characteristics and minimizing the eventual effort to construct models used for test cases generation.*

Resumo. *A combinação de técnicas de teste se apresenta como uma estratégia eficaz na avaliação de um produto de software. No entanto, a seleção de quais técnicas combinar para avaliar um projeto tem sido um problema na área de Engenharia de Software. Este artigo apresenta uma proposta que estende uma abordagem de apoio à seleção combinada de técnicas de teste baseado em modelos para projetos de software, chamada Porantim, aplicando estratégias de Otimização Combinatória Multiobjetivo, através da determinação do menor conjunto dominante em um grafo bipartido e biponderado. Sendo assim, são propostos algoritmos de estratégia gulosa e de busca local para a geração de soluções de compromisso, visando maximizar a cobertura das características do projeto e ao mesmo tempo minimizar o eventual esforço para construção dos modelos usados para geração dos casos de teste.*

1. Introdução

Teste de software é uma das tarefas mais importantes do desenvolvimento de software, pois representa o último recurso para avaliação do produto antes de sua entrega ao usuário final. Neste contexto, a qualidade dos testes é dependente das técnicas selecionadas e aplicadas em um projeto de software e dos casos de teste gerados para sua avaliação [Myers *et al.*, 2004]. Por esta razão, a seleção de quais técnicas de teste aplicar em um projeto deve ser realizada cuidadosamente e baseada em critérios técnicos. No entanto, sabe-se que a maioria das tomadas de decisão a respeito da escolha de técnicas de teste para um projeto é realizada por conveniência, ou seja, são aplicadas normalmente aquelas técnicas já conhecidas por uma equipe de teste, independentemente de sua adequação para o projeto de software [Vegas e Basili, 2005].

Um dos principais pontos que precisam ser considerados durante a escolha de técnicas de teste em um projeto é a possibilidade de sua combinação, de forma a ampliar a cobertura dos testes, e consequentemente a qualidade do software [Myers *et al.*, 2004].

Algumas abordagens que apóiam a seleção de técnicas de teste têm sido propostas na literatura técnica [Vegas e Basili 2005; Wojcicki e Strooper, 2007]. Tais abordagens apoiam a seleção de técnicas de teste provendo um catálogo de quais técnicas poderiam ser aplicadas tentando atender às características e requisitos definidos para o projeto e tipos de defeitos/falhas que as técnicas estão aptas a revelar. Dias-Neto e Travassos (2009) propuseram uma abordagem, denominada de *Porantim*, que apóia a seleção combinada de técnicas de teste baseado em modelos (TTBMs) em dois passos: primeiro passo, analisando a completude (adequação) das técnicas selecionadas para o projeto e, no segundo passo, analisando o impacto que tal combinação poderia prover no esforço para construção dos modelos usados para geração dos testes. Nesta solução, são utilizadas duas abordagens distintas usando funções mono-objetivo ponderadas, uma em cada passo, para avaliar cada objetivo de forma individual (completude e esforço para modelagem). Tal análise é provida pelo cruzamento dos dados do projeto x TTBMs, avaliando a interseção entre os dois conjuntos (conjunto das características e requisitos do projeto e conjunto das características das TTBMs selecionadas).

No entanto, embora importante para esta tomada de decisão, a análise da interseção entre os dois conjuntos provida por *Porantim* é limitada no que diz respeito a prover um direcionamento sobre quais TTBMs, quando combinadas, seriam mais adequadas a um projeto e analisar como os objetivos da seleção, muitas vezes conflitantes, estão sendo atingidos ou não. Os objetivos são: as TTBMs devem maximizar a cobertura das características e requisitos definidos para o projeto e ao mesmo tempo minimizar o esforço para a construção dos modelos usados para geração dos testes, tentando aproveitar os modelos já providos no processo de desenvolvimento.

Este cenário de seleção combinada de técnicas de teste, contextualizada a Testes Baseado em Modelos, otimizando-se dois objetivos, pode ser modelado como um problema de Otimização Combinatória Multiobjetivo (COM), consistindo na determinação do menor conjunto dominante em um grafo bipartido e biponderado¹, considerando-se que os múltiplos objetivos envolvidos podem ser agrupados em dois grandes grupos de objetivos conflitantes entre si, considerados de duas formas: a primeira, através da transformação dos dois objetivos em um só, determinando-se a razão “máxima cobertura dos atributos do projeto/mínimo esforço para construção dos modelos para geração dos testes”; e, a segunda, através da consideração dos dois objetivos simultaneamente na geração de conjuntos não-dominados de soluções, ou seja, soluções de compromisso entre máxima cobertura dos atributos e mínimo esforço de modelagem de tal forma que nenhuma solução do conjunto gerado tenha valores melhores do que outra solução deste conjunto para os dois objetivos considerados.

A aplicação de tal estratégia de Otimização Combinatória Multiobjetivo [Abraham *et al.*, 2005; Rodrigues, 1999] no cenário da seleção combinada de TTBMs para um projeto de software tem como propósito prover informações sobre o ganho de cobertura dos atributos de testes requeridos em um projeto de software (objetivo a ser maximizado) a partir da seleção combinada de mais de uma TTBM, analisando ao mesmo tempo o esforço requerido para construção dos modelos a serem usados para geração dos testes (objetivo a ser minimizado).

A abordagem de resolução envolve a elaboração de algoritmos de estratégia gulosa e de busca local, esta última baseada na heurística proposta por Rodrigues *et al.* (2008) e que utiliza a vizinhança 2-opt para a geração de soluções de compromisso, não-dominadas entre si, para os objetivos envolvidos.

¹ Grafo bipartido é aquele cujos vértices estão divididos em dois conjuntos independentes, onde não há arestas entre vértices do mesmo conjunto. Um grafo é ponderado quando possui um peso associado a cada aresta (ou vértice). No caso de um grafo biponderado, dois pesos são associados a cada aresta.

Sendo assim, este trabalho apresenta uma proposta de extensão da abordagem *Porantim* [Dias-Neto e Travassos, 2009] para apoiar a seleção combinada de TTBM_s utilizando o ferramental das áreas de Otimização Combinatória Multiobjetivo [Abraham *et al.*, 2005; Papadimitriou e Steiglitz, 2004; Rodrigues, 1999] e Teoria dos Grafos [Bondy e Murty, 2008; Szwarcfiter, 1984] para modelar e resolver tal problema de Engenharia de Software (ES), visando prover informações para apoiar uma equipe de testes na tomada de decisão a respeito da escolha de TTBM_s em projetos de software.

O presente artigo está estruturado da seguinte forma: a Seção 2 apresenta o problema da seleção combinada de TTBM_s, com a abordagem *Porantim*. A Seção 3 apresenta a modelagem do problema em ES como um problema em OCM, baseado na determinação do menor conjunto dominante considerando múltiplos objetivos. Por fim, na Seção 4 são feitas as considerações finais e próximos passos deste trabalho.

2. Seleção Combinada de TTBM_s – *Porantim*

Porantim é uma abordagem de apoio à seleção combinada de técnicas de teste baseado em modelos (TTBM_s) que se baseia em dois elementos principais (Figura 1):

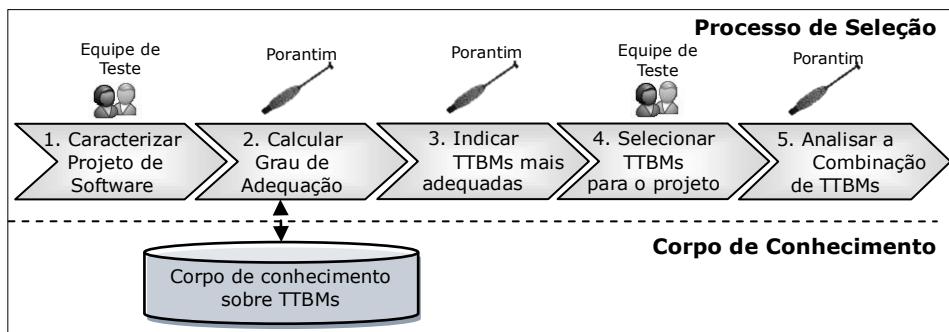


Figura 1. Elementos da Abordagem *Porantim*.

- **Corpo de conhecimento de TTBM_s**, que funciona como repositório de TTBM_s disponíveis para um projeto e que foram identificadas na literatura técnica;
- **Processo de Seleção de TTBM_s**, formado por cinco passos que visam prover informações sobre a adequação e o impacto das TTBM_s selecionadas em relação às características e requisitos de teste requeridos em um projeto de software.

No passo 1 do processo de seleção, a equipe de teste realiza a caracterização do projeto de software onde se deseja aplicar testes baseado em modelos. Isso funciona por meio de um formulário onde são preenchidos características e requisitos de teste do projeto. Em seguida, no passo 2 é realizado, individualmente, o cálculo do grau de adequação de cada TTBM que compõe o corpo de conhecimento em relação às características do projeto de software, definidas no passo 1 do processo de seleção. Este grau de adequação corresponde a um indicador numérico calculado a partir de fórmulas matemáticas apresentadas em Dias-Neto e Travassos (2009) e que utilizam o conceito matemático de **distância Euclidiana**, da seguinte forma:

- Primeiramente, as características do projeto de software e da TTBM são transformadas em valores numéricos utilizando regras pré-definidas que se baseiam em comparação entre os elementos de um conjunto;
- Em seguida, as características de cada TTBM são representadas em um vetor multidimensional, onde cada dimensão corresponde a um atributo de caracterização de TTBM_s (ao total, são 10 atributos de caracterização);
- As características do projeto de software, já transformadas em números, também passam a ser representadas como um vetor multidimensional;

- Calcula-se a distância euclidiana dos vetores que representam cada TTBM e o vetor que representa as características do projeto de software. A distância obtida representa este grau de adequação. Assim, quanto menor for a distância, mais adequada seria a TTBM em relação às características do projeto.

Após este cálculo, nos passos 3 e 4 do processo de seleção, a equipe de teste deve escolher as TTBMs a serem selecionadas de forma combinada. No entanto, a análise provida pelo grau de adequação é individual, caracterizando apenas a adequação de uma TTBM por vez em relação ao projeto de software. Desta forma, cabe ao responsável pela tarefa analisar os dados conjuntos das técnicas para julgar se quando combinadas, as técnicas selecionadas se mantêm adequadas ao projeto de software sem acarretar em aumento significativo de esforço para construção dos modelos a serem usados para geração dos testes.

No passo 5 do processo de seleção, é realizada a análise do impacto das TTBMs selecionadas em relação ao esforço para construção dos modelos a serem usados para geração dos testes. Para esta análise, a partir dos modelos requeridos pelas TTBMs selecionadas, avalia-se se estes modelos são providos ou não no processo de desenvolvimento adotado para o projeto (definido no momento da caracterização do projeto – passo 1), atribuindo-se pesos às diferentes situações e aplicando formulas matemáticas descritas em [Dias-Neto e Travassos, 2009]. Caso os modelos sejam providos, o esforço de adaptação para geração dos testes é baixo, quando comparado à situação em que os modelos não são providos no processo de desenvolvimento, e com isso deverão ser totalmente construídos.

Na próxima seção, será apresentada uma proposta de extensão da abordagem *Porantim* no que diz respeito ao cálculo do grau de adequação utilizando-se o ferramental de Otimização Combinatória Multiobjetivo e Teoria dos Grafos. Nesta extensão, as análises dos objetivos de cada TTBM selecionada em relação à cobertura das características e requisitos de teste desejados no projeto de software e do esforço para construção dos modelos a serem usados para geração dos testes, que eram feitas de forma individual e em passos distintos (passos 2 e 5, respectivamente), passarão a ser realizadas de forma simultânea em um mesmo passo do processo de seleção. Sendo assim, a equipe de teste será provida em um mesmo momento de informações a respeito do impacto das TTBMs selecionadas nos dois objetivos de seleção (completude e esforço) analisados pela abordagem *Porantim*. O objetivo com esta extensão é simplificar o processo de tomada de decisão a respeito de quais TTBMs, quando combinadas, melhor atendem às características de um projeto de software.

3. Determinação do Menor Conjunto Dominante em um Grafo Bipartido e Biponderado

O problema da seleção combinada de TTBMs pode ser modelado baseando-se no problema da determinação do menor conjunto dominante em um grafo bipartido, conforme descrito a seguir.

O problema clássico em Teoria dos Grafos, dado em sua versão de otimização, pode ser enunciado como:

Versão 1: Menor Conjunto Dominante (grafo bipartido não ponderado): dado um grafo $G=(V,E)$ com a bipartição $V_1, V_2 \subseteq V$, determinar o menor conjunto dominante $D \subseteq V_1$ em G , tal que para todo vértice $v \in V_2$ exista um vértice $u \in D$ para o qual a aresta $\{u,v\} \in E$.

Este problema possui uma dificuldade computacional inerente, pertencendo à classe de problemas **NP-difíceis** [Garey e Johnson, 1979] tanto para grafos gerais

quanto para grafos bipartidos [Bondy e Murty, 2008; Szwarcfiter, 1984], que é o caso citado acima.

A versão do problema de determinação do menor conjunto dominante considerando um grafo bipartido e ponderado (com um peso associado a cada aresta do grafo) pode ser enunciada da seguinte forma:

Versão 2: Menor Conjunto Dominante (grafo bipartido e ponderado): dado um grafo $G=(V,E)$ com a bipartição $V_1, V_2 \in V$ e com peso w_e associado a cada aresta $e=\{v_1, v_2\}$, onde $v_1 \in V_1$ e $v_2 \in V_2$, determinar o menor conjunto dominante $D \subseteq V_1$ em G , tal que para todo vértice $v \in V_2$ exista um vértice $u \in D$ para o qual a aresta $\{u, v\} \in E$ e que a soma total dos pesos w_e associados a cada aresta $\{u, v\} \in E$ seja a menor possível.

A versão do problema de determinação do menor conjunto dominante considerando múltiplos objetivos divididos em dois grandes grupos de objetivos (versão bi-objetivo) pode ser enunciada da seguinte forma:

Versão 3: Menor Conjunto Dominante (grafo bipartido e biponderado): dado um grafo $G=(V,E)$ com a bipartição $V_1, V_2 \in V$ e com pesos w_{1e} e w_{2e} associados a cada aresta $e=\{v_1, v_2\}$, onde $v_1 \in V_1$ e $v_2 \in V_2$, determinar o menor conjunto dominante $D \subseteq V_1$ em G , tal que para todo vértice $v \in V_2$ exista um vértice $u \in D$ para o qual a aresta $\{u, v\} \in E$ e que os pesos w_{1e} e w_{2e} associados a cada aresta $\{u, v\} \in E$ seja de melhor compromisso possível.

Desta forma, dado um conjunto de $n_1=|V_1|$ TTBMs (primeira partição do grafo) e um projeto de software a ser avaliado por tais técnicas, onde este projeto possui $n_2=|V_2|$ atributos em análise (segunda partição do grafo), o problema consiste em encontrar um subconjunto D de TTBMs que “domine” ou “cubra” todos os atributos do projeto em avaliação, tal que este subconjunto de TTBMs possua o melhor compromisso entre a máxima cobertura dos atributos e mínimo esforço de construção/adaptação dos modelos a serem usados para geração dos testes (Figura 2).

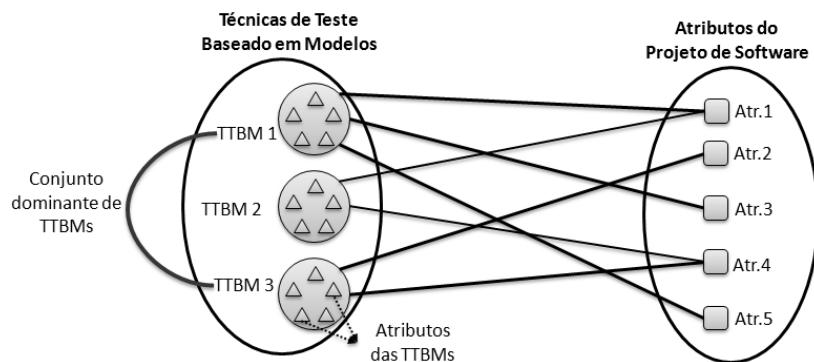


Figura 2. Grafo bipartido com menor conjunto dominante.

Dado o problema, deseja-se obter boas soluções de compromisso (subconjunto de TTBMs) entre os dois objetivos envolvidos, ou seja, idealmente deseja-se obter o **Pareto Ótimo** do problema, que constitui o conjunto das melhores soluções de compromisso ou as soluções não-dominadas da região de soluções factíveis do problema com dois objetivos a otimizar (Figura 3) [Abraham *et al.*, 2005; Rodrigues, 1999].

Todas as três versões do problema, citadas anteriormente nesta Seção, são NP-difícies e, portanto, gerar o menor conjunto dominante de maneira exata tem um custo computacional elevado. Na prática, é totalmente possível que este conjunto dominante não exista, ou seja, seria normal não existir um conjunto de TTBMs que cubra

exatamente todos as características e requisitos de teste do projeto em questão e que, além disto, minimize o esforço de construção dos modelos, ou seja, adotem apenas modelos já provados pelo processo de desenvolvimento de software da organização, sem requerer um esforço extra na sua construção para geração dos testes.

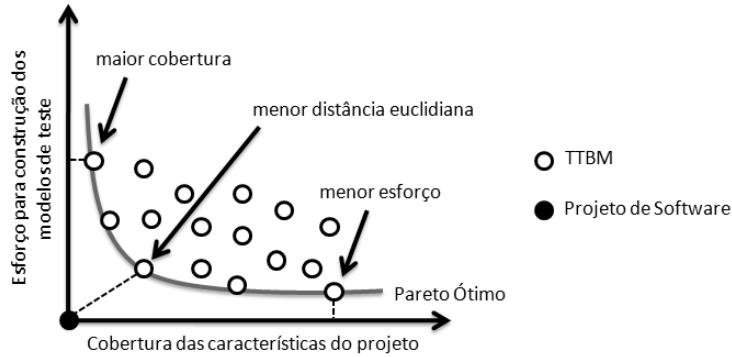


Figura 3. Representação visual do Grau de Adequação de TTBM.

Assim, neste trabalho são apresentadas propostas de duas abordagens algorítmicas aproximadas, buscando a geração de um subconjunto de TTBM que não necessariamente cubra todos os atributos do projeto, mas que satisfaça uma relação entre máxima cobertura dos atributos do projeto com mínimo esforço para construção dos modelos, visto que o objetivo da abordagem *Porantim* é apenas prover informações sobre diferentes possibilidades de combinação de TTBM que sejam adequadas a um projeto de software para apoiar na tomada de decisão a ser realizada pela equipe de teste, e não tomar a decisão em seu lugar. Uma restrição definida para ambas as propostas é que os resultados sejam computados com um custo computacional razoável, pois esta decisão precisa ser tomada durante o andamento de um projeto de software, e não haveria tempo e nem recursos disponíveis para execução de algoritmos com alto custo computacional. Assim, A primeira abordagem utiliza uma **estratégia construtiva gulosa** e a segunda utiliza uma **estratégia de busca local**, ambas heurísticas determinísticas, tal como descritas a seguir.

O Algoritmo 1 apresenta o pseudocódigo para a estratégia gulosa proposta, que constrói uma solução de compromisso entre: a maior cobertura em relação aos atributos requeridos para o projeto e o menor esforço para construção/adaptação dos modelos usados para geração dos testes. Ou seja, esta estratégia retorna como solução um subconjunto de TTBM que possua a maior razão entre os dois objetivos citados acima.

Algoritmo 1: esquema geral do algoritmo heurístico de estratégia construtiva gulosa.

Entrada: grafo $G(V,E)$, $V_1, V_2 \in V$, V_1 (família TTBM, onde cada $\tau \in V_1$ representa uma TTBM composta por um subconjunto de atributos, ou seja, subconjunto de all_A), V_2 (atributos do projeto);

$all_A \leftarrow$ inicia com todos os atributos do projeto em análise, ou seja, todo $v \in V_2$.

$TT_selec \leftarrow \emptyset;$

Enquanto ($all_A \neq \emptyset$)

- selecionar $\tau \in V_1$ com maior razão “máxima cobertura/mínimo esforço”;

- $TT_selec := TT_selec \cup \{\tau\}$;

- $all_A := all_A / \tau$;

fim_enquanto;

retornar TT_selec ;

Assim sendo, o Algoritmo 1 recebe como entrada um grafo bipartido e biponderado nas arestas, $G=(V,E)$, $V_1, V_2 \in V$, onde a partição V_1 representa o conjunto de

TTBMs, cada TTBM composta por um subconjunto de características $\tau \in V_1$, onde $\tau \subseteq V_2$, e a partição V_2 representa o conjunto de atributos do projeto em análise. A estratégia consiste, então, em primeiramente criar uma fila de prioridades mantendo no topo desta fila a TTBM com a maior razão “máxima cobertura/mínimo esforço”. O passo guloso consiste em inserir no conjunto TT_selec (lista de TTBMs que serão apresentadas à equipe de teste, para que esta decida quais serão selecionadas para o projeto – passo 3 do processo de seleção de *Porantim* – Figura 1), as TTBMs do topo da fila de prioridades (aqueles de maior razão) em cada iteração do algoritmo, eliminando do conjunto all_A os atributos do projeto “cobertos” (ou dominados) por tal TTBM. Isto é feito até que o conjunto all_A fique vazio ou que um limite mínimo desejado seja atingido.

Em relação à complexidade computacional e questões implementacionais, o processo de leitura da instância (grafo com n vértices) custa $O(n)$, onde $n=|V|$. O laço “enquanto” será executado no máximo $n_2=|V_2|$. E, no interior do laço, a operação mais custosa envolve a manutenção da fila de prioridades da fila de prioridades contendo TTBMs. Neste caso, um *heap* pode ser usado, custando $O(\log n_1)$ criar e manter tal estrutura, onde $n_1=|V_1|$. Assim, o Algoritmo 1 possui complexidade computacional de tempo $O(n_2 \log n_1)$, ou somente, $O(n \log n)$.

O Algoritmo 2, dado a seguir, apresenta o pseudocódigo para a estratégia de busca local proposta que recebe como entrada a solução construída pelo Algoritmo 1, TT_selec (ou qualquer outra solução aproximada para o problema), e altera esta solução no intuito de gerar uma solução com razão maior do que a solução inicial (ou de melhor compromisso), ou seja, realiza troca de arestas (a vizinhança de busca é definida por todas as possíveis trocas de arestas, duas a duas), mantendo sempre a solução de maior razão. Assim sendo, o Algoritmo 2 também recebe como entrada uma solução para o problema, ou seja, um grafo G' , contendo subconjuntos de vértices e arestas do grafo biponderado e bipartido $G(V,E)$, $V_1, V_2 \in V$, que representam justamente a solução construída pelo Algoritmo 1.

Algoritmo 2: esquema geral do algoritmo heurístico de busca local.

Entrada: TT_selec , grafo $G(V,E)$, $V_1, V_2 \in V$, V_1 (família TTBMs, onde cada $\tau \in V_1$ representa uma TTBM composta por um subconjunto de atributos, ou seja, subconjunto de all_A), V_2 (atributos projeto);

repita k iterações

- $\Pi :=$ subconjunto de d vértices, $d \in V_1$ (inicialmente, $\tau \in TT_selec$);
 - calcule os custos de uma solução usando Π (considerando os dois objetivos envolvidos, quando for o caso, guardando sempre o melhor conjunto não-dominado);
 - realize movimentos de 2-trocas (troca de duas arestas entre as duas partições do grafo, considerando Π como primeira partição) até que nenhuma melhoria ocorra.
-

Para o Algoritmo 2, em relação à complexidade computacional e questões implementacionais, o processo de leitura da instância (grafo com n vértices e solução TT_selec com no máximo n_1 vértices) custa $O(n)$, onde $n=|V|$. O laço “repita” será executado no máximo $k=n$. E, no interior do laço, a operação mais custosa envolve a realização de todas as possíveis troca de arestas duas-a-duas, custando $O(n^2)$. Assim, o Algoritmo 2 possui complexidade computacional de tempo $O(n^3)$.

Em relação aos resultados computacionais, a implementação dos algoritmos propostos está sendo realizada a fim de avaliar sua eficiência para apoiar a seleção de TTBMs, em comparação aos resultados obtidos pela versão original de *Porantim*.

4. Conclusões

Este trabalho abordou o problema da seleção combinada de TTBM, visando maximizar a cobertura em relação às características e requisitos de teste de um projeto de software e minimizar o esforço de construção dos modelos a serem usados para geração dos testes, como sendo o problema da determinação do menor conjunto dominado em um grafo bipartido e biponderado (menor conjunto dominante de melhor compromisso entre máxima cobertura e mínimo esforço). Para o mesmo, foram propostas duas abordagens algorítmicas, uma utilizando uma estratégia construtiva gulosa, que visa identificar as combinações de TTBM que possuem maior relação entre a “máxima cobertura / mínimo esforço”, e outra utilizando uma estratégia de busca local, que dada uma solução, realiza movimentos do tipo *2-opt* para gerar uma solução melhorada que seja de melhor compromisso entre os dois objetivos envolvidos.

Esta pesquisa está em fase inicial e, como próximos passos para sua evolução, já em andamento, está prevista a implementação das duas propostas apresentadas e avaliação dos algoritmos sob o ponto de vista da área de Otimização Combinatória, analisando o impacto da aplicação de tais estratégias em um problema da Engenharia de Software (seleção de técnicas de teste). Em seguida, será realizada uma análise experimental aprofundada dos algoritmos propostos sob o ponto de vista da área de Engenharia de Software, considerando instâncias de testes reais em projetos de software e comparando com os resultados obtidos em um estudo experimental que avaliou a versão original da abordagem *Porantim* publicado em Dias-Neto e Travassos (2009).

Referências Bibliográficas

- Abraham, A., Jain, L., Goldberg, R. (2005). “*Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*”, Springer, USA.
- Bondy, J., and Murty, U. (2008). “*Graph Theory*”, 1st ed., vol. 244 of Graduate Texts in Mathematics. Springer, London, England.
- Dias-Neto, A.C.; Travassos, G.H. (2009). “*Model-based Testing Approaches Selection for Software Projects*”, In: Information and Software Technology, v. 51, p. 1487-1504.
- Garey, M., Johson, D. (1979). “*Computers and Intractability: a Guide to the Theory of NP-Completeness*”, 1st ed. W.H. Freeman & Co, San Francisco, USA.
- Myers, G.J.; Sandler, C.; Badgett, T; Thomas, T.M. (2004), “*The Art of Software Testing*”. John Wiley & Sons, Inc., New Jersey.
- Papadimitriou, C., Steiglitz, K. (2004) “*Combinatorial Optimization: Algorithms and Complexity*”. Chapman & Hall and CRS Press, New York, USA.
- Rodrigues, R.; Pessoa, A.; Uchoa, E.; Poggi de Aragão, M. (2008). “*Heuristic algorithm for the parallel machine total weighted tardiness scheduling problem*”. Tech. Rep. RPEP Vol.8 no.08, Departamento de Engenharia de Produção da Universidade Federal Fluminense - UFF, Niterói-RJ, Brasil. Disponível em http://www.producao.uff.br/conteudo/rpep/volume8/RelPesq_V8_2008_08.pdf.
- Rodrigues, R. (1999). “Times Assíncronos para Problemas de Otimização Combinatória com Múltiplos Objetivos. Dissertação de Mestrado – Mestrado em Ciência da Computação – Instituto de Computação - IC, UNICAMP, Campinas-SP, Brasil.
- Szwarcfiter, J. “*Grafos e Algoritmos Computacionais*” (1984). Ed. Campus, Rio de Janeiro-RJ, Brasil.
- Vegas, S.; Basili, V. (2005). “A Characterization Schema for Software Testing Techniques”, Journal of Empirical Software Engineering, v.10 n.4, p.437-466.
- Wojcicki, M. A.; Strooper, P. (2007). “*An Iterative Empirical Strategy for the Systematic Selection of a Combination of Verification and Validation Technologies*”. Proceedings of the 5th international Workshop on Software Quality (May 20 - 26, 2007).

