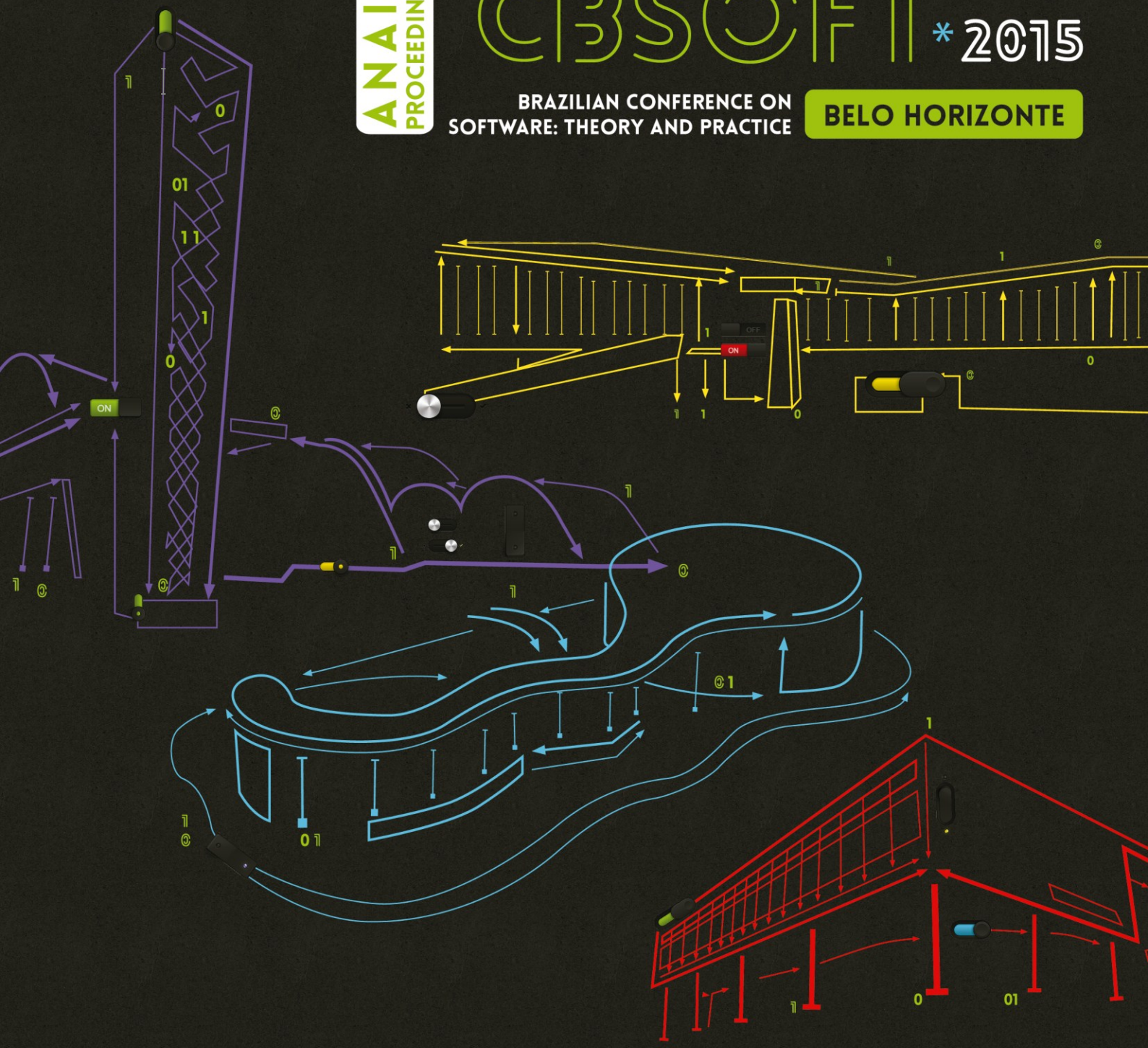
ANAIS PROCEEDINGS

CBSOFT * 2015

BRAZILIAN CONFERENCE ON SOFTWARE: THEORY AND PRACTICE

BELO HORIZONTE



SESSÃO DE FERRAMENTAS

CBSOFT.ORG

Sponsors:

ThoughtWorks · AvenueCode · Google Brasil · take.net · RARO LABS · Avanti · IBM

Promotion:

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico · CAPES · FAPEMIG · SBC Sociedade Brasileira de Computação

Organizing Institutions:

UFMG · PUC Minas · CEFET-MG

SESSÃO DE FERRAMENTAS

September 23rd –24th 2015
Belo Horizonte – MG, Brazil

ANAIS | *PROCEEDINGS*

COORDENADOR DO COMITÊ DE PROGRAMA DA SESSÃO DE FERRAMENTAS | *PROGRAM COMMITTEE CHAIR OF THE TOOLS SESSION*
TRILHA DE TRABALHOS TÉCNICOS | *TECHNICAL RESEARCH TRACK*
Marcel Vinícius Medeiros Oliveira (UFRN)

COORDENADORES GERAIS DO CBSOFT 2015 |*CBSOFT 2015 GENERAL CHAIRS*
Eduardo Figueiredo (UFMG)
Fernando Quintão (UFMG)
Kecia Ferreira (CEFET-MG)
Maria Augusta Nelson (PUC-MG)

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

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

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

COMITÊ DE ORGANIZAÇÃO | *ORGANIZING COMMITTEE*

CBSOFT 2015 GENERAL CHAIRS

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

CBSOFT 2015 LOCAL COMMITTEE

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

WEBSITE AND SUPPORT

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

COMITÊ TÉCNICO | *TECHNICAL COMMITTEE*

COORDENADOR DO COMITÊ DE PROGRAMA DA SESSÃO DE FERRAMENTAS | *TOOLS SESSION PC CHAIR*

Marcel Vinicius Medeiros Oliveira (UFRN)

COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE*

Alexandre Mota (UFPE)
Anamaria Martins Moreira (UFRJ)
Arilo Dias Neto (UFAM)
Bruno Costa (UFRJ)
Cláudio Sant'Anna (UFBA)
Daniel Lucrédio (UFSCar)
David Déharbe (UFRN)
Delano Beder (UFSCar)
Eduardo Almeida (UFBA)
Eduardo Figueiredo (UFMG)
Elder José Cirilo (UFSJ)
Elisa Huzita (UEM)
Fernando Trinta (UFC)
Frank Siqueira (UFSC)
Franklin Ramalho (UFCG)
Glauco Carneiro (UNIFACS)
Gledson Elias (UFPB)
Ingrid Nunes (UFRGS)
Juliana Saraiva (UFPB)
Leila Silva (UFS)
Luis Ferreira Pires (UniversityofTwente)
Marcel Oliveira (UFRN)
Marcelo dAmorim (UFPE)
Marcilio Mendonça (Amazon)
Marco Tulio Valente (UFMG)
Maria IstelaCagnin (UFMS)
Márcio Cornélio (UFPE)
Nabor Mendonca (UNIFOR)
Patricia Machado (UFCG)
Paulo Maciel (UFPE)
Paulo Pires (UFRJ)
Pedro Santos Neto (UFPI)
Ricardo Lima (UFPE)
Rita Suzana Pitangueira Maciel (UFBA)
Roberta Coelho (UFRN)
Rohit Gheyi (UFCG)
Rosângela Penteado (UFSCar)
Sandra Fabbri (UFSCar)

Tiago Massoni (UFCG)
Uirá Kulesza (UFRN)
Valter Camargo (UFSCar)
Vander Alves (UnB)

REVISORES EXTERNOS | *EXTERNAL REVIEWERS*

André Hora
Bruno Nogueira
Cesar Marcondes
Evaldo Silva
Fernando Figueira Filho
Juliana Alves Pereira
Kattiana Constantino
Marcelo Pitanga
Martin Musicante
Maurício Souza
Paloma Oliveira
Paulo Anselmo
Ricardo Menotti

TRILHA DE ARTIGOS TÉCNICOS | *TECHNICAL RESEARCH TRACK PAPERS*

SESSÃO 1: ARQUITETURA DE SOFTWARE E ENGENHARIA DE LINHAS DE PRODUTO DE SOFTWARE

SESSÃO2: AMBIENTES DE SUPORTE E COMPILADORES DE LINGUAGENS DE PROGRAMAÇÃO

SESSÃO3: VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE

Observatório do Investimento - Uma plataforma computacional para coleta, tratamento, mineração, análise, distribuição e visualização de dados de mídias Web para o mercado de capitais brasileiro

Renato Utsch[1], **Alef Miranda**[1], **Zilton Cordeiro Junior**[1], **Adriano C. M. Pereira**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte — MG — Brazil

{renatoutsch,alefwmm,zilton,adrianoc}@dcc.ufmg.br

Abstract. Observatório do Investimento *is a Web Portal portal built with an innovative client architecture that optimizes relevant data transference and uses* Web Components*, with the objective of supplying an analysis of the stock market by using news and data from social networks, obtained through data processing and computational intelligence techniques.*

Resumo. *O Observatório do Investimento é um portal Web construído com uma arquitetura de cliente inovadora que otimiza a transferência de dados relevantes e que utiliza* Web Components*, com o objetivo de fornecer uma análise do mercado de capitais a partir de notícias e informações de redes sociais, obtidas através de técnicas de tratamento de dados e inteligência computacional.*

URL para o vídeo: https://www.youtube.com/watch?v=X0b0vfq-mM8

1. Descrição e motivação do problema abordado pela ferramenta

O surgimento da Web 2.0 fez com que os usuários passassem de meros consumidores a produtores ativos de conteúdo. Ferramentas como fóruns, Twitter e redes sociais, como o Facebook, o Google+, o Instagram e o YouTube possibilitam que usuários comuns expressem suas opiniões sobre os mais diversos assuntos e propaguem, em tempo real, informações que consideram relevantes.

A partir do momento em que os usuários deixam de ter uma posição passiva na *Internet*, a interatividade e a avaliação do fluxo de informações em tempo real passa a ser um fator importante nas várias aplicações Web. Assim, a quantidade de dados disponível aumentou massivamente, em um curto período de tempo [Coffman and Odlyzko 2002]. Isso se manifesta no interesse cada vez maior por fenômenos dinâmicos, substituindo o paradigma anterior da Web como repositório estático de dados e informações sobre os mais variados temas e personalidades. Essa mudança de paradigma fez da Web uma ferramenta ainda mais poderosa em áreas cruciais, como a política e a economia.

Um aspecto interessante é que a Web deixou de ser um mecanismo de registro estático, como uma grande biblioteca digital para a qual foi concebida, tornando-se um ambiente marcado pelo dinamismo e pela demanda por interatividade em tempo real. Além disso, há um anseio crescente para sabermos o que está acontecendo agora e o que

aconteceu recentemente, para tentarmos explicar comportamentos e relações em diferentes contextos, como a relação entre uma empresa e seus produtos, entre uma nova notícia e os efeitos gerados no mercado financeiro, dentre outros.

Nesse contexto, o problema é definir a melhor maneira para coletar, analisar e apresentar dados massivos (*Big Data*) [Manyika et al. 2011] da Web, em tempo real. A coleta deve ser capaz de selecionar fontes e conteúdos relevantes, em particular aqueles que sejam gerados continuamente (*stream*), sendo ainda necessário identificar e extrair informações importantes desses dados. A análise deve sumarizar e extrair padrões interessantes dos dados, agregando valor a eles e facilitando o seu entendimento e desdobramentos. Finalmente, a visualização deve apresentar esses dados de forma intuitiva e significativa, facilitando o entendimento dos fenômenos e tendências, assim como o comportamento dinâmico. O requisito de análise em tempo real, aliado a outras características da Web, como o volume e a natureza diversificada dos dados, trazem um novo conjunto de desafios que as tecnologias correntes não estão habilitadas a superar.

O desafio deste problema está na diversidade de cenários a serem modelados, no volume de dados a serem processados, na complexidade das relações a serem analisadas, na efetividade das metáforas visuais a serem empregadas e na geração das análises em tempo real. É importante notar que vários desses requisitos são conflitantes, como a geração de análises complexas, baseada em algoritmos de descoberta do conhecimento (i.e. mineração de dados e aprendizado de máquina [Mitchell 1999]) que têm, frequentemente, complexidade exponencial, sobre grandes volumes de dados em tempo real. Em suma, não apenas os requisitos em si são desafiadores, mas resolvê-los conjuntamente se torna um desafio muito maior.

O objetivo geral deste trabalho é apresentar uma plataforma tecnológica, e um grande sistema de informação Web, que contemplará diferentes apresentações e análises de dados do mercado de capitais, correlacionando, em tempo real, os dados reais da bolsa de valores e dados de mídias Web , como redes sociais, portais de notícias, fóruns especializados, blog e microblogs.

O restante deste trabalho esta organizado da seguinte forma. Na Seção 2 são apresentadas as funcionalidades da ferramenta. A Seção 3 detalhada a arquitetura utilizada para o desenvolvimento da ferramenta. Na Seção 4 são apresentadas ferramentas relacionadas. Por fim, na Seção 6, é apresentada a licença de software da ferramenta proposta.

2. Funcionalidades e usuários

O Observatório do Investimento envolve uma série de tecnologias e desafios a serem resolvidos, em especial devido ao grande volume de dados, a importância do tratamento, da mineração, da análise e da visualização de diferentes tipos de informação em tempo real. Com o intuito de se tornar um grande sistema de informação Web, que contemplará diferentes apresentações e análises de dados do mercado de capitais, disponibilizando dados reais da bolsa de valores, mais especificamente do Índice Bovespa, como o preço e o volume financeiro e diversas outras informações a respeito de cada ativo das empresas. Todos esses dados advindos da bolsa de valores brasileira são correlacionados com informações coletadas de mídias Web, como redes sociais (Facebook e Google+), portais de conteúdo (p.ex., UOL, Terra, G1, Globo, etc.), fóruns, blogs, microblogs (p. ex., Twitter), portais especializados (Exame, Valor Econômico, etc.), dentre outros.

O foco da ferramenta, além da análise de sentimentos, é identificar a popularidade, as notícias e as menções relacionadas às empresas e ativos do índice bovespa. As informações extraídas das redes sociais e das notícias podem ser úteis aos investidores e a *softwares* de investimento automatizado.

Com o intuito de se tornar um grande aglutinador de informações do mercado de capitais brasileiro, o Observatório do Investimento se subdivide em dois produtos: O portal Web e a biblioteca de serviços Web (API).

Portal Web: é um sistema de informação Web integrado à plataforma tecnológica do Observatório do Investimento, onde as informações coletadas, relacionadas ao Mercado de Capitais, passam pelas análises descritas na Seção 3.2 e são disponibilizadas de maneira organizada e classificada já pelo teor e relevância das mesmas, podendo ser visualizadas e manipuladas pelo usuário. Este será o principal canal para acesso às informações geradas pela plataforma, sendo utilizado por investidores, corretoras e agentes autônomos de investimento. O Portal Web estará disponível em um sítio Web (domínio observatoriodoinvestimento.com).

Biblioteca de Serviços Web (API): é uma API (do inglês, *Application Programming Interface* ou Interface de Programação de Aplicativos), que representa um conjunto de rotinas e padrões estabelecidos por um sistema de software, para a utilização das suas funcionalidades por aplicativos que não devem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços. Assim, a API do Observatório do Investimento é composta por uma série de funções ou serviços acessíveis através da Web para fornecer indicadores e resultados providos pelo sistema de informação Web. Isso possibilita prover um meio padronizado e com interoperabilidade para que diferentes usuários interessados possam acessar e consumir informações geradas pela plataforma tecnológica, e com isso usar as informações diretamente ou mesmo incorporá-las a um serviço próprio para agregar valor. Para viabilizar o serviço, existe uma parceria com a empresa SmarttBot, que é certificada como *Subvendor* (distribuidor de informações) e Roteadora de Ordens (pedidos de compra e de venda de ações) na BM&FBOVESPA, que oferece ao mercado plataformas computacionais para o desenvolvimento, simulação e operacionalização de estratégias automatizadas de investimento. O Observatório do Investimento será a primeira plataforma nacional a disponibilizar, via API, dados sociais e de mídias Web em geral integrados a dados em tempo real da bolsa de valores (BM&FBOVESPA) para o desenvolvimento de estratégias automatizadas de investimento.

No atual cenário de investimentos no Brasil, discute-se fortemente uma tendência de migração das aplicações de renda fixa, hoje pouco privilegiadas em termos de rendimento, para outras com ganhos mais substanciais e, por consequência, com maior risco. Essa migração ocorreria por não ser mais possível a obtenção dos mesmos níveis de ganhos médios em aplicações mais conservadoras, dadas as diminuições de rentabilidade das mesmas. Um dos caminhos possíveis para esta migração é o mercado de capitais, em particular o mercado de ações.

Segundo a BM&FBOVESPA, existem aproximadamente 600 mil investidores pessoa física cadastrados na bolsa em novembro de 2013[BM&FBOVESPA 2013], dos quais 150 mil são ativos, distribuídos entre 89 corretoras de valores no Brasil, ou seja, há um grande potencial de crescimento. Além disso, segundo a ANCORD (Associação Nacional

das Corretoras), existem 9.957 agentes autônomos autorizados no Brasil.

Para este processo de consolidação e crescimento do mercado da bolsa de valores no Brasil, muitas ferramentas de suporte serão necessárias, principalmente no que tange ao auxilio na tomada de decisão de investimentos. Este trabalho visa auxiliar os diversos envolvidos na cadeia do mercado de capitais na escolha e acompanhamento de seus papéis. Dessa forma, o Observatório do Investimento irá auxiliar as corretoras, os agentes autônomos e, principalmente os investidores, no processo de definição de onde irão colocar seus recursos, de uma forma mais assertiva e informativa.

3. Arquitetura

Tradicionalmente, aplicações *Web* dinâmicas têm sido escritas seguindo o modelo em que cada usuário envia uma requisição ao servidor, que a processa, compõe o conteúdo e envia a página de volta para o usuário, que utiliza um navegador para visualizar os resultados. Esse modelo arquitetural possui problemas de desempenho, pois a cada recarga da página toda a estrutura precisa ser transmitida novamente, junto com as dependências da página. Ou seja, transmitir páginas Web completas do servidor para os clientes diminui o desempenho e escalabilidade necessária à aplicação. Esse problema pode ser contornado ao se separar os dados e os metadados da aplicação.

Uma maneira mais robusta de construir aplicações consiste na separação completa das responsabilidades entre cliente e servidor (Figura 1), onde o servidor deixa de ser responsável pela composição das páginas e delega essa tarefa para o navegador do cliente, passando a ser apenas um repositório de dados. O cliente, agora responsável por compor a página, passa a fazer requisições para o servidor sobre os dados a serem mostrados e constrói, por conta própria, a estrutura da página. Essa estrutura é implementada com base no *Representational State Transfer* (REST) [Fielding 2000], onde o servidor implementa suas responsabilidades fornecendo uma API, seguindo tais princípios. O REST foca sua atenção nos recursos da aplicação, permitindo acesso aos dados por meio de URLs (*Uniform Resource Locator*).

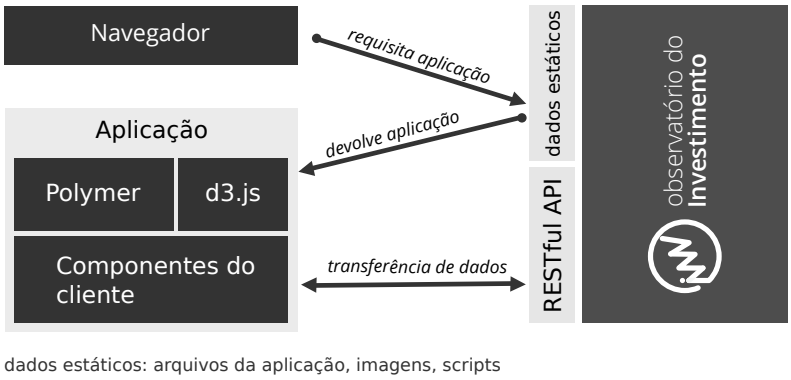


Figura 1. Estruturação simplificada entre cliente e servidor da aplicação.

Esse modelo possui diversas vantagens. Primeiro, há uma redução na complexidade do código do servidor, que se utiliza dos princípios do REST e passa a ser responsável apenas por enviar dados mediante requisições. O processo de compor uma página passa a ser responsabilidade do cliente, fazendo com que o servidor use menos

recursos, já que o tempo de processamento que antes era gasto na composição das páginas agora seja distribuído entre os vários clientes. Além disso, após o envio da estrutura da aplicação para o cliente, na primeira requisição, o tráfego de dados diminui satisfatoriamente, já que essa estrutura da página não é mais enviada. Toda comunicação entre cliente e servidor é feita estritamente a través da API.

A desvantagem nessa arquitetura é o aumento da carga de processamento no cliente, que agora precisa manipular os dados recebidos para compor a página. Porém, com o aumento da performance dos computadores nos últimos anos, esse impacto é dificilmente notado se a aplicação é bem construída.

As subseções a seguir explicam as tecnologias usadas no cliente e no servidor.

3.1. Cliente

O desenvolvimento de páginas Web na parte do cliente sempre sofreu com a falta de modularização. Por não existir nenhum tipo de encapsulamento em páginas Web, uma mudança, por exemplo, na folha de estilo direcionada a uma parte da página pode causar, não intencionalmente, uma mudança em outras áreas que não deveriam ser modificadas. Porém, com a chegada do HTML 5 [Hickson and Hyatt 2011], muitas melhorias foram introduzidas, sendo uma dessas melhorias a utilização de *Web Components*, um conjunto de elementos e APIs que permitem encapsulamento nas páginas *Web*. Com *Web Components* é possível criar elementos HTML personalizados, cuja implementação é encapsulada em uma árvore *Dynamic Object Model* (DOM) separada da árvore da página, conhecida como *Shadow DOM*. Desse modo, cada elemento passa a ser independente e os conflitos que antes eram comuns não acontecem mais.

O problema desse novo conjunto de funcionalidades é que a única forma de utilizá-las é a partir de funções em Javascript, de maneira imperativa. Para resolver esse problema, uma biblioteca chamada *Polymer* [Google 2015] foi criada para permitir que elementos sejam construídos de maneira declarativa, usando HTML, o que fornece uma maneira intuitiva para acessar as novas funcionalidades. Além disso, essa biblioteca fornece *polyfills*, que são implementações alternativas das funcionalidades do HTML 5, utilizada para garantir compatibilidade com os outros navegadores. Isso permite que utilizemos *Web Components* em todos os navegadores modernos.

O cliente do Observatório do Investimento utiliza o *Polymer* para implementar sua interface *Web*. O encapsulamento dos *Web Components* permite desenvolver, separadamente, cada componente da interface e apenas os integrar na página sem maiores problemas. Além disso, é fácil fazer requisições ao servidor no modelo REST e renderizar a página baseado na resposta.

A biblioteca *d3.js* [Bostock et al. 2011] é utilizada para processar e gerar visualizações a partir de dados enviados pelo servidor. Essa biblioteca fornece métodos para manipular uma imagem do tipo *Scalable Vector Graphics (SVG)*, em conjunto com *HTML* e *CSS* para desenhar gráficos na página. Uma interface da biblioteca para o *Polymer* foi implementada para facilitar a criação de gráficos em uma *Shadow DOM*.

3.2. Servidor

O servidor do Observatório consiste de um *pipeline* de processamento de dados, composto por cinco estágios, ilustrado na Figura 2. Seus cinco estágios estão descritos a seguir.

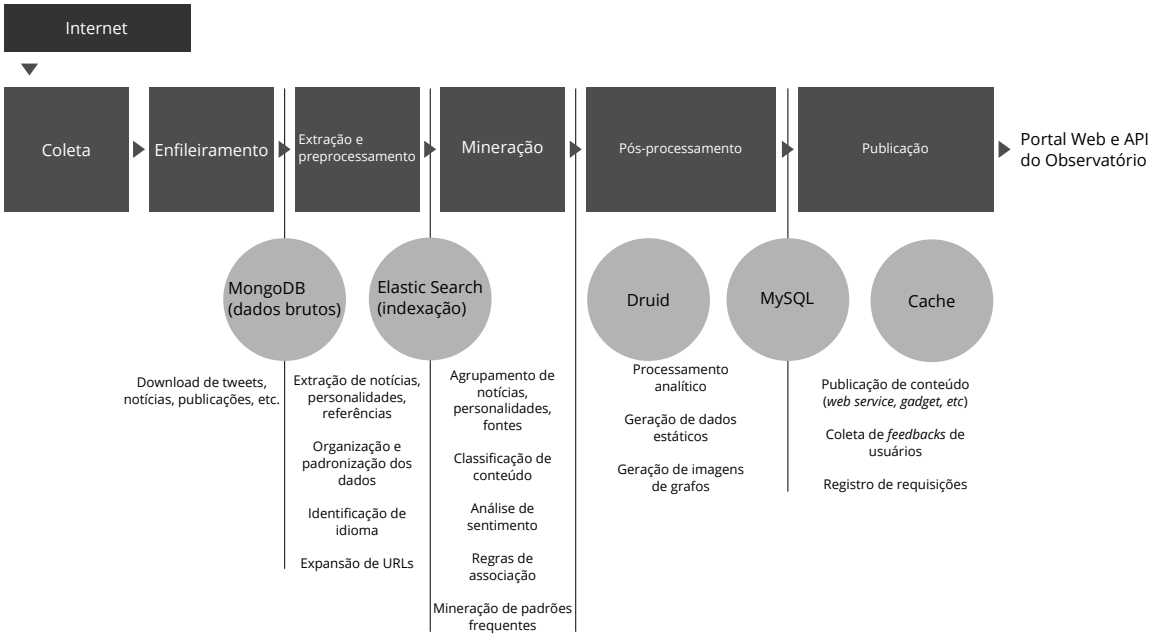


Figura 2. Estados do pipeline do servidor

Coleta: inicialmente os dados que serão utilizados são coletados de diversas fontes de dados públicos, que podem ser distinguidas em três categorias:

1. Redes sociais, como o Twitter, o Youtube e o Facebook, em que coletamos os dados através de APIs específicas para isso, que são fornecidas pelas próprias redes sociais.
2. Websites de jornais, revistas e portais de notícia voltados ao mercado de ações, em que os dados são coletados através de URLs publicadas em *feeds* (RSS e Atom).
3. Blogs e fóruns específicos, que são coletados através de palavras chave ou coletores especializados para a página.

Com a coleta de todas essas fontes de dados, é possível abranger os grandes disseminadores de informações e notícias do Mercado.

Enfileiramento: etapa de segmentação dos dados coletados, onde cada mensagem passa por um conjunto de filas para uma pré identificação dos rótulos, que podem ser empresas, setores e ativos da bolsa. Mensagens que não estão relacionadas ao contexto do mercado de capitais são descartadas.

Extração e Pré-processamento: essa etapa engloba apenas as categorias 2 e 3 do estágio anterior, visto que a mensagem retornada por APIs de coleta de dados já é estruturada. Para coleta em *feeds* de notícias é preciso extrair: título, URL da página, autor da matéria, data de publicação e resumo. Além disso, o texto da notícia normalmente não é disponibilizado nos canais RSS. Portanto, é necessário utilizar um extrator para separar o texto da página da notícia. Esse texto possui inúmeras informações relevantes que devem ser identificadas, como *hash tags*, empresas, ativos, setores, pessoas, organizações, lugares, datas, valores monetários etc. É nessa etapa em que o processo de descoberta do conhecimento é iniciado, em que informações, muitas vezes ainda difusas, são recuperadas dos dados.

Mineração: nessa etapa as notícias são agrupadas por ativo, empresa, setor e fontes. Além disso, o conteúdo dos textos é classificado através de técnicas como:

1. Análise de Sentimento, que mensura a positividade e negatividade de um texto automaticamente. Coleta-se informações sobre a polaridade (sentimento) das publicações apresentadas no portal a partir da opinião do usuário e um método de aprendizado de máquina (baseado em *Naive Bayes* [Lewis 1998]) é aplicado às notícias coletadas.
2. Regras de Associação, que são usadas para descobrir ocorrências em comum dentro de um conjunto de dados.
3. Mineração de Padrões Frequentes, que é utilizada para descobrir associações e correlações entre os dados.

Por ser capaz de separar os dados coletados por categorias e atrelar cada mensagem, texto, vídeo ou imagem coletada a uma entidade do contexto sendo monitorado, essa é uma das etapas mais importantes.

Pós-processamento: aqui as informações que foram extraídas dos dados são analisadas com o objetivo de gerar dados estatísticos, como contagens, somatórios, médias e outras medidas, que são úteis para resumir as negociações na bolsa de valores e as informações sociais. Esses dados podem ser visualizados, por exemplo, na forma de gráficos. Dentre as contagens realizadas nessa etapa, um destaque pode ser dado à medição da popularidade de uma URL nas redes sociais, o que permite identificar notícias populares sobre um ativo, empresa ou setor.

Publicação: as informações produzidas pelo pipeline do Observatório do Investimento são disponibilizadas através de uma API RESTful. Esses dados podem ser consumidos diretamente ou visualizados através de um cliente, como o portal Web, que consome da API para montar a página.

Esses estágios fazem parte dos desafios enfrentados para coletar e processar o grande volume de dados — também conhecido como *Big Data* [Manyika et al. 2011] — de fontes da Web. Como a incerteza, subjetividade e ambiguidade é inerente dos dados coletados das diversas fontes, existe uma constante evolução de padrões ao longo do tempo, que, em conjunto com a necessidade do processamento em tempo real, requer constante evolução tecnológica.

4. Ferramentas relacionadas

O Observatório do Investimento não tem nenhum concorrente direto no mercado brasileiro por ser essa uma aplicação pioneira, que explora e relaciona as interações entre os canais de notícias, os usuários da Internet, os investidores do mercado de capitais e os dados de ativos das empresas da BM&FBOVESPA. Com isso, o Observatório do Investimento está abrindo espaço no mercado, por meio de parcerias com empresas de ponta no mercado de ações (SmarttBot e Bússola do Investidor).

No entanto, pode-se elencar algumas empresas (ferramentas) no cenário nacional e internacional que extraem informações de dados difusos na Web. A exemplo, têm-se:

- Bright Planet [BrightPlanet 2015]: atua no mercado dos Estados Unidos, dedicada ao monitoramento de redes sociais, com o intuito de monitorar marcas e comportamentos, agregando valor aos dados ao introduzir conceitos e técnicas como a análise de sentimento e mineração de dados em tempo real.
- Zunnit [Technologies 2015]: atua no mercado brasileiro, dedicada a sistemas de recomendação, trabalhando com conceitos ligados a *Big Data* em tempo real e técnicas de aprendizagem de máquina e mineração de dados.

- Zahpee [Zahpee 2015]: atua no mercado nacional, dedicada ao monitoramento de redes sociais, em tempo real, para a extração de informações e padrões relevantes dos usuários, com relação a marcas, empresas e/ou produtos em diversos cenários.

5. Telas da Ferramenta

Como o Observatório do Investimento é uma aplicação Web, a ferramenta está acessível no domínio `https://www.observatoriodoinvestimento.com`.

6. Licença

As funcionalidades do aplicativo aqui descrito poderão ser utilizadas gratuitamente. Porém, melhorias e funcionalidades extras que forem implementadas no futuro podem fazer parte de um plano pago para os usuários que desejarem acesso a tais vantagens.

Agradecimentos

Esta pesquisa é parcialmente apoiada pelo Instituto Nacional de Ciência e Tecnologia para a Web (INWEB - CNPq no. 573871/2008-6), CAPES, CNPq, Finep e Fapemig.

Referências

BM&FBOVESPA (2013). Divulgação do balanço de operações de outubro. `http://tinyurl.com/ozx4ptq`. [Online; acessado 13-Nov-2013].

Bostock, M., Ogievetsky, V., and Heer, J. (2011). D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309.

BrightPlanet (2015). Brightplanet - deep web intelligence. `http://www.brightplanet.com`. [Online; acessado 30-Jul-2015].

Coffman, K. G. and Odlyzko, A. M. (2002). Internet growth: Is there a "moore's law" for data traffic? In *Handbook of massive data sets*, pages 47–93. Springer.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Google (2015). Polymer project. `https://www.polymer-project.org`. [Online; acessado 27-Mai-2015].

Hickson, I. and Hyatt, D. (2011). Html5. *W3C Working Draft WD-html5-20110525, May*.

Lewis, D. D. (1998). Naive (bayes) at forty: The independence assumption in information retrieval. In *Machine learning: ECML-98*, pages 4–15. Springer.

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A., and Institute, M. G. (2011). Big data: The next frontier for innovation, competition, and productivity.

Mitchell, T. M. (1999). Machine learning and data mining. *Comm. ACM*, 42(11):30–36.

Technologies, Z. (2015). Zunnit. `http://www.zunnit.com/`. [Online; acessado 30-Jul-2015].

Zahpee (2015). Zahpee. `https://zahpee.com/`. [Online; acessado 30-Jul-2015].

RipRop: A Dynamic Detector of ROP Attacks

Mateus Tymburibá, Rubens Emilio, Fernando Pereira

Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{mateustymbu,rubens,fernando}@dcc.ufmg.br

Abstract. *Return Oriented Programming (ROP) is presently one of the most effective ways to bypass modern software protection mechanisms. Current techniques to prevent ROP based exploits usually yield false positives, stopping legitimate programs, or false negatives, in which case they fail to detect attacks. Furthermore, they tend to add substantial overhead onto the programs they protect. In this paper we introduce RipRop – a tool to detect ROP attacks. RipRop monitors the frequency of indirect branches. If this frequency exceeds a threshold, then it aborts the program under guard. RipRop is very effective: on the one hand, it exposes all the exploits publicly available that we have experimented with; on the other hand, it does not prevent the legitimate execution of any program in the entire SPEC CPU 2006 benchmark suite. To solidify our claims, we provide an argument that building attacks with low-frequency of indirect branches is difficult, and show that RipRop improves current frequency-based techniques to detect ROP attacks.*

Link to Video: https://www.youtube.com/watch?v=EouKW1UUutU

1. Introduction

To make software more secure, we must find ways to prevent the class of exploits known as Return-Oriented Programming (ROP) attacks. ROPs are a relatively recent technique [Shacham 2007]. Nevertheless, in spite of its short history, ROP emerges today as one of the most effective ways to exploit vulnerable software [Lin et al. 2010, Roemer and Buchanan 2012]. This effectiveness stems from the fact that the available ROP prevention techniques do either suffer from false positives, false negatives, or high overhead. This paper presents RipRop, a tool to detect ROP exploits that is substantially different from all the techniques previously described. The key insight of this tool lies on the observation that public ROP-based exploits have a very high density of indirect branches. We demonstrate empirically that such a density is unlikely to be observed in actual programs. Thus, we have designed a binary instrumentation framework that keeps track of the frequency of indirect branches, and that fires an exception once this frequency exceeds a certain threshold. Even though simple, RipRop is effective. It recognizes all the exploits available in the "Exploits Database" [Anonymous 2014] that we have experimented with. Furthermore, we have not observed any false positive warning when applying our method on all the SPEC CPU 2006 [Henning 2006] programs. We reinforce our claims with two simple arguments. First, we show that although possible in some situations, circumventing our technique is hard enough to discourage attackers. Secondly, we explain that our approach has a low hardware footprint and does not affect the application's runtime.

2. Architecture

We have two versions of our tool: one for Windows, the other for Linux. It is built on top of Pin [Luk et al. 2005], a framework to instrument binary programs dynamically, i.e., at runtime. Thus, our prototype does not require us to recompile the programs that we analyze, and it handles seamlessly user and library code in the same way. RipRop is robust enough to detect ROP-based attacks in applications deployed in a production scenario. To meet such goal, RipRop gives us an *exact* view of the instructions that are processed by the hardware, including library code which is linked dynamically with the application.

Pin is a binary instrumentation framework maintained by Intel. It supports the architectures IA-32 and x86-64, and has versions that work in Android, Linux, OSX and Windows. Tools that use Pin are called *Pintools*. Our RipRop is a Pintool. We chose Pin to implement RipRop because it shows better performance than other binary instrumentation options publicly available [Bruening 2004, Nethercote and Seward 2007], and because it has a large community of users. Pin works in a just-in-time fashion: it intercepts an instruction before it is processed by the hardware; it then generates and runs new binary code, to handle that instruction; and finally, it ensures that, once the instruction is executed, the control flow will be given back to the resident pintool. Notice that, even though the monitored application, the resident pintool, and the Pin framework itself share the same address space, none of these processes share library code. In this way, Pin avoids undesired interactions between itself and the application that it instruments.

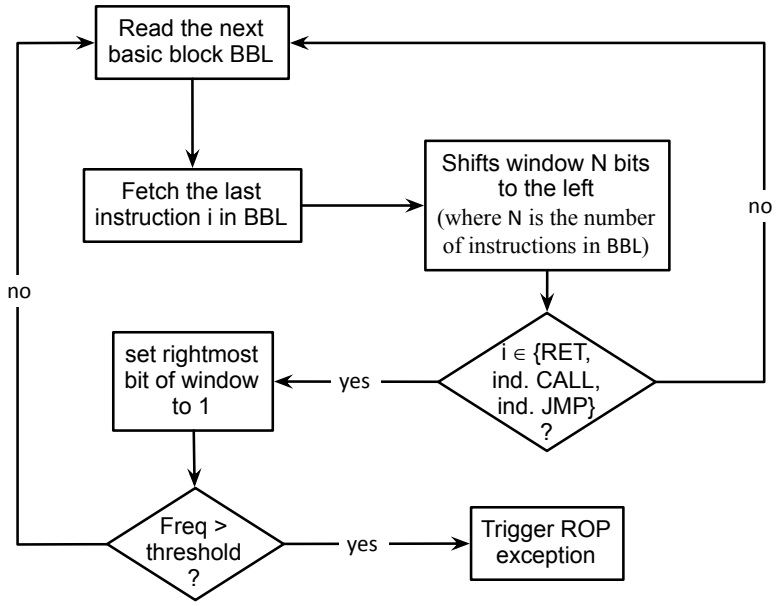


Figure 1. An overview of RipRop.