Método de seleção de Casos de Teste para Alterações Emergenciais

Fábio de A. Farzat, Márcio de O. Barros

Programa de Pós-Graduação em Informática - UNIRIO – Universidade Federal do Estado do Rio de Janeiro

Av. Pasteur 458, Urca – Rio de Janeiro, RJ – Brasil

{fabio.farzat, marcio.barros}@uniriotec.br

Resumo. *O teste de software é uma tarefa cara que contribui significativamente para o custo total de um projeto de desenvolvimento de software. Entre as várias estratégias disponíveis para testar um projeto de software, a criação de casos de teste automatizados que podem ser usados para garantir a qualidade de uma versão do software antes do seu lançamento ou para resolver um defeito específico, é cada vez mais utilizado na indústria. No entanto, alguns defeitos encontrados durante o uso do sistema podem bloquear as principais operações do negócio suportado por ele. Esses defeitos críticos são, por vezes, resolvidos diretamente no ambiente de produção, devendo ser resolvidos em um prazo restrito, onde não há tempo suficiente para executar o conjunto completo de casos de teste automatizados sobre a versão corrigida do software. Ao liberar uma versão para uso sem executar o conjunto completo de casos de teste, a equipe permite uma rápida liberação do software para a produção, mas também permite que outros defeitos sejam introduzidos no sistema. Este trabalho apresenta uma abordagem heurística para selecionar casos de teste que possam apoiar as mudanças emergenciais realizadas sobre um software, com o objetivo de maximizar a cobertura e a diversidade do conjunto de testes selecionado sob uma restrição severa de tempo de execução e dada a prioridade das funcionalidades que foram alteradas.*

1. Introdução

Uma das atividades mais onerosas do processo de desenvolvimento, chegando às vezes até 50% do custo total do projeto [10], a atividade de testes visa revelar defeitos introduzidos no software por atividades anteriores no processo de desenvolvimento. Embora muitas equipes de desenvolvimento possam ver a atividade de testes como uma forma de provar que uma determinada parte do software funciona corretamente, esta atividade não pode provar que um sistema está correto: ela só pode provar o contrário, apontando defeitos no software. Devido ao seu alto custo, freqüentemente menos esforço é dedicado às atividades de teste do que seria necessário para validar o software completamente. Em casos extremos, os testes são quase que praticamente eliminados do processo de desenvolvimento de uma ou mais versões do sistema. Mesmo empresas que têm um processo institucionalizado podem atribuir ao teste um papel secundário no contexto de algumas liberações, devido às pressões para a entrega de uma versão operacional do sistema.

Este comportamento é ainda mais comum quando são encontrados erros em ambiente de produção, erros estes que podem bloquear a correta execução de operações do negócio suportado pelo sistema. Em tais situações, as atividades de manutenção corretiva são freqüentemente executadas no ambiente de produção, lançando a versão corrigida do software em produção sem testes apropriados. Dada a necessidade de resolver a falha, a equipe realiza a mudança e implanta imediatamente. Além disso, o risco de inserir mais defeitos no sistema e

causar danos colaterais para a operação é ainda pior do que não testar uma mudança de emergência. Para minimizar o risco de um patch incorreto ou ainda de incluir mais defeitos, algum tipo de verificação deve ser aplicada sobre o software mesmo em uma situação crítica de atualização. Por exemplo, pode-se testar todo o código referente às funcionalidades afetadas pelo código escrito ou alterado como parte da liberação de emergência. No entanto, separar manualmente os casos de teste para um conjunto específico de mudanças também pode ser uma tarefa demorada. Uma vez que estamos tratando de uma alteração emergencial, o tempo consumido é um critério decisivo. Outra desvantagem é que considerar apenas o código novo ou alterado pode não garantir que outras partes essenciais do sistema continuem consistentes após a alteração.

Assim, é preciso selecionar um conjunto de casos de teste que possa ser executado em um determinado período de tempo e que cubra as funcionalidades mais importantes que foram afetadas pela mudança de emergência, de acordo com a prioridade de cada funcionalidade. Selecionar um conjunto apropriado de casos de teste é um problema de otimização e, em sistemas com um conjunto de testes de grande porte, talvez não exista um método exato capaz de encontrar a solução ótima em tempo razoável. Este artigo apresenta um modelo formal para este problema, um procedimento de coleta de dados para obter as informações corretamente e um método heurístico baseado em um algoritmo genético para propor soluções suficientemente boas para o problema. Este artigo está organizado em sete seções. A primeira compreende esta introdução. A segunda parte detalha o modelo proposto para o problema, listando os componentes do mesmo e como se relacionam. A terceira parte detalha o processo de coleta dos dados necessários para a otimização. A quarta parte descreve o processo de otimização e como as informações coletadas serão usadas. A quinta parte propõe um modelo formal para o problema de seleção dos casos de teste de emergência. A sexta parte propõe um estudo de caso baseado em um caso real da indústria de software e, por fim, a sétima parte contém a conclusão do artigo e considerações sobre o estado atual da pesquisa.

2. Modelando o problema de seleção dos casos de teste de emergência

Dada uma alteração de emergência, estabelecemos o problema de interesse como a seleção de um conjunto de casos de teste que podem ser executados em um prazo curto e previamente definido e que maximize a cobertura das funcionalidades que sofreram alterações realizadas neste contexto. Assumimos que as funcionalidades afetadas pelas alterações podem ter prioridades distintas, considerando aqui uma perspectiva dos *stakeholders* do negócio suportado pelo sistema. Também restringimos o problema a ser tratado aos testes de unidade, que são testes desenvolvidos para validar um módulo de código específico [12]. Os módulos de código-fonte, por outro lado, implementam uma ou mais funcionalidades. A Figura 1 apresenta as relações entre esses elementos.

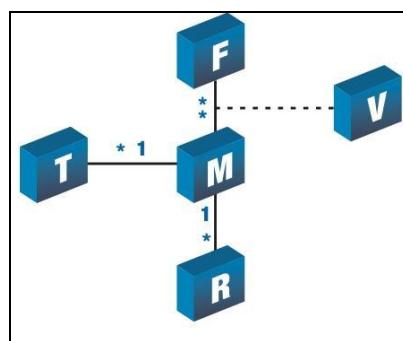


Figura 1. Modelo do problema de seleção de casos de teste para verificação de alterações emergenciais

O problema de seleção de casos de teste para alterações emergenciais está relacionado ao problema NP-completo que determina os itens a incluir em uma coleção [8], onde cada item que é descrito por um peso e um valor, de modo que o peso total do conjunto seja inferior a um determinado limite e seu valor total seja maximizado. Neste trabalho, um item é mapeado para um caso de teste. O peso de um item está relacionado ao tempo de execução do seu caso de teste. O limite de peso é a restrição de tempo máximo para execução do conjunto de casos de teste. Finalmente, o valor a ser maximizado é uma composição das prioridades das funcionalidades afetadas pela mudança emergencial com a cobertura dos casos de teste aplicáveis sobre os módulos de código-fonte que implementam estas funcionalidades.

Uma **funcionalidade** (F) representa um conjunto de requisitos funcionais fornecidos pelo software. Em geral, cada transação ou operação específica realizada pelo software é considerada uma funcionalidade. Em nosso contexto, uma funcionalidade é implementada por um conjunto de módulos de código-fonte (artefatos de análise e *design* não são tratados no método de seleção de casos de teste). Cada funcionalidade possui uma propriedade que indica a sua **prioridade** para a operação do sistema de software, ou seja, sua prioridade do ponto de vista dos usuários. Quanto mais importante é a funcionalidade no sentido de suportar o negócio, maior será o valor de sua prioridade. Recursos com a prioridade máxima serão sempre marcados como alterados, ou seja, receberão sempre atenção no processo de seleção dos casos de teste.

Um **módulo** (M) é um arquivo que contém parte do código-fonte que compõe o sistema. Cada módulo participa na implementação de uma ou mais funcionalidades. Assim, existe uma relação (V) entre as funcionalidades e os módulos de código-fonte, de modo que cada módulo implementa um percentual da funcionalidade. Para a execução do método de seleção de casos de teste proposto, assumimos que os desenvolvedores são capazes de manter rastreabilidade entre os módulos de código-fonte e as funcionalidades providas pelo sistema. Essa relação se dá através de uma ferramenta de controle de atividades, ou ferramenta de *issue tracking*. O Bugzilla¹ e o Trac² são exemplos de ferramentas que controlam as atividades dos desenvolvedores e associam essas atividades com um ou mais módulos. Cada módulo tem um conjunto de **revisões** (R) e um conjunto de **testes** (T) associados a ele. Cada revisão associada a um módulo representa uma mudança particular sofrida por ele. As revisões são estritamente relacionadas com o sistema de controle de versão utilizado para apoiar o desenvolvimento do software. Cada teste associado a um módulo cobre determinada parte do seu código-fonte. Uma vez que as funcionalidades são implementadas por módulos e que os módulos são verificados por casos de teste, os casos de teste cobrem indiretamente as funcionalidades providas pelo software.

Teste de software pode ser definido como um tratamento composto por um conjunto de valores de entrada, uma ação e um resultado esperado [2]. A ação é executada usando um conjunto de valores de entrada para obter uma saída, que é comparada com o resultado esperado. Os testes podem ser classificados como positivos ou negativos. Nyman [6] conceitua testes como segue:

Testes positivos: testes que visam demonstrar que um módulo de código-fonte particular faz o que deve fazer, de acordo com sua especificação. Por exemplo, um teste positivo pode verificar se uma rotina gera certo resultado quando um conjunto de valores em particular é lançado como parâmetro de entrada;

Testes negativos: testes que visam demonstrar que um módulo de código-fonte não faz nada que não se deve fazer, de acordo com as restrições especificadas nos documentos de projeto. Estes testes são desenvolvidos com a intenção de "quebrar" o sistema. Um exemplo de comportamento inesperado é quando o sistema exibe uma mensagem quando não deveria ou retorna um valor que não era esperado.

¹ <http://www.bugzilla.org>

² <http://trac.edgewall.org>

Em nossa abordagem para selecionar casos de teste de emergência, a proposta é permitir que o gestor atribua um peso para os testes positivos (W_p) e negativos (W_n) de uma forma que estes somem até 100%. Os pesos permitem que o gestor decida se mais testes positivos ou negativos devem ser selecionados. Tal decisão pode depender dos tipos de mudanças sofridas pelo sistema durante a atualização de emergência. Ao selecionar o conjunto ideal de casos de teste, o método proposto leva em conta essas ponderações.

3. Um procedimento de coleta de dados para seleção de casos de teste emergenciais

Para conseguir as informações requeridas pelo método de seleção de casos de teste, propomos usar dados sobre as últimas alterações sofridas pelos módulos de código-fonte que compõem o sistema. Essa informação determina o conjunto de módulos que precisam ser testados. Se os casos de teste disponíveis relacionados a estes módulos necessitam de mais tempo para ser executados do que a restrição de tempo desejado, um procedimento de otimização é acionado para selecionar um subconjunto destes casos de testes de acordo com a sua cobertura e a prioridade das funcionalidades implementadas por módulos de código-fonte que eles validam.

Controle de versão é a tarefa de manter sistemas de software consistentes em várias versões e configurações bem organizadas [5]. Subversion³ (ou SVN) é um dos muitos sistemas que são utilizados em projetos de software para gerenciamento de versão, sendo muito difundido na comunidade de código aberto por causa de sua distribuição sob uma licença *open source*. Projetos como o Apache⁴ e Debian⁵ usam o Subversion para apoiar o seu processo de gerenciamento de configuração. Para a coleta de informações sobre os módulos de código-fonte que foram alterados como parte do último *release*, capturamos informações do histórico de alterações mantido pelo Subversion. Ao ler o *log* de histórico de alterações, podemos identificar os módulos que compõem o código-fonte da aplicação e sua estrutura no sistema de arquivos (diretórios em que eles residem). Para cada módulo, o registro histórico de alterações possui as datas em que as mudanças foram feitas, o desenvolvedor que alterou o código e um comentário que descreve as mudanças.

Enquanto os módulos de código-fonte que compõem o sistema podem ser identificados através do registro histórico de alterações, as funcionalidades fornecidas pelo sistema podem ser diretamente obtidas a partir de uma ferramenta comumente utilizada na indústria ou projetos *open source*. Algumas ferramentas de engenharia de requisitos permitem ao usuário manter um conjunto de requisitos. Assim, se essas ferramentas estiverem disponíveis, as funcionalidades podem ser recuperadas a partir delas. Em um cenário geral, pretendemos desenvolver uma ferramenta que faça a leitura do log de histórico de alterações, permite que o usuário insira as funcionalidades oferecidas pelo software, e associar estas funcionalidades com os módulos de código-fonte. Esta ferramenta contribuirá com informações para o processo de otimização, descrevendo os módulos de código-fonte que implementam cada funcionalidade e o percentual relativo a cada módulo.

Posto que a abordagem proposta se restringe a testes de unidade, cada caso de teste está diretamente ligado a um e apenas um módulo de código-fonte. Ferramentas de gerenciamento de teste permitem aos desenvolvedores manter um repositório de casos de teste, por vezes relacionando-os com os módulos de código-fonte que são validados por eles. Mesmo que essas ferramentas não estejam disponíveis, os casos de teste de unidade geralmente seguem regras rígidas de desenvolvimento, que incluem as convenções de nomenclatura e restrições sobre os diretórios em que eles precisam ser salvos. Assim, o conjunto de casos de teste que foi desenvolvido para um sistema pode ser facilmente recuperado e associado aos módulos que os casos de teste avaliam.

³ <http://subversion.tigris.org>

⁴ <http://www.apache.org>

⁵ <http://www.debian.org>

Finalmente, precisamos obter dados sobre a cobertura de cada caso de teste. Este dado é lido automaticamente dos resultados gerados por ferramentas de execução de testes que também são comumente utilizados na indústria. Ferramentas como o **JCover** (para Java) e **NCover** (para o .NET Framework) executam um conjunto de testes de unidade e coletam dados sobre o seu tempo de execução e cobertura. Essa informação fica disponível em arquivos de *log* que podem ser consumidos pelo processo de otimização.

4. Modelo formal

O problema de seleção de casos de teste de emergência consiste em maximizar o alcance e a diversidade (testes positivos e negativos) da atividade de teste, respeitando uma restrição rigorosa sobre o tempo necessário para executar um conjunto de casos de teste e que a prioridade das características alteradas no último release. Formalmente, temos:

- Seja F o conjunto de funcionalidades. Cada elemento $F_i \in F$ é descrito por um nome e uma prioridade;

$$F = \{F_i\}$$

$$F_i = [\text{nome}_i, \text{prioridade}_i]$$

- Seja $Q = \{P1, P2, P3, P4, P5\}$ o conjunto de prioridades de funcionalidades que permite que cada elemento $q_i \in Q$ seja associado a um único peso inteiro positive w_i onde $w_1 > w_2 > w_3 > w_4 > w_5$;

- Seja M o conjunto de módulos de código-fonte. Cada elemento $M_i \in M$ é descrito por um nome, um conjunto de revisões e um conjunto de casos de teste;

$$M = \{M_i\}$$

$$M_i = [\text{nome}_i, R_i, T_i]$$

- Seja R_i o conjunto de revisões associadas a um determinado modulo de código-fonte M_i . Cada $R_{ij} \in R_i$ é descrito por um número, uma data, uma descrição e um autor responsável por aquela revisão;

$$R_i = \{R_{ij}\}$$

$$R_{ij} = [\text{número}_{ij}, \text{data}_{ij}, \text{descrição}_{ij}, \text{autor}_{ij}]$$

- Seja T_i o conjunto de casos de teste associados a um determinado modulo de código-fonte M_i . Cada $T_{ij} \in T_i$ é descrito por um nome, um tipo (positivo ou negativo), um tempo de execução e um percentual de cobertura;

$$T_i = \{T_{ij}\}$$

$$T_{ij} = [\text{nome}_{ij}, \text{tipo}_{ij}, \text{tempo}_{ij}, \text{cobertura}_{ij}]$$

- Seja V o conjunto de associações entre módulos de código-fonte e funcionalidades. Cada elemento $V_i \in V$ é descrito por um módulo de código-fonte, uma funcionalidade e um percentual de propriedade;

$$V = \{V_i\}$$

$$V_i = [M_i, F_i, perc_i]$$

onde $M_i \in M$,

$F_i \in F$,

$0 < V_i.\text{percent} \leq 100\%$,

$$\forall F_k \in F, \sum_{v \in V | v.F = F_k} v.\text{perc} = 100\%$$

- Seja S o conjunto de testes de emergência, ou seja, o conjunto de casos de teste selecionados para apoiar o lançamento de um *patch* de software sob restrições de tempo estritas e abrangendo os recursos afetados pelas mudanças mais recentes;

$$S = \{T_1, T_2, T_3 \dots T_n\}$$

O processo de otimização tem como objetivo selecionar um conjunto de testes emergencial suficientemente bom S^* para que ele possa ser executado em um determinado intervalo de tempo (L) e maximiza uma função de custo.

$$S^* \subseteq S \mid \text{time}(S^*) \leq L \wedge$$

$$\nexists s \subseteq S \therefore s \neq S^* \wedge \text{time}(s) \leq L \wedge \text{reward}(s) \geq \text{reward}(S^*)$$

$$\text{time}(s) = \sum_{t \in s} t.\text{time}$$

$$\text{reward}(s) = \text{cov}(s) * (1 - \text{penalty}(s))$$

$$\text{cov}(s) = \sum_{t \in s} t.\text{coverage} * \sum_{m \in \text{mod}(t)} \sum_{v \in \text{rel}(m)} v.\text{perc} * \text{weight}(v.F.priority)$$

$$\text{mod}(t) = \{ m \in M \mid t \in m.T \wedge m.R \neq \emptyset \}$$

$$\text{rel}(m) = \{ v \in V \mid v.M = m \}$$

$$\text{penalty}(s) = \sqrt{\frac{((\text{poscount}(s) - W_p)^2 + (\text{negcount}(s) - W_n)^2)}{2}}$$

$$\text{poscount}(s) = \frac{\text{count}(t \in s \mid t.type = \text{positive})}{\text{count}(t \in s)}$$

$$\text{negcount}(s) = \frac{\text{count}(t \in s \mid t.type = \text{negative})}{\text{count}(t \in s)}$$

As equações anteriores impõem uma série de restrições para a seleção de S^* : (a) S^* deve ser um subconjunto do conjunto de testes de emergência S , (b) S^* deve consumir menos do que L unidades de tempo para executar, e (c) nenhum outro subconjunto de S que possa ser executado em menos de L unidades de tempo deve produzir um valor superior ao gerado por S^* na função *reward()*. Para avaliar *reward()* antes é necessário avaliar a $\text{cov}(s)$ e $\text{penalty}(s)$. A função $\text{cov}(s)$ avalia um fator de cobertura global para um conjunto de casos de teste. Em seguida, a função $\text{penalty}(s)$ ajusta este valor com uma função de penalidade. O fator de cobertura global para um determinado conjunto de casos de teste é a soma de um fator de cobertura para cada caso de teste que compõe o conjunto. Finalmente, o fator de cobertura para um determinado caso de teste é calculado multiplicando-se a cobertura deste caso de teste pelo produto dos pesos das prioridades das funcionalidades cobertas pelo caso de teste (de acordo com os módulos avaliados pelo caso de teste).

Para avaliar a função de penalização, deve-se primeiramente determinar o equilíbrio desejado entre os casos de teste positivos e os casos de teste negativos na suíte de testes a ser selecionada, através do estabelecimento de valores de W_p e W_n . A função de penalização utiliza a distância euclidiana para calcular a diferença entre a proporção desejada para cada tipo de teste e a proporção observada em um conjunto de casos de teste sendo avaliado pela função de penalidade. Ela produz valores que residem no intervalo entre $[0, 1]$. Valores mais elevados desta função representam maiores distâncias entre a proporção desejada e a observada, tornando assim o conjunto que está sendo avaliado menos atraente para o processo de otimização.

5. Proposta de solução

Conforme citado na seção II, o problema de seleção dos casos de teste de emergência pode ser analisado como um problema de otimização combinatória complexo. A complexidade e o número de combinações possíveis para uma instância do problema tornam o desenvolvimento de uma solução exata inviável, visto que o tempo disponível para encontrar a solução a ser aplicada é um recurso crítico do problema.

Algoritmos Genéticos são heurísticas para problemas de otimização combinatória inspirados nos mecanismos da evolução de populações de seres vivos. Segundo Goldberg [15], algoritmos genéticos utilizam regras de transição probabilísticas e não determinísticas, o que lhes permite analisar simultaneamente uma grande área do espaço de busca e trabalhar com um grande número de parâmetros extremamente complexos. Essas características permitem que eles escapem dos mínimos locais, tornando-os excelentes escolhas para propor soluções boas para problemas de otimização combinatória.

O algoritmo genético proposto neste trabalho cria uma população inicial de cromossomos de forma aleatória. Nesta população, um cromossomo é definido por um conjunto de casos de teste e representa uma solução candidata, ou seja, uma solução que respeita as restrições impostas ao problema: nenhum caso de teste pode estar repetido dentro de uma solução, todos os casos de teste selecionados cobrem ao menos uma alteração e, em conjunto, respeitam o limite de tempo para sua execução. Um cromossomo possui N genes, sendo N o número total de casos de testes que cobrem algum aspecto da alteração de emergência. Desta forma, cada gene representa um caso de teste e é representado por um valor zero ou um. O valor zero em um gene significa que o caso de teste correspondente não está incluído na solução que o cromossomo representa. O valor um determina que o caso de teste está incluído na solução representada pelo cromossomo. Para exemplificar a proposta, seja T o conjunto de casos de testes que cobrem uma determinada alteração. Considere que T tem 10 casos de teste.

Sejam T_1 e T_2 duas possíveis soluções para o problema, ou seja, dois subconjuntos de T . As descrições C_1 e C_2 abaixo representam, respectivamente, os cromossomos de T_1 e T_2 .

$$T = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}\}$$

$$T_1 = \{T_1, T_3, T_4, T_5, T_6, T_8, T_{10}\}$$

$$C_1: (1,0,1,1,1,1,0,1,0,1)$$

$$T_2 = \{T_2, T_4, T_5, T_6, T_9, T_{10}\}$$

$$C_2: (0,1,0,1,1,1,0,0,1,1)$$

A criação da população inicial é feita gerando cromossomos S aleatoriamente e impondo a restrição de tempo descrita na função $time(s)$. Para tal, um conjunto aleatório de genes é escolhido e passado para o cromossomo. O algoritmo então avalia se o conjunto de genes gerado é uma solução viável. Em caso positivo, o indivíduo é criado e adicionado à população. Em caso negativo, o indivíduo é descartado. Esse procedimento continua até completar o número de indivíduos desejados na população.

Com a população inicial construída, calcula-se o valor de cada cromossomo S conforme definido na função $reward(S)$. Em seguida, os cromossomos são ordenados em ordem decrescente de valor calculado para a função, sabendo-se que quanto maior o valor do cromossomo melhor é a solução. A partir da população inicial ordenada, é executado o processo de reprodução que define a geração de novos cromossomos ao longo das iterações.

A estratégia de seleção adotada é baseada no conceito do Elitismo [13], onde um percentual dos melhores indivíduos (cromossomos) é escolhido para compor a nova geração. O método de Elitismo é um método de seleção que força o algoritmo a reter um certo número de melhores indivíduos a cada nova geração. Tais indivíduos poderiam ser perdidos se eles não fossem selecionados para reprodução ou se eles fossem alterados pela mutação.

Várias técnicas de reprodução são aplicáveis na criação algoritmos genéticos, mas as técnicas mais conhecidas são: *crossover* de um ponto, *crossover* de dois pontos, *crossover* de n pontos e *crossover* uniforme [14]. Neste trabalho, a técnica de reprodução escolhida foi o

crossover uniforme. Nela, cada gene do primeiro pai pode ser trocado com o gene correspondente do segundo pai de acordo com uma determinada probabilidade. Para cada gene, é sorteado um número aleatório e, caso ele seja inferior ou igual a probabilidade estabelecida, os genes são trocados. A escolha dessa técnica garante a diversidade da população e ajuda a garantir que nenhum ponto do espaço de busca tenha probabilidade zero de ser examinado [14].

Finalmente, após a reprodução aplicamos a operação de mutação. Nesta operação, os indivíduos e genes que sofrerão mutação são escolhidos aleatoriamente. Para cada gene do cromossomo escolhido um número aleatório entre $[0, 1]$ é gerado. Se esse número for menor que a P_m (probabilidade de mutação) então o gene para incluir ou retirar o caso de teste que ele representa da solução é trocado pelo oposto do seu valor. Logo em seguida o algoritmo repete a avaliação para verificar se o novo indivíduo representa uma solução viável. Caso negativo, o mutante é descartado.

Portanto o procedimento de formação dos cromossomos para composição de uma nova geração consiste nas três etapas abaixo:

1. Selecionar os E% melhores indivíduos (cromossomos de maior valor) da população corrente;
2. R% dos novos indivíduos serão gerados pelo processo de reprodução. Para cada reprodução, seleciona-se aleatoriamente um cromossomo pertencente aos E% melhores indivíduos (pai 1) e outro cromossomo (pai 2) do conjunto total de indivíduos da população corrente. Para cada gene do cromossomo filho gera-se um número real aleatório $X \in [0, 1]$. Se X for menor ou igual ao fator C% (percentual de *Crossover*), então o gene do pai 1 é selecionado. Caso contrário, seleciona-se o gene do pai 2. Assim, o cromossomo filho será composto por uma seqüência de genes dos dois cromossomos pais. Da mesma forma que na criação da população inicial, os indivíduos filhos resultantes do processo de reprodução são avaliados para saber se respeitam o critério de tempo. Caso uma operação de reprodução gere um indivíduo inválido, o mesmo será descartado e o processo será repetido até gerar um novo indivíduo único e válido.
3. $(1 - E\% - R\%)$ dos novos indivíduos são aleatoriamente gerados (mutantes).

O processo descrito acima é executado até que um dos critérios de parada seja alcançado. Os critérios de parada adotados neste trabalho são a avaliação de um número G de gerações, encontrar um indivíduo de custo ótimo (ou seja, que cubra 100% do código alterado dentro do limite de tempo imposto com o percentual escolhido do tipo de casos de teste) ou ainda não conseguir melhorar o custo global após avaliar um número Z de gerações.

6. Estudo de caso

Para validar o método proposto, um estudo de caso será realizado com dados reais da indústria de seguros. Uma companhia de seguros brasileira concordou em fornecer informações sobre um projeto de software real para fins de pesquisa. Estes dados serão recolhidos de acordo com o procedimento apresentado na seção II e alimentados em nosso algoritmo genético. A empresa tem um processo de desenvolvimento de software definido e utiliza um conjunto de ferramentas para apoiar os processos de manutenção corretiva e evolutiva. Eles usam **Subversion** como o sistema de controle de versão e **Trac** para apoiar a gestão de atividades. O projeto foi desenvolvido utilizando linguagem de programação C#, usa o **Microsoft Visual Studio** como ambiente de desenvolvimento, e **NUnit** para automatizar testes de unidade.