Figure 1 provides an overview of RipRop. RipRop keeps track of a window of bits that represent the last W instructions observed during the execution of a program. W is parameterizable, and in Section 4 we experiment with different values of it. Whenever we read a basic block of size N we shift the window N bits to the left, zeroing the rightmost bits. After shifting the window, we analyze the instruction at the end of the basic block. If this instruction is a function terminator, then we set the rightmost bit of the window to

one, otherwise we leave it as zero. Before moving on to read the next basic block, we check the frequency of function terminators. If the window contains a number of them that is higher than a threshold, then RipRop issues a warning that indicates a potential ROP attack. Like the size of the window, the threshold is also parameterizable.

To run RipRop, one must first download and install Pin. RipRop is then executed alongside with Pin, via command line, giving as argument the name of the application that shall be monitored. At 3:28s in the video[1], we show how to execute our tool to monitor a music player program.

As a drawback, the application under monitoring is slowered down: since each and every basic block must be processed by our Pintool before actual execution, RipRop adds an overhead to the original performance. Despite this, the running time of programs being monitored by RipRop is feasible, as we show in the video and present in the experimental results available in Section 4.

3. Core Insights

ROPs are built around sets of *gadgets*. A gadget is a small sequence of instructions that ends with an indirect branch (IB). In the x86 architecture, for instance, IBs are return instructions (RETs), indirect jumps or indirect call instructions. As an example of attack, Shacham *et al.* [Shacham et al. 2004, p.299] shows how to chain these gadgets to overcame a protection known as *Write⊕Execute*, which marks memory addresses where data is stored as non-executable. A common trait among gadgets observed in practice is their size: they tend to be small. In general, ROP-based exploits use gadgets with 1-3 instructions only [Yuan et al. 2011]. When crafting an exploit, the attacker can only build gadgets out of sequences of instructions that are already part of the program code. Long sequences may create undesirable side effects, such as overwriting registers or the stack in unwanted ways. The contents of the stack need to be carefully prepared, so that a ROP-based attack can be successful. The probability that a store operation changes this content, thus breaking the progress of the attack, is non-negligible.

3.1. The Key Observation

As mentioned before, the gadgets tend to be formed by small sequences of instructions. Therefore, the *density* of IBs during the execution of a ROP-based exploit is likely to be high. We measure the density of IBs as the number of such instructions within a given *window*. In our context, a window is a sequence of instructions in the program's execution trace. If we consider, for instance, the last 32 instructions executed by a program under attack, then gadgets containing between 1 and 3 instructions would give us a density of at least 10 IBs in a 32-instruction window.

The table in Figure 2 shows numbers taken from 15 actual ROP-based attacks publicly available in the Exploit Database [Anonymous 2014]. We report the number of instructions that are part of the gadgets used in the exploit, and we report the number of IBs present in this sequence. This last value equals the number of gadgets themselves. These two quantities are readily available in the code of the exploits. In each one of these exploits, a window of 32 instructions would register, at its peak, a high number of IBs. We

[1] `https://youtu.be/EouKW1UUutU?t=3m28s`

Application	Instructions in Exploit	Function Terminators	Density
	Windows		
Wireshark*	49	24	0.49
DVD X Player*	62	28	0.45
Zinf Audio Player*	95	38	0.4
D. R. Audio Converter*	69	36	0.52
Firefox - use aft free*	51	21	0.41
Firefox - integer ovf*	36	18	0.5
PHP	53	21	0.40
AoA Audio Extractor	212	81	0.38
ASX to MP3 Conv	150	59	0.39
Free CD to MP3 Conv	47	19	0.40
	Linux		
ProFTPD Debian*	66	24	0.36
ProFTPD Ubuntu*	45	19	0.42
PHP	81	27	0.33
Wireshark	69	30	0.43
NetSupport	45	14	0.31

Figure 2. Sum of instructions in the gadgets used to craft a ROP-based attack, and number of cross-IBs among these instructions. We mark with * the exploits that use, in addition to RET instructions, also indirect jumps or indirect calls. We have run all the exploits in an x86 machine.

have observed a minimum of at least 11 IBs in each example that we have tested. On the other hand, during a legitimate execution of the applications, none of them has presented more than 9 IBs per window. This fact leads us to our key observation:

Observation 3.1 *In the 15 examples of exploits evaluated, there exists a well-defined separation between the density of IBs in the exploit and in the actual execution of the application.*

3.2. A Brief Discussion about False Positives

If we assume that we have a ROP-based attack in course whenever we observe a high-density of IBs, then false positives may happen if such density is achieved by the legitimate execution of code. We studied four situations in which we can expect a high density of IBs: (i) loops with very large bodies; (ii) invocation of short functions within loops having small bodies; (iii) bytecode interpreters and (iv) recursive function calls. We have designed and implemented a series of micro-benchmarks that exercises each one of these scenarios. Our toils let us conclude that even under these extreme situations we can expect a relatively low density of IBs. Further, using both `gcc` and MS Visual Studio 2013, we have not been able to produce a binary that yields a density of one IB per each two instructions.

3.3. A Brief Discussion about False Negatives

RipRop will produce a *false-negative* if an attacker can craft a sequence of gadgets that gives us a low density of IBs. We have tried to build such sequences on top of 10 exploits

publicly available for Windows on x86. These efforts show that it is possible, although difficult, to circumvent our protection. To achieve such a purpose, we have trodden two different avenues: first, we have tried to simply increase the number of instructions after gadgets that we already knew for some of the publicly available exploits. These attempts were fruitless: the extra instructions break the exploits for the applications that we have analyzed. Second, we have tried to interpose *no-op gadgets* in between sequences of effective gadgets. A no-op gadget is a sequence of instructions ending with an IB, which does not cause side effects that could invalidate an exploit. In this case, we have been able to build effective exploits for two applications. Do these findings indicate that our defense is weak? We say: No! All the operative no-op gadgets belong into one of two categories: (i) static initialization sequences, and (ii) alignment blocks. In the former category we have long sequences of stores into fixed addresses, which are used to initialize static memory in C. In the latter we have no-op gadgets that correspond to code that the compiler inserts in-between functions to force an alignment of 16 bytes. We claim that it is possible to modify the compiler to remove these gadgets.

4. Experiments

Figure 3 shows the frequency of IBs for programs compiled with Visual Studio, running on Windows 7 in 32-bit mode. Each dot in the charts shows the maximum number of IBs observed on a sliding window of either 32, 64, 96 or 128 bits. When running the SPEC CPU programs, we have used their reference input, which is the largest set of input data that the benchmark provides. This data amounts to billions of instructions. From these charts, it is possible to draw three conclusions: (i) ROP-based exploits show higher peak density of IBs; (ii) larger window sizes tend to blur the distinction between legitimate and fraudulent executions; and (iii) at least for the benchmarks that we have tested, it is possible to determine a universal threshold that separates legitimate and illegal traces of instructions using a 32-instructions sliding window. We have observed similar behavior when running the exploits in the Linux system.

Universal vs Specific Thresholds. In our experiments, we have found a perfect distinction between legitimate and fraudulent executions in all the benchmarks that we have used, and in the two different operating systems that we have fiddled with. However, in Linux this threshold is made of only two instructions, a rather small gap. This small difference leads us to believe that, instead of searching for a universal threshold, a frequency-based mechanism to detect ROP attacks should consider thresholds that are specific to each application. It is possible to determine such thresholds either dynamically or statically. Dynamically, we can use an instrumentation framework to find out the peak frequency of IBs of each application. Statically, we can analyze the call graph of programs, to determine the shortest path between return instructions. We have already experimented with the first approach.

If we study again the chart in Figure 3, we see that only a few applications achieve an IB frequency that is close to the universal threshold. The table in Figure 4 makes this observation more explicit. We have grouped benchmarks according to their peak frequency. We notice that the peak frequency tends to be higher in the Windows OS; however, the frequency of the exploits is higher as well in this environment, as Figure 3 reveals. On the other hand, Linux, the system where we found the smallest gap between legal and illegal executions, has a very low peak ($6/32$ IBs per instruction) in almost half

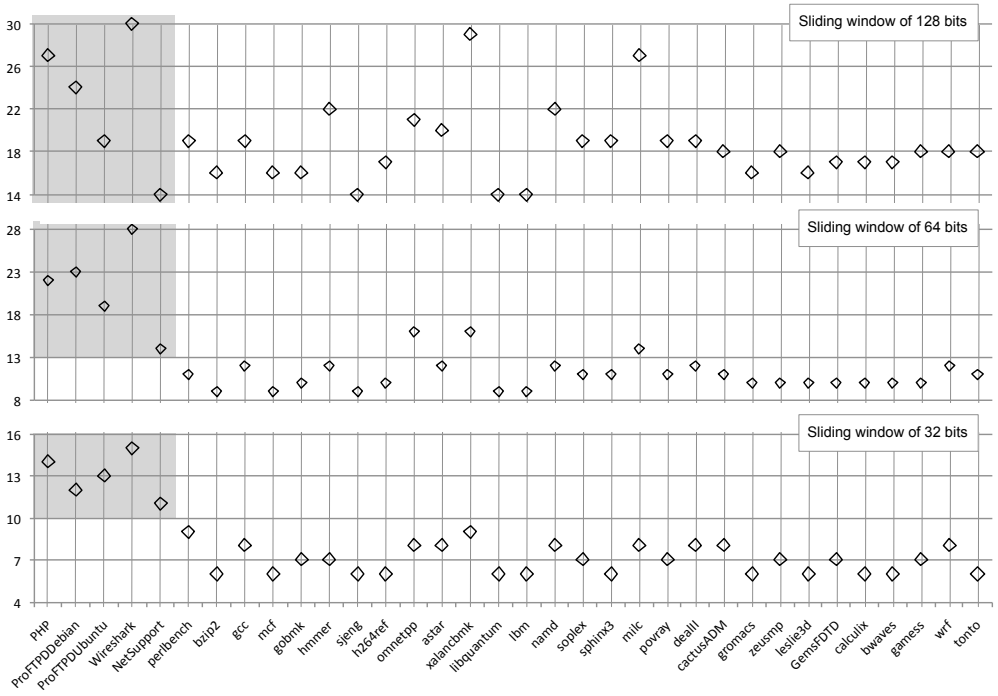


Figure 3. Maximum density of IBs in Windows 7, given different window sizes. The publicly available exploits are in the grey area.

the applications. The immediate conclusion that we draw from Figure 4 is that the most common frequency of IBs in our benchmarks is also the one that is the farthest away from the lowest peak observed in the actual exploits. Therefore, the use of a specific threshold for each application provides us with a more robust and accurate system.

Peak	Linux	Windows
10	0	11.1%
9	6.9%	88.9%
8	27.6%	0
7	24.1%	0
6	41.4%	0

Figure 4. Distribution of peak occurrences of IBs within a sliding window of 32 bits for SPEC CPU2006. The higher the peak, the higher the frequency of IBs, e.g., a peak of 10 instructions means that 10 IBs have been observed within a group of 32 successive opcodes.

Regarding the performance of applications monitored by RipRop, experiments with the SPEC CPU2006 suite of benchmarks show that our tool brings in an average overhead of 727%. This value is comparable to the overhead between 217% and 530% imposed by other protections that use dynamic code instrumentation mechanisms and unlike RipRop do not provide protection against all types of gadgets (terminated with returns, jumps or calls) [Chen et al. 2009, Davi et al. 2011]. Those overheads stem mainly from the use of dynamic instrumentation tools, which need to perform constant

context switches with the instrumented application, disassembly codes and generate instructions, all while running the instrumented application. Nevertheless, various code optimizations aggregated to RipRop assure a competitive performance. When compared to DROP [Chen et al. 2009], which also uses Pin framework, RipRop imposes a lower overhead for the two SPEC benchmarks tested by the authors of DROP. In experiments with bzip2, DROP resulted in a computational cost of 1540%, considerably higher than the 809% recorded with RipRop. In tests with gcc DROP imposed an overhead of 960% while RipRop raised the CPU time 729%.

5. Related Work

There are strategies to identify ROP-based attacks that are similar to ours. The closest works are due to Chen *et al.* [Chen et al. 2009], Davi *et al.* [Davi et al. 2009], Pappas *et al.* [Pappas et al. 2013], and - more recently - Cheng *et al.* [Cheng et al. 2014]. These researchers propose to monitor the sequence of instructions that a program produces during its execution, flagging as exploits instruction streams that show a high frequency of RET operations. For instance, *Chen et al.* [Chen et al. 2009] fire warnings if they observe sequences of three RET operations and if each is separated from the others by five or less instructions. This approach yields more false positives than our sliding window: Chen *et al.* have reported one false alarm in a smaller corpus of benchmarks than the one we have used in this paper. Davi *et al.* have not implemented their approach, but discuss four situations that lead to false positives.

Pappas *et al* [Pappas et al. 2013] and Cheng *et al.* [Cheng et al. 2014] use the Last Branch Record (LBR) registers to detect chains of gadgets. LBR refers to a collection of register pairs that store the source and destination addresses of recently executed branches. They are present in Intel Core 2, Intel Xeon and Intel Atom. ARM has similar capabilities in some processors. We propose a different approach, which consists of a sliding window. We believe that our strategy may be impemented in hardware easier and faster than using the LBR approach. Carline *et al.* [Carlini and Wagner 2014] and Goktas *et al.* [Göktas et al. 2014] propose ways to bypass the LBR-based defense. However, we do not believe that these strategies work easily against our protection mechanism. For attackers to succeed, they must insert no-ops between gadgets. Just adding them at the end, or after 10 gadgets, like Carline does against the defense proposed by Pappas *et al.* and Cheng *et al.* is not enough to circumvent our approach.

6. Conclusion

This paper has presented RipRop, a novel technique to detect Return-Oriented Pogramming attacks: we check the density of indirect branches in the stream of executed instructions. A sliding window implemented over Pin lets us keep track of this information efficiently. Our approach does not make ROP exploits impossible to be implemented. The recent history of construction and defeat of ROP prevention mechanisms has shown us that this goal would be rather ambitious. Instead, we propose a cheap, original and efficient method to make ROP attacks substantially more difficult to craft.

References

[Anonymous 2014] Anonymous (2014). Exploit-DB. `www.exploit-db.com/`.

[Bruening 2004] Bruening, D. L. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology.

[Carlini and Wagner 2014] Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *Security Symposium*, pages 385–399. USENIX.

[Chen et al. 2009] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In *ISS*, pages 163–177. IEEE.

[Cheng et al. 2014] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). Ropecker: A generic and practical approach for defending against ROP attacks. In *NDSS*. Internet Society.

[Davi et al. 2011] Davi, L., Dmitrienko, A., and Egele, M. (2011). ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, pages 1–21.

[Davi et al. 2009] Davi, L., Sadeghi, A., and Winandy, M. (2009). Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *WSTC*, pages 49–54.

[Göktas et al. 2014] Göktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Security Symposium*, pages 417–432. USENIX.

[Henning 2006] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

[Lin et al. 2010] Lin, Z., Zhang, X., and Xu, D. (2010). Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *DSN*, pages 281–290. IEEE.

[Luk et al. 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM.

[Nethercote and Seward 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM.

[Pappas et al. 2013] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *SEC*, pages 447–462. USENIX.

[Roemer and Buchanan 2012] Roemer, R. and Buchanan, E. (2012). Return-oriented programming: Systems, languages and applications. *Trans. Inf. Syst. Secur.*, V.

[Shacham 2007] Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552–561. ACM.

[Shacham et al. 2004] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298–307. ACM.

[Yuan et al. 2011] Yuan, L., Xing, W., Chen, H., and Zang, B. (2011). Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *APSys*, pages 6:1–6:5. ACM.

ArchRuby: Conformidade e Visualização Arquitetural em Linguagens Dinâmicas

Sergio Miranda[1], **Marco Tulio Valente**[1], **Ricardo Terra**[2]

[1]Universidade Federal de Minas Gerais, Belo Horizonte, Brasil

[2]Universidade Federal de Lavras, Lavras, Brasil

{sergio.miranda,mtov}@dcc.ufmg.br, terra@dcc.ufla.br

Resumo. *Erosão arquitetural é um problema recorrente na evolução de software. Isso se agrava em sistemas desenvolvidos em linguagens dinamicamente tipadas devido (i) a alguns recursos providos por tais linguagens tornarem desenvolvedores mais propícios a quebrar a arquitetura planejada, e (ii) a comunidade de desenvolvedores sofrer da falta de ferramentas voltadas a propósitos arquiteturais. Assim, este artigo apresenta ArchRuby, uma ferramenta de conformidade e visualização arquitetural baseada em técnicas de análise estática de código e em uma heurística de inferência de tipo para sistemas desenvolvidos em linguagem Ruby. A ideia central é prover à comunidade de desenvolvedores formas de controlar o processo de erosão arquitetural através da detecção de violações arquiteturais e da visualização do modelo de alto nível da arquitetura implementada.*

Vídeo da ferramenta. $https://youtu.be/iltehaohgew$

1. Introdução

Arquitetura de software geralmente define como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir [5]. No entanto, no decorrer do projeto – devido a falta de conhecimento, prazos curtos, etc. – esses padrões tendem a se degradar fazendo com que benefícios proporcionados por um projeto arquitetural (manutenibilidade, escalabilidade, portabilidade, etc.) sejam anulados [6, 4].

Isso se agrava em linguagens dinamicamente tipadas por duas principais razões: (i) alguns recursos providos por tais linguagens (e.g., invocações dinâmicas, construções dinâmicas, *eval*, etc.) tornam os desenvolvedores mais propícios a quebrar a arquitetura planejada, e (ii) a comunidade de desenvolvedores sofre da falta de ferramentas voltadas a propósitos arquiteturais. De fato, este estudo é centrado na conjectura de que sistemas desenvolvidos em linguagens dinâmicas podem ter um projeto arquitetural bem definido.

Este artigo apresenta `ArchRuby`, uma ferramenta que provê conformidade e visualização arquitetural baseada em técnicas de análise estática de código e em uma heurística de inferência de tipo para sistemas Ruby [3]. Este documento estende [3] detalhando a arquitetura da ferramenta e um exemplo de uso utilizando o próprio `ArchRuby` como sistema alvo. O objetivo é prover à comunidade de desenvolvedores uma forma simples e objetiva de controlar a erosão arquitetural mediante a detecção de violações arquiteturais e da visualização do modelo de alto nível da arquitetura implementada.

O restante desse artigo está organizado como descrito a seguir. A Seção 2 provê uma visão geral da ferramenta `ArchRuby` apresentando um exemplo de uso completo que envolve as principais funcionalidades da ferramenta (2.1), limitações (2.2), interface e arquitetura (2.3)

e resultados da avaliação com dois sistemas reais (2.4). A Seção 3 discute ferramentas relacionadas e a Seção 4 apresenta as considerações finais.

2. A Ferramenta `ArchRuby`

`ArchRuby` é uma ferramenta de conformidade e visualização arquitetural baseada em técnicas de análise estática de código e uma inferência de tipos para sistemas Ruby, uma linguagem dinamicamente tipada. A ideia central é detectar violações arquiteturais (conformidade) e prover uma visão de alto nível da arquitetura (visualização).

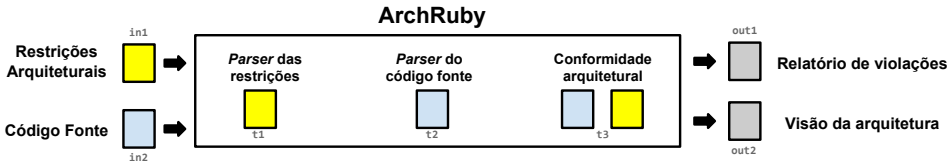


Figura 1. Funcionamento da ferramenta `ArchRuby`

A Figura 1 ilustra o funcionamento da ferramenta. `ArchRuby` recebe como entrada apenas o arquivo de especificação das restrições arquiteturais e código fonte do sistema. Após o *parse* das restrições arquiteturais e do código fonte, a ferramenta realiza o processo de conformação arquitetural, i.e., a detecção de decisões de implementação não coerentes com a arquitetura planejada do sistema alvo. Como resultado, `ArchRuby` apresenta um relatório textual reportando e detalhando as violações arquiteturais detectadas (localização, restrição, etc.). Além disso, apresenta uma visão de alto nível da arquitetura implementada.

2.1. Exemplo de Uso

Sistema Motivador: A própria ferramenta `ArchRuby`[1] será utilizada para ilustrar o funcionamento da solução proposta. A ferramenta foi desenvolvida em Ruby e utiliza dois Gems[2] para auxiliar na implementação: Ruby Parser para auxiliar no `parser` do código fonte e Graphviz para desenhar o modelo de alto nível da arquitetura do sistema analisado. A Figura 2 ilustra o diagrama das classes mais importantes do sistema.

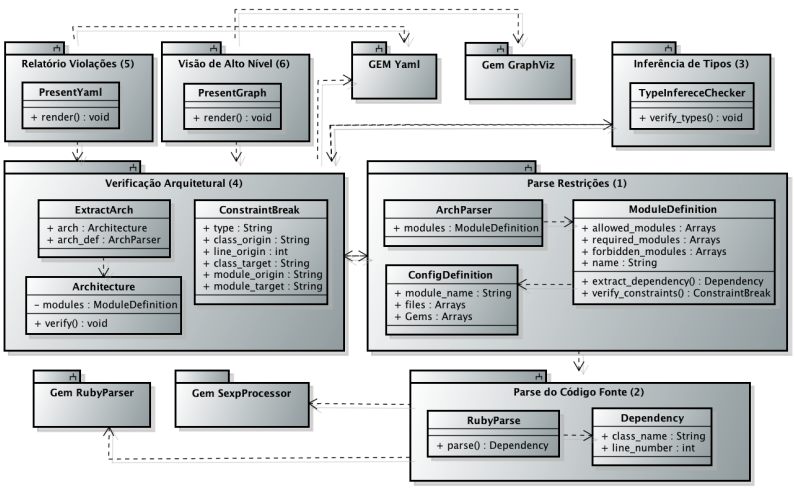


Figura 2. Arquitetura do `ArchRuby`

[1]O código fonte do sistema está disponível em `http://github.com/sergiotp/archruby`.
[2]Gem representa um pacote ou uma aplicação reusável escrita na linguagem Ruby.

Especificação de Regras Arquiteturais: As regras arquiteturais são especificadas em uma linguagem de domínio específico em YAML, formato largamente utilizado no ecossistema Ruby. Logo, a tarefa de definição de regras pode ser facilmente realizada por um desenvolvedor pleno/sênior do sistema. Para ilustrar uma especificação YAML, a Figura 3 reporta a definição de regras arquiteturais para o sistema `ArchRuby`. Por exemplo, o módulo `module_definition` é formado pelo arquivo `module_definition.rb` e *pode* depender de classes do módulo `config_definition`, `ruby_parser`, `dependency`, `constraint_break` e `file_extractor`. Por outro lado, o módulo `multiple_constraints_validator` é formado pelo arquivo `archruby.rb` e *não* pode depender de classes do módulo `architecture`. É importante ressaltar que módulos formados por Gems não definem restrições arquiteturais (e.g., `sexp_processor`, `yaml_parser`, `graphviz` e `parser_ruby`), pois não são partes integrantes (i.e., internas) do sistema alvo. Por restrições de espaço, a descrição completa da especificação, incluindo a formalização em Extended Backus-Naur Form, pode ser encontrada em [3].

```
config_definition:
  files: 'lib/archruby/architecture/config_definition.rb'

module_definition:
  files: 'lib/archruby/architecture/module_definition.rb'
  allowed: 'config_definition, ruby_parser, dependency, constraint_break, file_extractor'

architecture:
  files: 'lib/archruby/architecture/architecture.rb'

architecture_parser:
  files: 'lib/archruby/architecture/parser.rb'
  allowed: 'config_definition, module_definition, type_inference, yaml_parser'

constraint_break:
  files: 'lib/archruby/architecture/constraint_break.rb'

dependency:
  files: 'lib/archruby/architecture/dependency.rb'

type_inference:
  files: 'lib/archruby/architecture/type_inference_checker.rb'

presenters:
  files: 'lib/archruby/presenters/**/*.rb'
  allowed: 'architecture, graphviz'

ruby_parser:
  files: 'lib/archruby/ruby/parser.rb'
  allowed: 'dependency'
  required: 'parser_ruby, sexp_processor'

sexp_processor:
  gems: 'SexpInterpreter'

yaml_parser:
  gems: 'YAML'

graphviz:
  gems: 'GraphViz'

parser_ruby:
  gems: 'RubyParser'

file_extractor:
  files: 'lib/archruby/architecture/file_content.rb'

multiple_constraints_validator:
  files: 'lib/archruby.rb'
  forbidden: 'architecture'
```

Figura 3. Regras arquiteturais `ArchRuby`

Conformidade Arquitetural: Nesta etapa, a ferramenta (i) extrai os módulos alvo e suas restrições arquiteturais pelo *parse* da especificação YAML; (ii) extrai o grafo de dependências de todo o sistema; e (iii) verifica se as dependências extraídas no passo ii seguem as restrições impostas no passo i. O processo de conformidade arquitetural gera como saída um relatório reportando todas as violações arquiteturais encontradas

(ausências e divergências). Considere as regras arquiteturais especificadas para o sistema ArchRuby (Figura 3). Nessa especificação, o módulo module_definition não pode depender do módulo type_inference (linha 6). No entanto, assuma que uma classe do módulo module_definition acesse um método da classe do módulo type_inference. Tal dependência representa uma divergência arquitetural que será reportada ao desenvolvedor conforme ilustrado na Figura 4.[3]

```
1 divergence:
2   origin_module: module_definition
3   origin_class: Archruby::Architecture::ModuleDefinition
4   origin_line: 29
5   target_module: type_inference
6   target_class: Archruby::Architecture::TypeInferenceChecker
7   constraint: module 'module_definition' cannot depend on module 'type_inference'
```

Figura 4. Reporte textual de uma violação arquitetural

Inferência de Tipos: É importante mencionar que todo o processo de conformidade arquitetural é baseado em técnicas de análise estática de código. Portanto, a ferramenta não é capaz de detectar divergências geradas por meio de construções dinâmicas, tais como, reflexão (e.g., Kernel.const_get().new|send) e avaliação dinâmica de código (eval). A maior limitação seria, na prática, a identificação dos tipos em linguagens dinâmicas. Embora dinamicamente tipada, a linguagem alvo da ferramenta é fortemente tipada, logo é possível inferir em Ruby parte dos tipos de variáveis e parâmetros formais. Para isso, foi proposta uma heurística [3] – mais especificamente, uma simplificação da heurística formalizada por Furr et al. [2] – que visa construir um conjunto TYPES de triplas $[\texttt{method}, \texttt{var\_name}, \texttt{type}]$, onde type é um dos possíveis tipos inferidos para a variável ou parâmetro formal var_name do método method. Esse conjunto é construído com a seguinte definição recursiva:

i) *Base*: Para cada inferência direta (e.g., instanciação) de um tipo T de uma variável x em um método f, então $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$.

ii) *Passo recursivo*: Se $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$ e existir em f uma chamada $\texttt{g}(\texttt{x})$, então $[\texttt{g}, \texttt{y}, \texttt{T}] \in \texttt{TYPES}$, onde y é o nome do parâmetro formal de g.

iii) *Fechamento*: $[\texttt{method}, \texttt{var\_name}, \texttt{type}] \in \texttt{TYPES}$ somente se puder ser obtido a partir de (i) com um número finito de aplicações de (ii).

Embora pareça contrassenso, não há problema algum em usar análise estática em linguagens dinamicamente tipadas. Tal decisão foi tomada com o intuito de verificar a conformidade arquitetural sem a necessidade de execução. Como trabalho futuro, a precisão desta heurística está sendo analisada e será reportada em trabalho futuro.

Visualização Arquitetural: Após o processo de conformidade arquitetural, ArchRuby apresenta uma visão de alto nível da arquitetura inspirado no modelo de reflexão proposto por Murphy et al. [4]. O modelo de alto nível é um grafo de dependências orientado, onde os vértices representam os módulos definidos na especificação YAML e as arestas representam as dependências estabelecidas entre os módulos, as quais são diferenciadas quando se tratar de violações arquiteturais. A Figura 5 ilustra o modelo de alto nível do sistema ArchRuby. Os vértices em retângulos na cor cinza claro representam módulos internos do sistema (e.g., module_definition) e vértices em trapézios na cor cinza escuro representam módulos externos do sistema (e.g., parser_ruby).

[3]O relatório de violações também está em formato YAML para facilitar o reúso.

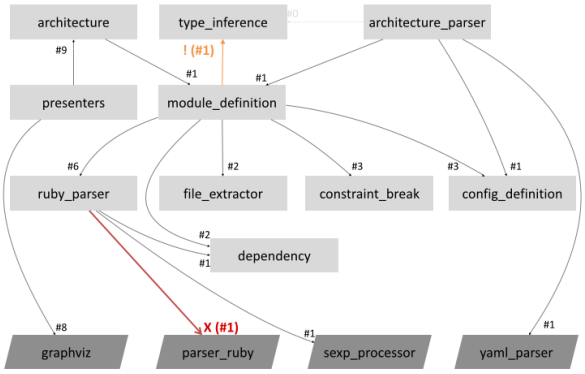


Figura 5. Visualização automática do `ArchRuby` provida pelo `ArchRuby`

Arestas na cor preta indicam dependências permitidas (*allowed*) entre módulos. Por exemplo, o módulo `ruby_parser` estabelece uma (#1) dependência com o módulo `dependency` (veja linha 30, Figura 3). Arestas na cor vermelha indicam violações do tipo ausência. Por exemplo, existe uma classe do módulo `ruby_parser` que não estabelece dependência com o módulo `parser_ruby`, mesmo tal dependência sendo obrigatória (*required*, veja linha 31, Figura 3). Arestas na cor laranja indicam uma violação do tipo divergência. Por exemplo, assuma que o módulo `module_definition` depende do módulo `type_inference`, mas tal dependência não é permitida (veja linha 6, Figura 3). Arestas na cor cinza não indicam violações, mas sim, um alerta de não existem dependências entre módulos que supostamente deveriam ter, i.e., foram prescritas como permitidas (*allowed*). Por exemplo, foi explicitamente permitido que o módulo `architecture_parser` dependa do módulo `type_inference` (veja linha 13, Figura 3), porém tal dependência não é estabelecida.

2.2. Interface e Arquitetura

`ArchRuby` é um Gem para Ruby que implementa a solução proposta. A execução da ferramenta ocorre por linha de comando, a fim de permitir que qualquer organização – independentemente do ambiente de desenvolvimento adotado – possa incorporar a ferramenta em seu processo de desenvolvimento. Um exemplo pode ser visto a seguir:

```
archruby --arch_def_file=/fmot/arch_def.yml --app_root_path=/fmot
```

O executável `archruby` requer como entrada o caminho do arquivo de restrições arquiteturais (`--arch_def_file`) e o caminho do sistema (`--app_root_path`) para prover como saída o relatório de violações (`archruby_report.yml`) e uma visão de alto nível da arquitetura (`archruby_architecture.png`), conforme previamente ilustrado na Figura 1. A implementação segue uma arquitetura dividida nos seguintes módulos, os quais estão ilustrados na Figura 2:

1. *Parser das restrições*: Responsável por extrair e armazenar o conteúdo do arquivo de restrições (e.g., `/fmot/arch_def.yml`) em uma estrutura de dados interna para consultas posteriores. Caso o usuário entre com restrições inválidas, por exemplo, `allowed` e `forbidden` em conjunto, é papel desse módulo avisá-lo. O *parser* do arquivo YAML é realizado pela Gem `YAML` padrão da linguagem Ruby.
2. *Parser do código fonte*: Responsável por extrair e armazenar todas as dependências do sistema (e.g., `/fmot`) em uma estrutura de dados interna para consultas posteriores. O *parser* do código fonte de cada classe é realizado pelo Gem `ruby_parser`,

o qual gera suas saídas em *s*-expressions, uma estrutura em forma de árvore. Ao percorrer essa árvore, os tipos de cada variável pertencente a algum método é armazenado em uma tabela interna para consulta posterior. Ainda, todos os tipos dos parâmetros formais e possíveis invocações de métodos utilizando os parâmetros formais como argumentos também são armazenados.

3. *Inferência de Tipos*: Responsável por inferir os tipos das variáveis utilizadas, de acordo com o algoritmo descrito na Seção 2.1.
4. *Verificação arquitetural*: Responsável por verificar se a arquitetura implementada (código fonte) segue a arquitetura planejada (restrições arquiteturais). O objetivo é identificar dependências que *não* respeitam as restrições arquiteturais e, caso detectadas, armazenar as informações detalhadas de tais violações. Para a detecção das dependências que não respeitam as restrições arquiteturais é feito o uso das informações coletadas pelo *parse* do código fonte e pela heurística de inferência de tipos. Ou seja, toda a tabela gerada nos dois passos anteriores é percorrida, extraindo as informações armazenadas, para realizar a verificação de possíveis violações. Ao encontrar uma violação, a ferramenta armazena informações detalhadas – e.g., nome do módulo que está violando a regra, nome do módulo que é proibido acessar, número da linha, nome da classe, classe de destino, etc. (veja Figura 2, classe `ConstraintBreak`). – para consulta posterior.
5. *Geração do relatório de violações*: Responsável por estruturar os dados das violações arquiteturais detectadas em um arquivo no formato YAML (`archruby_report.yml`). O relatório apresentado mostra as informações geradas pelo processo de verificação arquitetural reportando as violações encontradas.
6. *Geração da visão de alto nível da arquitetura*: Responsável pela geração da visão arquitetural do sistema alvo. O grafo é gerado por meio do Gem `Ruby-Graphviz`. O grafo de dependência gerado pelo *parse* do código fonte é utilizado nessa etapa para gerar a figura de alto nível da arquitetura implementada.

Os módulos previamente descritos possuem mais de uma classe para realizar suas tarefas. Através dessa separação, é possível ter um maior controle sobre as partes integrantes do sistema, facilitando assim a adição de novas funcionalidades e manutenção em funcionalidades existentes. Ainda, a ferramenta possui testes automatizados para garantir que, ao serem feitos manutenções na ferramenta, as alterações não quebrem o comportamento esperado do sistema. Todas as dependências (`ruby_parser`, `Ruby-Graphviz`), que não fazem parte da biblioteca padrão do Ruby, citadas anteriormente são instaladas de forma automática quando o usuário realizar a instalação do Gem Archruby.

2.3. Avaliação

Em um artigo recente [3], `ArchRuby` foi avaliada em dois sistemas reais[4]: `Tim Beta`, que é um canal de comunicação da TIM com o público jovem; e `Dito Social`, plataforma social da Dito para atender aos seus clientes. Tabela 1 reporta principais informações dos sistemas.

Metodologia: Os arquitetos responsáveis definiram as restrições arquiteturais de cada sistema. Para o `Tim Beta`, definiu-se 43 módulos e 7 restrições arquiteturais. Para o `Dito Social`, definiu-se 62 módulos e 43 restrições arquiteturais. É impraticável em termos de tempo despendido verificar todas essas restrições manualmente. Assim, a partir de tais

[4] `http://www.timbeta.com.br` e `http://www.dito.com.br`

Tabela 1. Sistemas avaliados com `ArchRuby`

	`Tim Beta`	`Dito Social`
LOC	17.817	13.304
# classes	141	142
# gems	50	34
Principais Tecnologias	Ruby On Rails, Resque, Twitter, YoutubeIt, Google Plus, Instagram, Devise, Foursquare2	Ruby on Rails, Resque, Rspec, RSA, Twitter, Google Plus, Koala, Suspot Rails, Mysql2

especificações, `ArchRuby` foi aplicada nos sistemas. Uma descrição completa da metodologia utilizada pode ser encontrada em [3].

Conformidade Arquitetural: `ArchRuby` pôde detectar 22 e 24 violações nos sistemas `Tim Beta` e `Dito Social`, respectivamente. Como um exemplo, dez divergências foram detectadas no `Tim Beta` por classes de modelo acessando classes responsáveis pela comunicação com a API do Orkut, mesmo não mais existindo a integração com a plataforma Orkut. Similarmente, cinco divergências foram detectadas no `Dito Social` por classes de modelo acessando classes responsáveis por envio de e-mails, mesmo não mais existindo tal funcionalidade. Como um outro exemplo, uma ausência foi detectada no `Dito Social` por uma classe de modelo não estabelecer dependência com o Gem nativo de persistência do *framework* Rails (similar violação foi simulada no sistema motivador `FindMeOnTwitter`).

Discussão: É importante destacar alguns pontos: (i) os arquitetos tiveram que refinar as restrições arquiteturais para que violações apontadas por `ArchRuby` fossem de fato verdadeiros positivos; (ii) o arquiteto do `Tim Beta` fez a especificação arquitetural de mais alto nível, assim justificando o número de apenas sete restrições arquiteturais; (iii) foi detectado um maior número de violações do tipo divergência em ambos os sistemas, i.e., existe uma tendência dos desenvolvedores estabelecerem comunicação com módulos não permitidos pela arquitetura; e (iv) os arquitetos não tinham prévio conhecimento das violações arquiteturais apontadas pela ferramenta e alegaram que tais violações impactam negativamente a manutenibilidade dos sistemas.

3. Ferramentas Relacionadas

No nosso melhor conhecimento, não existe uma ferramenta de conformidade e visualização arquitetural como a proposta neste artigo no ecossistema Ruby. `DCLsuite` é uma ferramenta de conformidade e reparação arquitetural para sistemas Java, em que nossa ferramenta foi inspirada [7, 8]. A partir de uma especificação arquitetural na linguagem DCL, a ferramenta detecta violações arquiteturais e ainda provê sugestões de como resolvê-las. `ArchRuby`, por sua vez, implementa uma heurística de inferência de tipo por ser voltada a uma linguagem dinamicamente tipada, permite a especificação de restrições arquiteturais em arquivos YAML e provê uma visão de alto nível da arquitetura implementada, embora não contemple técnicas de reparação arquitetural [3].