7. Conclusão

Este trabalho propõe uma estratégia de otimização para selecionar um conjunto de casos de teste que pode ser executado em um determinado período de tempo e cobertura das características mais importantes que foram afetados por uma mudança de emergência, de acordo com a prioridade de cada funcionalidade. Nós apresentamos um procedimento de coleta de informações para apoio à aquisição de dados exigidos pelo processo de otimização, considerando que o projeto é apoiado por um conjunto de ferramentas. Finalmente, apresentamos um modelo formal para o problema, nossas primeiras impressões sobre um algoritmo genético para implementar o processo de otimização e sobre um estudo de caso para validar o modelo proposto, utilizando informações fornecidas por uma empresa de seguros brasileira.

Este trabalho é parte de uma dissertação de mestrado e é um trabalho em andamento. Atualmente, estou concluindo a pesquisa bibliográfica e a modelagem formal do problema de otimização. Os próximos passos incluem a implementação dos conceitos apresentados no modelo formal, personalizar uma aplicação já existente de um algoritmo genético para apoiar o método proposto, avaliando a aplicação com os dados fictícios, a coleta de informações fornecidas pela empresa e avaliar o método através desses dados.

Referências

- R. Krishnamoorthi, S.A.Sahaaya Arul Mary. Regression Test Priorization using Genetic Algorithms, International Journal of Hybrid Information Technology. Vol.2, No.3, July, 2009.
- Myers, Glenford. The Art of Software Testing. John Wiley & Sons, 1979. Primeira edição.
- Rothermel, G.; R. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," IEEE Trans. Software Eng., vol. 27, no. 10, pp. 929-948, Oct. 2001.
- SWEBOK – Guide to the Software Engineering Body of Knowledge, 2004, www.swebok.org, acesso em 14/10/2009
- TICHY, WALTER. RCS – A System for Version Control, Software - Practice & Experience 1.1 ed., p 3, July 2005.
- NYMAN, Jeff, Positive and Negative Testing, GlobalTester, TechQA, <http://www.sqatest.com/methodology/PositiveandNegativeTesting.htm>, last access in March 29th, 2010.
- IEEE Standards Collection - Software Engineering. IEEE, New York - NY, 1994
- Sami Khuri, Thomas BÄack, and JÄorg HeitkÄotter. The zero/one multiple knapsack problem and genetic algorithms. In Proceedings of the ACM Symposium on Applied Computing, pages 188{193. ACM Press, 1994.
- PRESSMAN, S., ROGER (1997), Software Engineering - A Practitioner's Approach, 4 ed., New York, McGraw-Hill.
- HARROLD, M. J.; SOFFA, M. L. Selecting and using data for integration test. IEEE software, v. 8, n. 2, p. 58-65, Mar 1991.
- Walcott; Performing Regression Test Prioritization for Time-Constrained Execution Using a Genetic Algorithm, Master of Science Thesis, 2004.
- BARTIÉ, ALEXANDRE (2002), Garantia da qualidade de software, São Paulo, Elsevier.
- Rafal Kicinger, Tomasz Arciszewski, and Kenneth A. De Jong. Evolutionary computation and structural design: A survey of the state of the art. Computers & Structures, 83(23-24) : 1943-1978.
- Livramento, S. Algoritmo genético para o problema de localização de recursos em rede telefônica. Master's thesis, Universidade Estadual de Campinas (2004).
- GOLDBERG, D.E., Genetic Algorithms in Search, Optimization & Machine Learning. Massachusetts: Adison Wesley Logman, 1989.

Gerando Dados de Teste para Programas Orientados a Objeto com Um Algoritmo Genético Multi-Objetivo

Gustavo Henrique de Lima Pinto¹, Silvia Regina Vergilio¹

¹Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-980 – Curitiba – PR – Brasil

{gustavo,silvia}@inf.ufpr.br

Resumo. *O teste evolutivo de software usa algoritmos de busca heurística para a geração de dados de teste. Os diversos trabalhos existentes utilizam diferentes técnicas e funções de fitness. Entretanto, as funções utilizadas são baseadas em um único objetivo, geralmente a cobertura de um critério de teste. Mas existem diferentes fatores que podem influenciar a tarefa de geração de dados de teste. Para considerar estes fatores, este trabalho descreve um algoritmo genético multiobjetivo, integrado com a ferramenta de teste JABUTi que apóia o teste estrutural de programas orientados a objeto. Dois objetivos são avaliados experimentalmente, a cobertura de um critério de teste e o tempo de execução. Os resultados mostram um bom desempenho do algoritmo implementado.*

1. Introdução

A tarefa de geração de dados de teste para satisfazer um dado critério de teste geralmente consome muito esforço e não pode ser completamente automatizada. Isso se deve a inúmeras limitações da atividade de teste, tais como a existência de caminhos não executáveis. Devido a isto, o tema geração de dados de teste, ainda é um desafio, associado a inúmeros trabalhos científicos. Dentre estes, trabalhos que utilizam algoritmos de busca, tais como Algoritmos Genéticos e outros evolutivos, têm apresentado bastante sucesso, o que originou uma nova área de pesquisa chamada Teste Evolutivo de software [McMinn 2004]. Os trabalhos destas áreas diferem no tipo de técnica utilizada, e na função de fitness utilizada. A maioria dos trabalhos são orientados a cobertura de critérios de teste estruturais no teste de programas procedurais [McMinn 2004]. No contexto de programas orientados a objeto, existe um número mais reduzido de trabalhos, destacando-se [Araki and Vergilio 2010, Ribeiro 2008, Sagarna et al. 2007, Seesing and Gross 2006, Tonella 2004, Wappler and Wegener 2006].

O problema desses trabalhos é que a maioria deles não está integrado a uma ferramenta de teste. Além disso, todos eles tratam a geração de dados de teste como um problema de um único objetivo, geralmente a cobertura de um critério, e utilizam uma única função de fitness. Entretanto, existem diversos fatores que precisam ser considerados na geração de dados de teste, tais como a capacidade do dado de teste revelar certos tipos de defeitos, o tempo de execução, consumo de memória, e outros que podem ser específicos do tipo de programa sendo desenvolvido e ambiente de desenvolvimento.

Portanto percebe-se que o problema de geração de dados de teste é um problema que envolve múltiplas variáveis (objetivos) e para o qual uma solução ótima satisfazendo todos os objetivos nem sempre é possível, pois eles podem ser conflitantes. Para avaliar

e determinar o grupo destas boas soluções, utiliza-se o conceito de dominância de Pareto [Pareto 1927]. A idéia é descobrir soluções que não são dominadas por nenhuma outra no espaço de busca. Dado um conjunto de possíveis soluções para um problema, uma solução A domina uma solução B, se e somente se, A é melhor que B em satisfazer pelo menos um dos objetivos, sem ser pior que B em nenhum dos outros objetivos.

Na Engenharia de Software a utilização de algoritmos de otimização multiobjetivo é um tema emergente de pesquisa. No teste de software, eles têm sido utilizados para priorização de casos de teste [Yoo and Harman 2007] e também para geração de dados de teste [Cao et al. 2009, Ghiduk et al. 2007, Harman et al. 2007]. Esses trabalhos apresentam resultados promissores, entretanto, eles não se encontram integrados a uma ferramenta de teste, nem consideram o contexto de programas orientados a objetos ou a cobertura de diferentes critérios de teste, e além disto, há outros fatores não investigados a serem considerados na geração de dados de teste. Diferentemente dos trabalhos encontrados na literatura, o presente trabalho apresenta uma proposta de geração de dados de teste com algoritmos multiobjetivos no contexto de programas orientados a objeto, descrevendo um framework integrado a uma ferramenta de teste, utilizando como objetivos, a cobertura e tempo de execução. O algoritmo implementado é o NSGA-II (Nondominated Sorting Genetic Algorithm) concebido como uma modificação no NSGA proposto em [Deb and Srinivas 1994]. Resultados experimentais mostram que o algoritmo implementado é melhor que uma geração aleatória e que consegue apresentar um conjunto de boas soluções para ambos objetivos.

O trabalho está organizado como segue. Na Seção 2 alguns detalhes de implementação do algoritmo são descritos. Na Seção 3 são apresentados os resultados experimentais. Na Seção 4 estão as conclusões e possíveis trabalhos futuros.

2. A Geração de Dados de Teste Como Um Problema Multiobjetivo

Diversos fatores devem ser considerados para selecionar um dado de teste, por exemplo, é desejado maximizar a cobertura de um critério de teste e a capacidade do teste em revelar defeitos. Deve-se também procurar minimizar o tempo de execução, ou o tamanho do dado de teste, ou ainda o conjunto de dados de teste.

Esta seção descreve um algoritmo integrado com a ferramenta de teste JaBUTi (Java Bytecode Understanding and Testing) [Vincenzi et al. 2006] que apoia o teste estrutural em programas Java. O algoritmo implementa o NSGA-II, que é um algoritmo genético multiobjetivo dos mais conhecidos e utilizados e considera dois objetivos conflitantes: maximizar a cobertura de um critério de teste implementado pela JaBUTi e minimizar o tempo de execução do dado de teste.

Em geral o indivíduo na população representa um dado de teste. Neste trabalho, foi adotado um formato diferenciado para o indivíduo, que será um conjunto de dados de teste com um tamanho definido pelo usuário. Com essa abordagem procura-se aumentar as chances de encontrar boas soluções no espaço de busca. A codificação do dado de teste para programas orientados a objetos é uma tarefa complexa, pois não trata somente de uma sequência de valores de entrada, como o teste de programas procedurais. O dado de teste deve ser codificado de forma que possa ser decodificado posteriormente, sem perder informações sobre construtores, invocações de métodos e valores passados por parâmetro. A representação escolhida para cada elemento do indivíduo é exemplificada na gramática

ilustrada a seguir, baseada em [Tonella 2004].

```

<chromosome> ::= <actions> @ <values>
<actions> ::= <action> { : <actions> } ?
<action> ::= $id = constructor ( {<parameters>}? )
| $id = class # null
| $id . method ( {<parameters>}? )
<parameters> ::= builtin-type {<generator>} ?
| $id
<generator> ::= [ low ; up ]
| [ genClass ]
<values> ::= <value> { , <values> } ?
<value> ::= integer
| real
| boolean
| string
  
```

Para exemplificação, segue um dado de teste resultante da codificação para o programa TriTyp.java. Este programa possui como entrada três números inteiros positivos e verifica se estes formam um triângulo equilátero, escaleno, isósceles ou se não formam um triângulo. Os métodos que atribuem os valores para os lados do triângulo são setI, setJ, setK, cujas invocações de métodos são separados por ':'. Após o '@', são passados os valores referentes aos métodos, estes limitados aos tipos primitivos da linguagem Java.

```
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8
```

As seguintes funções de fitness foram utilizadas: a) Tempo de execução: refere-se à minimização do tempo de execução de um dado de teste, que é o período em que o teste permanece em execução (tempo final - tempo inicial); b) Cobertura de um critério: A cobertura de um critério é mensurada com auxílio da ferramenta de teste JaBUTi e tem como foco a maximização. O cálculo da cobertura do critério é feito pela fórmula:

$$Fitness_x = \frac{\text{nr. elementos cobertos}_x * 100}{\text{nr. total de elementos requeridos}}$$

Inicialmente, o usuário testador deverá fornecer os parâmetros do algoritmo. Tais parâmetros são separados em dois conjuntos. O primeiro conjunto é referente aos parâmetros do algoritmo evolutivo, e incluem: número de gerações, número de dados de teste que irão compor o indivíduo, objetivos e probabilidades relacionadas aos operadores genéticos. Os operadores de mutação realizam mudanças (ou remoções) nos valores de entrada, em chamadas de construtores e invocações de métodos. O operador de crossover de ponto único foi implementado. Esses operadores estão descritos em [Araki and Vergilio 2010, Tonella 2004]. Foi ainda implementado um operador especial, chamado de Relacionamento. Sua finalidade é alternar dados de teste entre indivíduos para garantir a diversidade da população como um todo. Este conta ainda com dois métodos: Half relation e Random relation. Para ilustração, considere como exemplo inicial o conjunto $A = \{a, b, c, d, e, f\}$ e conjunto $B = \{g, h, i, j, k, l\}$. De acordo com a abordagem *half relation* os novos conjuntos derivados dessa união deverão conter a primeira metade do conjunto A, e a segunda metade do conjunto B. Como resultado, haverá os conjuntos $HR_i = \{a, b, c, j, k, l\}$ e $HR_{ii} = \{g, h, i, d, e, f\}$. Por outro lado, se adotada a estratégia *Random relation*, serão sorteados os índices dos elementos que deverão ser alternados.

O segundo conjunto de parâmetros diz respeito aos parâmetros de teste: unidade (classe) a ser testada (cut) e um dos critérios de teste da ferramenta Ja-

BUTi [Vincenzi et al. 2006] que são: todos-nos, todos-arcos, todos-usos e todos-potenciais-usos.

Dados de testes são gerados e codificados de maneira que um único indivíduo represente um conjunto de dados de teste. Posteriormente, as principais variáveis são inicializadas e é criada a população inicial aleatoriamente. Um grande laço dá início ao processo de evolução. No primeiro momento, cada elemento (dado de teste) do indivíduo deve ser avaliado pela ferramenta individualmente afim de encontrar o fitness de cada elemento. Após, é avaliado o fitness do indivíduo, que nada mais é que a relação entre os fitness de todos os dados de teste. Posteriormente são identificadas as soluções dominadas e não-dominadas, de acordo com o critério de dominância de Pareto. Após a identificação, é aplicado o critério de seleção multiobjetivo e os individuos selecionados darão início à outra evolução, a dos dados de teste.

Neste momento, aplica-se a técnica da roleta para selecionar os dados de teste do indivíduo que sofrerão a ação dos operadores genéticos (mutação e cruzamento). Inicia-se então uma evolução referente ao conjunto de dados de teste, pois espera-se que o conjunto retenha os melhores testes no decorrer das iterações. No final da iteração, os dados de teste são alternados entre os indivíduos através da técnica denominada de Relacionamento, preservando assim a diversidade. Essa etapa caracteriza a segunda evolução do algoritmo, relacionada ao aperfeiçoamento dos conjuntos dos dados de teste.

Finalmente o grupo das melhores soluções são acrescentadas em uma nova população, e por conseguinte, apresentadas ao usuário. O ciclo será retomado até que o critério de parada seja satisfeito, ou seja, um número máximo de gerações for atingido. Após o fim do laço, uma última avaliação é feita nos indivíduos da última geração, e as soluções não dominadas por hora encontradas, são adicionadas novamente no montante de soluções não dominadas encontradas durante todo o processo.

3. Experimentos Realizados

Para avaliar o algoritmo implementado, alguns experimentos foram conduzidos com dois programas Java: TriTyp.java e Find.java (que busca por um inteiro em um vetor). Os programas foram extraídos do site <http://www.inf-cr.uclm.es/www/mpolo/stvr/> e têm sido utilizados em vários trabalhos da literatura [Polo et al. 2009] e apesar de simples dão uma idéia do funcionamento do algoritmo. Nestes experimentos procurou-se avaliar dois tipos de critérios: o critério baseado em fluxo de controle todos-arcos (AC) e o critério baseado em fluxo de dados todos-usos (US).

A configuração foi ajustada empiricamente (Tabela 1). Os resultados obtidos pelo NSGA-II foram comparados com uma estratégia aleatória configurando o número de gerações para 1, visto que a primeira população é obtida aleatoriamente. Ambas estratégias foram executadas 10 vezes e após isto, o conjunto de soluções não dominadas foi obtido considerando todas as execuções. As soluções encontradas para cada programa e critério estão na Tabela 2.

Para o programa TriTyp e o critério todos-arcos o NSGA-II apresentou 5 soluções não dominadas e a estratégia aleatória apenas uma, sendo esta dominada pela solução do algoritmo genético. As soluções do algoritmo genético variam de 278 a 344ms com cobertura de 52 a 81%. O NSGA-II selecionou os pontos mais relevantes no espaço de

Tabela 1. Configurações do Algoritmo

Programa	Critério	T. Cross	T. Muta	T. Relation	numGerações	numIndividuos	numDadosTeste
TriTyp	AC	0.8	0.2	0.7	100	100	50
TriTyp	US	0.8	0.2	0.7	100	100	50
Find	AC	0.8	0.2	0.7	50	50	50
Find	US	0.8	0.2	0.7	50	100	50

Tabela 2. Resultados

Programa	Critério	NSGA-II		RS	
		Cov.(%)	Exec.Time (ms)	Cov.(%)	Exec.Time (ms)
TriTyp	AC	81	319	65	284
		76	297		
		74	281		
		72	278		
		52	344		
	US	69	315	42	349
				40	331
Find	AC	95	297	88	327
		93	292		
	US	88	303	86	247
		87	220	76	240
		77	200		

busca. Para o critério todos-usos, o NSGA-II apresentou somente uma solução que também domina ambas soluções encontradas pela estratégia aleatória. Resultados similares foram obtidos para o programa Find. NSGA-II apresentou uma maior número de soluções que domina todas a soluções encontradas pela estratégia aleatória. Observa-se como esperado que é mais difícil satisfazer o critério baseado em fluxo de dados e que o NSGA melhorou em torno de 20% a cobertura obtida pela estratégia aleatória.

4. Conclusões

Este trabalho explorou uma abordagem que utiliza algoritmos genéticos multi-objetivos para geração de dados de teste no contexto de programas orientados a objetos. Foi implementado o algoritmo NSGA-II e dois objetivos: cobertura e tempo de execução no teste de programas Java. O algoritmo foi integrado a uma ferramenta de teste, que extrai informações de teste diretamente do código objeto Java, o que permite o teste mesmo na ausência do código fonte, o que é muito comum no teste de componentes. A representação do indivíduo também é independente do código fonte, as funções objetivo implementadas também permitem o uso dos diferentes critérios implementados pela JaBUTi.

Resultados experimentais foram conduzidos com os dois objetivos implementados. Os resultados mostram que as soluções obtidas pelo algoritmo NSGA-II são melhores do que as soluções obtidas por uma estratégia aleatória. Além disso, um número maior de soluções foi obtido, sendo estas bem distribuídas no espaço de busca, o que dá ao testador diferentes opções (soluções) não dominadas para escolher.

O algoritmo implementado poderá ser especializado e aperfeiçoado para tratar de outros problemas que envolvam a geração de dados de teste e/ou otimização multiobjetivo, podendo este ser aprofundado, aumentando o número de objetivos ou diversificando-os, como tratando outras questões como consumo de memória, ou teste baseado em defeitos. Além disso, novos experimentos com programas mais complexos e outros contextos

deverão ser conduzidos.

Referências

- Araki, L. and Vergilio, S. R. (2010). Um framework de geração de dados de teste para critérios estruturais em código objeto Java. In *Workshop de Testes e Tolerância a Falhas*.
- Cao, Y., Hu, C., and Li, L. (2009). Search-based multi-paths test data generation for structure-oriented testing. In *ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 25–32.
- Deb, K. and Srinivas, N. (1994). Multiobjective optimization using nondominated sorting in genetic algorithms. In *IEEE Transactions on Evolutionary Computation*, pages 221–248.
- Ghiduk, A. S., Harrold, M. J., and Girgis, M. R. (2007). Using genetic algorithms to aid test-data generation for data-flow coverage. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*, pages 41–48.
- Harman, M., Lakhotia, K., and McMinn, P. (2007). A multi-objective approach to search-based test data generation. In *Genetic and Evolutionary Computation Conference*.
- McMinn, P. (2004). Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 2(14):105–156.
- Pareto, V., editor (1927). *Manuel D’Economie Politique*. Ams Pr.
- Polo, M., Piattini, M., and García-Rodríguez, I. (2009). Decreasing the cost of mutation testing with second-order mutants. *Software Testing Verification Reliability*, 19(2):111–131.
- Ribeiro, J. C. B. (2008). Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*.
- Sagarna, R., Arcuri, A., and Yao, X. (2007). Estimation of distribution algorithms for testing object oriented software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pages 438–444.
- Seesing, A. and Gross, H.-G. (2006). A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134.
- Tonella, P. (2004). Evolutionary testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128.
- Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., and Wong, W. E. (2006). Establishing structural testing criteria for Java bytecode. *Software: Practice and Experience*, 36(14):1513–1541.
- Wappler, S. and Wegener, J. (2006). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*.
- Yoo, S. and Harman, M. (2007). Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis*.

Applying Search-Based Techniques for Requirements-Based Test Case Prioritization

Camila Loiola Brito Maia, Fabrício Gomes de Freitas, Jerffeson Teixeira de Souza

Optimization in Software Engineering Group (GOES.UECE)

State University of Ceará (UECE)

Fortaleza, 60740-903, Ceará, Brazil

camila.maia@gmail.com, fabríciogf.uece@gmail.com, jeff@larces.uece.br

Abstract. *Although software test is very important, there may be situations in which there is no time to execute all test cases. It is important to order the test cases so that the most important ones come first. Most of the works about search-based test case prioritization have used unit tests techniques, and we have to know the code in advance. This work considers requirement-based metrics to prioritize test cases, like volatility and importance. In other words, no code has to be known to prioritize the test cases by this approach. Some search-based techniques were applied to this multi-objective requirement-based formulation of the test case prioritization problem. The results show the efficiency of search-based techniques to order the test cases.*

Resumo. *Apesar de ser muito importante testar o software, há situações em que não há tempo suficiente para que todos os casos de teste sejam executados. É importante ordenar os casos de teste de modo que os mais importantes sejam priorizados. A maioria dos trabalhos sobre priorização de casos de teste utilizando técnicas de otimização tem usado técnicas para testes unitários, quando é necessário conhecer o código-fonte. Este trabalho considera métricas baseadas nos requisitos, não sendo necessário conhecer o código-fonte. Algumas técnicas de otimização foram aplicadas à formulação multiobjetiva proposta para este problema. Os resultados mostram a eficiência das técnicas de otimização para a ordenação dos casos de teste.*

1. Introduction

Nowadays, software is intensively present in our lives: in the phone, airplanes, surgeries, security, home activities, and so on. Consequently, these software systems are getting more and more complex.

Despite software testing being a very important process to be executed, often there is not enough time or resources to execute all planned test cases. In this case, it is desirable to prioritize the test cases in a way that the most important ones are in the first positions in an attempt to guarantee their execution.

In the context of software engineering, a new research field has emerged by the application of search techniques to well-known software engineering problems. This new research field has been titled Search-Based Software Engineering (SBSE), which concerns on finding solutions to complex software engineering problems formulated as search problems.

In this work we propose a new formulation to prioritize test cases, based on requirements characteristics, such as volatile, complexity and importance. Search-based techniques had been used to evaluate the new formulation.

The remaining of the paper is organized as follows: Section 2 describes previous works regarding test case prioritization, Section 3 discusses some details of the problem and Section 4 shows the mathematical formulation proposed for this problem. Section 5 shows the design of the experiments and Section 6 presents the results from the experiments and describes the analysis of these results. Section 7 concludes the work and points out some future directions to this research.

2. Related Works

This section reports some works regarding test case prioritization, some of them using search-based approaches and metaheuristics.

In [Srikanth; Williams; Osbome 2005], authors proposed a test case prioritization technique called PORT (Prioritization of Requirements for Test). PORT prioritizes test cases based upon four factors: requirements' volatility, customer priority, implementation complexity, and fault proneness of the requirement.

The authors of [Walcott; Soffa; Kapfhammer; Roos 2006] proposed a test case prioritization technique with a genetic algorithm which reorders test suites considering two objectives: testing time constraints and code coverage. This technique significantly outperformed other prioritization techniques described in the paper.

In reference [Yoo and Harman 2007], authors described a Pareto approach to prioritize test cases based on code coverage, execution cost and fault-detection history. Three algorithms were compared: a reformulation of a Greedy algorithm (Additional Greedy algorithm), Non-Dominating Sorting Genetic Algorithm (NSGA-II), and a variant of NSGA-II, vNSGA-II.

Finally, reference [Li; Harman; Hierons 2007] compared five algorithms: Greedy algorithm, Additional Greedy algorithm, 2-Optimal Algorithm, Hill Climbing, and genetic algorithm. For small programs, the performance was almost identical for all algorithms and coverage criteria. The Greedy algorithm performed the worst and the genetic algorithm and Additional Greedy algorithm produced the best results.

3. Aspects Considered in the Problem

Requirements are the basis for software development. They represent what the customer wants and what he/she expects from the system. Because of its extreme importance for the system, we considered three requirements characteristics to compose the formulation of the problem, as described below.

3.1. Complexity

Each requirement has its complexity, which is a scalar value that represents how complex the requirement is to be implemented. In this work we consider requirement's complexity ranging from 1 to 5, where 1 represents the lowest complexity and 5 represents the more complex level. It is desirable that the most complex requirements are tested first, since they have more chance to be implemented incorrectly, and they probably will have more fails than a less complex requirement.

3.2. Importance

The client company has a set of stakeholders (customers) responsible for approving the developed system. Each of them has an importance value to the development company, from 1 (minor importance) to 10 (major importance).

Each stakeholder assigns an importance value for each one of the requirements. In our experiments, we considered these importance values from 1 (minor importance) to 5 (major importance). The requirement's importance takes into account those importance values assigned by stakeholders considering each stakeholder's importance.

It is desirable that the most important requirements are tested first because they reflect the most important functionalities to the client's business. Besides that, a problem found in a most important feature is a more serious error; therefore, it is necessary to take into account the customer's importance.

For example, there are two stakeholders in customer's company that will be responsible for the system's requirements: A and B. Client A has importance 2 for the company, and he assigns 5 as importance value to requirement 1 and 6 to requirement 2. Client B has importance 8 for the company, and he assigns value 9 to requirement 1 and 4 to requirement 2. In this case, it is important to consider requirement 1 first, since client B has a major importance for the company and he assigns a higher value to requirement 1 than requirement 2.

3.3. Volatility

Volatility of a requirement represents the amount of changes that occurred in it since its first release. The greater the number of changes, the greater the scalar value for the volatility. As used for the two characteristics above, we considered values from 1 (little volatility) to 5 (major volatility) for the experiments. It is desirable that the most volatile requirements are tested first because it is more likely to find errors when requirements change a lot.

4. Problem Formulation

The proposed multi-objective formulation for test case prioritization can be seen in the box below. Let S be a solution of the problem, i.e., a test suite, or a set of test cases ordered for execution. The three following objective function must be maximized:

- (1) $F_I(S)$: Total importance of S . For each requirement, the sum of all importance values assigned by all stakeholders, weighted by stakeholder's importance, is calculated. Then, a weight function is applied, based on the first test case that cover this requirement on suite S , i. e., higher values are assigned when the first test case that covers the requirement is placed at the beginning of test suite. After all requirements have known their importance values, these values are added and will compute total importance of S .
- (2) $F_V(S)$: Total volatility of S . For each test case i of S , we sum the volatility of all requirements covered by i , and the total sum is weighted by the position of test case i on suite S .
- (3) $F_C(S)$: Total complexity of S . For each test case i of S , we sum the complexity of all requirements covered by i , and the total sum is weighted by the position of test case i on suite S .

$$\begin{aligned}
 (1) \text{Maximize } F_I(S) &= \sum_{i=1}^n [(N - P_i) * (\sum_{j=1}^m I(r_i, c_j) * C(c_j))] \\
 (2) \text{Maximize } F_V(S) &= \sum_{i=1}^N (i * \sum V(r)), \forall r | cover(i, r) = 1 \\
 (3) \text{Maximize } F_C(S) &= \sum_{i=1}^N (i * \sum X(r)), \forall r | cover(i, r) = 1
 \end{aligned}$$

where:

- a) S is a solution of the problem, i.e., a test suite with the test cases ordered;
- b) N is the lenght of test suite (number of test cases)
- c) $cover(i, r) = \begin{cases} 1, & \text{if test case } i \text{ covers requirement } r \\ 0, & \text{otherwise} \end{cases}$

Specific to the equation (1):

- d) $F_I(S)$ is the total importance assigned to a test suite S ;
- e) n is the number of requirements of the system
- f) P_i is the position of first test case that covers the requirement i
- g) m is the number of client's stakeholders of the system
- h) $I(r_i, c_j)$ is the importance value assigned by customer j to requirement r
- i) $C(c_j)$ is the importance value assigned to customer j

Specific to the equation (2):

- j) $F_V(S)$ is the total volatility assigned to a test suite S ;
- k) $V(r)$ is the volatility value assigned to requirement r

Specific to the equation (3):

- l) $F_C(S)$ is the total complexity assigned to a test suite S ;
- m) $X(r)$ is the complexity value assigned to requirement r

5. Experimental Design

This section describes all aspects related to the design of the experiments.