As seguintes ferramentas têm o intuito de aumentar a qualidade de sistemas de software através de técnicas de análise estática de código. `Code Climate` é uma ferramenta – com foco no *framework* Rails – que auxilia o processo de revisão de código [1]. A ferramenta reporta pontos onde o sistema pode ser melhorado, e.g., métodos complexos, potenciais falhas de segurança, oportunidades de refatoração, etc. Ainda, a ferramenta sugere literatura de consulta para que os desenvolvedores entendam melhor o problema e o processo de correção. `Rubocop` é um Gem que realiza análise estática de código em sistemas

Ruby a fim de verificar erros e regras de estilo. Similarmente, `LASER` e `ruby-lint` também verificam erros e regras de estilo, porém baseadas em outras heurísticas. Em outra linha, `Brakeman` é uma ferramenta voltada à detecção de vulnerabilidades em aplicações Ruby on Rails. Por fim, `Pelusa` é uma ferramenta que indica padrões e boas práticas no desenvolvimento orientado a objetos em Ruby. `ArchRuby`, por sua vez, complementa tais ferramentas uma vez que auxilia o desenvolvedor na garantia de sua arquitetura planejada.

4. Considerações Finais

A erosão arquitetural é um problema recorrente no desenvolvimento de software. Violações em relação à arquitetura planejada fazem com que o sistema se torne cada vez mais difícil de se manter e evoluir, podendo até mesmo ocasionar a reescrita de componentes. Ainda mais crítico, o processo de erosão se agrava em linguagens dinâmicas devido pois (i) os recursos dinâmicos providos por essas linguagens tornam os desenvolvedores mais propícios a violar a arquitetura planejada, e (ii) a comunidade de desenvolvedores em linguagens dinâmicas carece de ferrramentas voltadas a propósitos arquiteturais. Para mitigar esse problema, este artigo apresentou `ArchRuby`, uma ferramenta que provê formas de controlar o processo de erosão arquitetural através da detecção de violações baseada em técnicas de análise estática e em uma heurística de inferência de tipos, além da visualização do modelo de alto nível da arquitetura implementada. Como resultado prático, `ArchRuby` foi integrada ao processo de desenvolvimento adotado pela Dito Internet, empresa responsável pelos sistemas avaliados.

A ferramenta `ArchRuby` e seu código fonte estão publicamente disponíveis em: `http://aserg.labsoft.dcc.ufmg.br/archruby`

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] Blue Box and Code Climate. Code Climate. `http://codeclimate.com`, 2014.

[2] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *24th Symposium on Applied Computing (SAC)*, pages 1859–1866, 2009.

[3] Sergio Miranda, Marco Tulio Valente, and Ricardo Terra. Conformidade e visualização arquitetural em linguagens dinâmicas. In *XVIII Conferencia Iberoamericana de Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 1–14, 2015.

[4] Gail Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, 1995.

[5] David Lorge Parnas. Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.

[6] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010.

[7] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[8] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–28, 2013.

SmartHomeRiSE: An DSPL to Home Automation

Matheus Lessa G. da Silva[1], **Michelle Larissa L. Carvalho**[1], **Alcemir Rodrigues Santos**[1], **Eduardo Santana de Almeida**[1]

[1]RiSE Labs - Federal University of Bahia

Abstract. *Smart Home is considered the new trend of housing for the future. They can be defined as an intelligent residence that aims to provide services to turn the life of their inhabitants easier and safer. Moreover, they need to be efficient, fast and adaptable to the interests of residents and environmental changes. There are many works that address methods and techiniques about how to implement a smart home, however aspects as reuse, evolution and maintenability remains poorly prioritized by the existent literacture. In order to deal this issues, we develop the SmartHomeRiSE DSPL, a Dynamic Software Product Line that acts in the core of the smart home and changes its state based on user actions and environment changes. In this paper, we describe the SmartHomeRiSE DSPL and its features and architecture, as well an example of use of the tool, and a preliminary evaluation.*
Video link: http://youtu.be/QtXFgy316p8

1. Introduction

The application of domestic use technologies is dated to almost 100 years and since than it has been diversifying following the evolution of the society. This evolution creates different ways to apply the existent technology, in addition to creating never before perceived needs [Harper 2003]. In this scenario, create an smart home application depends on robust technology of hardware and software to make it able to handle with all the demands of interaction inherent in this domain.

The smart home is an intelligent environment that is able to acquire and apply knowledge about its inhabitants and their surroundings either via hardware installed in its rooms or online information [Yang et al. 2014]. In this way, a smart home can be defined as a residence endowed of computational intelligence and information technology that which anticipates and responds to the needs of the dwellers by changing its state and the environment to promote services in order to increase the comfort, convenience, security and so on [Harper 2003].

Dynamic Software Product Lines (DSPL), is an extension of the concept of Software Product Lines (SPL). Both concepts support large-scale reuse, improve the maintainability and quality of the resulting product, however the product of an DSPL have to be derivated at runtime as response to the environmental stimuli.

Based on this scenario, we developed a DSPL named SmartHomeRiSE. The application aims to provide practicality for users since it adapts itself responding both to user commands and environmental changes, *i.e.*, features are activated, deactivated, and updated dynamically at runtime to satisfy the specifics of each moment. In addition, it allow the insertion of new devices and features, without compromising the smooth operation of the house. Therefore, the variation points of the SmartHomeRiSE were defined from the

user's perspective and the needs of the context where it is inserted. The SmartHomeRiSE is open source and the source code is available on link[1].

This paper is organized as follows. Next section presents a background about SPL and more specifically about DSPL. Section 3 introduces the SmartHomeRiSE and its functioning, architecture and one application case is explained. This section also discusses an Arduino communication framework developed to connect the DSPL structure with the Arduino board. In Section 4, a preliminary evaluation is presented. Section 5 discusses related work of smart home applications. Finally, Section 6 concludes the paper and presents some future work.

2. Background

The diversification in software development has been a key issue for companies employing new engineering practices. In this way, some software industry players adopted the Software Product Line Engineering (SPLE) approach aiming faster product development with high quality and low cost [Linden et al. 2007]. The SPL approach allows that development companies supply the large demand of software systems. There are many benefits achieved from the use of SPL in software companies, much as the improvement of the process side of software development [Linden et al. 2007]. In addition, SPL supports large-scale reuse, which contributes to cost reduction, decreases the time to market and improves the quality of the resulting products.

SPL are developed based on core assets [Northrop 2002] and each member of the product family is known as a variant, which is instantiated according to their needs and rules of the common architecture. The feature variability management, that in SPL occours at compilation time, is a key concept for SPL engineering practices. However, in some cases, there is a new need of adaptation of the products where the generation of variations occurs at runtime which produces systems with higher level of automation [Talib et al. 2010].

DSPL extends the SPL concept to deal with dynamic variability [Hallsteinsen et al. 2008] and their approach allows its variants reconfigure themselves during their execution [Bencomo et al. 2008]. Besides, DSPL engineering is an appropriate way of achieving and managing the adaptations at runtime that operates in a predefined configuration space.

In DSPL, when the binding time occurs at runtime, the features are maintained regardless of whether the product will be used or not. Moreover, are produced configurable products (CP) that has autonomy to reconfigure themselves and receive constant updates. The CP is produced by the product line, as well as conventional SPL, however, in DSPL case, was needed two artifacts to control them: the decision maker and the reconfiguration rules. The former has function of capturing all the information in its environment that suggests changes, as external sensors information or information from a user for example. The latter is responsible for executing the decision by using the standard SPL runtime binding [Cetina et al. 2008].

Such properties, indicates that DSPLs can benefit from research in many other areas, such as situation monitoring and adaptive decision making that are also character-

[1]https://subversion.assembla.com/svn/risesmarthome

istics of autonomic computing.

3. The SmartHomeRiSE DSPL

This section presents the technical aspects of the SmartHomeRiSE's build. Besides, each part of the development is discussed along the section which includes the project architecture, an arduino communcation framework used as support, the DSPL feature model, reconfiguration rules and so on.

3.1. Architecture

The SmartHomeRiSE was developed in Java and manages the features variability for addition, removing and request by other features. It was also built a house scale model and attached some devices, such as, Arduino, sensors (e.g., temperature, gas and smoke, luminosity, and presence), and LEDs. These hardware are used to act in the house and simulate environmental changes to help us to explore the dynamic variability. Figure 1 shows the architecture of the SmartHomeRiSE prototype.

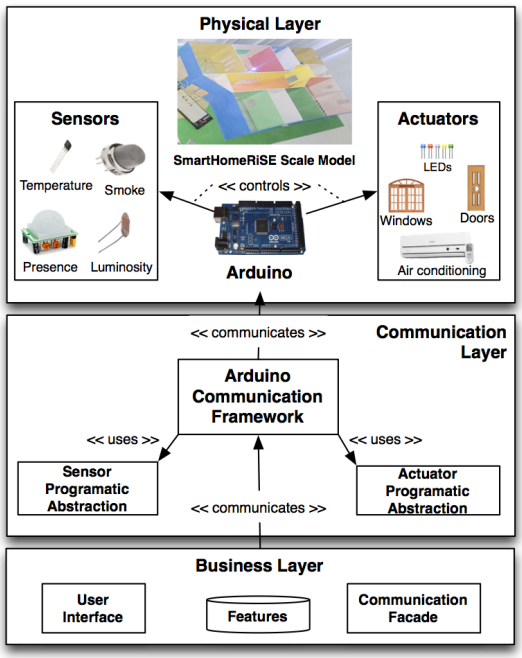


Figure 1. SmartHomeRiSE Architecture

The architecture is divided into physical layer, that comprises a scale model, Arduino, sensors, and actuators beyond the business layer where the features and user interface of the product line are inserted.

Figure 2 shows a generic model of the adopted dynamic variability control structure. It allows that the DSPL features assume the behavior modeled in the feature model and quickly react to reconfiguration triggers. Such adaptability was implemented through native Java capabilities, such as annotations, reflection and other techniques.

In this model, all features extends the `FeatureBase` abstract class, which allow them to share common properties, such as the list of required features. Such approach helps in the features identification by polymorphism and reflection. Besides, the

context-aware features implement the `AdaptableFeature` interface, which provides means to the DSPL kernel to recognize and execute them. The `MandatoryFeature` and `AlternativeFeature` annotations are used to identify mandatory and alternative features of the feature model. The former has no fields and is sufficient to categorize mandatory features. The latter allows to specify alternative features to the annotated feature. The `Optional` features are identified consequently where none of this annotations are used.

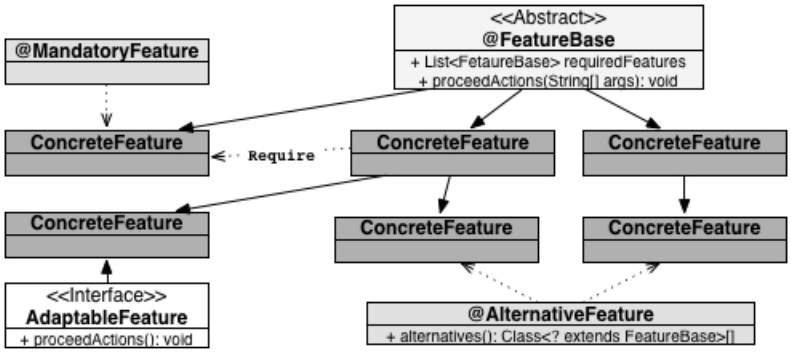


Figure 2. SmartHomeRiSE Generic Class Model

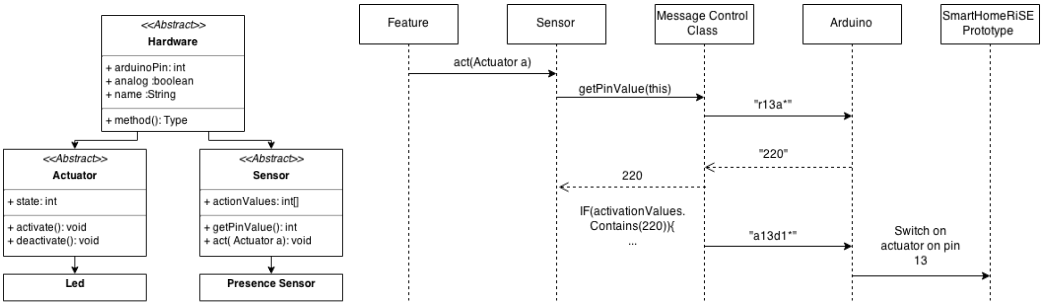
Additionally, it was necessary to develop a communication framework that allows the message exchanging between the architecture layers. This framework uses the programmatic abstractions of hardware in order to spend the information collected through Arduino for the system since it could be interpreted with the aim to control the physical sensors and actuators.

3.2. Arduino Communication Framework

The SmartHomeRiSE Arduino Communication Framework can be seen as an intermediate communication layer between the business layer and the prototype. Being written in two parts, one in Java in the main application, and another in C within the Arduino, this tool allows programmatic use of all hardware connected to the prototype.

Inside the Java application, the framework can be divided into two parts: programmatic abstraction of sensors and actuators and Arduino messages manager class.The programmatic abstraction of sensors and actuators is a hierarchical structure of classes (Figure 3(a)), which defines the type of hardware (sensor or actuator) and its actions and is the interface between the framework and features.

Figure 3(b) shows a sequence diagram of the Arduino Communication Framework. The message control class is the class responsible for absorbing all the logic of communication with Arduino and only can be seen by the hardware abstraction classes. In this arrangement, every action requested by a feature to a hardware abstraction class instance is processed and sent to the microcontroller through a standard message via USB port. In Arduino, the messages are received, decoded and appropriate action is taken besides responses can also be sent back to the message managent class.



(a) Hardware Abstraction Model (b) Sequence Diagram of the framework functioning example

Figure 3. Arduino Communication Framework

3.3. Feature Model

The SmartHomeRiSE was developed with the Object-Oriented Paradigm, and followed a feature-based fashion in order to adapt the system at runtime in accordance with variability described by a feature model. Figure 4 shows its feature model. The features are represented according to type such as: `Mandatory`, `Optional`, `Alternative`, `OR`, `Abstract`, and `Concrete`. In addition, they were managed to satisfy a range of system adaptations. Such set of features enables to explore the DSPL nature of the system and identify and manage variation points to support domain specific needs regarding users viewpoint.

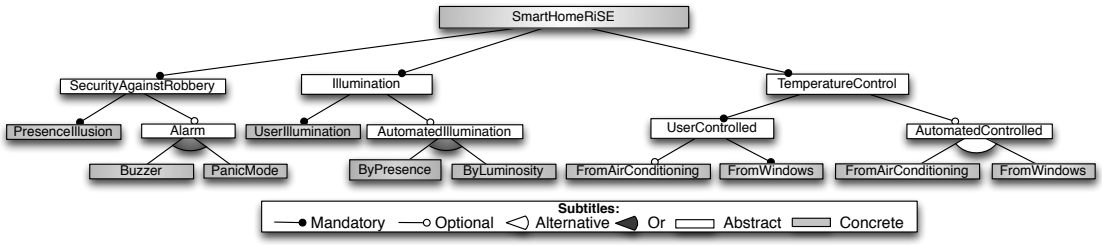


Figure 4. SmartHomeRiSE Feature Model

Regarding the adaptations at runtime of features in DSPL, they are often based on components and are described at the architectural level [Gomaa and Hussein 2004]. It allows the domain engineer to specify adaptation rules for reconfiguration components. However, it is necessary to adopt different adaptation policies to manage the variability in DSPL [Bencomo et al. 2008]. The modification when detected at the operational context actives the product reconfiguration to provide context-relevant services or collect quality request [Lee et al. 2011]. Table 1 shows some of the possible adaptations scenarios of the SmartHomeRiSE DSPL.

3.4. Tool Application

The implementation of SmartHomeRiSE addresses variability mechanisms considered essential in DSPL with the main goal of variability management and adaptation control at runtime. Although the tool has been implemented in the smart home domain, it can benefit the research in several related areas such as:

Table 1. Reconfiguration Rules

Illumination Control: The illumination system behaves according to user's action and environmental behavior.	
Rule 01:	If low luminosity is detected on the environment, the ByLuminosity feature turns on light, otherwise the feature turns off, light.
Rule 02:	If presence is detected on the environment, the *ByPresence* feature turns on light, otherwise the feature turns off light.
Rule 03:	The user turns on/off light by the *UserIllumination* feature.
Constraint:	If presence is detected on the environment with high luminosity, the *ByPresence* feature keeps deactivated.
Security Control: The security system behaves according to the user's action.	
Rule:	The *PresesenceIllusion* feature turns on/off light alternately.
Constraint:	The *PresenceIllusion* feature requires *UserIllumination* feature.
Temperature Control: The temperature system behaves according to user's action and environmental behavior.	
Rule 01:	If high temperature is detected on the environment, the *FromAirConditioning* feature (automated control) turns on the air-conditioning, otherwise the feature turns off the air-conditioning.
Rule 02:	If high temperature is detected on the environment, the *FromWindows* feature (automated control) opens the windows, otherwise the feature closes the window.
Constraint:	If the *FromWindows* feature controls actuator, the *FromAirConditioning* feature keeps deactivated.

- **Smart Buildings:** It integrates sensors into control systems for lighting, ventilation, safety (*e.g.*, fire monitoring, evacuation), and air conditioning.

- **Environmental Control:** It requires the integration of several control systems to improve life quality for users with weakness at home or hospital.

- **Wireless Sensor and Actuator Networks (WSAN):** It encompasses a set of interconnected sensors and actuators to respond to environmental changes at runtime. Particularly, the implementation of network systems by using SmartHomeRiSE also brought interest, because it requires more dynamic capabilities.

Indeed, the SmartHomeRiSE tool disposes a variety of devices that reacts to contextual informations at runtime and requires the activation/deactivation/update of features autonomously. With its use the researchers can explore aspects of dynamic reconfiguration, identify how to simplify user interaction with the system, and find solutions to integrate multiple control systems.

4. Preliminary Evaluation

In order to validate the application developed regarding the constructed DSPL model, well as to their function and performance, was applied a survey with Master and Ph.D. students. In this survey, we used a design that suggests the use of closed questions in self-administered questionnaires [Pfleeger and Kitchenham]. In order to perform a complete analysis of all developed artifacts of SmartHomeRiSE DSPL, the survey was divided into five parts: respondent identification, DSPL modeling, implementation techniques evaluation, questions about the use, and performance of the tool.

As result, all participants related that does no modeling errors, but one of them suggested that the feature model must contain more clearly the possible adaptations of the DSPL. Besides, the most experienced subject criticized the lack of explicit elements to represent environmental adaptations and reconfiguration rules. About the questions of implementations techniques, was used a scale format to judge the techniques under the aforementioned aspects. All responses were similar, and the arithmetic mean of all evaluations was around 95% of the maximum score. This result shows the good acceptance by the respondents about the used techniques and their applications in each case. Regarding the software use, all surveyed students reported that the features worked without errors and followed the constraints of the feature model. On the questions about the adaptation speed of the smart home, all respondents judged the time as satisfactory, considering the adaptations instantaneous or almost.

5. Related Work

There are many studies [Cook et al. 2003], [Yang et al. 2014], [Cetina et al.] that address issues concerning the development of smart homes. Yang et al. in his work presented an architecture based on services to achieve the problem of the interoperability among multiple appliances in home environment. In our work all the smart home application is built as an DSPL and the interoperability issue is treated by the recofiguration rules of product line, providing centralized control of the smart home operation.

Another architectural approach is proposed by Cook et al. that uses artificial intelligence to control the smart home actions based on predictive algorithms. Their findings are relevants but their most important drawback appears on the evolution and reuse aspects since that this algorithms are not sufficiently flexible and can not deal with not projected context changes. On the other hand, DSPL applications are designed to respond to unexpected changes and to keep at least minimum functions started, however the agent-based decision model can be a improvement to our DSPL based smart home.

Cetina et al. showed a RFID-based technique to specify DSPL contexts. Their work discussed key issues to identify how to manage variation points in dynamic applications. On the other hand, their approach is an insight of the application of prototyping techniques at variability modeling do not presenting details on the implementation of DSPL applications.

6. Conclusions and Future Work

In this paper, SmartHomeRiSE was presented, a DSPL that offers automation and security features applied to smart homes. We describes its architecture, feature model, reconfiguration rules, functioning and so on. Aditionaly, it was conducted a survey to evaluate the assets developed.

It was possible to realize that although noticed some lack of understanding caused by possible failures in the construction of the models that represent the system architecture and feature model, the majority of respondents considered positively both the model of DSPL and the implementation techniques used to build the tool. In addition, none of the respondents reported failures in the implementation of product line and all judged its implementation as being quite fast.

As future work, we would like to use the lessons learned with this work in order to build a guideline on DSPL implementation techniques to each product line characteristic with one or more techniques. After that, we intend refine this work with the finds obtained by the construction of the guideline and then define a general DSPL implementation framework.

References

Bencomo, N., Sawyer, P., Blair, G., and Grace, P. (2008). Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *Proc. 2nd Int'l. Work. Dynamic Software Product Lines*, pages 23–32.

Cetina, C., Giner, P., Fons, J., and Pelechano, V. Designing and prototyping dynamic software product lines: Techniques and guidelines. In *Proc. of the 14th Int'l. Conf. on Software Product Lines*, pages 331–345. Springer-Verlag.

Cetina, C., Trinidad, P., Pelechano, V., and Ruiz-Cortés, A. (2008). An architectural discussion on dspl. In *Proc. of the 2nd Work. on Dynamic Software Product Line*, page 59–68. Irish Software Engineering Research Centre.

Cook, D., Youngblood, M., Heierman, E.O., I., Gopalratnam, K., Rao, S., Litvin, A., and Khawaja, F. (2003). Mavhome: an agent-based smart home. In *Proc. of the 1st Int'l. Conf. on Pervasive Computing and Communications*.

Gomaa, H. and Hussein, M. (2004). Software reconfiguration patterns for dynamic evolution of software architectures. In *In Proceedings of the Fourth Working IEE/IFIP Conference on Software Architecture*, pages 79–88. IEEE.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). Dynamic software product lines. *Computer*, 41:93–95.

Harper, R. (2003). *Inside the smart home*. Springer Science & Business Media.

Lee, J., Whittle, J., and Storz, O. (2011). Bio-inspired mechanisms for coordinating multiple instances of a service feature in dynamic software product lines. *Journal of Universal Computer Science*, 17:670–683.

Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Northrop, L. M. (2002). Sei's software product line tenets. *IEEE*, 19(4):32–40.

Pfleeger, S. L. and Kitchenham, B. A. Principles of survey research: Part 1: Turning lemons into lemonade. *SIGSOFT Softw. Eng. Notes*, 26(6):16–18.

Talib, M. A., Nguyen, T., Colman, A. W., and Han, J. (2010). Requirements for evolvable dynamic software product lines. In *Proc. of the 4th Work. on Dynamic Software Product Lines*, pages 43–46.

Yang, C., Yuan, B., Tian, Y., Feng, Z., and Mao, W. (2014). A smart home architecture based on resource name service. In *Proc. of the 17th Int'l. Conf. on Computational Science and Engineering*.

SACRES 2.0: Suporte à Recomendação de Configurações considerando Múltiplos *Stakeholders*

Lucas Lazzari Tomasi[1], **Jacob Stein**[1], **Ingrid Nunes**[1]

[1]Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brasil

{lltomasi,jacob.stein,ingridnunes}@inf.ufrgs.br

Abstract. *Feature models (FMs) are used to represent and organize the variability in a software product line, and the task of configuring a FM in order to generate a valid product configuration thus becomes a key activity. This task, which is known to be hard and time consuming, gets even harder when many stakeholders are involved in the process of configuration. The tool presented in this paper has the purpose of supporting the configuration process with multiple stakeholders, recommending optimal configurations based on their preferences.*

Resumo. *Feature Models (FMs) são usados para representar e organizar a variabilidade em uma linha de produto de software, e a tarefa de configurar um FM de modo a gerar a uma configuração de produto válida torna-se, portanto, uma atividade chave. Essa tarefa, conhecida por ser difícil e demorada, torna-se ainda mais complicada quando múltiplos stakeholders estão envolvidos no processo de configuração. A ferramenta apresentada neste artigo tem o propósito de dar suporte ao processo de configuração com múltiplos stakeholders, recomendando configurações ótimas baseadas em suas preferências.*
URL do video: `http://www.youtube.com/watch?v=HeHT8JXlgR4`.

1. Introdução

O reuso de componentes não é um conceito novo, é uma das estratégias usadas para a redução de custos e esforço no desenvolvimento de sistemas de software. As linhas de produto surgiram com esta motivação, sistematizando o reuso, criando famílias de produtos com características comuns entre si, permitindo um aumento de produção, mas com a possibilidade de customização. Na indústria de software a combinação dos conceitos de famílias de produtos e customização em massa deram origem às chamadas de Linhas de Produto de Software (LPS) [Clements and Northrop 2002]. Segundo Linden e Pohl (2005), LPS referem-se às técnicas de engenharia para a criação de sistemas de software similares a partir de um conjunto compartilhado de partes, através de uma forma sistemática de construção de aplicações. Por meio da organização e sistematização do reuso é possível diminuir custos de desenvolvimento, melhorar a qualidade do software e reduzir o tempo de produção.

Nas LPS, o uso de *feature models* (modelos de *features*) destaca-se como uma forma de representar e gerenciar a variabilidade dentro de uma LPS. Essa variabilidade é expressa por meio de *features*. Uma *feature* representa uma característica ou propriedade do sistema que é relevante para algum *stakeholder* [Czarnecki and Eisenecker 2000]. O

Feature Model (FM) é uma representação gráfica em forma de árvore que permite visualizar as features disponíveis de serem selecionadas de forma a criar um novo produto customizado [Kang et al. 1990]. O FM modela as propriedades comuns e variáveis dos produtos possíveis de uma LPS, incluindo suas interdependências. Também é possível a definição de restrições entre certas combinações de *features*, quando for necessário que duas sejam mutualmente exclusivas, por exemplo.

A partir de um FM definido é possível a criação de configurações de produtos. Uma configuração de FM (i.e. configuração de um produto) é a seleção de um conjunto de *features* contidas nesse modelo. Quando esta seleção de *features* respeita as restrições presentes no modelo ela é chamada de configuração válida. Dado que FMs podem ter um grande número de *features* disponíveis, temos um número exponencial de configurações possíveis. Desta forma, escolher um conjunto válido de *features*, satisfazendo as restrições do FM e as preferências de *múltiplos* stakeholders torna-se uma tarefa nada trivial, exigindo suporte computacional [Mendonca and Cowan 2010].

Quando há um número maior de indivíduos envolvidos no processo de configuração de um mesmo produto torna-se complicado de encontrar um consenso para determinar quais *features* devem estar presentes no produto final, de forma que os participantes do processo fiquem suficientemente satisfeitos. É perceptível a necessidade de uma ferramenta para dar suporte ao processo e aos participantes. Portanto, apresentamos neste artigo o SACRES 2.0, um plugin para a IDE Eclipse que implementa uma nova abordagem [Stein et al. 2014] para dar suporte à configuração de FMs com base nas preferências de múltiplos *stakeholders*.

Este artigo está organizado da seguinte forma. A Seção 2 descreve o problema, exemplificando os tipos de conflitos endereçados em nossa ferramenta. A Seção 3 apresenta detalhadamente a ferramenta desenvolvida e um pequeno exemplo explicativo. A Seção 4 discute brevemente alguns trabalhos relacionados e, finalmente, a conclusão é feita na Seção 5.

2. O Problema

Para chegar na definição de uma configuração de um produto é comum termos opiniões de múltiplas partes envolvidas no processo. Nesta abordagem consideramos um cenário onde vários *stakeholders* envolvidos no processo de configuração de um produto podem expressar suas opiniões e preferências. *Stakeholders* podem ser qualquer indivíduo que tem um interesse no domínio, como por exemplo: gerentes técnicos e de marketing, programadores, usuários finais e clientes. Considerando esses diversos indivíduos, que possuem necessidades, interesses e preocupações distintas com relação ao produto a ser desenvolvido, podemos perceber que conflitos devam surgir devido às preferências divergentes entre os *stakeholders*.

A participação de mais de um *stakeholder* nesse processo o torna mais complexo, pois podem existir preferências conflitantes entre eles, de modo que seja impossível recomendar uma única configuração que satisfaça-os razoavelmente. Isso acontece, por exemplo, quando dois *stakeholders* expressam suas preferências sobre uma mesma *feature* de modo que um deles deseja que ela esteja presente no produto, porém o outro prefere que ela não esteja. Um segundo tipo de conflito acontece nos grupos de *features* alternativas, onde somente uma das *features* do grupo pode ser selecionada. No momento

que um *stakeholder* selecionar F1, ele automaticamente impede que outro *stakeholder* selecione F2, considerando que as duas (F1 e F2) estão em um mesmo grupo. Um terceiro tipo de conflito surge com relação às restrições de integridade do FM onde, caso um primeiro *stakeholder* selecione a *feature* F1 que requer a seleção de F2, ele automaticamente faz com que as ambas sejam selecionadas, podendo causar um futuro conflito caso algum stakeholder não queira uma delas no produto final.

Considerando os conflitos descritos, percebe-se a necessidade de uma abordagem para endereçá-los. Desta forma, a presente ferramenta implementa uma abordagem [Stein et al. 2014] que nós desenvolvemos a qual propõe um metamodelo que permite a criação de configurações de *stakeholder*. Cada configuração de stakeholder representa as preferências estabelecidas por um indivíduo sobre as features do FM. Com base nessas configurações buscamos uma configuração final que satisfaça da melhor maneira os *stakeholders* envolvidos. Assim, na próxima seção mostraremos as principais funcionalidades e características da ferramenta SACRES 2.0.

3. Funcionalidades e Arquitetura da Ferramenta

O SACRES 2.0 é um plugin para a IDE Eclipse desenvolvido em Java que funciona de maneira integrada com outro plugin chamado FeatureIDE[1]. Dentre as possíveis ferramentas existentes para a integração com o SACRES 2.0, o FeatureIDE foi o escolhido por sua fácil usabilidade e por possuir cobertura de todo o processo de desenvolvimento de uma LPS, incluindo assim, funcionalidades desejadas para dar suporte à nossa implementação. Além disso, ele dispõe de uma documentação bem detalhada e está em constante desenvolvimento. O FeatureIDE objetiva a redução de esforços para a construção de ferramentas para novas e existentes abordagens de LPS. O plugin também possui diversas funcionalidades pertinentes para a aplicação da nossa ferramenta, entre as principais citamos: criação e edição de modelos de *features*, análise automática do modelo, criação e edição de configurações de produtos. Ele utiliza *views* específicas que permitem a visualização com detalhes dos FMs e das possíveis configurações.

A abordagem acima mencionada, implementada pelo SACRES 2.0, foi anteriormente implementada na forma de um *protótipo*, para viabilizar um experimento para avaliar diferentes estratégias da teoria da escolha social, que foram instanciadas para o contexto de FMs [Stein et al. 2014]. Este protótipo, referenciado como SACRES 1.0, carece das funcionalidades relacionadas à criação e edição de FMs disponíveis no FeatureIDE, pois utiliza um banco de dados para armazenamento dos FMs previamente definidos. Sua interface permite apenas a criação e edição de configurações de *stakeholders* e grupos de SHs. Assim sendo, nossa ferramenta tem como propósito a integração das funcionalidades do FeatureIDE com a recomendação de configurações ótimas envolvendo múltiplos *stakeholders* realizada pelo SACRES 1.0. Além disso, a recomendação de configurações é realizada de forma mais otimizada.

A Figura 1 mostra a arquitetura do plugin desenvolvido com a interação entre as partes envolvidas. Nela, podemos ver que o FeatureIDE é o responsável pela criação e edição do modelo de *features* e pela visualização das configurações recomendadas. O SACRES 2.0 contém as classes com suporte ao metamodelo e a implementação dos algoritmos de recomendação para as estratégias de escolha social usadas. Além disso, o plugin

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

possibilita a criação e edição de configurações que são salvas em arquivos individuais e organizadas em pastas para cada grupo de *stakeholders* associando-os a um modelo de *features*. Desse modo, ela também é responsável pelo gerenciamento dos arquivos, desde os novos arquivos com as configurações de stakeholder baseadas no metamodelo até os resultados das recomendações que podem ser visualizados no FeatureIDE. O SACRES 2.0 faz o uso do CSP Solver CHOCO[2] de forma a eliminar de maneira mais eficiente as configurações consideradas inválidas pelas restrições duras, gerando o que chamamos de *conjunto de consideração*, utilizado pelas estratégias de escolha social para buscar as configurações ótimas.

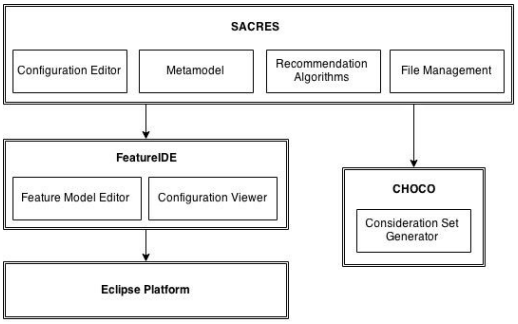


Figura 1. Arquitetura da ferramenta.

Para uma melhor compreensão do funcionamento da ferramenta iremos detalhar suas funcionalidades através de um pequeno exemplo. Para isso, criamos um FM simples e associamos um grupo de três *stakeholders* a esse modelo.

3.1. Metamodelo

Na solução proposta por Stein (2014), na qual o plugin se baseia, foi definido um metamodelo para melhorar a expressão de preferências dos *stakeholders*, flexibilizando o processo, através de definições de preferências de *features* de maneira quantitativa, representando o nível de preferência sobre as *features*. Em outras palavras, os *stakeholders* definem um número no intervalo [-1,1] que expressa o quanto eles querem que uma *feature* esteja presente na configuração (representado por valores positivos) ou o quanto não desejam que uma *feature* esteja na configuração final (representado por valores negativos). Essas preferências são chamadas de restrições do tipo *soft* ou brandas.

Além das restrições brandas o metamodelo também permite que sejam estabelecidas restrições do tipo *hard* ou duras, que por sua vez representam a obrigatoriedade de uma *feature* estar presente na configuração, quando utilizada uma restrição *hard* positiva, ou de não estar presente, quando utilizada uma restrição *hard* negativa. Dessa forma podemos dizer que uma configuração de um *stakeholder* é composta por um conjunto de restrições *hard* positivas e negativas e por um conjunto de restrições *soft* positivas e negativas associados a um modelo de *features*. Para medirmos o nível de satisfação de um *stakeholder* em relação à uma configuração usaremos as restrições do tipo *soft*. A soma das restrições brandas satisfeitas em uma configuração nos dá um valor numérico, chamado de satisfação de *stakeholder*, e é esse valor que desejamos maximizar. Já as restrições duras servem para eliminar possíveis configurações conflitantes em uma primeira etapa, diminuindo o espaço de busca a ser considerado pela segunda etapa.

[2] `http://choco-solver.org/?q=Choco3`

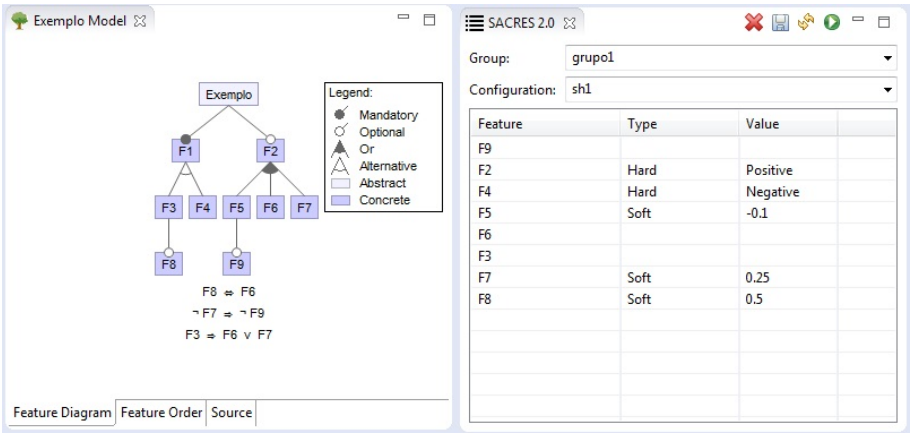


Figura 2. *Feature Model* e possível configuração de *stakeholder*.

A Figura 2 apresenta o FM criado através do FeatureIDE como exemplo e a *view* do plugin SACRES 2.0 em execução. A *view* pode ser dividida em três partes: (i) a seção que seleciona as pastas e arquivos; (ii) a tabela com as *features* para alteração de configurações de *stakeholder*; e (iii) os botões de funcionalidade.

Grupos de *stakeholders* são representados por pastas dentro de um projeto do FeatureIDE e configurações de *stakeholders* são representadas por arquivos dentro dessas pastas. Na *view* primeiramente deve-se selecionar um grupo pertencente à lista de grupos associados ao FM. Depois é possível abrir configurações salvas dentro desse grupo ou criar uma nova configuração.

A tabela de configuração é dividida em três colunas, que listam o nome das *features*, o tipo da restrição associada a ela e o valor dessa restrição. O modelo de exemplo possui dez nodos na árvore, porém apenas oito estão sendo listados na tabela, pois o nodo raiz é abstrato não sendo uma *feature* propriamente dita e o nodo F1 representa uma *feature* obrigatória, assim sendo não é possível expressar preferências sobre ele. Ao estabelecer preferências, primeiramente o usuário seleciona o tipo (*hard* ou *soft*), e depois escolhe seu valor, caso seja *soft*.

No canto superior direito da *view* encontram-se quatro botões com as seguintes funções: apagar a configuração atual, salvar a configuração atual, atualizar as informações da *view* e executar os algoritmos de recomendação de configurações. Ao salvar uma configuração, o usuário deve escolher um nome para ela, digitando na caixa "Configuration". Quando executamos os algoritmos de recomendação, uma pasta com os resultados é criada e, caso não existam configurações válidas para serem recomendadas, uma mensagem de erro é exibida.