5.1. The Data

For these experiments, two sets of instances, A and B with increasing sizes, were synthetically generated. The instance A is the simplest one, with 10 customers, 40 requirements and 30 test cases. Instance B was composed by 20 customers, 100 requirements and 90 test cases.

5.2. The Algorithms

The test case prioritization problem was solved by the metaheuristics NSGA-II [Deb; Pratap; Agarwal; Meyarivan 2002], MoCell [Nebro; Durillo; Luna; Dorronsoro; Alba 2009], SPEA2 [Zitzler; Laumanns; Thiele 2001], and a random algorithm. No details of these algorithms will be discussed in this paper.

5.3. Details of Experiments

The three multiobjective algorithms were executed 100 times, and each execution generated 100 solutions to be compared. The random algorithm generated 100 solutions, among which only the nondominated solutions were used in comparison between the algorithms and to calculate the comparison metrics.

5.4. Comparison Metrics

To compare the results obtained by the algorithms we used the execution time of the algorithms and a quality indicator named hypervolume, which measures both convergence and diversity. Large values for the hypervolume indicator are desirable.

6. Experimental Results and Analysis

The results are presented in the tables and figures below, and discussed in this section.

Figures 1, 2 and 3 show results for instance A, while Figure 4 shows the results for instance B. The objective functions are compared two by two. For the Figures 1, 2 and 3, the first graph contains only the three metaheuristics, while the second graph contains the same metaheuristics and the random algorithm.

Because results for random algorithm were very worse comparing with the other three algorithms, as can be seen in Figures 1, 2 and 3, we will consider this algorithm only in comparative graphics for instance A. Its execution time can be seen in Table 1.

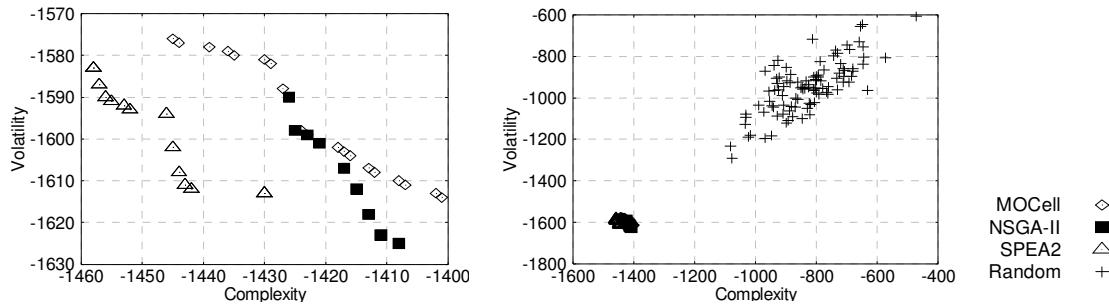


Figure 1. Objectives Complexity and Volatility for instance A with only metaheuristics (left) and all algorithms (right).

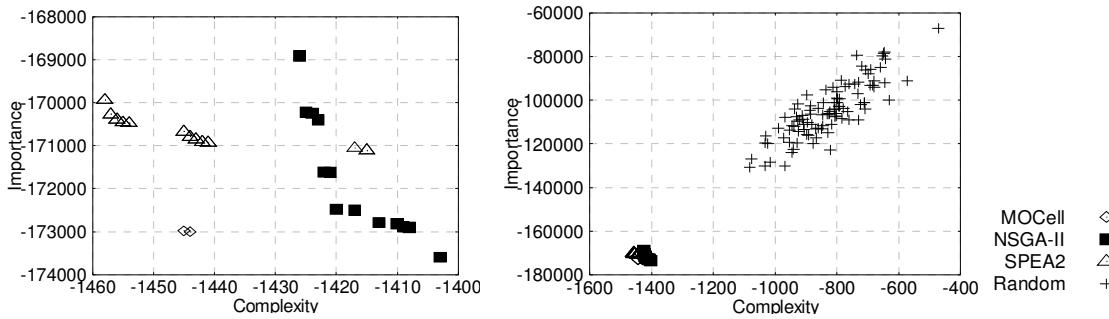


Figure 2. Objectives Complexity and Importance for Instance A with only metaheuristics (left) and all algorithms (right).

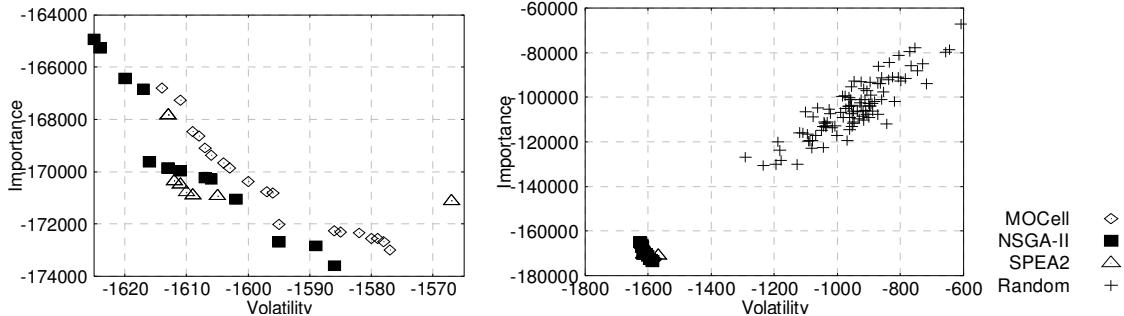


Figure 3. Objectives Volatility and Importance for Instance A with only metaheuristics (left) and all algorithms (right).

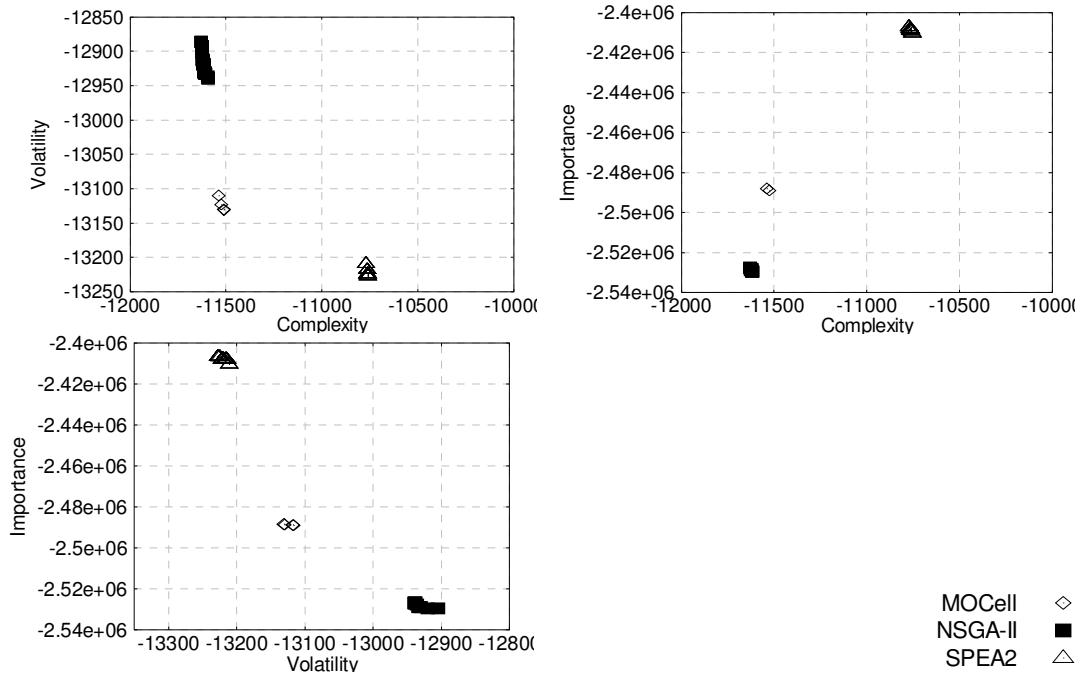


Figure 4. Results of search-based algorithms for instance B.

For instance A, it is possible to see in figures that SPEA2 generated solutions closer to the axis. SPEA2 was followed by NSGA-II and finally by MOCell, but the objective values were very similar, meaning that the three algorithms found good solutions for the problem.

For instance B, represented by Figure 4, each algorithm focused on a search region. All algorithms found good solutions, but, for this case, SPEA2 obtained the worst solutions. The SPEA2 seems to have a better performance for smaller search spaces, while NSGA-II and MOCell have better performance for larger search spaces.

Table 1 presents average and standard deviation for execution time, while Table 2 presents average and standard deviation for hypervolume quality indicator, per

instance. For the three metaheuristics, it has been considered the average between all executions for instance A and instance B, while for the random algorithm, the average was calculated by taking into account the 100 solutions generated by single executions.

As can be seen in Table 1, random algorithm had the best lowest execution time. Nevertheless, as cited previously, this algorithm produced the worst solutions. The second best execution time was obtained by MOCell, followed by NSGA-II. SPEA2 algorithm was the slowest algorithm, about 7.64 times slower than MOCell and 6.13 slower than NSGA-II.

Table 1. Execution time (ms) for each algorithms

Algorithm	Instance A	Instance B
MOCell	3412.50±188.656	15233.28±1322.352
NSGA-II	4255.16±141.770	14429.93±1488.813
SPEA2	26072.99±2944.850	35931.92±7571.272
Random	47	79

Table 2. Hypervolume for the metaheuristics

Algorithm	Instance A	Instance B
MOCell	0.971812±0.0196	0.896479±0.2121
NSGA-II	0.980380±0.0188	0.930851±0.1069
SPEA2	0.973822±0.0173	0.912884±0.1426

The hypervolume indicator, showed in Table 2, was calculated for the three search-based algorithms. The best hypervolume value was obtained by NSGA-II. The second best value was obtained by SPEA2, followed by MOCell algorithm. NSGA-II had the best hypervolume and MOCell had the lowest execution time.

7. Conclusion and Future Work

This paper proposed a multiobjective formulation for the test case prioritization problem, considering requirements' characteristics. Regarding the comparison of algorithms, all results were consistent to show that metaheuristics can be applied to solve the test case prioritization problem.

As future work, we will consider major instances and we will compare the results found by metaheuristics with human responses, beyond analyzing the improving of fault detection of test suites generated by the metaheuristics.

References

- Walcott, K., Soffa, M. L., Kapfhammer, G., and Roos, R. (2006) "Time-Aware Test Suite Prioritization", Proceedings of the International Symposium on Software Testing and Analysis, pp. 1-12.
- Yoo, S. and Harman, M. (2007) "Pareto Efficient Multi-Objective Test Case Selection", Proceedings of the International Symposium on Software Testing and Analysis, pp. 140-150.

- Li, Z., Harman, M., and Hierons, R. (2007) "Search Algorithms for Regression Test Case Prioritization", IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225-237.
- Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., and Alba, E. (2009) "MOCell: A cellular genetic algorithm for multiobjective optimization". Int. J. Intell. Syst. vol. 24, pp. 726-746.
- Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002) "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II", IEEE Transactions on Evolutionary Computation, Vol. 6, Issue 2, 182-197.
- Srikanth, H., Williams, L., and Osbome, J. (2005) "System Test Case Prioritization of New and Regression Test Cases", International Symposium on Empirical Software Engineering.
- Zitzler, E., Laumanns, M., and Thiele, L. (2001) "SPEA2: Improving the strength Pareto evolutionary algorithm", EUROGEN, Citeseer, 95-100.
- Meisel, W. S. (1973) "Tradeoff decision in multiple criteria decision making", Multiple Criteria Decision Making. University of South Carolina Press, Columbia-SC, 461-476.
- Smith, A. and Jones, B. (1999). On the complexity of computing. In *Advances in Computer Science*, pages 555–566. Publishing Press.

Uma Abordagem de Otimização Multiobjetiva para o Problema da Priorização da Correção de Defeitos

Tarciane C. Andrade, Fabrício G. Freitas, Daniel P. Coutinho, Jerffeson T. Souza

Grupo de Otimização em Engenharia de Software (GOES.UECE)

Universidade Estadual do Ceará - Fortaleza, Ceará, Brasil

tarcianeandrade@gmail.com, fabríciogf@uece.br,
daniel.pintocoutinho@gmail.com, jeff@larces.uece.br

Abstract. *Tests are essential to improve software quality and they generate as result a list of defects. The prioritization of which defects should be corrected aids in a better allocation of resources, effort and planning of future versions. However, prioritize them requires that some factors should be considered according to different views of the team and the client. This paper proposes an multiobjective optimization formulation to solve this problem and validates the approach with the metaheuristic NSGA-II and MOCell. The results with the proposed approach outperformed the results of other techniques frequently applied in the context of the problem.*

Resumo. *Os testes são essenciais para melhorar a qualidade do software e possuem como resultado uma lista de defeitos encontrados. A priorização de quais defeitos devem ser corrigidos auxilia em uma melhor distribuição dos recursos, do esforço e no planejamento das futuras versões, porém, priorizá-los exige que alguns fatores sejam ponderados de acordo com diversas visões, da equipe e do cliente. Este artigo propõe uma formulação de otimização multiobjetiva para resolver o problema de priorização de defeitos e valida a abordagem com as metaheurísticas NSGA-II e MOCell. Os resultados mostraram-se melhores em comparação com outras técnicas normalmente utilizadas na prática.*

1. Introdução

A possibilidade de ocorrer erros humanos durante o processo de desenvolvimento de software é alta, independente da fase do processo, como levantamento de requisitos, análise, projeto e implementação. Assim, é fundamental que o desenvolvimento do software seja acompanhado de atividades de testes para assegurar que o software funcione conforme o especificado e que atenda aos requisitos dos clientes.

A correção de defeitos é uma tarefa custosa para a empresa e, como tal, deve ser analisada e executada de acordo com certos critérios a fim de corrigir primeiro aqueles defeitos de maior prioridade. O efeito colateral direto da priorização incorreta dos defeitos é a má utilização dos recursos e do esforço necessário em função do enfoque na correção de defeitos menos prioritários [Caetano 2007].

No contexto de defeito de software, entre os diversos critérios que podem ser associados aos defeitos estão aqueles relacionados às visões de diferentes atores do processo. Por exemplo, na visão do desenvolvedor, o critério mais relevante está relacionado ao tempo necessário para correção, enquanto para o gerente do projeto trata

principalmente o risco do defeito, e o cliente tem mais interesse no fator de impacto do defeito para o negócio. Percebe-se, assim, a existência de várias visões em relação à priorização de correção de defeitos de software.

O problema de priorização de defeitos é caracterizado como tendo um elevado espaço de soluções possíveis, por se tratar das permutações de ordem dos defeitos. Além disso, como já citado, o contexto do problema pode ser caracterizado por diversos critérios a serem considerados simultaneamente. A resolução de tal problema de forma manual se mostra então um processo difícil, devendo portanto ser realizada por meio de um método automatizado e não exaustivo. Através disso, as pessoas envolvidas com o processo de desenvolvimento podem se preocupar com atividades onde a capacidade da inteligência humana é mais conveniente.

Nesse contexto, tal problema faz parte do contexto da área conhecida como *Search-based Software Engineering* que trata da utilização de métodos de otimização matemática para a resolução de problemas da Engenharia de Software em várias fases do desenvolvimento, como planejamento de projeto [Alvim; Barros; Neto 2009], teste de software [Maia 2009] [Yoo; Harman 2007], entre outros. Neste artigo, propõe-se uma formulação multiobjetiva para o problema de priorização de defeitos de software. A técnica de otimização matemática utilizada procura encontrar as melhores soluções de acordo com os aspectos severidade, importância e freqüência, que serão definidos a seguir. Assim, as contribuições principais do trabalho são:

- Uma formulação multiobjetiva para o problema da priorização de defeitos para correção, considerando os objetivos de antecipar, simultaneamente, a correção dos defeitos com maior severidade, importância e freqüência;
- Resolução do problema com o uso de metaheurísticas multiobjetivas e realização de avaliação empírica para demonstrar a aplicabilidade da formulação apresentada e a eficiência da técnica proposta na resolução do problema.

Este trabalho está organizado da seguinte forma. A seção 2 detalha um breve levantamento de alguns trabalhos relacionados com a área de priorização de defeitos de software. A seção 3 apresenta a formulação multiobjetiva do problema e apresenta uma breve análise da complexidade do problema. A seção 4 discute uma avaliação empírica do problema ao comparar a sua solução com otimização utilizando o NSGA-II e MOCell, e a seção 5 detalha as conclusões e trabalhos futuros a cerca dessa pesquisa.

2. Trabalhos Relacionados

Algumas ferramentas de registro e rastreamento de defeitos existentes [Bugzilla 2009] [Mantis 2009] possibilitam a classificação dos defeitos de acordo com sua prioridade. Existem também na literatura várias maneiras de classificar os defeitos para posterior priorização da correção. Entre as classificações mais usadas estão os aspectos de prioridade (defeitos com alta prioridade são corrigidos imediatamente ou num curto prazo de tempo) e severidade (impacto do defeito no desenvolvimento da aplicação) [Bugzilla 2009] [Whalen 2009] [Zeller 2009] [Caetano 2007]. Outros trabalhos indicam que prioridade é a urgência de correção de defeito na visão do cliente e severidade é o impacto do defeito na funcionalidade requerida pelo cliente [Kelsey 2006] [Tsiu 2004]. Outras delas classificam os defeitos de forma mais completa de acordo com a severidade (impacto no desenvolvimento), frequência (percentual de ocorrência do defeito) e importância (visão do cliente) [Horch 2003]. Cada uma das classificações possuem

valores para os defeitos que podem variar entre: crítico, urgente, médio e baixo, por exemplo.

Em [Chandler 2010], o autor utiliza da abordagem de RPN (*Risk Priority Number*) presente no FMEA (*Failure Model and Effect Analysis*) [Six Sigma 2008] para priorizar a correção de defeitos. O objetivo do FMEA é detectar falhas antes que se produza um produto ou processo. A análise consiste basicamente na formação de um grupo de pessoas que identificam para o produto/processo os tipos de defeitos que podem ocorrer, os defeitos e as possíveis causas deste defeito e riscos de cada causa de falha é feito por meio de índices. Com base nesta avaliação, são tomadas as ações necessárias para diminuir estes riscos. Na fase de avaliação de riscos existem três índices: severidade, ocorrência e detecção. Esse três índices, possuem critérios e quando multiplicados formam o RPN. O RPN é então utilizado como forma de analisar as possíveis falhas e tentar mitigá-las. Em [Chandler 2010], o RPN é modificado para ser usado na priorização da correção de defeitos. Para isso, o índice de detecção foi substituído pelo índice de prioridade de correção do defeito para o usuário. Assim, quanto maior o RPN, mais rápida a correção do defeito deverá ser realizada.

Em [Alvim; Barros; Neto 2009], os autores propõem uma abordagem com heurística híbrida para planejar as tarefas de correção de erros do sistema para os desenvolvedores de forma que minimize o custo para corrigi-lo. Os autores consideram custo como sendo três fatores: minimizar o esforço requerido para a próxima versão do sistema, garantir que os erros mais urgentes sejam corrigidos na próxima versão e maximizar a taxa de ocupação da equipe de testes. A abordagem constrói um cronograma viável para um determinado conjunto de tarefas de correção de erros com prioridades diferentes para cada desenvolvedor.

3. Formulação Multiobjetiva do Problema da Priorização de Defeitos

3.1 Descrição Formal do Problema

Considere inicialmente um conjunto, D , de defeitos descobertos durante a fase de testes de um dado sistema. Seja N o número de defeitos contidos no conjunto D . Denota-se como D_i , o defeito i do conjunto D , com i variando de 0 até $N - 1$. Cada defeito $D_i \in D$, possui uma severidade, S_i , uma importância, I_i , e uma freqüência, F_i . Consideramos severidade como sendo o fator relacionado à visão técnica, importância como sendo o fator relacionado à visão do cliente e freqüência como sendo a porcentagem de ocorrência do defeito no sistema. Tal escolha foi realizada dado o uso na literatura. Contudo, destacamos que quaisquer outros fatores podem ser usados ou adaptados na abordagem proposta.

Nesse contexto, uma solução para o problema da priorização de correção de defeitos é uma ordenação, O , dos elementos de D . Pretende-se, como discutido anteriormente, que essa ordenação contenha em suas posições iniciais os defeitos com maior severidade, importância e freqüência, simultaneamente.

3.2 Formulação de Otimização Multiobjetiva

O problema da priorização de defeitos para correção pode ser definido matematicamente como um problema de otimização multiobjetiva da seguinte forma:

Encontrar uma ordenação, O , dos elementos de D , maximizando simultaneamente as seguintes funções:

$$F_S(O) = \sum_{i=0}^{N-1} [S_i \times (N - X_i)] \quad (1)$$

$$F_I(O) = \sum_{i=0}^{N-1} [I_i \times (N - X_i)] \quad (2)$$

$$F_F(O) = \sum_{i=0}^{N-1} [F_i \times (N - X_i)] \quad (3)$$

onde,

- D representa o conjunto de defeitos a serem corrigidos
- S_i representa a severidade do defeito i
- F_i representa a freqüência do defeito i
- X_i representa a posição do defeito i na ordenação O
- I_i representa a importância do defeito i
- N representa o número de defeitos

Na formulação acima, a equação (1) considera a priorização dos defeitos em D que possuem um maior valor de severidade. Perceba que nessa equação, quanto maior for o valor da severidade atribuída a certo defeito i , dado por S_i , e mais próximo do início estiver esse defeito i na ordenação O , ou seja, quanto menor o valor de X_i , maior será o fator adicionado a função $F_S()$.

No sentido de ilustrar essa computação, considere o seguinte exemplo ilustrativo. Sejam três defeitos, D_0 , D_1 e D_2 , logo, $N = 3$. Considere que os valores 1, 2 e 3 foram atribuídos como sendo a severidade desses três defeitos, respectivamente, ou seja, $S_0 = 1$, $S_1 = 2$ e $S_2 = 3$. Nessa situação, se considerarmos somente o objetivo de priorizarmos os defeitos com maior severidade, a ordenação ideal seria $D_2D_1D_0$.

Usando a equação (1) para calcularmos o valor da função $F_S()$ para todas as ordenações possíveis dos três defeitos, temos:

$$\begin{aligned} F_S(D_0D_1D_2) &= 1 \times (3 - 0) + 2 \times (3 - 1) + 3 \times (3 - 2) = 10 \\ F_S(D_0D_2D_1) &= 1 \times (3 - 0) + 2 \times (3 - 2) + 3 \times (3 - 1) = 11 \\ F_S(D_1D_0D_2) &= 2 \times (3 - 0) + 1 \times (3 - 1) + 3 \times (3 - 2) = 11 \\ F_S(D_1D_2D_0) &= 2 \times (3 - 0) + 1 \times (3 - 2) + 3 \times (3 - 1) = 13 \\ F_S(D_2D_0D_1) &= 3 \times (3 - 0) + 1 \times (3 - 1) + 2 \times (3 - 2) = 13 \\ F_S(D_2D_1D_0) &= 3 \times (3 - 0) + 1 \times (3 - 2) + 2 \times (3 - 1) = 14 \end{aligned}$$

Os cálculos acima mostram que a ordenação $D_2D_1D_0$ de fato obteve o maior valor de $F_S()$. A partir desses cálculos, percebe-se que a função $F_S()$, descrita pela equação (1), expressa exatamente o nosso objetivo de antecipar os defeitos com maior severidade, visto que as ordenações que antecipam esses defeitos terão maior valor de $F_S()$. Assim, tem-se o desejo de se buscar uma ordenação que maximize o valor dessa função.

Analogamente, a mesma análise realizada acima pode ser feita para as equações (2) e (3), que expressam, respectivamente, os objetivos de anteciparmos a correção dos defeitos com maior importância e freqüência.

3.3 Análise da Complexidade do Problema

Para entender a complexidade de resolução desse problema, basta analisar o tamanho do seu espaço de soluções, visto que o mesmo pode ser facilmente identificado como um problema de busca, que varre o espaço de soluções, ordenações, a procura daquela com maiores valores, simultaneamente, de $F_S()$, $F_I()$ e $F_F()$.

Considerando a existência de N defeitos, o espaço de soluções do problema de priorização da correção de defeitos possui $N!$ (n fatorial) soluções, visto que existem $N!$ possíveis ordenações para um conjunto de N elementos. Dada essa característica, percebe-se a impossibilidade da utilização de estratégias exaustivas que percorram todo o espaço de soluções do problema. Assim, percebe-se a necessidade de soluções eficientes, mesmo sem garantia de otimalidade do resultado gerado por essa solução.

4. Avaliação Empírica

4.1 Descrição dos Dados

Foram utilizadas sete instâncias com o intuito de demonstrar a validade da abordagem em diferentes contextos. A diferença entre as instâncias reside na quantidade de defeitos existentes para correção: 20, 30, 40, 50, 100, 200, 300. Em cada instância, é definido para cada defeito um valor entre 1 e 5 para cada um dos três fatores. Para maior verossimilhança com situações reais de projetos de software, os valores gerados para cada fator em cada defeito é independente dos demais. Os significados dos valores estão apresentados na Tabela 1 a seguir:

Tabela 1. Significados dos Valores em Cada Fator Considerado

Valor	Severidade	Importância	Frequência
1	Sem valor	Insignificante	1% a 20%
2	Baixa	Inferior	21% a 40%
3	Média	Moderada	41% a 60%
4	Alta	Importante	61% a 80%
5	Crítica	Urgente	81% a 100%

Para ilustrar os dados de entrada utilizados, a Tabela 2 a seguir apresenta o conjunto de defeitos na instância de 20 defeitos:

Tabela 2. Dados da Instância com 20 Defeitos

Defeito	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	D ₈	D ₉
<i>Severidade</i>	4	4	1	1	5	3	1	1	5	5
<i>Importância</i>	3	5	1	5	2	2	1	3	5	3
<i>Frequência</i>	1	1	3	1	4	1	5	5	4	2
Defeito	D ₁₀	D ₁₁	D ₁₂	D ₁₃	D ₁₄	D ₁₅	D ₁₆	D ₁₇	D ₁₈	D ₁₉
<i>Severidade</i>	4	5	3	2	5	2	1	4	2	4
<i>Importância</i>	1	5	4	2	5	5	2	1	1	1
<i>Frequência</i>	5	3	5	2	1	1	1	5	1	1

4.2 Metodologia

Após a geração independente de cada conjunto de dados, utilizando os algoritmos NSGA-II [Deb et al 2000] e MOCell [Nebro et al. 2009]. Os algoritmos foram executados e posteriormente comparados a partir dos resultados tempo e dos resultados em si para as instâncias. Os parâmetros utilizados para a metaheurística NSGA-II em cada teste foram: tamanho da população de 100, quantidade de gerações de 3000, e taxa de mutação igual a 2% e taxa de cruzamento de 90%. As configurações para a metaheurística MOCell foram: matriz populacional de 100 por 100, arquivo externo de tamanho 100, e número de gerações de 3000.

De posse dos resultados obtidos a partir da aplicação das metaheurísticas, e da determinação do método mais adequado, comparamos os resultados gerados com resoluções tipicamente humanas. Para isso, analisamos como a indústria resolve o problema de priorização de defeitos [Bugzilla 2009] [Chandler 2007] [Mantis 2009] e comparamos essas abordagens humanas com os resultados gerados pela metaheurística descoberta como a mais adequada. Nesse sentido, analisamos três formas de resolução: ordenação por soma (estratégia humana I), ordenação por produto (estratégia humana II) e aleatório [Harman 2007]. A abordagem de ordenação por soma resolve o problema de acordo com o resultado da soma dos três fatores para cada defeito e seguinte ordenação de forma decrescente. A ordenação por produto é semelhante à ordenação por soma, com a alteração que nesta abordagem os valores dos fatores em cada defeito são multiplicados. Essas abordagens foram consideradas, pois descrevem formas de resolução do problema na prática.

4.3 Apresentação e Análise dos Resultados

Em relação ao tempo de execução, percebe-se pela análise da Tabela 5, que a metaheurística NSGA-II consumiu cerca de 40% do tempo requerido pelo MOCell.

Tabela 5. Resultados do Tempo (em milisegundos) para NSGA-II e MOcell

Instância (# de defeitos)	NSGA-II	MOCell
20	38703.20±2476.462	48716.60±2530.685
40	41359.90±4316.934	64431.30±2756.200
60	45814.90±7738.550	83762.80±2215.176
80	43843.40±2704.329	101722.50±2876.614
100	50900.60±2615.228	130468.10±6408.998
200	69278.40±3723.749	206521.60±5879.176
300	94939.90±4750.750	273624.60±4205.762

A visualização de um gráfico tridimensional no plano não se mostra adequada dada a dificuldade de leitura do mesmo. Assim, apresentamos os resultados com três gráficos bidimensionais, sendo cada um com os resultados a dois objetivos. Além disso, vale ressaltar que os gráficos são apresentados como se o problema fosse de minimização, no sentido de permitir uma melhor visualização da Frente de Pareto.

Os experimentos com todas as instâncias mostraram resultados similares sob o aspecto da validade da abordagem proposta. Por restrições de espaço, apresentaremos apenas o gráfico da instância de 100 defeitos na Figura 1 com apenas o algoritmo MOCell para facilitar a visualização. Para maior análise, apresentamos também a seguir as soluções para a instância com 20 defeitos de acordo com os dados da Tabela 2.

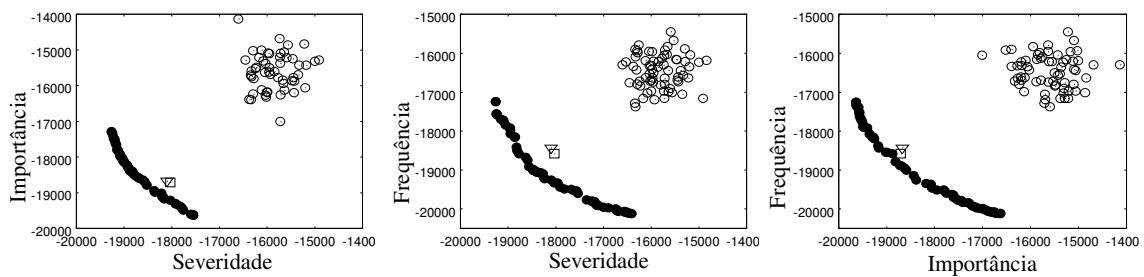


Figura 1: Resultados de MOCell e técnicas humanas com 100 defeitos, legenda
 ● MOCell, ○ randômica, ▼ “Estratégia Humana I”, □ “Estratégia Humana II”.

Como a Figura 1 indica, a abordagem proposta apresenta melhores resultados do que as outras técnicas em todos os cenários na instância com 100 defeitos. Isso é constatado devido ao fato de sempre existir alguma solução da Frente de Pareto que é melhor nos dois objetivos que qualquer solução das outras técnicas. Por exemplo, para as soluções de “Estratégia Humana I” e “Estratégia Humana II”, existe uma solução na Frente de Pareto que está “mais para esquerda” e “mais para baixo” no gráfico. Em outros termos, tal solução na Frente de Pareto é melhor nos dois.

Para melhor apresentação da eficácia do resultado obtido, realizamos a análise do teste executado com a instância com 20 defeitos que foi realizado com os dados apresentados na Tabela 2. A solução da “Estratégia Humana I” indica a ordem “D8 D11 D12 D4 D14 D1 D9 D10 D17 D7 D0 D15 D3 D2 D5 D13 D12 D2 D16 D18” que resulta nos valores 779 para Severidade e 712 para Importância de acordo com as funções apresentadas na seção 3. Uma das soluções geradas pela metaheurística indica a ordem “D8 D11 D14 D1 D15 D9 D4 D12 D0 D5 D10 D19 D17 D7 D18 D13 D3 D16 D6 D2” que resulta nos valores 797 e 726, respectivamente. Assim, mostramos que existe uma ordem melhor para a correção dos defeitos quando este problema é resolvido com a abordagem proposta, pois os resultados dos somatórios dos valores priorizados na ordem são maiores. Em relação aos resultados com as soluções aleatórias, observa-se que de fato tal técnica apresenta os piores resultados. Assim, motiva-se o uso de uma abordagem estruturada, como a proposta neste artigo.

Resultados obtidos nos experimentos sugerem que a metaheurística MOCell é capaz de produzir soluções para o problema da priorização da correção de defeitos de forma mais precisa. Esse fato, aliado a vantagem de obtenção de diferentes soluções dispostas na Frente de Pareto gerada pelo MOCell, o que permite uma melhor interpretação e seleção de uma solução em particular, qualifica esse método automático como extremamente adequado para resolver o problema tratado nesse artigo.

5. Conclusões

A fase de teste de software é uma das mais custosas durante o desenvolvimento de software e apresenta como um artefato a lista de defeitos encontrados no sistema. Dada o contexto de vários critérios e do impacto do momento onde cada defeito é corrigido, uma abordagem automática que realize esta priorização é adequada.

Nesse trabalho, propomos uma abordagem multiobjetiva para esse problema. Os resultados indicam que a resolução com as técnicas da otimização consideradas retornam soluções melhores em relação a outras abordagens encontradas na literatura e na prática. Trabalhos futuros são: uso de instâncias reais e análise de soluções humanas.

Referências

- Alvim, A.; Barros, M.; Netto, F. (2009) “A Hybrid Heuristic Approach for Scheduling Bug Fix Tasks to Software”. 1st International Symposium on Search Based Software Engineering. Windsor, 2009.
- Bagnall, A., Rayward-Smith, V. J., Whittle, I. (2001) “The next release problem”, Information and Software Technology, pp. 883–890, 2001.
- Bugzilla. (2009) “Bugzilla”. <<http://www.bugzilla.org/>>. Acesso em: 12 Maio 2010.
- Caetano, C. (2007) “Gestão de Defeitos”. Engenharia de Software Magazine, v. 1, n. 1, p. 60-67, 2007.
- Chandler, L. (2010) “Software Defects: Severity, Priority and Impact”, 2007. <<http://sccaffadaffa.com/2007/06/software-defects-severity-priority-and.html>>. Acesso em: 05 Maio 2010.
- Deb, K. et al. (2000) “A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II”, IEEE Transactions on Evolutionary Computation, V. 6, pp. 182-197, 2000.
- Harman, M. (2007) “The Current State and Future of Search-based Software Engineering”. Proceedings of International Conference on Software Engineering / Future of Software Engineering 2007, Minneapolis Minnesota USA, pp. 342-357, 2007.
- Kelsey, R. B. (2006) Software Project Management: Measures for Improving Performance. Management Concepts, 2006. 92-93 p. ISBN 978-1567261738.
- Horch, J. W. (Ed.). (2003) Practical Guide to Software Quality Management. 2^a. ed. [S.I.]: Artech House, 2003. ISBN 9781580535274.
- Maia, C. L. B., Carmo, R. A. F, Freitas, F. G, Campos, G. A. L, Souza, J. T. (2009) “A Multi-Objective Approach For The Regression Test Case Selection Problem”, Anais do XLI Simpósio Brasileiro de Pesquisa Operacional, Porto Seguro, 2009.
- Mantis. MantisBT Group, (2009). <http://www.mantisbt.org/bugs/main_page.php>. Acesso em: 03 Maio 2010.
- Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B. & Alba, E.. (2009) “MOCell: A Cellular Genetic Algorithm for Multiobjective Optimization”. International Journal of Intelligent Systems , 24, 7, Julho, 2009.
- Six Sigma. “Failure Modes and Effects Analysis (FMEA)”, (2008). <<http://www.isixsigma.com/tt/fmea/>>. Acesso em: 05 Maio 2010.
- Whalen, D. (2009) “Software Testing. Severity vs Priority”, 2009. <<http://www.softwaretestingclub.com/profiles/blogs/severity-vs-priority>>. Acesso em: 05 Maio. 2010.
- Yoo, S., Harman, M. (2007) “Pareto Efficient Multi-Objective Test Case Selection”, Proceedings of the International Symposium on Software Testing and Analysis, 2007, pp. 140-150.
- Zeller, A. (2009) Why Programs Fail: A Guide to Systematic Debugging. 2^a. ed. [S.I.]: Elsevier Science & Technology Books, 2009. 32-34 p. ISBN 978-1558608665.

Uma Nova Abordagem de Otimização Multiobjetiva para o Planejamento de *Releases* em Desenvolvimento Iterativo e Incremental de Software

Márcia Maria Albuquerque Brasil, Fabrício Gomes de Freitas, Thiago Gomes Nepomuceno da Silva, Jerffeson Teixeira de Souza, Mariela Inês Cortés

Grupo de Otimização em Engenharia de Software (GOES.UECE)
Universidade Estadual do Ceará (UECE) – Fortaleza, CE – Brasil

marcia.abrasil@larces.uece.br, fabriciogf@uece.br, thi.nepo@gmail.com,
{jeff, mariela}@larces.uece.br

Resumo. *Processos de desenvolvimento rápido de software são projetados para criar software útil rapidamente. Nesse contexto, o desenvolvimento iterativo e incremental se destaca como uma abordagem caracterizada por pequenos e freqüentes releases do sistema. O planejamento de releases visa à alocação de requisitos em releases de modo que todas as restrições sejam respeitadas. Um planejamento eficiente representa uma importante e complexa atividade. O presente trabalho tem por finalidade apresentar uma abordagem baseada em otimização multiobjetiva para o problema tendo em vista a satisfação de clientes e o gerenciamento de riscos de forma adequada. Uma avaliação realizada evidencia a viabilidade da formulação proposta.*

1. Introdução

O software é parte de quase todas as operações de negócio; assim, é imprescindível que o seu desenvolvimento seja realizado rapidamente para que seja capaz de aproveitar novas oportunidades e responder às pressões competitivas [Sommerville 2007].

Nesse sentido, o desenvolvimento incremental possibilita aos clientes receberem funcionalidades do software mais cedo sendo, normalmente, um processo iterativo no qual a especificação, o projeto, a implementação e o teste são intercalados. Assim, o software não é desenvolvido e disponibilizado integralmente, mas através de pequenos e freqüentes *releases* (ou incrementos). Uma filosofia incremental também é seguida pelas denominadas “metodologias ágeis”. Esse modelo de desenvolvimento apresenta uma série de vantagens: *feedback* rápido dos clientes e demais *stakeholders*, tratamento antecipado de riscos, maior execução de testes, dentre outras.

Como parte integrante deste modelo de desenvolvimento está o planejamento de *releases*, onde estágios de entrega são definidos; cada um fornecendo um subconjunto das funcionalidades do sistema. O planejamento de *releases* envolve todos os aspectos e decisões relacionados à seleção, priorização e alocação de requisitos em seqüências de *releases* do software. Assim, realizar um planejamento adequado é uma atividade não somente importante como intrinsecamente complexa, que normalmente é realizada informalmente através do conhecimento e experiências anteriores de gerentes.

O Planejamento de *Releases* de Software, ou *Software Release Planning*, é o tema discutido neste trabalho, o qual apresenta uma abordagem baseada em otimização para auxiliar gerentes de projeto em um planejamento eficaz. O trabalho está inserido no

contexto de uma área de pesquisa recente denominada SBSE – *Search-Based Software Engineering* [Harman and Jones 2001], que tem por finalidade a aplicação técnicas de otimização na resolução de problemas complexos da Engenharia de Software.

Assim, a principal contribuição desta pesquisa é apresentar uma nova formulação multiobjetiva de otimização matemática para o problema do planejamento de *releases* considerando importantes fatores, como: satisfação de clientes, priorização, valor de negócio, riscos envolvidos, recursos disponíveis, precedência entre requisitos.

Além desta seção introdutória, o trabalho é estruturado conforme segue: Seção 2, que descreve brevemente os trabalhos relacionados; Seção 3, a qual apresenta a formulação do problema e discute os aspectos relacionados à sua definição; Seção 4, que reporta os resultados de uma avaliação inicial realizada para demonstrar a viabilidade da abordagem; e, Seção 5, onde são feitas as considerações finais do artigo.

2. Trabalhos Relacionados

Na área de SBSE, a seleção de requisitos foi tratada inicialmente em [Bagnall et al. 2001]. O trabalho, denominado “The Next Release Problem”, consiste em selecionar um conjunto de clientes cujos requisitos serão atendidos no próximo *release*. Com uma formulação mono-objetiva, realiza o planejamento apenas para o próximo *release* e não leva em consideração a importância que os requisitos têm para cada cliente. O problema é tratado de forma multiobjetiva em [Zhang et al. 2007], onde o custo do projeto e a satisfação de clientes são os objetivos a serem otimizados. Embora considere a importância dos requisitos para os clientes, a formulação não contempla a precedência entre requisitos, algo que dificilmente ocorre em projetos reais.

[Greer and Ruhe 2004] e [Ruhe and Saliu 2005] apresentam uma abordagem evolutiva e iterativa, baseada em algoritmos genéticos, para o planejamento de *releases*. A abordagem considera a prioridade atribuída aos requisitos pelos *stakeholders*, o valor de negócios agregado e os recursos disponíveis em cada *release*, bem como precedência entre requisitos. Embora o replanejamento seja contemplado, esses trabalhos não tratam do risco inerente a cada requisito e somente no segundo a quantidade de *releases* é fixa. Uma abordagem multiobjetiva mais completa é apresentada em [Colares et al. 2009], onde um conjunto de requisitos é selecionado para ser implementado em um número fixo de *releases*, cujos objetivos são: maximizar a satisfação de clientes e minimizar os riscos do projeto (através da implementação antecipada dos requisitos de maior risco); respeitando os recursos disponíveis e a precedência entre os requisitos. A satisfação dos clientes é alcançada apenas pela implementação antecipada dos requisitos prioritários.

3. Definição do Problema

Esta seção descreve a formulação matemática proposta para o problema e apresenta, previamente, alguns aspectos relacionados à sua definição.

3.1. Fatores considerados

A abordagem proposta tem como objetivos: maximizar a satisfação dos clientes, implementando cada requisito no *release* mais desejado pelo cliente e priorizando a implementação dos mais importantes; minimizar os riscos do projeto, implementando mais cedo os requisitos de maior risco. As restrições consideradas são: os limites de tempo e orçamento disponíveis em cada *release*; e as precedências existentes entre os

requisitos. As escalas numéricas aqui utilizadas são apenas uma forma de avaliar valores de risco, importância, prioridade, custo etc. e permitir a modelagem matemática do problema. Outras escalas podem ser utilizadas em diferentes contextos, sem perda de generalidade. Assim, os conjuntos e relacionamentos a seguir são definidos.

* **Requisitos**: $R = \{r_i \mid i = 1, 2, \dots, N\}$ é o conjunto de funcionalidades a serem implementadas e alocadas em *releases*. A implementação de cada requisito r_i demanda certo custo e tempo denotados por $cost_i$ e $time_i$, respectivamente. Cada requisito r_i possui também um risco $risk_i$ associado, que varia de 1 (menor risco) a 5 (maior risco). Um relacionamento de precedência entre requisitos pode ser identificado quando a implementação de certo requisito depende da implementação prévia de outro requisito.

* **Clientes**: $C = \{c_j \mid j = 1, 2, \dots, M\}$ é o conjunto de clientes cujos requisitos devem ser atendidos. Cada cliente c_j possui uma importância para a organização definida por w_j , que varia de 1 (menor importância) a 10 (maior importância).

* **Releases**: $S = \{s_k \mid k = 1, 2, \dots, P\}$ é o conjunto de *releases* a serem desenvolvidos. Cada *release* s_k tem um orçamento disponível ($budgetRelease_k$), assim como um tempo máximo para entrega ($timeRelease_k$) que não devem ser ultrapassados.

* **Clientes versus Requisitos** – Diferentes clientes possuem diferentes interesses com a implementação de cada requisito. Assim, $importance(j, i)$ quantifica a percepção de importância ou valor de negócio que cada cliente c_j atribui a cada requisito r_i , associando um valor que varia de 0 (nenhuma importância; o cliente não tem interesse no requisito) à 9 (alta importância).

* **Requisitos versus Clientes versus Releases** – Os clientes também têm valores de preferência ou urgência de implementação para cada requisito em cada *release*. Assim, $urgency(j, i, k)$ representa o grau de preferência definida por um cliente c_j para implementação de um requisito r_i em cada *release* s_k . Exemplo: supondo que, para a implementação de um requisito r_1 no primeiro *release* (s_1), o cliente c_1 tem grau de preferência 5; para que seja no segundo *release* (s_2), preferência 10; e, para que seja no terceiro *release* (s_3), preferência 1. Assim, o cliente c_1 prefere mais fortemente que o requisito r_1 seja atendido no segundo *release* (s_2). Os valores de preferência variam entre 0 e 10.

3.2. Formulação matemática do problema

A partir dos fatores apresentados, a seguinte formulação é definida para o problema do planejamento de *releases* de software:

$$(1) \ Max f(y) = \sum_{i=1}^N score_i \cdot y_i$$

$$(2) \ Max g(x, z) = \sum_{j=1}^M \sum_{i=1}^N (profit(j, i, x_i) - penalty(j, i, x_i, z_{j,i}))$$

$$(3) \ Min h(x) = \sum_{i=1}^N risk_i \cdot x_i$$

Sujeito a:

$$(4) \ \sum_{i=1}^N cost_i \cdot v_{i,k} \leq budgetRelease_k, \forall k \in \{1, 2, \dots, P\}$$

$$(5) \ \sum_{i=1}^N time_i \cdot v_{i,k} \leq timeRelease_k, \forall k \in \{1, 2, \dots, P\}$$

$$(6) \ x_a \leq x_b, \text{ sempre que } r_a \text{ preceder } r_b$$

A variável x_i indica o *release* de implementação do requisito r_i , sendo um valor em $\{0, 1, 2, \dots, P\}$, para $i = 1, 2, \dots, N$. A variável y_i indica se o requisito r_i será implementado em algum *release* ($y_i = 1$) ou se não será implementado de forma alguma ($y_i = 0$), para $i = 1, 2, \dots, N$. Se $x_i > 0$, $y_i = 1$; se $x_i = 0$, $y_i = 0$. A variável $v_{i,k}$ indica se o requisito r_i foi implementado no *release* s_k ($v_{i,k} = 1$) ou não ($v_{i,k} = 0$). Se $x_i > 0$, $v_{i,k} = 1$; se $x_i = 0$, $v_{i,k} = 0$.

Função (1): expressa a satisfação dos clientes pela implementação dos requisitos considerados mais importantes, onde: $score_i = \sum_{j=1}^M w_j \cdot importance(j, i)$ representa, de forma ponderada, o valor de negócios agregado pela implementação do requisito r_i .

Função (2): expressa, de forma ponderada, a satisfação dos clientes pela implementação de cada requisito no *release* mais desejado, aplicando uma penalidade quando esta condição não for atendida, onde:

$profit(j, i, x_i) = urgency(j, i, x_i) \cdot w_j$: representa o benefício pela implementação do requisito r_i no *release* s_k conforme grau de preferência do cliente c_j .

$$penalty(j, i, x_i, z_{j,i}) = \begin{cases} w_j \cdot importance(j, i) \cdot \left(\frac{x_i - z_{j,i}}{P} \right) \cdot d & \text{se } x_i > z_{j,i} \\ 0 & \text{caso contrário} \end{cases}$$

: representa a penalidade pela implementação do requisito r_i após o *release* mais desejado pelo cliente c_j . Em caso contrário, nenhuma penalidade é aplicada.

$d = urgency(j, i, z_{j,i}) - urgency(j, i, x_i)$: representa a diferença entre o grau de preferência de implementação do requisito r_i no *release* mais desejado pelo cliente c_j e o grau de preferência de implementação do requisito r_i no *release* em que foi implementado.

$z_{j,i} = \{max(k) \text{ em } \{1, 2, \dots, P\} \mid max(urgency(j, i, k))\}$: define o incremento com maior grau de preferência por um cliente c_j para implementação de um requisito r_i . Se mais de um *release* tiver o mesmo grau de preferência máximo, é selecionado o de maior número.

Função (3): expressa o gerenciamento de riscos do projeto como um todo, indicando que os requisitos com maior risco devem ser implementados mais cedo.

As restrições do problema são expressas em 4, 5 e 6. Desta forma, (4) e (5) são as restrições que limitam o custo e o tempo de implementação de requisitos em cada *release* ao orçamento disponível e à duração do *release*, respectivamente; e (6) é a restrição que expressa relacionamentos de precedência entre requisitos. Se um requisito r_a precede um requisito r_b , então r_a deve ser implementado em um *release* anterior ao *release* de implementação de r_b ; no máximo, ambos devem ser implementados no mesmo *release* ($x_a \leq x_b$).

4. Avaliação

Uma avaliação preliminar foi conduzida para demonstrar a validade da formulação proposta. Por limitações de espaço, apenas os resultados principais são apresentados.

4.1. Abordagens Metaheurísticas e Busca Randômica

Para resolução do problema formulado na seção anterior, as metaheurísticas NSGA-II [Deb et al. 2002] e MOCell [Nebro et al. 2009] foram aplicadas. NSGA-II é um algoritmo multiobjetivo baseado em algoritmos genéticos que implementa os conceitos de dominância e de elitismo, classificando a população total em diferentes categorias de qualidade (*fronts*). MOCell é um modelo celular de algoritmo genético para otimização multiobjetiva que utiliza um arquivo externo para armazenar as soluções não dominadas encontradas durante o processo de evolução. A implementação da abordagem proposta através desses algoritmos foi realizada por meio do *framework* jMetal [Durillo et al. 2006], que disponibiliza diversas metaheurísticas multiobjetivas.

Um algoritmo de busca randômica também foi implementado como um referencial e permitir que os resultados obtidos pelas metaheurísticas pudessem ser comparados e validados. Na busca randômica, como o próprio nome sugere, o problema é resolvido obtendo-se uma solução gerada de forma aleatória.

4.2. Descrição das instâncias e configuração dos parâmetros

Três instâncias de problemas foram geradas e utilizadas para verificar a validade da abordagem em contextos distintos. Assim, a instância A foi definida com 5 *releases*, 30 requisitos e 3 clientes; a instância B com 10 *releases*, 50 requisitos e 5 clientes; e a instância C com 20 *releases*, 80 requisitos e 8 clientes. Os valores de $cost_i$, $time_i$ foram gerados aleatoriamente em um intervalo de 10 a 20. Da mesma forma, os valores para risco $risk_i$, w_j , $importance(j, i)$, $urgency(j, i, k)$ também foram gerados randomicamente, obedecendo às escalas definidas anteriormente. Em cada *release*, para $budgetRelease_k$ e $timeRelease_k$ foi considerado um percentual de 70% do orçamento e tempo necessários para desenvolver todos os requisitos (e dividido pela quantidade de *releases*). Para simular a precedência entre os requisitos, foi considerado que 10% dos requisitos apresentavam relacionamentos de precedência.

Cada instância foi resolvida através das metaheurísticas NSGA-II e MOCell. Os parâmetros utilizados por cada uma foram configurados a partir da execução de testes preliminares. Assim, para o NSGA-II, foram utilizados: tamanho da população de 1000 e número máximo de avaliações (ou critério de parada) de 1000000, resultando em 1000 gerações. Para o MOCell, a configuração foi: tamanho da população de 1024 e número de avaliações de 1024000, resultando em 1000 gerações. As taxas de cruzamento e mutação foram as mesmas nos dois algoritmos e iguais a 90% e 5%, respectivamente.

4.3. Apresentação e análise dos resultados

Os gráficos das Figuras 1, 2 e 3 apresentam os resultados obtidos em uma execução dos três algoritmos para cada uma das instâncias.

Os resultados comparam o desempenho dos algoritmos através de conceitos da frente de *Pareto*, onde uma frente gerada por cada metaheurística em cada instância foi utilizada na construção dos gráficos. Como este trabalho apresenta três objetivos a serem otimizados, um gráfico tridimensional deveria ser utilizado para apresentação das soluções. No entanto, a utilização de tal gráfico dificultaria a leitura e interpretação dos resultados. Assim, para cada instância, os resultados são apresentados através de três gráficos de forma bidimensional, onde cada gráfico indica os resultados relativos a dois objetivos, representados em cada eixo.

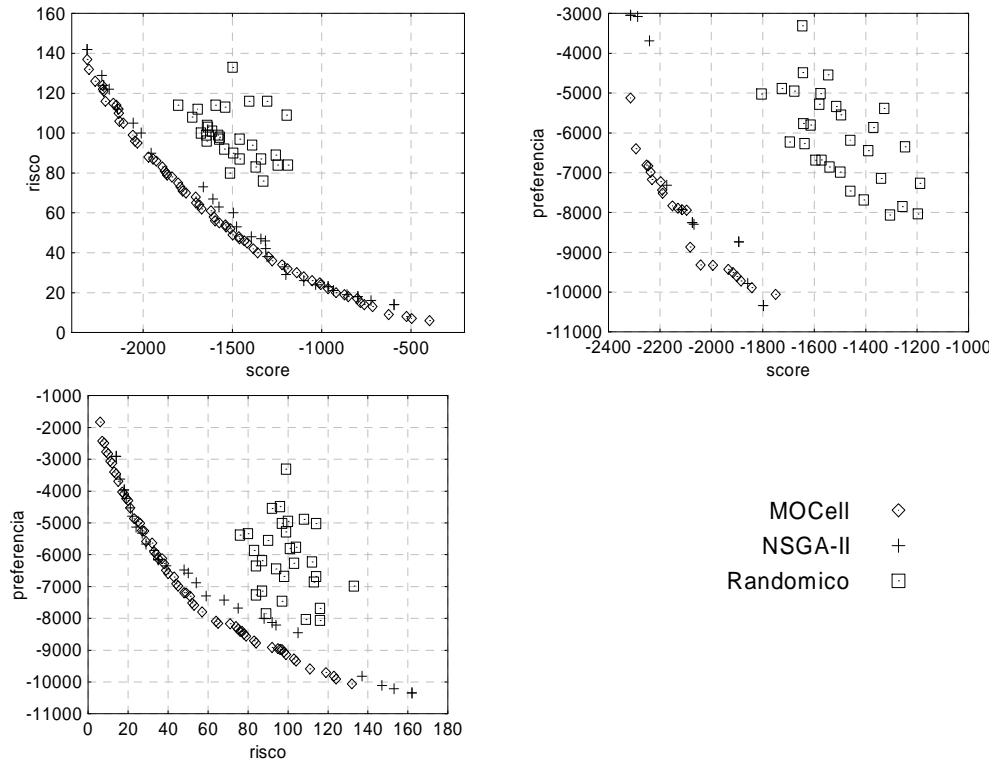


Figura 1. Resultados para a instância A (5 releases, 30 requisitos, 3 clientes).