3.2. Estratégias de Escolha Social

Foram utilizadas sete estratégias baseadas na teoria da escolha social para a recomendação das configurações. Cada uma delas gera uma configuração de produto, de acordo com suas características. Sendo assim, após a execução do algoritmo da ferramenta, teremos até sete configurações como resultado, visto que estratégias diferentes podem gerar a mesma configuração. O algoritmo consiste de quatro passos principais: (i) tradução do FM e as preferências *hard* dos stakeholders para um CSP; (ii) geração de configurações

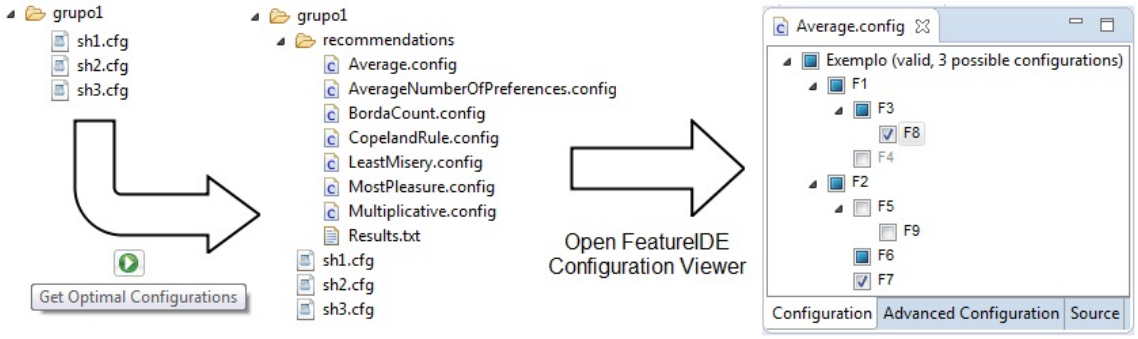


Figura 3. Arquivos e configurações salvas.

válidas utilizando o CHOCO; (iii) seleção da configuração ótima dentre as soluções viáveis de acordo com o critério de cada estratégia; e (iv) tradução das configurações ótimas para um arquivo de configuração seguindo o padrão do FeatureIDE.

As estratégias usadas são chamadas de: *Average (AVG), Average Number of Preferences (ANP), Borda Count (BC), Copeland Rule (CR), Least Misery (LM), Most Pleasure (MP) e Multiplicative (MUL)*. As estratégias *Average* e *Multiplicative* se preocupam em encontrar a satisfação global máxima dos *stakeholders*, a primeira calcula a média das satisfações individuais dos *stakeholders*, enquanto a segunda multiplica as satisfações individuais, e para ambas é selecionado o maior valor entre os calculados. A *Most Pleasure* se preocupa em encontrar a melhor satisfação de *stakeholder* possível, então ela seleciona o máximo entre o máximo de satisfações individuais de *stakeholders*. *Least Misery* se preocupa em não deixar os *stakeholders* muito insatisfeitos, o que pode ocorrer com as três estratégias acima. Para isso, ela seleciona o máximo entre o mínimo de satisfações individuais de *stakeholders*. A *Borda Count* usa a satisfação do *stakeholder* para ordenar as configurações. Assim temos uma classificação de configurações da satisfação mais alta (melhor) até a satisfação mais baixa (pior), e baseando-se nessa classificação cada configuração recebe um número de pontos. A pior configuração recebe 0 pontos, a segunda pior recebe 1 ponto e assim sucessivamente. Finalmente, os pontos de cada configuração individual de *stakeholder* são somados e o máximo é selecionado. *Copeland Rule* considera a relação de ordenação dada pela satisfação do *stakeholder*. Ela seleciona a configuração que tem a máxima diferença entre o quão frequente uma configuração ganha de outras configurações e o quão frequente ela perde, usando um sistema de votos. A *Average Number of Preferences* considera apenas a satisfação das restrições *soft*, ignorando os graus de preferência dos *stakeholders*. Ela seleciona a configuração com o número máximo de restrições *soft* satisfeitas.

A Figura 3 mostra uma parte do gerenciamento de arquivos feito pelo SACRES 2.0. Dentro de qualquer projeto do FeatureIDE existe uma pasta chamada “configs”. Nossa ferramenta trabalha com os arquivos contidos dentro dessa pasta. Cada subpasta presente nela representa um grupo de *stakeholders*, que podem ser criados, editados e apagados diretamente na *view* do projeto no Eclipse. Dentro das pastas de grupo encontram-se as configurações dos *stakeholders* associadas a ele. Estas configurações são baseadas no metamodelo descrito, são salvas com a extensão “.cfg” e podem ser exibidas usando a *view* da ferramenta (o conteúdo do arquivo “sh1.cfg” pode ser observado na Figura 2).

A. Configurações de stakeholders.

Feature	SH1	SH2	SH3
F2	+	0.6	+
F3			0.3
F4	-		
F5	-0.1		
F6			-0.15
F7	0.25	+	0.1
F8	0.5	0.2	
F9			

B. Recomendações das estratégias.

Total	AVG	ANP	BC	CR	LM	MP	MUL
Satisfação	2.2	2.2	2.1	2.2	2.1	2.05	2.2
Preferências	8	8	7	8	7	7	8

Tabela 1. Exemplo de recomendações usando preferências de stakeholders.

Na Figura 3 podemos também ver a pasta "grupo1" com três configurações diferentes. Após o clique no botão "*Get Optimal Configurations*" os algoritmos de recomendação são executados e uma nova pasta chamada "*recommendations*" é criada. Dentro dela estão os arquivos de configuração correspondentes a cada estratégia de escolha social no formato padrão do FeatureIDE ".config". Dessa forma é possível usar a funcionalidade de visualização de configurações do próprio FeatureIDE.

A Tabela 1(A) nos mostra as preferências de cada um dos três *stakeholders* pertencentes ao grupo. Os símbolos + e - representam restrições do tipo *hard* positivas e negativas, respectivamente. Os valores numéricos representam o grau de preferência de restrições do tipo *soft*. Na Tabela 1(B) exibimos alguns dos resultados para cada estratégia (coletados do arquivo "Results.txt" localizado na pasta "recommendations"). Ela mostra o total de satisfação dos *stakeholders*, ou seja, a soma das satisfações individuais e o número total de preferências satisfeitas para cada estratégia. Podemos observar que algumas estratégias conseguiram satisfazer todas as oito restrições do tipo *soft* presentes no grupo. Outras informações como o mínimo, o máximo, a média e o desvio padrão, tanto da satisfação individual de *stakeholder* quanto da quantidade de preferências para cada estratégia também podem ser encontradas no arquivo de resultados.

4. Trabalhos Relacionados

A ferramenta apresentada neste artigo permite a especificação de configurações associadas a uma LPS. Entretanto, essas configurações não são as usualmente utilizadas, mas configurações onde preferências *soft* e *hard* são especificadas. Além disso, recomendações são realizadas que resultam nessas configurações tradicionais. Assim, SACRES tem como parte de sua infraestrutura uma ferramenta para o gerenciamento de FMs. Foram analisadas diversas ferramentas a fim de encontrar o melhor candidato para a integração com o SACRES, entre elas citamos: SPLOT[3], XFeature[4] e FMP[5]. Estas ferramentas carecem de algumas funcionalidades desejadas no nosso plugin (como discutido na Seção 3), por isso foram descartadas e o FeatureIDE foi escolhido.

O plugin SPLConfig [Machado et al. 2014] também visa a recomendação automática de configurações ótimas maximizando a satisfação dos clientes, porém não suporta a participação de múltiplos stakeholders no processo de configuração de um mesmo produto. Ele mantém o foco em outros requisitos e limitações, como custo, benefício do consumidor e orçamento.

[3] `http://www.splot-research.org/`
[4] `http://www.pnp-software.com/XFeature/`
[5] `http://gsd.uwaterloo.ca/fmp`

5. Considerações Finais

Configuração de *feature model* é uma atividade importante para o desenvolvimento de produtos em uma LPS. O envolvimento de múltiplos *stakeholders* nesse processo pode torná-lo difícil devido aos possíveis conflitos entre suas preferências. Neste artigo apresentamos a ferramenta SACRES 2.0, um plugin para o Eclipse que visa dar suporte à recomendação de configurações considerando múltiplos *stakeholders*. A ferramenta utiliza estratégias de escolha social anteriormente propostas para buscar configurações que melhor satisfaçam o grupo de *stakeholders*.

O foco deste trabalho não foi a análise de cada estratégia usada, e sim as funcionalidades fornecidas pela ferramenta. Detalhes mais aprofundados sobre o desempenho das estratégias e avaliação do resultado das recomendações, com relação à satisfação individual e justiça, podem ser encontrados em [Stein et al. 2014]. Dois aspectos poderiam ser futuramente avaliados com relação à ferramenta: usabilidade e escalabilidade. Como um estudo anterior [Mendonca et al. 2009] concluiu que o processamento de *feature models* com um CSP Solver em geral não tem problemas de escalabilidade, temos indícios que o SACRES 2.0 não terá problemas de performance. Também, futuramente, possíveis melhorias podem ser adicionadas à ferramenta. Uma possibilidade seria considerar pesos para determinadas preferências dos *stakeholders* de acordo com sua expertise ou preferências sobre grupos de *features*, além de dar um maior suporte à colaboração distribuída dos *stakeholders*. Atualmente, as configurações dos *stakeholders* podem ser criadas em máquinas diferentes, entretanto os arquivos de configuração gerados devem ser agrupados para a geração de recomendações.

Reconhecimento

Este trabalho recebeu o apoio financeiro do programa PIBIC CNPq-UFRGS, com concessão de bolsa de iniciação científica ao primeiro autor deste trabalho.

Referências

Clements, P. and Northrop, L. (2002). *Software product lines: practices and patterns*, volume 59. Addison-Wesley Reading.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document.

Machado, L., Pereira, J., Garcia, L., and Figueiredo, E. (2014). Splconfig: Product configuration in software product line. In *CBSoft, Tools Session*, pages 1–8.

Mendonca, M. and Cowan, D. (2010). Decision-making coordination and efficient reasoning techniques for feature-based configuration. *Science of Computer Programming*, 75(5):311–332.

Mendonca, M., Wasowski, A., and Czarnecki, K. (2009). Sat-based analysis of feature models is easy. In *SPLC '09*, pages 231–240, USA. Carnegie Mellon University.

Stein, J., Nunes, I., and Cirilo, E. (2014). Preference-based feature model configuration with multiple stakeholders. In *SPLC '14*, pages 132–141, USA. ACM.

RAwTIM – Uma Ferramenta para Rastreabilidade da Informação em Análises de Riscos

Paulo Barbosa[1], Fábio Leite[1], Raphael Mendonça[1], Melquisedec Andrade[1], Luana Sousa[1], Pablo Oliveira Antonino[2]

[1]Núcleo de Tecnologias Estratégicas em Saúde – Universidade Estadual da Paraíba(UEPB) – Campina Grande – PB – Brasil

[2]Embedded Systems Division Franhoufer IESE – Kaiserslautem – Germany

{paulo.barbosa, fabio.leite, raphael.mendonca, melqui.andrade, luana.janaina}@nutes.uepb.edu.br, pablo.antonino@iese.franhoufer.de

Abstract. *Safety critical systems have required new tools for assuring and demonstrating that safety requirements are addressed in the development project. This paper presents the RAwTIM (Risk Analysis with Traceability Information Model), an extension of the Enterprise Architect tool that uses a model for decomposition of safety artifacts. Through this tool, designers can: (1) automatically verify the integrity between the system architectural elements and the risk analysis previously performed; (2) guarantee the traceability between modules and safety goals; and (3) verify if not acceptable risk elements were mitigated during the risk analysis.*

Resumo. *Sistemas críticos têm requisitado novas ferramentas para assegurar e demonstrar que requisitos de segurança estejam contemplados no projeto de desenvolvimento. Este artigo apresenta o RAwTIM (Risk Analysis with Traceability Information Model), uma extensão da ferramenta Enterprise Architect que utiliza um modelo de decomposição de elementos de segurança. Através desta ferramenta, os projetistas podem: (1) verificar automaticamente a integridade entre os elementos arquiteturais do sistema e a análise de riscos previamente realizada; (2) garantir a rastreabilidade entre os módulos e as respectivas metas de segurança; e (3) verificar se elementos de risco não aceitáveis foram mitigados durante a análise de riscos.*

Link do Vídeo Demonstrativo: https://youtu.be/QPKfafy98qk

1. Introdução

O contexto atual da indústria de sistemas críticos tais como dispositivos eletromédicos, transportes, geração de energia, entre outros, tem aumentado a exigência sobre as ferramentas de garantia e promoção da qualidade dos seus produtos. Se por um lado, a indústria é impulsionada com um alto grau de inovação através do avanço das tecnologias relacionadas, perigos surgem com o aumento do poder de processamento e da conectividade dos sistemas embarcados. Desta forma, os órgãos reguladores de qualidade do setor estão requerendo níveis mais refinados e acurados de informações em seus respectivos processos de certificação dos produtos (J. HATCLIFF, 2014). Diante desta conjuntura, faz-se necessária a promoção de novas ferramentas e técnicas que propiciem a rastreabilidade entre requisitos (identificados junto às normas de segurança, análise de risco, especialistas, usuários e demais envolvidos) e artefatos

arquiteturais produzidos desde as primeiras fases do projeto até os estágios finais de produção, testes e operação (P. MADER, 2013). Através da adoção destas técnicas, temos observado um avanço na qualidade dos sistemas críticos e na redução de esforços no processo de certificação.

Um dos principais aspectos de qualidade abordados pelo setor é o grau de segurança, de acordo com o nível de risco associado ao contexto de uso dos sistemas. Por isso, estes sistemas devem ser submetidos a processos de certificação antes de serem disponibilizados no mercado (ANVISA, 2011). No contexto de sistemas eletromédicos, a indústria deve submeter os equipamentos médicos para as agências reguladoras. Estas agências verificam a conformidade dos produtos de acordo com as normas reguladoras do setor, como é o caso da (ISO 14971, 2007), da (IEC 62304, 2006), ou do conjunto de normas IEC 60601. Sendo assim, o fabricante deve prover toda a informação necessária para que os agentes reguladores identifiquem que os requisitos de segurança estão garantidos no projeto do sistema.

De acordo com as normas (ISO 42010-2011 seção 5.7, ISO 14971-2007 seção 3, IEC 61508-2006 Parte 3 seção Anexo A, C), um dos aspectos fundamentais no processo de certificação é que os sistemas embarcados eletromédicos precisam apresentar evidências claras que requisitos de segurança identificados na fase de análise estão contemplados pelos elementos arquiteturais que constituem o sistema na fase de projeto.

Embora a importância da rastreabilidade entre os artefatos de projeto e de análise seja bastante evidenciada, esta não é uma prática trivial a ser adotada. Dentre vários problemas, podemos citar as dificuldades de comunicação entre as equipes responsáveis pela segurança e arquitetos de sistemas. Estes problemas decorrem, por existirem diferentes de abstração no projeto agravadas pela falta de ferramentas (linguagens, modelos, ambientes integrados de desenvolvimento, etc.) onde possam unificar seus esforços (P. MADER, 2013) (J. HATCLIFF, 2014).

Tendo em vista os problemas apresentados, a ferramenta RAwTIM implementa um modelo de rastreabilidade da informação na ferramenta Enterprise Architect (EA)[1]. Este projeto foi desenvolvido para realizar análises de riscos em sistemas, de acordo como proposto pela norma para dispositivos médicos, a ISO 14971 (ISO 14971, 2007), e seguindo o Modelo de Informação de Rastreabilidade (TIM – *Traceability Information Model*) definido em (Antonino & Trapp, 2012). O Modelo de Informação de Rastreabilidade fornece um framework conceitual no qual as metas de mais alto nível de abstração são decompostas em requisitos de nível atômico, onde apenas um elemento arquitetural do sistema afeta diretamente o respectivo requisito de segurança.

A principal ferramenta relacionada do gênero é o Risk Analysis Designer da Obeo[2]. Esta ferramenta é um *plug-in* para o Eclipse, robusto, mas que tem a limitação de ser generalizada para vários domínios industriais, tais como sistemas aviônicos, automotivos e ferroviários, seguindo as normas ISO 26262, DO178C, EN 50126, EN 50128 e EN 50129. Ela não possui aplicabilidade para as normas da indústria médica e desta forma, não provê o tipo de análise que estamos interessados.

2.Rastreabilidade dos Requisitos de Segurança

O objetivo do Modelo de Informação de Rastreabilidade (TIM – *Traceability Information Model*) (Antonino & Trapp, 2012) é planejar a rastreabilidade entre

[1] sparxsystems.com
[2] obeodesigner.com

elementos de segurança e componentes do sistema. Os elementos de segurança se relacionam entre si seguindo regras pré-determinadas. O modelo é representado por um diagrama de classes UML e está divido em três camadas de abstração, são elas: nível de contexto, nível funcional e nível técnico. Maiores informações sobre os itens que compõem o TIM e os principais termos para dispositivos médicos podem ser encontradas em (Antonino P. , Trapp, Barbosa, Gurjao, & Rosario, 2015).

O nível de contexto representa o levantamento preliminar de todas as possíveis ameaças (*Hazards)* ao sistema. Faz-se a identificação da sequência de eventos que, caso aconteçam, originarão algum dano ao sistema, usuário, operador ambiente ou outro sistema. Esta sequência de eventos previstos é representada pelo elemento *Foreseeable Sequence*, que juntamente com uma dada ameaça, podem levar o sistema a uma situação perigosa (*Hazardous Situation*) e causar algum dano ao sistema (*Harm*). O levantamento desses elementos (*Hazard, Foreseeable Sequence, Hazardous Situation* e *Harm*) é chamado de análise de risco de acordo com ISO 14971. Terminado o processo de análise de riscos do sistema, pode-se iniciar o processo de rastreabilidade dos requisitos de segurança do sistema.

O nível funcional é motivado por um *Hazard* elucidado na fase de análise de riscos. Para garantir que o risco do *Hazard* causar algum dano ao sistema seja aceitável, se faz necessário que uma meta de segurança seja alcançada. Tal meta inicia a construção do nível funcional do modelo e é representada pelo elemento *Top-most Safety Requirement*. Para garantir que *Top-most Safety Requirement* seja alcançado, um conjunto de metas de segurança deve ser garantido. Tais metas são representadas no sistema como *Composite Functional Safety Requirement*, que por sua vez pode ser motivado por várias causas de falhas (*Functional Failure Cause*). Cada *Functional Failure Cause* será decomposta até que exista apenas uma causa de falha associada. Tal falha é representada por um requisito de segurança atômico, o *Atomic Safety Requirement*. Cada requisito de segurança possui uma classificação de severidade de dano que varia de "A" até "C" em dispositivos médicos, sendo "A" o menos severo e "C" o mais severo, herdado da norma IEC 62304, como *Software Safety Class*.

Estabelecidos os requisitos de segurança no nível funcional, analisamos como os componentes do sistema garantirão tais requisitos. Essa descrição é feita no nível técnico, que apresenta os seguintes componentes: *Technical Safety Requirement, Technical Fault Tolerance Requirement, Technical Detection Requirement, Functional Containment Requirement* e *Technical Containment Requirement*.