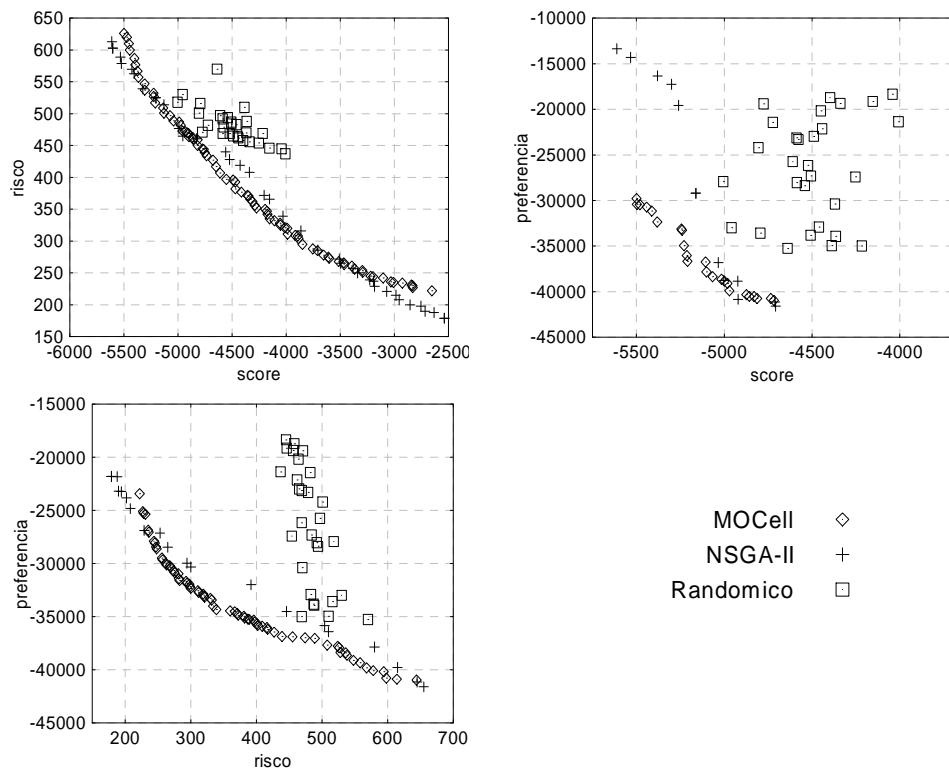


Figura 2. Resultados para a instância B (10 releases, 50 requisitos, 5 clientes).

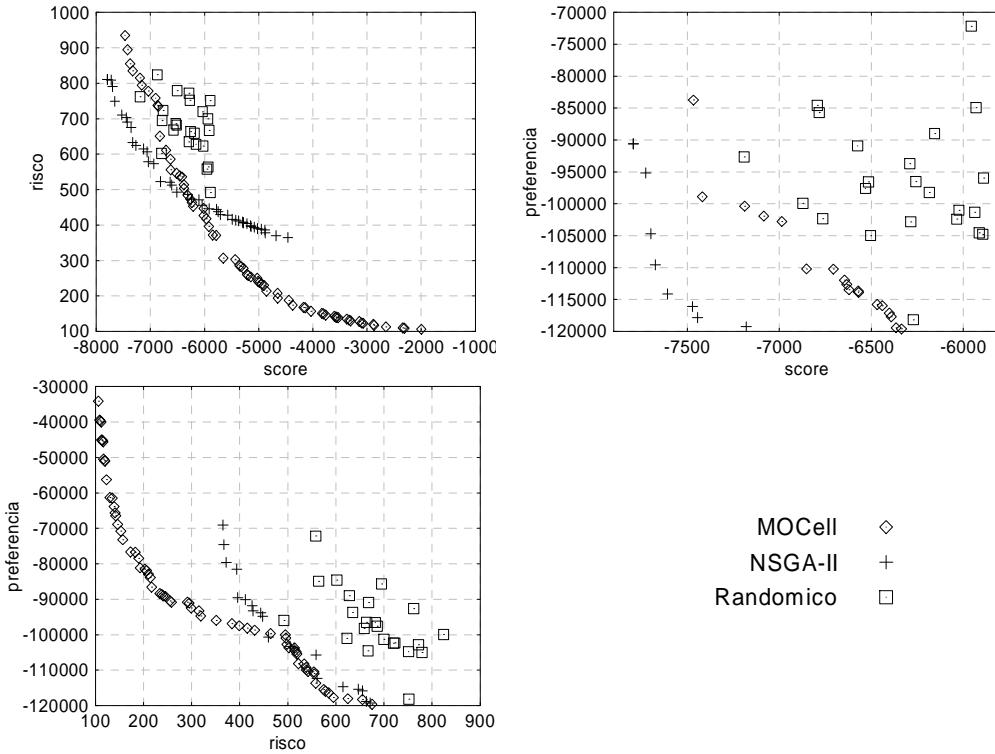


Figura 3. Resultados para a instância C (20 releases, 80 requisitos, 8 clientes).

Os resultados indicam que a estratégia randômica foi superada pelas duas metaheurísticas em todas as instâncias. De fato, NSGA-II e MOCell geraram soluções melhores considerando os três objetivos definidos, o que reforça a utilização das abordagens metaheurísticas. Como a busca randômica não considera qualquer objetivo, as soluções geradas são de baixa qualidade.

Observando-se os gráficos da figura 1, percebe-se que o comportamento das metaheurísticas foi semelhante para a instância A, justificável pelo fato de esta ser a menor das instâncias, com um espaço de busca mais reduzido, possibilitando a definição de frentes semelhantes. Para a instância B, percebe-se no primeiro gráfico (*score versus risco*) que, nos extremos da frente de *Pareto*, as melhores soluções são as apresentadas pelo NSGA-II. No entanto, nas outras regiões da frente, as melhores soluções foram as obtidas pelo MOCell. Diante deste fato, pode-se sugerir que, para este tipo de problema, as duas metaheurísticas devem ser utilizadas de forma complementar, não existindo uma melhor que a outra, pois cada uma se comportou melhor em uma determinada região da frente. Para a instância C, pode-se visualizar no primeiro gráfico que há um cruzamento entre as duas frentes. É importante notar também que cada metaheurística conseguiu soluções em diferentes regiões do gráfico, diferentemente da instância A que apresentou comportamentos similares. Este fato reforça a importância da utilização conjunta de diferentes metaheurísticas.

5. Considerações Finais

Devido à quantidade de aspectos envolvidos, o planejamento de *releases* de software apresenta uma complexidade tal que torna o problema adequado para resolução por meio da aplicação de técnicas automatizadas na tentativa de se obter bons resultados.

Assim, esse trabalho teve por objetivo apresentar uma abordagem consistente para o problema considerando diversos aspectos que influenciam a tomada de decisões em projetos reais. É o primeiro a propor uma formulação com três objetivos, apresentando duas funções de satisfação de clientes que visam implementar os requisitos mais importantes e de acordo com os *releases* de preferência dos clientes; e uma função de gerenciamento de riscos. A abordagem também busca respeitar os recursos disponíveis e o relacionamento entre requisitos, através das restrições impostas. Os resultados preliminares obtidos nos experimentos demonstram a viabilidade da aplicação da formulação proposta.

Os próximos passos em direção à evolução da pesquisa são: realizar estudos com mais instâncias avaliando o comportamento das metaheurísticas de modo mais aprofundado; comparar o desempenho das metaheurísticas através da análise de tempo de execução, *hypervolume* e *spread*; utilizar outras metaheurísticas multiobjetivas; conduzir um estudo com dados reais, no contexto de projetos que aplicam metodologias ágeis; comparar as soluções geradas pela formulação com as de profissionais da área.

Referências

- Bagnall, A. J., Rayward-Smith, V. J., Whittle, I. M. (2001) "The Next Release Problem", *Information and Software Technology* 43, 14, 883-890, 2001.
- Colares, F., Souza, J., Carmo, R., Pádua, C., Mateus, G. R. (2009) "A New Approach to the Software Release Problem", In: XXIII Simpósio Brasileiro de Engenharia de Software, 2009.
- Deb, K., Pratap, A., Agarwal, S., Meyarivan, T. (2002) "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II", *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.
- Durillo, J. J., Nebro, A. J., Luna, F., Dorronsoro, B., Alba, E. (2006) "jMetal: a Java Framework for Developing Multi-Objective Optimization Metaheuristics", Technical Report: ITI 2006-10, University of Málaga, 2006.
- Greer, D., Ruhe, G. (2004) "Software Release Planning: An Evolutionary and Iterative Approach", *Information and Software Technology*, 46, 243-253, 2004.
- Harman, M., Jones, B. F. (2001) "Search-Based Software Engineering", *Information and Software Technology*, 43(14):833–839, 2001.
- Nebro, A. J., Durillo, J. J., Luna, F., Dorronsoro, B., Alba, E. (2009) "MOCell: A Cellular Genetic Algorithm for Multiobjective Optimization", *International Journal of Intelligent Systems*, vol. 24, pp. 726-746, Julho 2009.
- Ruhe, G., Saliu, M. (2005) "The Art and Science of Software Release Planning", *Software, IEEE*, 22(6):47–53, 2005.
- Sommerville, I. (2007) "Engenharia de Software", 8.ed., São Paulo: Pearson Addison-Wesley, 2007.
- Zhang, Y., Harman, M., Mansouri, S. A. (2007), "The Multi-Objective Next Release Problem", In Proceedings of the 9th annual conference on Genetic and evolutionary computation, pages 1129–1137. ACM, 2007.

Explorando Técnicas de Agrupamento de Dados na Indexação de Repositórios de Componentes de Software

Marcos Paulo Paixão¹, Tales Brito², Leila Silva¹, Gledson Elias²

¹Departamento de Computação
Universidade Federal de Sergipe (UFS), Aracaju, SE – Brasil

²Departmento de Informática
Universidade Federal da Paraíba (UFPB), João Pessoa, PB – Brasil

marcosppsp@dcomp.ufs.br, tales@compose.ufpb.br,
leila@ufs.br, gledson@di.ufpb.br

Abstract. *In order to promote software reuse, component repositories require search mechanisms based on component description models, capable of representing their syntactic and semantic features. As a solution, in general, semi-structured data models are adopted, implying in challenges related to the design of indexing techniques, which must be efficient in terms of storage space, processing time and precision level. In such a context, this paper proposes an approach for the reduction of storage space and processing time by applying data clustering techniques when indexing component repositories. Based on an illustrative repository, outcomes indicate a significant optimization of storage space.*

Resumo. *Para promover o reuso de software, repositórios de componentes demandam mecanismos de busca baseados em modelos de descrição de componentes, capazes de representar características sintáticas e semânticas. Para tal, em geral, modelos de dados semi-estruturados são adotados, implicando em desafios no projeto de técnicas de indexação eficientes em termos de espaço de armazenamento, tempo de processamento e precisão das consultas. Neste contexto, este artigo propõe uma abordagem para redução do espaço de armazenamento e tempo de processamento, através da aplicação de técnicas de agrupamento de dados na indexação de grandes repositórios de componentes. Baseado em um repositório exemplo, os resultados indicam uma otimização significativa do espaço de armazenamento.*

1. Introdução

Repositórios de componentes têm o potencial de facilitar o reuso de software ao possibilitar o compartilhamento de componentes entre diferentes desenvolvedores. Entretanto, reuso de componentes de software é freqüentemente uma tarefa difícil, particularmente quando a busca e a seleção devem ser conduzidas sobre grandes coleções de componentes. Portanto, em sistemas de repositório, é importante o desenvolvimento de mecanismos de busca que possam auxiliar na busca, seleção e recuperação de componentes de software. Como consequência, vários sistemas de busca têm sido propostos na academia e indústria. Alguns desses extraem as características dos componentes diretamente do código fonte ou documentação, não expressando adequadamente e precisamente as características sintáticas e semânticas dos mesmos.

Conforme Orso *et al.* (2000), sistemas de repositório não devem apenas armazenar componentes, mas também metadados que descrevem suas características. Tais metadados provêem as informações usadas por mecanismos de busca para indexar

e classificar os componentes. Nesta direção, como reforçado por Vitharana (2003), modelos de descrição de componentes podem adotar conceitos de alto nível para descrever metadados dos componentes, tornando possível expressar suas características sintáticas e semânticas, e, assim, facilitar a busca e seleção dos mesmos.

Na prática, as iniciativas de definição de modelos de descrição de componentes têm adotado abordagens baseadas em dados semi-estruturados, mais especificamente XML, permitindo que as relações estruturais dos elementos XML possam agregar semântica aos valores textuais. Como exemplos de iniciativas propostas pela indústria e academia, podemos citar o RAS [OMG 2005] e a sua extensão X-ARM [Elias 2006], que é o contexto deste trabalho.

No entanto, técnicas de indexação baseadas em termos textuais não são eficientes para dados semi-estruturados, pois não são capazes de indexar as relações estruturais entre os termos, comprometendo a precisão das consultas com resultados falso-positivos. Logo, a adoção de dados semi-estruturados implica em desafios relacionados ao projeto de técnicas de indexação que mantenha eficiência nos requisitos de espaço de armazenamento, tempo de processamento e nível de precisão das consultas, que podem conter restrições textuais e estruturais de acordo com as necessidades de informação dos usuários.

Diversos trabalhos têm sido propostos para lidar com tais problemas. Apesar das relevantes contribuições, as técnicas existentes ainda comprometem o espaço de armazenamento e o tempo de processamento [Meier 2002], bem como a precisão das consultas [Goldman e Widom 1997]. Uma proposta que se destaca neste cenário é apresentada em [Brito *et al.* 2010], que define uma técnica de indexação de dados semi-estruturados precisa e eficiente em termos de tempo de processamento das consultas, mas que, embora em relação a outras propostas seja também eficiente quanto ao espaço de armazenamento, ainda é limitada pois os arquivos de índices ainda são bem maiores que a base de entrada. Logo, considerando grandes repositórios de componentes, técnicas de indexação que minimizem o espaço de armazenamento, mas que não impactam de forma demasiada no tempo de processamento e na precisão das consultas, ainda representa um desafio, constituindo-se assim em um problema em aberto.

Neste contexto, este artigo propõe uma alternativa para otimização do espaço de armazenamento e tempo de processamento de consultas, através da aplicação de técnicas de agrupamento de dados [Jain e Dubes 1988]. Propõe-se a construção de um índice de elementos representativos de artefatos do repositório real. Cada elemento representativo referencia os componentes pertencentes ao grupo que ele representa. Uma métrica de similaridade é aqui proposta para indicar quais componentes pertencem a um mesmo grupo. Quanto maior a similaridade entre os componentes do repositório, menos grupos são formados, e, por conseguinte, menos elementos representativos devem ser indexados, permitindo economia de armazenamento e agilidade de consulta.

Como o problema de agrupamento de dados é NP-Difícil, inúmeras heurísticas já foram propostas. O trabalho de Xu e Wunsch (2005) é um interessante artigo de revisão na área. Já o trabalho de Feng (2007) mostra que algoritmos de agrupamento (em particular, os algoritmos hierárquico e *K*-Means [Jain e Dubes 1999]) são equivalentes a um algoritmo de otimização de uma função global. Neste trabalho, propomos uma heurística de agrupamento que compreende dois estágios e é baseada no algoritmo hierárquico clássico e no *K*-Means. Para a validação da estratégia adotada

uma base aleatória de 27 mil artefatos foi gerada e os resultados indicam que há uma otimização significativa do espaço de armazenamento dos componentes do repositório.

Embora os autores desconheçam o uso de técnicas de agrupamento no contexto de indexação de repositórios de componentes, vários trabalhos já aplicaram estas técnicas na Engenharia de Software. Por exemplo, Wu *et al.* (2005) realizam um estudo comparativo de várias abordagens de agrupamento propostas no contexto de evolução de software. Já Li *et al.* (2007) utilizam agrupamentos para realizar o encapsulamento de requisitos. Chiricota *et al.* (2003) investigam o domínio de engenharia reversa, em particular, usam técnicas de agrupamento para recuperar a estrutura de sistemas.

O restante deste artigo está estruturado da seguinte forma. A Seção 2 apresenta o modelo de descrição X-ARM, adotado nesta proposta, identificando os principais artefatos e suas relações. A Seção 3 apresenta a técnica de agrupamento proposta para otimizar o espaço de armazenamento e o tempo de processamento de consultas. Por fim, a Seção 4 apresenta algumas considerações finais e os trabalhos futuros.

2. X-ARM

Para expressar as características sintáticas e semânticas dos componentes, Frakes (2005) sugere a adoção de modelos de descrição de componentes, provendo assim um conjunto de informações que possibilitem a indexação e classificação dos artefatos por sistemas de busca. Nesta direção, este trabalho explora o modelo de descrição X-ARM, que adota um modelo de dados semi-estruturado baseado em XML, expressando não apenas informações sintáticas, mas também propriedades semânticas [Elias 2006]. Além disso, o X-ARM permite descrever os diversos tipos de artefatos produzidos em processos de DBC (Desenvolvimento baseado em componentes), provendo a semântica necessária para representar seus relacionamentos.

Conforme ilustrado na Figura 1, o X-ARM permite descrever especificações de interfaces e componentes, bem como implementações de componentes. As especificações de interfaces e componentes podem ser descritas de forma independente ou dependente de modelo de componentes, considerando assim as características particulares do modelo adotado, tais como CCM, JavaBeans, EJB e Web Services.

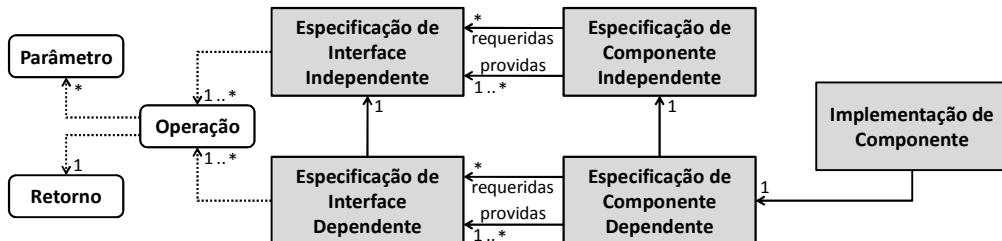


Figura 1. Relacionamentos entre artefatos

No X-ARM, as especificações de interfaces, dependentes e independentes, são descritas na forma de um conjunto de operações. Cada operação possui um nome, um conjunto de parâmetros de entrada ou saída, e um valor de retorno. As especificações de componentes, dependentes e independentes, podem referenciar um conjunto de especificações de interfaces providas e requeridas, dependentes e independentes, respectivamente. As versões dependentes de interfaces e componentes devem estar em conformidade com suas respectivas versões independentes. Além disso, para cada

versão independente, diferentes versões dependentes podem ser especificadas para diferentes modelos de componentes. As especificações de interfaces e componentes dependentes referenciam suas respectivas especificações independentes. Por sua vez, a implementação de componente referencia sua especificação de componente dependente.

Como exemplo de descrição de um artefato no X-ARM considere a Figura 2, que descreve um fragmento de uma especificação de componente dependente, onde as linhas foram enumeradas e alguns detalhes foram suprimidos por razões didáticas. A Linha 2 introduz o cabeçalho do artefato, no qual se encontra o identificador (id) do mesmo. Já as linhas 12 a 14 introduzem a especificação de componente independente referenciada pela especificação de componente dependente em questão. As linhas 15 a 24 introduzem as interfaces independentes que são providas pela especificação de componente dependente corrente.

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <asset last-change="2009-09-20" owner="COMPOSE" date="2009-09-20" language="pt"
   id="compose.especificacaoDependente-C-2.0-beta" name="especificacaoDependenteC"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=
   "http://www.compose.ufpb.br/X-ARM Interaction Profile - Dependent.xsd">
...
12.<model-dependency>
13.<related-asset name="especificacaoIndependenteD" required="true" relationship-type=
   "independentComponentSpec" id="compose.especificacaoIndependente-D-3.0-stable" />
14.</model-dependency>
15.<component-specification>
16.<interface>
17.<provided>
18.<related-asset name="interfaceDependenteD" id="compose.interfaceDependente-D-3.0-
   stable" relationship-type="dependentInterface" required="true"/>
19.</provided>
20.<provided>
21.<related-asset name="interfaceDependenteB" id="compose.interfaceDependente-B-3.0-
   stable" relationship-type="dependentInterface" required="true"/>
22.</provided>
23.</interface>
24.</component-specification>
25.</asset>
```

Figura 2. Descrição em xml de um artefato do tipo especificação de componente dependente.

Como as descrições X-ARM são documentos XML, a indexação baseada em dados semi-estruturados de descrições X-ARM torna possível selecionar os artefatos que satisfazem critérios informados através de restrições textuais e estruturais e de acordo com as necessidades de informação do usuário.

3. Otimizando a Busca em Repositórios de Componentes

A indexação de repositórios de dados semi-estruturados é um problema relevante [Meier 2002][Goldman e Widom 1997][Brito *et al.* 2010]. Um dos maiores desafios é prover um mecanismo de indexação que economize espaço de armazenamento, mas sem impactar demasiadamente no tempo de processamento e nível de precisão das consultas.

Neste trabalho propomos uma solução para otimização do espaço de armazenamento do índice através da construção de um índice composto de elementos representativos de um conjunto de componentes reais do repositório. A busca consideraria, então, o conjunto (reduzido) de elementos representativos, os quais referenciam os componentes reais. Para a identificação de grupos de componentes

similares, e, por conseguinte, para a construção de elementos representativos destes grupos, propõe-se a aplicação de técnicas de agrupamento de dados (*clustering*).

As técnicas de agrupamento [Jain e Dubes 1988] consistem em três etapas básicas: (i) extração de características, que expressam o comportamento dos elementos a serem agrupados; (ii) definição da métrica de similaridade entre elementos analisados; e (iii) algoritmo de agrupamento adotado. A fase de extração de característica consiste em estabelecer quais informações são relevantes para expressar o elemento analisado e como são quantificadas. Estas informações compõem um vetor de atributos e desta forma um elemento pode ser representado como um ponto no espaço multidimensional. A métrica de similaridade expressa, em termos quantitativos, a similaridade entre elementos. Em geral uma fórmula é estabelecida para este fim, sendo a distância Euclidiana [Jain e Dubes 1988] entre dois pontos (elementos) uma das métricas mais comumente adotadas. Por fim, o algoritmo de agrupamento de dados é uma heurística que objetiva gerar grupos de tal forma que todos os elementos pertencentes a um mesmo grupo sejam similares entre si, de acordo com o critério de similaridade estabelecido.

A solução aqui proposta aplica a técnica de agrupamento considerando, em separado, cada um dos cinco tipos de artefatos existentes no repositório, a saber, especificações de componentes dependentes e independentes, especificações de interfaces dependentes e independentes, e implementações de componentes. Assim, um vetor de atributos diferente é construído para cada tipo de artefato. No caso das implementações de componentes, a característica relevante é a especificação de componente dependente referenciada. Para as especificações de componentes dependentes são consideradas as especificações de componentes independentes referenciadas, bem como as suas especificações de interfaces providas dependentes. Em relação às especificações de componentes independentes são adotadas as especificações de interfaces providas independentes. Já para as especificações de interfaces dependentes são consideradas as especificações de interfaces independentes referenciadas, assim como as suas operações. Por fim, para especificações de interfaces independentes são consideradas as suas operações. Para o artefato ilustrado na Figura 2, a Figura 3 apresenta o correspondente vetor de atributos. Observe que como o artefato da Figura 2 corresponde a uma especificação de componente dependente, o vetor é composto da especificação de componente independente que este referencia (Linha 13) e das interfaces de componentes dependentes providas (linhas 15 a 24).

Id	Especificação de Componente Independente	<i>Interfaces de componente dependentes</i>
<i>compose.especificacaoDependente-C-2.0-beta</i>	<i>compose.especificacaoIndependente-D-3.0-stable</i>	compose.interfaceDependente-D-3.0-stable compose.interfaceDependente-B-3.0-stable

Figura 3. Vetor de atributo do artefato apresentado na Figura 2.

A métrica de similaridade é definida com base no vetor de atributos de cada tipo de artefato. Por razões de concisão, a métrica não será totalmente descrita (para detalhes veja [Paixão *et al.* 2010]). Para ilustrar a composição da métrica, considere o caso da determinação da similaridade entre duas especificações de componentes dependentes. Neste caso, se duas especificações de componentes dependentes possuem a mesma referência para uma especificação de componente independente é atribuído um valor de

similaridade entre ambas, denominado *distância*. Além disso, computam-se a interseção e a união das especificações de interfaces providas dependentes. Um peso é atribuído à razão *interseção/união*, de tal forma que especificações de componentes dependentes são consideradas mais similares se essa razão se aproxima de um.

Como exemplo do cálculo da métrica de similaridade considere uma especificação dependente de componente cujo vetor de atributos é dado pela Figura 4. A similaridade entre o artefato da Figura 2 e o que foi utilizado para construção do vetor de atributos da Figura 4 é estabelecida utilizando o vetor de atributos destas. Esta similaridade é expressa por um valor numérico obtido segundo a fórmula

$$d_f = d_i - k - (\text{interseção}/\text{união}) * 100,$$

onde d_i é uma distância inicial padrão ($d_i = 300$); k uma variável que pode assumir o valor 0 se as duas especificações de componentes dependentes referenciarem a mesma especificação de componente independente e 200, caso contrário; *interseção* que expressa a quantidade de interfaces de componentes dependentes comum a ambas especificações de componente dependente e *união* que denota o total das interfaces de componentes dependentes das duas especificações. Para o exemplo em questão, $d_i = 300$, $k = 0$, $\text{interseção} = 1$ e $\text{união} = 3$. Logo, $d_f = 300 - 0 - 33 = 267$.

Id	Especificação de Componente Independente	<i>Interfaces de componente dependentes</i>
<i>compose.especificacaoDependente-C-3.0-beta</i>	<i>compose.especificacaoIndependente-C-3.0-stable</i>	compose.interfaceDependente-A-4.0-mature compose.interfaceDependente-B-3.0-stable

Figura 4. Vetor de atributo de uma especificação de componente dependente

O algoritmo de agrupamento proposto possui dois estágios. No primeiro estágio aplica-se o algoritmo de agrupamento hierárquico clássico [Jain e Dubes 1988] até um limiar no qual a distância entre os grupos é maior do que o parâmetro de corte (*threshold*) estabelecido pelo usuário. Para cada grupo formado, constrói-se um elemento representativo. Nesta etapa os elementos a agrupar são escolhidos aleatoriamente no repositório até o limite de memória primária. Os elementos representativos são armazenados em disco. A Figura 5 exibe os principais passos: (a) artefatos do repositório são aleatoriamente escolhidos; (b) grupos de artefatos similares são formados aplicando-se o algoritmo hierárquico; e (c) elementos representativos dos grupos formados são construídos (denotados por quadrados).

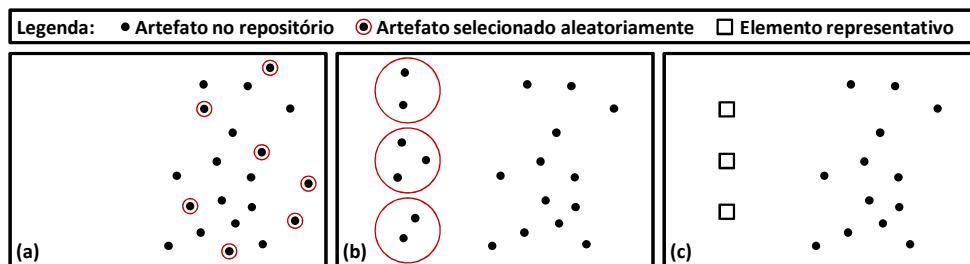


Figura 5. Primeiro Estágio do Algoritmo de Agrupamento

No segundo estágio aplica-se um algoritmo baseado no *K-Means* [Jain e Dubes 1988]. De maneira geral, os elementos representativos são considerados “centróides”,

mas diferentemente do *K-Means*, centróides não são recalculados. Na realidade, compara-se cada artefato ainda não agrupado na primeira etapa com os elementos representativos. Caso este artefato esteja a uma distância menor que o limiar de corte em relação a um dado elemento representativo, este artefato é agrupado junto com o elemento representativo em questão; caso contrário, o artefato se torna um novo elemento representativo, simbolizando um novo grupo.

Para validar a heurística proposta, foi gerada aleatoriamente uma base de artefatos. A Tabela 1 apresenta os resultados obtidos. Observe que há uma redução da base original quando o algoritmo de agrupamento é aplicado. Por exemplo, para o limiar de corte de 175, a base original de dados reduz a indexação de 27.000 para 2.907 artefatos, correspondendo em uma redução de espaço físico de 18 MB para 10,24 MB. A medida que o limiar de corte diminui, mais elementos representativos são formados decorrentes da formação esperada de mais grupos. No entanto, para um corte de 150, o resultado obtido ainda representa uma significativa otimização de espaço.

Tabela 1. Resultados do Agrupamento do Repositório de Dados

ARTEFATO BASE	Imp. Comp.	Esp. Comp. Dep.	Esp. Comp. Ind.	Esp. Int. Dep.	Esp. Int. Ind.	Total
Base Original	15000	6000	2000	2000	1000	27000
Base Agrupada (corte 150)	328	330	1000	575	861	3094
Base Agrupada (corte 175)	321	325	1000	511	750	2907

4. Considerações Finais

Com base nos resultados preliminares, podemos destacar como evidências importantes do benefício da técnica proposta a redução considerável dos requisitos de armazenamento. Vale ressaltar que, quanto maior o repositório fonte, maior a probabilidade de agrupamento de artefatos, e, portanto, maior o ganho no espaço de armazenamento.

Considerando que o algoritmo proposto por Brito *et al.* (2010) será adotado para indexar a base agrupada, e, tendo em mente que o mesmo já apresenta um excelente desempenho no processamento de consultas, a expectativa é que a redução do tamanho do repositório a ser indexado implique na redução substancial do tamanho dos arquivos de índices, e, por conseguinte, do tempo de processamento das consultas.

Vale ressaltar que este trabalho ainda não investigou o compromisso entre o melhor limiar de corte e o nível de precisão nos resultados de consultas. No entanto, já é esperada uma redução do nível de precisão tendo em vista que o agrupamento introduz algum grau de perda de informações. Neste sentido, a avaliação do impacto do agrupamento de artefatos no tempo de processamento e no nível de precisão das consultas constitui-se em alguns de nossos trabalhos futuros imediatos. Além disso, faz-se necessário uma análise comparativa entre a heurística de agrupamento proposta e outras abordagens existentes na literatura.

5. Agradecimentos

Este trabalho foi apoiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES)ⁱ, financiado pelo CNPq processo 573964/2008-4.

Referências

- Brito, T., Ribeiro, T. e Elias, G. (2010) "Indexing Semi-Structured Data for Efficient Handling of Branching Path Expressions", 2nd International Conference on Advances in Databases, Knowledge, and Data Applications, France, pp. 197-203.
- Chiricota, Y., Jourdan, F. e Melançon, G. (2003) "Software component capture using graph clustering", IEEE International Workshop on Program Comprehension.
- Elias, G., Schuenck, M., Negócio, Y., Dias, J. e Miranda, S. (2006), "X-ARM: An Asset Representation Model for Component Repository", 21st ACM Symposium on Applied Computing (SAC 2006), France, pp. 1690-1694.
- Feng ,A. (2007) "Document Clustering – An Optimization Problem", ACM SIGIR 2007, pp. 819-820.
- Frakes, W. e Kang, K. (2005) "Software Reuse Research: Status and Future", IEEE Transactions on Software Engineering, vol.31, issue 7, July, pp. 529-536.
- Goldman, R. e Widom, J. (1997) "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", 23rd International Conference on Very Large Data Bases (VLDB 1997), Greece, pp. 436-445.
- Jain, A.K. e Dubes, R.C. (1984), Algorithms for Clustering Data. Prentice Hall.
- Li, Z., Rahman, Q.A. e Madhavji, N.H. (2007) "An Approach to Requirements Encapsulation with Clustering", 10th Work. on Requirement Engineering, pp. 92-96.
- Meier, W. (2002) "eXist: An Open Source Native XML Database", NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems.
- OMG (2005). Reusable Asset Specification: OMG Available Specification – v2.2.
- Orso, A., Harrold, M.J. e Rosenblum, D.S. (2000) "Component Metadata for Software Engineering Tasks", 2nd Int. Work. on Engineering Distributed Objects, pp. 126-140.
- Paixão, M.P.S., Viana, T.B., Silva, L., e Elias, G. (2010) "Otimizando o Espaço de Busca em Repositórios de Componentes Distribuídos", Relatório Técnico, Junho. <http://www.compose.ufpb.br/reports/component-repository-cluster.pdf>.
- Vitharana, P., Zahedi, F. e Jain, H. (2003) "Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis", IEEE Trans. on Soft. Eng., vol. 29, issue 7, July, pp. 649-664.
- Wu, J., Hassan, A. E. e Holt, R.C. (2005) "Comparison of Clustering Algorithms in the Context of Software Evolution", 21st Int. Conf. on Soft. Maintenance, pp. 525-535.
- Xu, R. e Wunsch, D. II (2005) "Survey of Clustering Algorithms", IEEE Transactions networks, vol.16, issue 3, May, pp. 645-678.

ⁱ [1 www.ines.org.br](http://www.ines.org.br)

Formulando a Adaptação de Processos de Desenvolvimento de Software como um Problema de Otimização

Andréa M. Magdaleno^{1,3}, Marcio Barros², Cláudia Werner¹, Renata Araujo^{2,3}

¹Programa de Engenharia de Sistemas e Computação (PESC) – COPPE/UFRJ
Caixa Postal 68.511 – 21945-970 – Rio de Janeiro – RJ – Brasil

²Programa de Pós Graduação em Informática (PPGI) – UNIRIO

³Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec) – UNIRIO
22290-240 – Rio de Janeiro – RJ – Brasil

andrea@cos.ufrj.br, marcio.barros@uniriotec.br,
werner@cos.ufrj.br, renata.araujo@uniriotec.br

Resumo. *Uma das principais atividades do gerente no início do projeto é a adaptação do processo de desenvolvimento de software. Esta atividade é complexa, pois exige experiência, conhecimento do contexto do projeto e a harmonização das restrições existentes. Neste cenário, defende-se que a colaboração e a disciplina são importantes na adaptação do processo e podem ser balanceadas. Para facilitar este balanceamento, é possível automatizar a solução do problema, reduzindo o esforço necessário para executar esta tarefa e melhorando o processo obtido. Assim, este trabalho formula um problema de otimização e apresenta a sua definição, modelagem e análise. Como próximos passos para a construção da solução, pretende-se investir na sua implementação e avaliação.*

1. Introdução

Como o universo dos projetos de desenvolvimento de software é muito extenso e diversificado, nenhum processo de desenvolvimento consegue preencher todos os requisitos dos projetos ou organizações, pois eles têm objetivos, características e necessidades diferentes. Diante deste cenário, surge a necessidade de adaptar os processos de desenvolvimento às necessidades dos projetos e organizações. A adaptação de processos é o “ato de particularizar um processo geral para derivar uma definição aplicável a um contexto mais específico” [Ginsberg e Quinn, 1995; Pedreira *et al.*, 2007]. Como o orçamento do projeto, o prazo de desenvolvimento e a qualidade do produto dependem diretamente da qualidade do processo de software, a adaptação de processos precisa ser feita corretamente [Pedreira *et al.*, 2007].

Apesar da adaptação de processos ser uma das principais tarefas executadas pelo gerente no início do projeto, ela não é simples. Esta tarefa exige experiência, conhecimento e requer a harmonização de muitos fatores do contexto da equipe, do projeto ou da organização. Devido a esta complexidade, geralmente, o gerente de projeto não é capaz de avaliar todas as opções de adaptação do processo e acaba fazendo esta adaptação de forma *ad-hoc*. Como resultado, o processo adaptado pode não ser a melhor alternativa para o projeto em questão.

Neste trabalho, defende-se que a adaptação de processos deve levar em consideração dois aspectos fundamentais: colaboração e disciplina. A colaboração foca na interação entre as pessoas durante o desenvolvimento do projeto e é um fator

importante para que as organizações de software possam alcançar seus objetivos de produtividade, qualidade e compartilhamento de conhecimento. A disciplina está relacionada à rigidez do controle empregue na execução do processo de desenvolvimento. A colaboração e a disciplina são complementares e essenciais para qualquer projeto de desenvolvimento de software, mas em diferentes proporções, dependendo do contexto da organização, do projeto e da equipe. Por isso, a colaboração e a disciplina precisam ser balanceadas [Magdaleno, 2010a].

Com o propósito de facilitar a adaptação de processos, é possível fornecer apoio ao gerente de projeto na execução desta atividade, uma vez que alguns dos passos para a solução do problema podem ser automatizados, possivelmente diminuindo o esforço necessário para a sua execução e melhorando os resultados obtidos. Este trabalho apresenta a modelagem inicial de uma abordagem baseada em otimização para buscar uma solução boa e viável entre as candidatas, levando em consideração um balanceamento de qualidade entre colaboração e disciplina.

O restante deste texto está organizado da seguinte forma: a Seção 2 descreve a modelagem e analisa o problema de balanceamento na adaptação de processos. A Seção 3 conclui este artigo.

2. Problema de Balanceamento na Adaptação do Processo

Para sistematizar e estimular a reutilização na adaptação de processos de desenvolvimento de software foi criada uma abordagem de Engenharia de Linha de Processos Baseada em Contexto (ELPBC) [Nunes *et al.*, 2010]. Uma linha de processos corresponde a um conjunto de elementos de processo que compartilham um conjunto de características comuns e variáveis dentro de um domínio específico e são desenvolvidas a partir de artefatos que podem ser reutilizados e combinados entre si, segundo regras de composição e recorte, para compor e adaptar processos dinamicamente.

Este dinamismo é importante porque a colaboração e a disciplina variam durante todo o ciclo de vida de desenvolvimento do software. Como os requisitos e o ambiente do projeto também mudam, a adaptação de processos deve ser executada de forma contínua, considerando o contexto de execução do processo e mantendo o alinhamento do processo com as necessidades atuais do projeto.

De forma análoga à Engenharia de Linha de Produtos [Atkinson *et al.*, 2001], a abordagem ELPBC é organizada em duas fases principais: Engenharia de Domínio do Processo e Engenharia de Aplicação do Processo. Na primeira fase, a organização cria a linha de processos capturando as similaridades e diferenças de um domínio. Na fase Engenharia de Aplicação do Processo, o processo específico do projeto é adaptado a partir da linha de processos utilizando as informações de contexto.

Durante a construção da linha de processos, dois artefatos principais são gerados: o diagrama de características da linha de processos e a arquitetura da linha de processos. O diagrama de características representa as similaridades e variações existentes entre os processos da linha. A arquitetura de processos é composta por componentes de processos que possuem propriedades internas, interfaces e variabilidades. Estes dois artefatos estão relacionados, pois cada característica possui um conjunto de componentes de processo que a implementa.

O Problema de Balanceamento na Adaptação do Projeto (PBAPP) inicia na fase de Engenharia de Aplicação do Processo, quando começa a adaptação do processo a partir da linha de processos.

2.1. Definição e Modelagem do Problema

A visão geral das etapas que compõem o problema de otimização PBAPP é apresentada pela Figura 1. A primeira etapa do problema envolve o recorte das características da linha de processos. Neste ponto, as *informações de contexto* da organização, do projeto e da equipe atual (IC) e as *regras de seleção de características* (RSCAI e RSCEA) atuam para selecionar as características apropriadas para o projeto (Figura 1a). Em seguida, as *regras de composição de características* (RCCAI e RCCAE) recortam as características que foram selecionadas (Figura 1b). Este recorte provoca um filtro no universo de componentes da linha de processos.

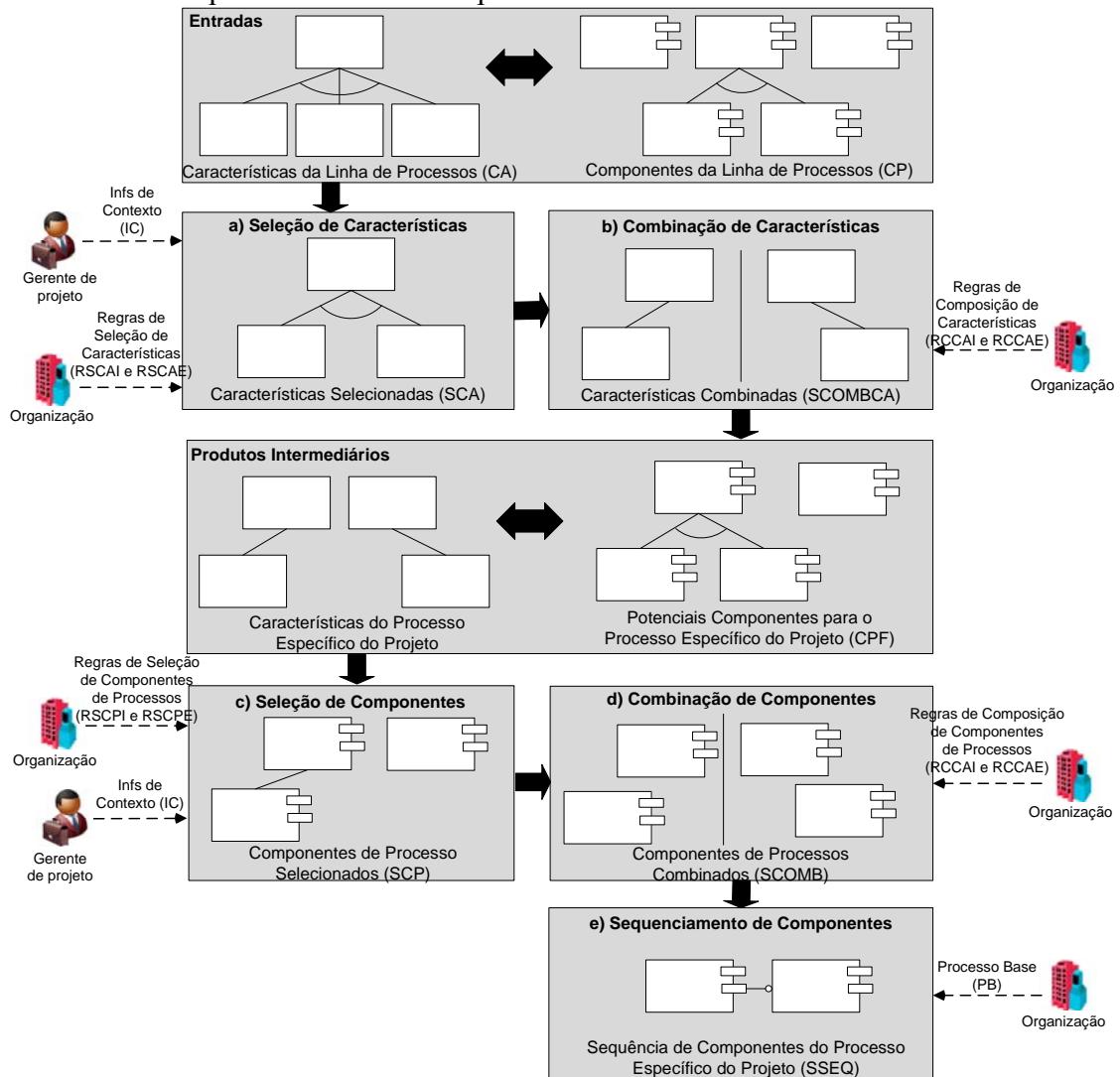


Figura 1 – Resumo das etapas do problema de balanceamento

Na etapa seguinte, as IC e as *regras de seleção de componentes* baseadas em contexto (RCSPI e RCSPE) filtram aqueles componentes de processos que são mais adequados para o projeto (Figura 1c). A partir daí, devem ser feitas as combinações possíveis dos componentes de processos de acordo com as *regras de composição de componentes* (RCCPI e RCCPE) (Figura 1d). Estas combinações serão avaliadas, considerando-se o seu potencial de colaboração e disciplina, para que seja escolhida a melhor combinação. Por fim, será composto o processo adaptado para o projeto através do encaixe dos componentes selecionados à *sequência de atividades* (ATIV) do *processo base* (PB) (Figura 1e).

2.1.1. Seleção das características do processo

A primeira etapa (Figura 1a) corresponde à seleção das características compatíveis com o contexto do projeto e pode ser modelada da seguinte forma:

- Seja CA o conjunto de características da linha de processos. Cada característica $CA_i \in CA$ possui um nome e um conjunto de componentes de processos.
 $CA = \{ CA_i \}$
 $CA_i = [nome_i, categoria_i, CP_i] \therefore CP_i \in CP$
- Seja CP o conjunto de componentes de processos da linha de processos. Cada componente de processo $CP_i \in CP$ é composto por um nome, um potencial de colaboração, um potencial de disciplina, um conjunto de interfaces requeridas (artefatos de entrada) e um conjunto de interfaces produzidas (artefatos de saída).
 $CP = \{ CP_i \}$
 $CP_i = [nome_i, pcolab_i, pdisc_i, INTRCP_i, INTCP_i]$
- Sejam $INTRCP_i$ e $INTCP_i$, respectivamente, conjuntos de interfaces requeridas (artefatos de entrada) e de interfaces providas (artefatos de saída) de um determinado componente de processo. Cada interface requerida $INTRCP_{ij} \in INTRCP_i$ possui um nome. A definição de $INTCP_{ij}$ é análoga.
 $INTRCP_i = \{ INTRCP_{ij} \}$
 $INTRCP_{ij} = [nome_{ij}]$
- Seja IC o conjunto de informações do contexto atual. Cada elemento $IC_i \in IC$ é composto por um nome e valor.
 $IC = \{ IC_i \}$
 $IC_i = [nome_i, valor_i]$
- Seja RSCAI um conjunto de regras de implicação para seleção de características. Estas regras indicam características que devem estar presentes em função das informações de contexto. Cada regra $RSCAI_i \in RSCAI$ é composta por um identificador, um antecedente (uma expressão de informações de contexto) e um consequente (um conjunto de características).
 $RSCAI = \{ RSCAI_i \}$
 $RSCAI_i = [id_i, ANTCAI_i, SEQCAI_i]$
 $\therefore ANTCAI_i = ecx$
 $SEQCAI_i = \{ CA_k \}, \text{ onde } CA_k \in CA$
 $ecx = (ecx \text{ AND } ecx) \vee (ecx \text{ XOR } ecx) \vee (ecx \text{ OR } ecx) \vee CTX$
 $\text{onde, } CTX = \text{nome } op \text{ valor}$
 $\text{onde, } \exists IC_i \in IC \rightarrow IC_i.nome = nome$
 $\text{e } op \in \{ =, \neq, >, \geq, <, \leq \}$
- Seja RSCE um conjunto de regras de exclusão para seleção de características. Estas regras indicam características que não devem estar presentes em função das informações de contexto. Cada regra $RSCE_i \in RSCE$ é composta por um identificador, um antecedente (uma expressão de informações de contexto) e um consequente (um conjunto de características).
 $RSCE = \{ RSCE_i \}$
 $RSCE_i = [id_i, ANTCAE_i, SEQCAE_i]$
 $\therefore ANTCAE_i = ecx$
 $SEQCAE_i = \{ CA_k \}, \text{ onde } CA_k \in CA$

Para resolver esta primeira parte do problema, é necessário selecionar as características que são indicadas pelas regras de implicação de contexto e não são indicadas pelas regras de exclusão de contexto:

$$SCA = \{ SCA_i \in CA \mid \alpha(SCA_i) \wedge \beta(SCA_i) \}$$

$$\begin{aligned} \alpha(s) &\leftarrow \exists RSCAI_i \in RSCAI \mid s \in RSCAI_i.SEQCAI \wedge eval(RSCAI_i.ANTCAI, IC) = \text{true} \\ \beta(s) &\leftarrow \nexists RSCAE_i \in RSCAE \mid s \in RSCAE_i.SEQCAE \wedge eval(RSCAE_i.ANTCAE, IC) = \text{true} \end{aligned}$$

onde $eval(eca, IC)$ assume um valor booleano de acordo com os valores expressos no conjunto do contexto.

2.1.2. Combinação de características

A segunda etapa (Figura 1b) recebe como entrada um conjunto de características compatíveis com o contexto do projeto selecionadas na etapa anterior (SCA) e recorta estas características de acordo com as regras de composição, gerando como resultado uma lista de potenciais combinações destas características.

- Seja SCOMBCA₁ o conjunto com todas as combinações de características vindas de SCA.

$$SCOMBCA_1 = \{ SCA_i \}$$

$$SCA_i \subseteq SCA \wedge \nexists s \subseteq SCA \mid s \notin SCOMBCA_1$$

- Sejam RCCAI e RCCAE, respectivamente, um conjunto de regras de implicação e de exclusão para composição de características. Estas regras indicam características que devem ou não estar presentes em função da presença de outras características. Cada regra RCCAI_i ∈ RCCAI é composta por um identificador, um antecedente (uma expressão de características) e um consequente (um conjunto de características). As regras RCCAE são definidas de forma similar.

$$RCCAI = \{ RCCAI_i \}$$

$$RCCAI_i = [id_i, ANTCCAI_i, SEQCCAI_i]$$

$$\therefore ANTCCAI_i = eca$$

$$SEQCCAI_i = \{ CA_k \}, \text{ onde } CA_k \in SCA$$

$$eca = (eca \text{ AND } eca) \vee (eca \text{ XOR } eca) \vee (eca \text{ OR } eca) \vee CAR$$

$$\therefore CAR \in CA$$

Neste caso, é necessário selecionar o conjunto de características que são indicadas pelas regras de composição e não são indicadas pelas regras de exclusão:

$$SCOMBCA = \{ SCOMBCA_i \in SCOMBCA_1 \mid \alpha(SCOMBCA_i) \wedge \beta(SCOMBCA_i) \}$$

$$\alpha(r) \leftarrow \forall RCCAI_i \in RCCAI \mid evaluate(RCCAI_i.ANTCCAI, r) = \text{true}$$

$$\Rightarrow \nexists CA_k \in RCCAI_i.SEQCCAI \mid CA_k \notin r$$

$$\beta(r) \leftarrow \nexists RCCAE_i \in RCCAE \mid evaluate(RCCAE_i.ANTCCAE, r) = \text{true}$$

$$\Rightarrow \nexists CA_k \in RCCAE_i.SEQCCAE \mid CA_k \in r$$

onde $evaluate(eca, SCOMBCA)$ assume um valor booleano de acordo com a forma como as características SCOMBCA satisfazem a expressão eca .

2.1.3. Seleção dos componentes de processo

Para um determinado conjunto de características pertencente à SCOMBCA, a etapa de seleção dos componentes de processo compatíveis com o contexto do projeto (Figura 1c) pode ser modelada da seguinte forma:

- Seja CPF o conjunto de componentes de processos filtrados da arquitetura da linha de processos após o recorte das características. A definição do componente de processo $CPF_i \in CPF$ é feita da mesma forma que a definição dos componentes CP. $CPF = \{ CPF_i \} \therefore CPF_i \in CP$
- Sejam RSCPI e RSCPE, respectivamente, um conjunto de regras de implicação e de exclusão para seleção de componentes. Estas regras indicam componentes que devem ou não estar presentes em função das informações de contexto. Cada regra $RSCPI_i \in RSCPI$ é composta por um identificador, um antecedente (uma expressão de informações de contexto) e um consequente (um conjunto de componentes de processos). As regras RSCPE são definidas de forma análoga.
 $RCSPI = \{ RCSPI_i \}$
 $RCSPI_i = [id_i, ANTCI_i, SEQCI_i]$
 $\therefore ANTCI_i = ecx$
 $SEQCI_i = \{ CP_k \}, \text{ onde } CP_k \in CPF$

Nesta parte do problema, é necessário selecionar o conjunto de componentes de processo disponíveis para o projeto, que são indicados pelas regras de implicação de contexto e não são indicados pelas regras de exclusão de contexto:

$$SCP = \{ SCP_i \in CPF \mid \alpha(SCP_i) \wedge \beta(SCP_i) \}$$

$$\alpha(u) \leftarrow \exists RSCPI_i \in RSCPI \mid u \in RSCPI_i.SEQCI \wedge eval(RSCPI_i.ANTCI, IC) = true$$

$$\beta(u) \leftarrow \nexists RSCPE_i \in RSCPE \mid u \in RSCPE_i.SEQCE \wedge eval(RSCPE_i.ANTCE, IC) = true$$

2.1.4. Combinação dos componentes de processo

Esta etapa (Figura 1d) recebe como entrada uma lista de componentes de processo compatíveis com o projeto de acordo com suas informações de contexto (SCP) e gera como resultado uma lista de potenciais combinações destes componentes que podem ser utilizadas na formação do processo base (SCOMB).

- Seja $SCOMB_1$ o conjunto com todas as combinações de componentes de processos vindas de SCP.
 $SCOMB_1 = \{ SCP_i \}$
 $SCP_i \subseteq SCP \wedge \nexists s \subseteq SCP \mid s \notin SCOMB_1$
- Sejam RCCPI e RCCPE, respectivamente, um conjunto de regras de implicação e de exclusão para composição de componentes. Estas regras indicam componentes que devem ou não estar presentes em função da presença de outros componentes. Cada regra $RCCPI_i \in RCCPI$ é composta por um identificador, um antecedente (uma expressão de componentes de processos) e um consequente (um conjunto de componentes de processos). As regras RCCPE são definidas de forma análoga.
 $RCCPI = \{ RCCPI_i \}$
 $RCCPI_i = [id_i, ANTI_i, SEQI_i]$
 $\therefore ANTI_i = ecp$
 $SEQI_i = \{ CP_k \}, \text{ onde } CP_k \in SCP$
 $ecp = (ecp \text{ AND } ecp) \vee (ecp \text{ XOR } ecp) \vee (ecp \text{ OR } ecp) \vee COMP$
 $\therefore COMP \in SCP$

Para selecionar a lista de todas as combinações de componentes viáveis segundo as regras de composição de componentes, é necessário filtrar o conjunto $SCOMB_1$ e remover os elementos que não atendem a estas regras:

$$SCOMB = \{ SCOMB_i \in SCOMB_1 \mid \alpha(SCOMB_i) \wedge \beta(SCOMB_i) \}$$

$$\alpha(t) \leftarrow \forall RCCPI_i \in RCCPI \mid evaluate(RCCPI_i.ANTI, t) = true$$

$$\Rightarrow \nexists CP_k \in RCCPI_i.SEQI \mid CP_k \notin t$$

$$B(t) \leftarrow \forall RCCPE_i \in RCCPE \mid evaluate(RCCPE_i.ANTI, t) = true$$

$$\Rightarrow \nexists CP_k \in RCCPE_i.SEQE \mid CP_k \in t$$

onde $evaluate(epc, SCOMB)$ assume um valor booleano de acordo com a forma como os componentes SCOMB satisfazem a expressão epc .

2.1.5. Sequenciamento dos componentes de processos

A etapa de sequenciamento dos componentes de processos (Figura 1e) para preencher as atividades do processo base ocorre após a combinação dos componentes de processos:

- Seja PB um processo base (ou modelo de ciclo de vida) definido pela organização. PB é composto por um nome e um conjunto de atividades.

$$PB = [nome, ATIV]$$

- Seja ATIV um conjunto de atividades pertencente a um dado processo base PB. Cada atividade $ATIV_i \in ATIV$ é composta por um nome, um conjunto de pré-atividades, um conjunto de interfaces requeridas (artefatos de entrada) e um conjunto de interfaces produzidas (artefatos de saída).

$$ATIV = \{ ATIV_i \}$$

$$ATIV_i = [nome_i, PRE-ATIV_i, INTRATIV_i, INTPATIV_i]$$

$$\therefore PRE-ATIV_i = \{ ATIV \}$$

- Sejam INTRATIV_i e INTPATIV_i, respectivamente, um conjunto de interfaces requeridas (artefatos de entrada) e de interfaces providas (artefatos de saída) de uma determinada atividade. Cada interface requerida INTRATIV_{ij} ∈ INTRATIV_i possui um nome. INTPATIV_{ij} é definida de modo equivalente.

$$INTRATIV_i = \{ INTRATIV_{ij} \}$$

$$INTRATIV_{ij} = [nome_{ij}]$$

Para resolver o problema, é necessário associar os componentes de processos que farão parte de cada atividade do processo base:

$$SCPATIV = \{ SCPATIV_i \}$$

$$SCPATIV_i = [AT_i, \{ C_{ij} \}]$$

$$\therefore AT_i \in ATIV$$

$$\forall ATIV_i \in ATIV \subset PB \mid \exists SCPATIV_i.AT = ATIV_i$$

$$C_{ij} \leftarrow componentes(AT_i, SCOMB_i)$$

$$componentes(AT_i, SCOMB_i)$$

$$SUB = \{ SUB_i \}$$

$$\therefore SUB_i \subseteq SCOMB_i \wedge \nexists s \subseteq SCOMB_i \mid s \notin SUB$$

$$\Phi \leftarrow \phi; componentes selecionados$$

$$Para cada SUB_i \in SUB$$

$$\forall r \in AT_i.INTRATIV \rightarrow \exists c \in SUB_i \mid r \in c. INTRCP$$

$$\forall s \in AT_i.INTPATIV \rightarrow \exists c \in SUB_i \mid s \in c. INTPCP$$

$$\forall c \in SUB_i \rightarrow \forall r \in c. INTRCP \mid r \in \bigcup SUB_i.INTPCP \vee r \in AT_i.INTRATIV$$

Se todas estas condições são satisfeitas, então:

$$\Phi \leftarrow \Phi \cup SUB_i$$

2.2. Funções de Custo

Normalmente, há várias soluções candidatas para um problema de otimização. As funções de custo caracterizam o que é considerada uma boa solução e ajudam a identificar a técnica de busca mais aplicável e a entender melhor a natureza das soluções candidatas em termos dos seus custos [Harman e Jones, 2001].

Como o problema deste trabalho de pesquisa envolve o balanceamento dos aspectos de colaboração e disciplina na adaptação do processo, estes aspectos foram usados como funções de custo para determinar a qualidade da solução do problema. A função de custo de colaboração é apresentada a seguir e a função de custo de disciplina é definida de modo equivalente.

- **Maior Colaboração:** função responsável por determinar a seleção de componentes de processos que satisfaz a todas as restrições do problema e representa a maior colaboração para o projeto. A colaboração é calculada pelo somatório do potencial de colaboração existente em cada componente de processo incluído no processo.

○ Seja S o conjunto de soluções possíveis que satisfazem a todas as restrições.

Seja $\text{Colab}(S_k)$ uma função que retorne a colaboração de um dado componente de processo S_k . O componente selecionado $S_i \in S$ é escolhido de acordo com a seguinte equação:

$$\nexists(S_j \in S) (\text{Colab}(S_j) < \text{Colab}(S_i))$$

2.3. Complexidade e Tamanho do Espaço de Busca

A complexidade do problema é influenciada pelo número de componentes de processos (p) e de atividades (a). Considerando estes fatores, a complexidade do problema é: $O(p!^a)$. Entretanto, quando as restrições são adicionadas na formulação do problema, o esforço necessário para encontrar as soluções tende a decrescer, devido ao número de caminhos que não atendem às restrições e têm menos utilidade do que a melhor solução. Desta forma, o número de possíveis componentes de processos para preencher cada atividade do processo base diminui para p' , onde $p' < p$.

Na prática, o espaço de busca cresce proporcionalmente a $(p'!)^a$. Por exemplo, considerando um processo base, como o ciclo de vida em cascata, com uma média de 5 atividades e supondo, com base no OpenUP (*Open Unified Process*)¹, que um modelo de desenvolvimento tem uma média de 20 componentes, podemos estimar que uma linha de processos preparada para trabalhar com 5 modelos de desenvolvimento distintos, vai possuir aproximadamente 100 componentes de processos. Assim, o espaço completo de busca tem aproximadamente $(100!)^5$ possíveis soluções.

Como uma solução exata para o problema não pode ser encontrada em um tempo exequível, o uso de algoritmos heurísticos, tais como os algoritmos genéticos, parece ser a opção mais indicada para implementar a solução deste problema.

3. Conclusão

Este artigo abordou a adaptação de processos de software como um problema de otimização. A definição e modelagem formal do problema foram apresentadas. A modelagem inicial do problema foi descrita em Magdaleno [2010b], mas desde então foi modificada com o acréscimo de novas etapas. Esta modelagem do problema é crítica para compreender a natureza do problema e apontar as técnicas de otimização mais

¹ Site: <http://epf.eclipse.org/wikis/openup/>

indicadas para serem implementadas. A análise do espaço de busca também mostrou que ele é grande o suficiente para justificar o uso de métodos heurísticos.

Como próximos passos, pretende-se definir mais claramente como as funções de custo devem ser calculadas, modelar computacionalmente o problema e selecionar qual técnica é mais apropriada para ser aplicada na solução deste problema. Talvez a solução envolva inclusive a combinação de mais de uma técnica de otimização, como foi feito em outros trabalhos [Barreto *et al.*, 2008; Netto, 2010]. Clarke *et al.* [2003] sugerem que se comece pela técnica mais simples para verificar se os resultados são encorajadores antes de investir em técnicas mais sofisticadas que podem representar um esforço adicional desnecessário. Por fim, a solução deve ser implementada utilizando a técnica selecionada e validada para avaliar sua viabilidade e validade.

Agradecimentos

Este trabalho é parcialmente financiado pelo CNPq sob o processo no. 142006/2008-4 e faz parte do projeto de pesquisa da FAPERJ no. E-26/103.049/2008.

Referências

- Atkinson, C., Bayer, J., Bunse, C., et al. (2001). "Component-Based Product Line Engineering with UML". Addison-Wesley Professional.
- Barreto, A., Barros, M. D. O., e Werner, C. M. (2008). "Staffing a software project: A constraint satisfaction and optimization-based approach". *Computers & Operations Research*, v. 35, n. 10, pp. 3073-3089.
- Clarke, J., Dolado, J. J., Harman, M., et al. (2003). "Reformulating software engineering as a search problem". *IEEE Proceedings-Software*, v. 150, n. 3, pp. 161–175.
- Ginsberg, M., e Quinn, L. (1995). "Process Tailoring and the Software Capability Maturity Model", CMU/SEI-94-TR-024, SEI-CMU, <http://www.sei.cmu.edu/publications/documents/94.reports/94.tr.024.html>.
- Harman, M., e Jones, B. F. (2001). "Search-Based Software Engineering". *Information and Software Technology*, v. 43, pp. 833-839.
- Magdaleno, A. M. (2010a). "Balancing Collaboration and Discipline in Software Development Processes". Doctoral Symposium of International Conference on Software Engineering (ICSE), Cape Town, South Africa, pp. 331-332.
- Magdaleno, A. M. (2010b). "An optimization-based approach to software development process tailoring". PhD Track - International Symposium on Search Based Software Engineering (SSBSE), Benevento, Italy (to appear).
- Netto, F. D. C. (2010). "Um Método Automático para Geração de Cronogramas de Tarefas de Correção de Bugs". Dissertação de Mestrado, Rio de Janeiro: Universidade Federal do Estado do Rio de Janeiro (UNIRIO).
- Nunes, V. T., Werner, C., e Santoro, F. M. (2010). "Context-Based Process Line". International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira, Portugal, pp. 277-282.
- Pedreira, O., Piattini, M., Luaces, M. R., et al. (2007). "A systematic review of software process tailoring". *SIGSOFT Software Engineering Notes*, v. 32, n. 3, pp. 1-6.

Uma Estratégia de Otimização para Agrupamento de Componentes de Software Baseada em DSM

Thaís Alves Burity Pereira, Glêdson Elias

COMPOSE, Departamento de Informática
Universidade Federal da Paraíba (UFPB) – Brasil

thais@compose.ufpb.br, gledson@di.ufpb.br

Abstract. In distributed Software Product Line (SPL) projects, dependencies between components influence on communication needs between their respective development teams. Thus, an alternative to reduce such needs is to cluster tightly coupled components into loosely coupled modules as long as each module is developed by a single team. In such a context, since numerous clustering possibilities exist, this paper describes an optimization strategy for clustering software components based on the technique named numerical DSM (Design Structure Matrix), which is adopted to represent dependencies among components of the SPL architecture.

Resumo. Em projetos distribuídos de Linhas de Produto de Software (LPS), dependências entre componentes exercem relevante influência sobre a necessidade de comunicação entre suas respectivas equipes de desenvolvimento. Dessa forma, uma alternativa para reduzir tal necessidade é agrupar componentes fortemente acoplados em módulos fracamente acoplados entre si, desde que cada módulo seja desenvolvido por uma única equipe. Nesse contexto, dado que são inúmeras as possibilidades de agrupamento, este artigo descreve uma estratégia de otimização para agrupamento de componentes de software baseada na técnica de DSM (Design Structure Matrix) numérica, que é adotada para representar as dependências entre os componentes da arquitetura de uma LPS.

1. Introdução

Visando encontrar melhores oportunidades de negócio e profissionais mais qualificados, grandes empresas têm investido no desenvolvimento distribuído de software (DDS). Nesse contexto, Linhas de Produto de Software (LPS) são consideradas uma estratégia facilitadora, por favorecer a definição de uma arquitetura mais modularizada, cujos componentes podem ser paralelamente desenvolvidos por equipes geograficamente distribuídas [Paulish 2003]. No entanto, componentes mantêm certa dependência com outros componentes, já que componentes devem ser integrados para instanciar aplicações, exercendo assim influência sobre a necessidade de comunicação entre suas respectivas equipes de desenvolvimento, conforme evidenciado em [Grinter *et al.* 1999] e [Sosa *et al.* 2002]. Por conseguinte, a produtividade, a qualidade dos produtos, o cronograma de atividades e os custos de desenvolvimento podem ser comprometidos.

Para o desenvolvimento de software ser mais eficiente, de acordo com [Mockus e Weiss 2001], a distribuição geográfica da organização e a estrutura de comunicação devem corresponder à divisão de tarefas no desenvolvimento de software, o que significa que tarefas fortemente relacionadas que requerem coordenação e sincronização freqüentes devem ser executadas em um só local.

Considerando que componentes para serem implementados consistem em tarefas de desenvolvimento, nesse artigo é apresentada uma estratégia de otimização para agrupamento de componentes de software fortemente acoplados em módulos de software fracamente acoplados entre si, a fim de que possam ser implementados por diferentes equipes distribuídas com reduzida necessidade de comunicação. A estratégia proposta é baseada na técnica de otimização denominada DSM (*Design Structure Matrix*) numérica, que é adotada para representar as dependências entre os componentes constituintes da arquitetura da LPS.

Algoritmos de agrupamento para DSM em geral operam segundo a definição de uma função objetivo e de uma estratégia de otimização da solução. No contexto deste trabalho, a função objetivo expressa o custo das dependências entre um conjunto de componentes, e a estratégia de otimização visa encontrar o menor valor possível para essa função a fim de encontrar a melhor configuração de agrupamento. Apesar de existirem diversos algoritmos desse tipo na literatura, tais algoritmos não lidam com o desenvolvimento de produtos de software e, além disso, apresentam limitações para serem aplicados no contexto da presente estratégia. O algoritmo proposto em [Idicula 1995] admite que um elemento pertença a mais de um agrupamento e também já foi constatado que tanto ele quanto o algoritmo em [Fernandez 1998] nem sempre levam a uma solução realmente próxima da ótima. Os algoritmos apresentados em [Whitfield *et al.* 2002] e [Wang e Antonsson 2004] geram agrupamentos que definem uma relação hierárquica entre componentes, o que foge do escopo da presente abordagem. O algoritmo descrito em [Yu *et al.* 2007] requer que seja especificado um número máximo de agrupamentos a serem gerados e também permite que um elemento pertença a mais de um agrupamento. Por fim, em [Helmer 2008] é proposto um algoritmo que lida com DSM numérica que contém informação sobre cinco tipos de dependência (estrutural, de energia, de sinal, material e espacial), o que não se aplica a produtos de software.

A estratégia aqui proposta é fortemente baseada no algoritmo de [Thebeau 2001], que foi originalmente desenvolvido para o agrupamento de componentes físicos, tais como os componentes de um sistema de elevador, a fim de possibilitar uma melhor separação entre as atividades desempenhadas pelas equipes de desenvolvimento. Apesar desse algoritmo não garantir a eliminação de sobreposições entre agrupamentos, reduz consideravelmente a sua ocorrência, tendo sido utilizado com sucesso em estudos recentes, tal como [Bonjour *et al.* 2009].

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta a estratégia de agrupamento proposta, ressaltando suas principais contribuições. A Seção 3 apresenta as considerações finais e os trabalhos futuros.

2. Estratégia para Agrupamento de Componentes

O algoritmo de [Thebeau 2001] é um algoritmo estocástico que interativamente tenta diminuir o valor de uma função de custo. No contexto deste trabalho, a função representa o custo total de coordenação dos componentes segundo o custo de cada componente individualmente. Dado o componente c_i , o seu custo de coordenação é definido com base em suas dependências em relação aos demais componentes na DSM, conforme expresso pela equação (1):

$$custo(c_i) = \sum_{j=1}^t DSM(i, j) * n^{powcc} \quad (1)$$

Na equação (1), t é o número de componentes na DSM; $DSM(i, j)$ é o valor da dependência do componente i em relação ao componente j segundo a DSM do projeto.

O termo n representa o número de componentes no agrupamento contendo os componentes i e j , caso pertençam ao mesmo agrupamento, ou caso contrário, representa o valor t , como se todos os elementos na DSM formassem um agrupamento único. O termo $powcc$ é um parâmetro que permite ajustar a relevância do tamanho do agrupamento sobre o custo de coordenação. Por exemplo, se $powcc$ é igual 0, o tamanho dos agrupamentos não exerce qualquer influência sobre o cálculo do custo de coordenação. Porém, se $powcc$ é igual a 1, o custo de coordenação irá aumentar proporcionalmente ao tamanho dos agrupamentos, enquanto que se $powcc$ é igual a 2 é mantida uma relação quadrática.

Dessa forma, o custo total de coordenação dos componentes consiste na soma do custo de coordenação de todos os componentes, conforme mostra a equação (2), sendo t o número de componentes na DSM.

$$custoTotal = \sum_{i=1}^t custo(c_i) \quad (2)$$

O algoritmo inicialmente atribui cada componente a um agrupamento. Com essa configuração, o custo inicial total, o maior que deve ser encontrado, é calculado. O esperado é que a formação de qualquer agrupamento reduza o custo total. O sistema é então configurado como instável, isto é, ele é configurado para repetir o procedimento de agrupamento até que não seja mais possível melhorar o custo total depois de repetidas tentativas. O algoritmo randomicamente escolhe um componente e calcula o vínculo deste com cada agrupamento, conforme a equação (3). Tal vínculo é uma medida do quanto dependente os membros de um agrupamento são do componente selecionado.

$$vínculo(c_i, m_k) = \frac{(\sum_{j=1}^{n_k} DSM(i, j) + DSM(j, i))^{powdep}}{n_k^{powbid}} \quad (3)$$

Na equação (3), $vínculo(c_i, m_k)$ é o vínculo do componente c_i escolhido com o módulo m_k , o numerador consiste na soma das dependências entre o componente escolhido e cada componente do módulo m_k segundo a DSM do projeto, e n_k consiste no número de componentes no módulo m_k . Os expoentes $powdep$ e $powbid$ permitem definir a relevância da dependência entre componentes e do tamanho dos agrupamentos sobre o vínculo entre um componente e um agrupamento, respectivamente. É recomendado atribuir um valor entre 0 e 2 para $powdep$ e entre 0 e 3 para $powbid$.

Para evitar que o algoritmo fique preso a um ótimo local, é empregada a técnica de arrefecimento simulado, que introduz comportamentos aleatórios. Tais comportamentos são representados pelos parâmetros $randBid$ e $randAccept$, que devem ser configurados antes do algoritmo ser iniciado. O parâmetro $randBid$ indica quando o algoritmo deve escolher o segundo maior valor de vínculo, expresso pela equação (3), ao invés de escolher o maior valor. Já o parâmetro $randAccept$ indica quando o algoritmo deve aceitar a inserção de um componente em um agrupamento mesmo quando esta não causa a redução do custo total. É recomendado que ambos os parâmetros recebam um valor entre a metade e o dobro do tamanho da DSM ($t/2 \leq valor \leq 2t$), sendo admitido por default o tamanho da DSM.

Então, o algoritmo seleciona o maior ou segundo maior vínculo diferente de zero, dependendo do valor de $randBid$. Por exemplo, se $randBid$ é 10, o maior vínculo será selecionado 90% das vezes, e o segundo maior vínculo será escolhido 10% das vezes. O componente escolhido é temporariamente atribuído ao agrupamento com o vínculo selecionado e é calculado o custo total resultante. Em geral, a mudança se torna

permanente quando ela promove a redução do custo total. Contudo, de acordo com o parâmetro *randAccept*, em momentos aleatórios é possível que uma mudança se torne permanente apesar da falta de melhoria do custo.

Quando uma mudança é feita, o algoritmo analisa a composição dos agrupamentos para remover agrupamentos que possuem conteúdo idêntico, agrupamentos vazios e agrupamentos que são subconjuntos de outros agrupamentos. Então, o sistema é configurado como instável e um novo componente é randomicamente escolhido (o que acontece $t \times m$ vezes, sendo t o tamanho da DSM e m o número de repetições, cujo valor *default* é 2). Quando não ocorrem mais mudanças, um novo componente é selecionado e o processo se repete até que, após várias tentativas (definida pelo parâmetro *limite de estabilidade*, cujo valor *default* é 2), o algoritmo não obtenha mais melhorias no custo total.

Para evitar que o algoritmo resulte em uma solução final inferior a outra encontrada em suas etapas intermediárias, o algoritmo deve salvar a informação sobre a melhor solução que encontrar. Assim, antes de inserir um componente em um módulo, se o custo total for aumentar, o algoritmo salva a solução atual e o seu valor de custo. Assim, antes de finalizar ele compara o custo da solução final com o custo da solução memorizada. Se o custo da solução final é igual ou inferior ao custo da solução memorizada, então a execução é finalizada. Caso contrário, o algoritmo substitui a solução final pela solução memorizada e continua a tentar otimizar os agrupamentos.

A Figura 1 ilustra o funcionamento do algoritmo com *powdep* igual a 4, e *powbid* e *powcc* iguais a 1, descrito partindo de uma DSM hipotética (Figura 1a) cujos valores de dependência estão no intervalo [0, 10]. Inicialmente cada componente é colocado em um agrupamento individual e o custo total é calculado, conforme mostrado na Figura 1b. Então, o componente C_3 é escolhido randomicamente e o cálculo de vínculo sugere que esse componente seja agregado ao módulo M_4 , que, até então, contém apenas o componente C_4 , conforme mostrado na Figura 1c. Como essa mudança promove a redução do valor de CDT, a mudança é efetivada. Na Figura 1d é mostrada uma situação semelhante àquela da Figura 1c, onde o componente C_2 é integrado ao módulo M_4 .

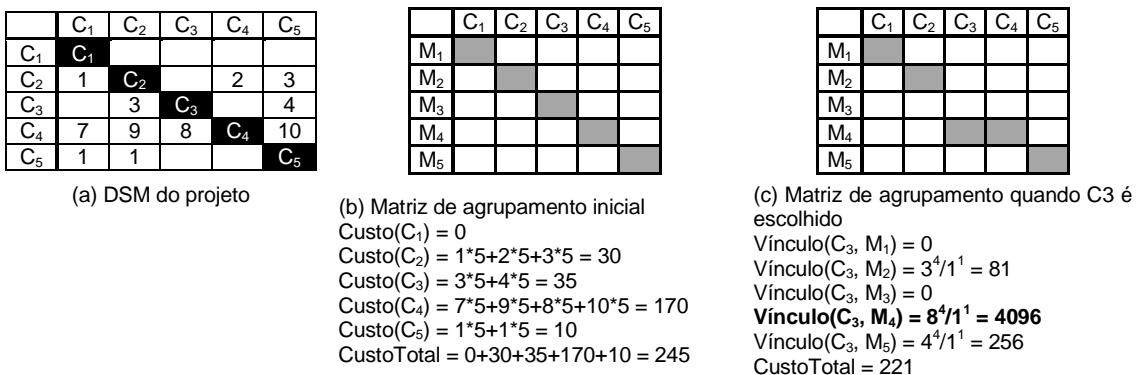


Figura 1. Exemplo de aplicação do algoritmo de [Thebeau 2001].

No contexto da presente estratégia, o custo total pode ser entendido como a medida da necessidade de comunicação em um projeto. A partir da análise das equações é possível perceber que, no algoritmo original, essa medida é proporcional ao peso da dependência entre componentes, o que também faz sentido para o objetivo da presente estratégia. Já o custo para um componente se tornar membro de um agrupamento aumenta proporcionalmente ao número de agrupamentos com os quais o componente mantém interação. Isso garante que componentes de integração (em se tratando de

software, tal como um componente de controle segundo o padrão arquitetural de controle centralizado¹, por exemplo), os quais estão acoplados a componentes pertencentes a agrupamentos diferentes, não sejam agrupados com outros componentes.

Além disso, pode ser observado que a formação de agrupamentos é restrinida pela quantidade de elementos, pois é pressuposto que o aumento do número de componentes implica em aumento de custo de coordenação. Em se tratando de componentes de software é mais significativo restringir a formação de agrupamentos com base na medida de complexidade ou esforço de desenvolvimento destes, já que essa medida influí na necessidade de comunicação entre as equipes responsáveis por desenvolver os agrupamentos. Em nível arquitetural, o esforço de desenvolvimento de componentes pode ser estimado a partir das métricas de complexidade de componentes propostas em [Cho *et al.* 2001]. Além disso, métricas de pontos de função, geralmente empregadas para medir a funcionalidade de sistemas de software como um todo, também podem ser utilizadas, desde que seja admitido que o aumento do número de funcionalidades implique no aumento do esforço de desenvolvimento dos componentes.

Por fim, o algoritmo original não garante que não existirá um elemento pertencendo a mais de um agrupamento, apesar de introduzir um custo de penalidade para soluções que o fazem. Contudo, para que agrupamentos de componentes sejam implementados por uma única equipe é preciso que cada componente pertença a um único agrupamento. Dessa forma, após ser feito o cálculo de vínculo entre o componente escolhido e os agrupamentos existentes, o algoritmo deve calcular o custo para as duas configurações distintas e preservar a configuração de menor custo: considerando o componente no agrupamento atual em que ele se encontra e considerando que ele seja deslocado para o agrupamento indicado pelo valor de vínculo.

3. Considerações Finais

Considerando a relevância do agrupamento de componentes no contexto de desenvolvimento de software e as deficiências das abordagens de agrupamento baseadas em DSM em geral, a estratégia apresentada no presente artigo propõe a remodelagem das funções de custo e de vínculo definidas no algoritmo de [Thebeau, 2001] para que estas considerem a complexidade dos agrupamentos, ao invés do tamanho destes, bem como a mudança dos passos do algoritmo após o cálculo da função de vínculo para prevenir a inserção de um elemento em mais de um agrupamento.

Vale ressaltar que a estratégia proposta está inserida no contexto de um abordagem que define um processo de análise para identificar candidatos à módulos para serem desenvolvidos de forma (parcialmente) independente por equipes geograficamente dispersas com base em evidência quantitativa. Como parte desse processo, a abordagem define: (i) medidas quantitativas que descrevem a dependência entre componentes de software; (ii) um algoritmo para recomendar módulos (que consiste na contribuição deste artigo), que depende, por sua vez, da definição de medidas quantitativas que descrevem a complexidade destes; e (iii) medidas quantitativas que descrevem as dependências entre módulos, que, dentro do framework de recomendação proposto em [Pereira *et al.* 2010], devem ser empregadas para guiar a alocação das equipes de desenvolvimento aos módulos.

¹ No padrão arquitetural de controle centralizado (*Centralized Control Architectural Pattern*) existe um componente de controle o qual conceitualmente executa um diagrama de estado e oferece o controle total e o seqüenciamento sobre o sistema.

Atualmente, está sendo definida a expressão de complexidade de componentes a ser empregada no algoritmo. Conforme apresentado, há diferentes métricas que expressam tal medida, sob diferentes perspectivas. Como atividade futura é planejada a implementação e posterior avaliação da estratégia a partir de um estudo de caso. Nesse sentido, é válido enfatizar que a qualidade das soluções de agrupamento depende não somente do algoritmo de otimização, mas também da estratégia de representação de dependência adotada. Além disso, testes devem ser realizados para serem definidos os valores para os parâmetros do algoritmo, já que os valores *default* não são adequados para todos os casos.

Agradecimentos. Este trabalho foi apoiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES)² e financiado pelo CNPq, processo 573964/2008-4.

Referências

- Bonjour, E. et al. (2009) "A Fuzzy Method for Propagating Functional Architecture Constraints to Physical Architecture", Journal of mechanical design, ISSN 1050-0472, 131(6).
- Cho, E. S. et al. (2001) "Component Metrics to measure Component Quality", 8th Asia Pacific Software Engineering Conference, pp. 419-426.
- Fernandez, C. I. G. (1998) "Integration analysis of product architecture to support effective team co-location", Master Thesis, Massachusetts Institute of Technology.
- Grinter, R. E. et al. (1999) "The Geography of Coordination: Dealing with Distance in R&D Work", International Conference on Supporting Group Work, pp. 306-315.
- Helmer, R. et al. (2008) "Systematic Module and Interface Definition Using component DSM", Journal of Engineering Design, Forthcoming.
- Idicula, J. (1995) "Planning for Concurrent Engineering", Master Thesis, Nanyang Technological University.
- Mockus, A., Weiss, D. M. (2001) "Globalization by Chunking: A Quantitative Approach", IEEE Software, 18(2), pp. 30-37.
- Paulish, D.J. (2003) "Product Line Engineering for Global Development", International Workshop on Product Line Engineering: The Early Steps.
- Pereira, T. A. B. et al. (2010) "A Recommendation Framework for Allocating Global Software Teams in Software Product Line Projects", 2nd International Workshop on Recommendation Systems for Software Engineering.
- Sosa, M. E. et al. (2002) "Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry", IEEE Transactions on Engineering Management, 49(1), pp. 45-58.
- Thebeau, R. E. (2001) "Knowledge management of system interfaces and interactions for product development processes", Master of Science Thesis, MIT.
- Wang, B., Antonsson, E. (2004) "Information measure for modularity in engineering design", ASME International Design Engineering Technical Conferences.
- Whitfield, R. et al. (2002) "Identifying component modules", 7th International Conference on Artificial Intelligence in Design AID02, pp. 571-592.
- Yu, T.-L. et al. (2007) "An information theoretic method for developing modular architectures using genetic algorithms", Research in Engineering Design, Springer-Verlag, 18(2), pp. 91-109.

² <http://www.ines.org.br/>