3. A Ferramenta RAwTIM

A RAwTIM foi desenvolvida com o objetivo de realizar análises de risco de sistemas críticos e apresentar uma garantia que as devidas medidas de segurança para mitigar riscos no sistema foram tomadas.

A ferramenta possui as seguintes funcionalidades:

- Cadastrar elementos de análise riscos em conformidade com a ISO 14971 (*Hazards*, *Foreseeable Sequences*, *Hazardous Situations* e *Harms*);
- Gerar matriz de análise de riscos com base nos elementos de risco cadastrados;
- Gerar um diagrama de análise de riscos com base na matriz de análise de riscos;
- Associar elementos da modelagem a elementos da matriz de análise de risco;
- Criar um diagrama de decomposição de requisitos de segurança.

Após instalação, o RAwTIM fica disponível conforme apresentado na Figura 1.

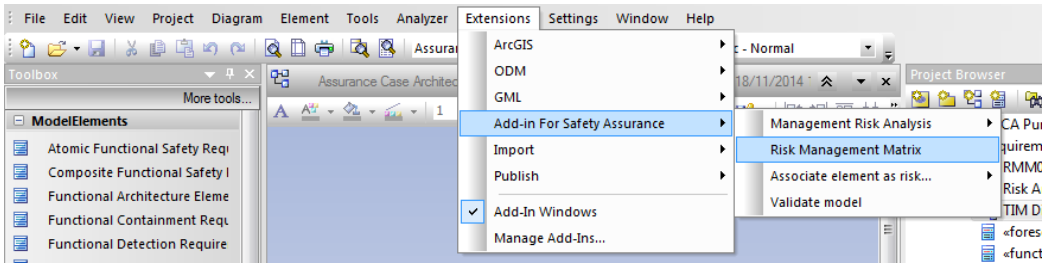
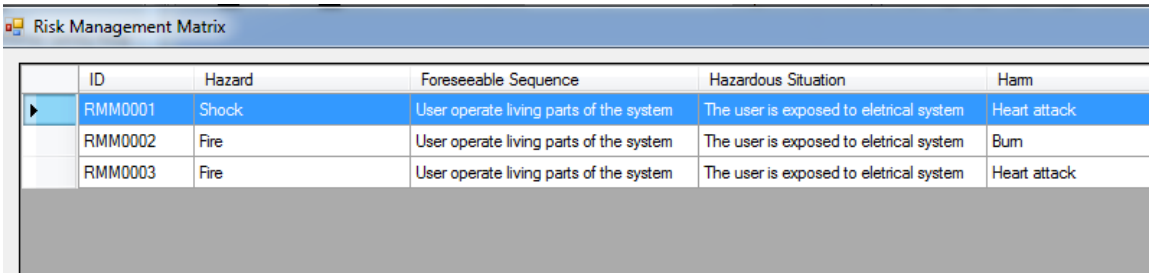


Figura 1 - Menu do RAwTIM no Enterprise Architect

A partir do menu inicial "*Add-in For Safety Assurance*", tem-se quatro sub-menus. São eles:

- *Management Risk Analysis*: Gerenciamento da Análise de Risco. É onde se faz a criação, edição e deleção dos elementos de risco. Os elementos de risco são: *Hazard, Foreseeable Sequence, Hazardous Situation* e *Harm*.
- *Risk Management Matrix*: Matriz de Gerenciamento de Risco. Nessa opção se faz a análise de risco propriamente dita das ameaças levantadas no sistema. Como se pode ver na Figura 2, cada linha da matriz representa uma análise de riscos realizada previamente onde, para um dado *Hazard*, tem-se um *Foreseeable Sequence*, uma *Hazardous Situation* e um *Harm* associado. Para cada linha da matriz, pode-se gerar um diagrama de análise de risco, bastando para isso, clicar em *Generate diagram*.



ID	Hazard	Foreseeable Sequence	Hazardous Situation	Harm
RMM0001	Shock	User operate living parts of the system	The user is exposed to eletrical system	Heart attack
RMM0002	Fire	User operate living parts of the system	The user is exposed to eletrical system	Burn
RMM0003	Fire	User operate living parts of the system	The user is exposed to eletrical system	Heart attack

Figura 2 - Matriz de Gerenciamento de Risco

- *Associate element as risk...*: Associar elemento como elemento de risco. Permite associar um elemento do sistema a ser modelado a algum elemento da análise de risco realizada. O intuito dessa funcionalidade é fazer a rastreabilidade dos componentes do sistema com a matriz de risco, dessa forma um componente "x" do sistema está relacionado com determinada ameaça à segurança, por exemplo.
- *Validate model*: Validar modelo. É a opção de validação do diagrama de decomposição dos elementos de segurança. Nessa opção, todas as regras para a construção do diagrama TIM serão verificadas para garantir que o diagrama modelado está em conformidade com o modelo TIM.

A ferramenta inclui ainda outros diagramas para modelagem: O diagrama de decomposição de requisitos de segurança (Assurance Case Architectural Diagram) e o diagrama de Matriz de Análise (Matrix Analysis Diagram).

O diagrama de decomposição é específico para refinar requisitos de segurança de alto nível em requisitos menores, relacionando as causas de falha associadas e medidas para detecção e contenção das falhas. Ao realizar a validação no diagrama, os elementos que apresentam algum tipo de inconsistência são destacados de vermelho e o resultado dessa validação é descrito na aba "*System Output*" do EA. A Figura 3 mostra

um exemplo de validação dos elementos do diagrama. Neste exemplo os elementos "*composite functional safety requirement*" e "*atomic functional safety requirement*" foram pintados de vermelho, considerando as regras do modelo TIM, onde se define a necessidade de um requisito de segurança "*Composite*" ser refinado por um requisito de segurança "*Atomic*". A verificação do sentido do relacionamento entre os elementos no diagrama também é uma característica presente na validação. A Figura 3 apresenta um exemplo desse relacionamento, onde o elemento "*Top-Most Safety Requirement*" é refinado pelo "*Composite Functional Safety Requirement*", dessa forma o único sentido de associação permitido é a partir do "Top-Most" para o "Composite", o sentido inverso não é permitido. Se a modelagem do diagrama estiver em conformidade com o modelo TIM, será presentada uma mensagem informando que o diagrama corrente está validado.

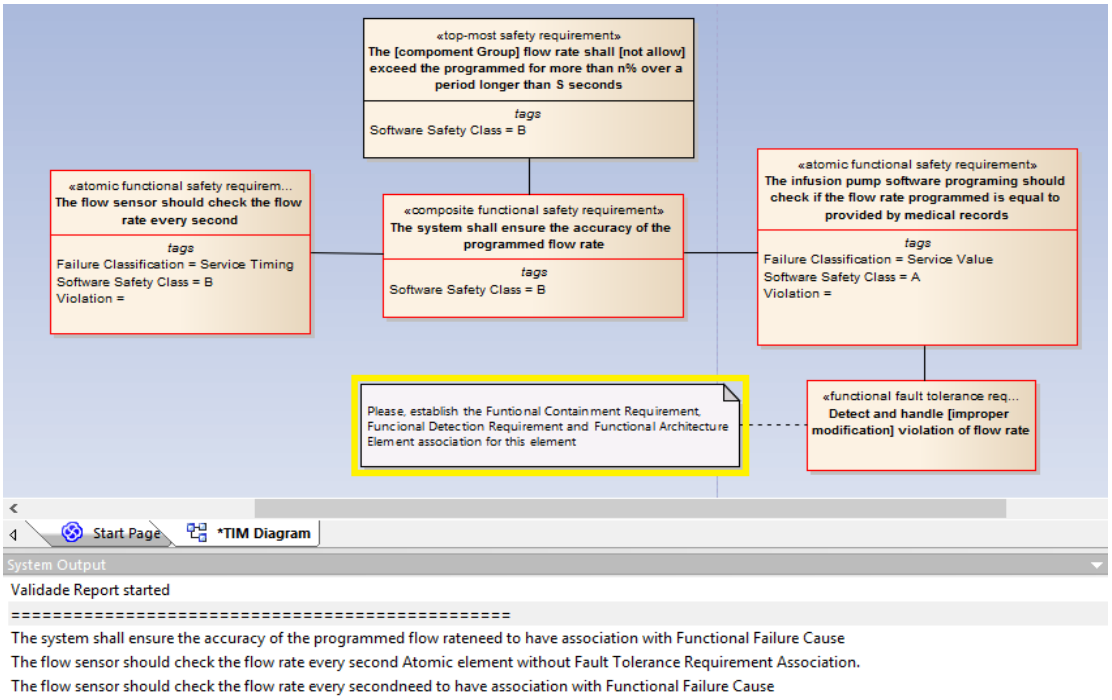


Figura 3 - Validação dos elementos do diagrama

Outra medida de validação presente na ferramenta é a inclusão de notas informativas sobre a ação que deve ser realizada para elementos que dependem obrigatoriamente de outros. A Figura 3 (destacado de amarelo) representa essa situação onde é adicionada uma nota informando os elementos que devem ser obrigatoriamente associados ao "*Functional Fault Tolerance*".

4. Utilização da ferramenta RAwTIM na Modelagem de um Desfibrilador Cardíaco (DEA)

Nesta seção, será apresentado um estudo de caso da utilização do RAwTIM na modelagem da arquitetura de um Desfibrilador Cardíaco (DEA – Desfibrilador Externo Automático). Este estudo de caso está inserido no contexto do NUTES[3] (Núcleo de Tecnologias Estratégicas em Saúde), que é uma iniciativa do Ministério da Saúde, em uma preparação de competências para transferência de tecnologias com a empresa

[3] nutes.uepb.edu.br

LIFEMED[4]. O NUTES é um núcleo de desenvolvimento de tecnologias para a garantia da confiabilidade de dispositivos produzidos pela indústria médica.

4.1. Descrição da arquitetura do Desfibrilador Cardíaco (DEA)

A Figura 4 mostra a modelagem do diagrama de contexto para o sistema do DEA. Neste diagrama são apresentadas as principais variáveis e os relacionamentos entre os módulos do sistema e entidades externas. Existem 2 atores importantes dentro desse contexto: O Paciente (Patient), que será conectado ao Desfibrilador, e o Socorrista (Rescuer), que irá interagir com a interface do Desfibrilador (*Operator Interface)* . A principal variável de entrada para o DEA é o Sinal do ECG *(ECG Signal)* que obtém os pulsos cardíacos do Paciente (Patient). Com base nessa informação, o Analisador do Sinal *(Signal Analyzer)* irá avaliar se aconteceu uma parada cardíaca e em caso afirmativo, o Gerador de Choque *(Schock Generator)* será acionado para entregar uma descarga de energia controlada ao peito do paciente através das pás *(Pads).*

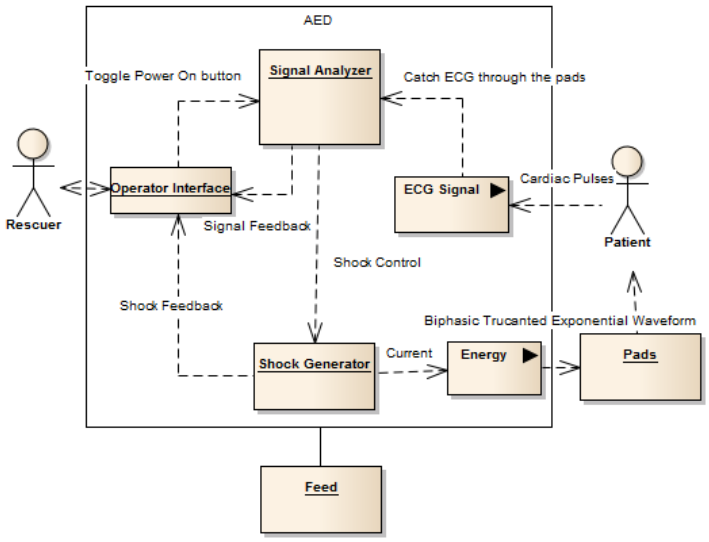


Figura 4 - Diagrama de contexto do DEA

Entre os principais casos de uso do DEA, está a operação normal que inclui analisar o sinal de ECG para obter a energia desejada. O *Rescuer* também pode abortar o choque ou executar um auto-teste. Os principais cenários de falhas no uso do sistema são: "*Failure to deliver the shock*", quando o sistema não é capaz de entregar o choque necessário ao paciente, "*Failure to deliver the shock at the right time*", quando o sistema entrega o choque após exceder o tempo limite "*Failure to obtain the ECG signal*", quando as pás (*pads)* não conseguem obter o sinal ECG do paciente, e "*Failure to charge the circuit for shock deliver*", quando ocorre alguma falha no carregamento do circuito de conversão de energia no modo Tensão Contínua - Tensão Contínua.

De acordo com a modelagem definida no projeto do DEA, o principal perigo identificado para este sistema foi o "*Overshocking"*, que acontece quando o Gerador de Choque entrega uma corrente elétrica acima do valor estabelecido à caixa torácica do paciente. Este risco está relacionado a falha "*Failure to deliver the shock*". A Tabela 1 ilustra a análise de risco para o "*Overschocking"*. Nesta análise são considerados os

[4] lifemed.com.br

elementos de risco, onde o *Hazard* representa o risco "*Overshocking"*, a *Foreseeable Sequence* é a sequência de passos realizados para ocorrer o perigo. Neste caso os passos são: estar realizando a operação normal do DEA e ocorrer uma falha na entrega do choque ao Paciente. O *Hazard Situation* é a situação de risco quando as pás estão conectadas ao paciente e o DEA está ligado, e o *Harm* é o dano à pele do paciente, que pode ocorrer se o paciente receber uma descarga maior que o que está especificado.

Tabela 1 - Análise para o risco *Overshoching*

Hazard	Foreseeable Sequence	Hazard Situation	Harm
Overshocking	Normal Operation of the AED ; Failure to deliver the shock	The Pads are connected to the Patient; The AED is turned ON	Skin damage

4.2.Utilização da ferramenta RAwTIM

Para ilustração da ferramenta, será considerada a análise de risco apresentada na Tabela 1. Conforme descrito anteriormente, a ferramenta possui uma funcionalidade para cadastrar todos os elementos de risco e com base nesses elementos cadastrados gerar uma Matriz de Gerenciamento de Risco (apresentada na Figura 5). Com o RAwTIM é possível associar elementos da arquitetura aos elementos da análise de risco, de forma a manter o relacionamento entre eles. Na Figura 5, por exemplo, temos que o componente do diagrama de contexto "*Energy*" foi associado ao *perigo "Overshocking"* considerando que a energia é fator principal para geração do perigo. Já o ator "Patient" está associado ao *dano "Skin damage*", pois caso aconteça o risco de *Overshocking* será o paciente quem sofrerá o dano informado.

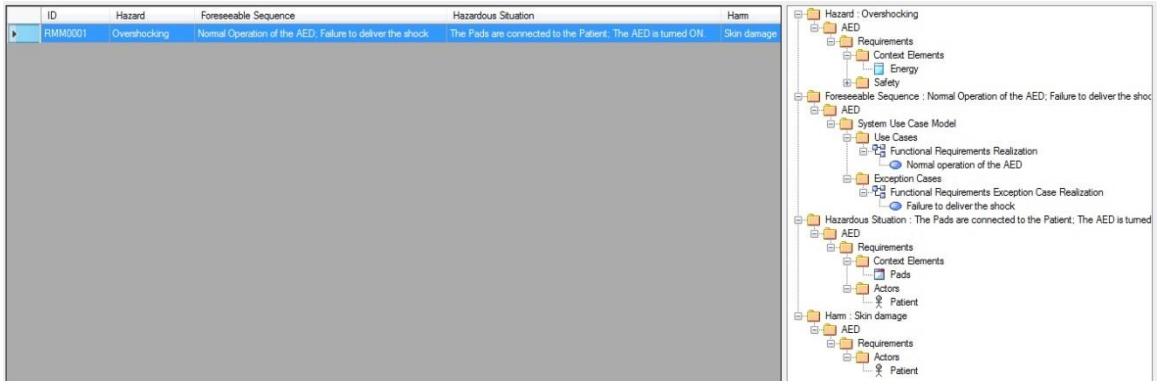


Figura 5 - Matriz de Gerenciamento de Risco

Além da matriz de gerenciamento de riscos, foi criado um diagrama de decomposição de requisitos de segurança, onde um trecho está ilustrado na Figura 6. Este diagrama foi utilizado para especificar os requisitos de segurança associados ao risco *Overshocking* e as principais causas de falha que podem gerar o risco. Um requisito de segurança para o *Overshocking* é garantir que o Gerador de Choque não entregue uma energia maior que o necessário ao peito do paciente. Esse requisito de alto nível foi especificado pelo elemento "*top-most safety requirement*". Para evitar o excesso de energia entregue, o gerador de choque deve garantir que a tensão da energia esteja na faixa entre 0 a 5k volts. Isso foi descrito no elemento "*composite safety requirement"* por ser uma especificação do elemento "*top-most safety requirement*". Por fim, o bloco SysML Discharge é o responsável pela realização do requisito de segurança e deverá apresentar medidas protetivas, tais como um Snubber, sensores de tensão e de corrente ou atuadores durante o carregamento.

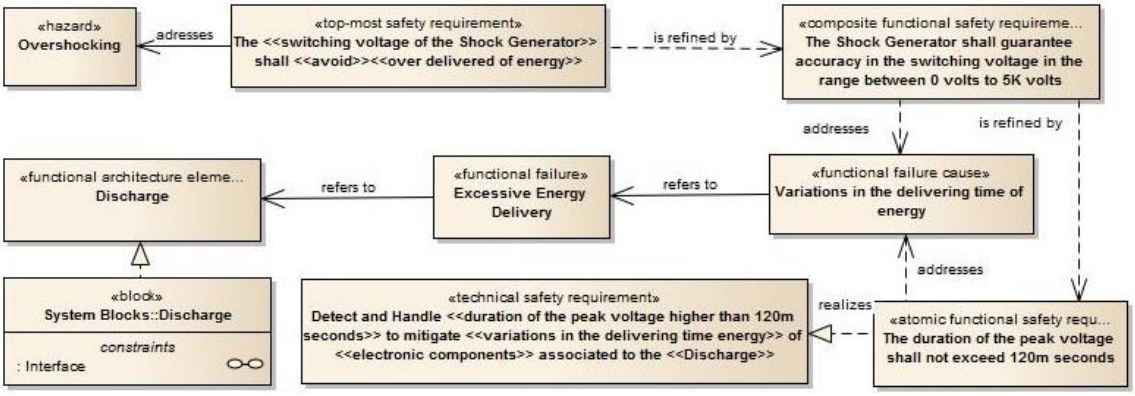


Figura 6 – Trecho da decomposição de requisitos para o Hazard Overshocking

5.Conclusão

Neste trabalho, apresentamos a ferramenta RAwTIM para decomposição de requisitos de segurança e rastreabilidade para elementos arquiteturais. A ferramenta se encontra em um estágio avançado de desenvolvimento, com outros tópicos que são considerados estado da arte em engenharia de segurança sendo incorporados a ela. Gerenciamento de evidências, padrões de argumentação estruturada, são alguns dos tópicos em evolução no momento.

Bibliografia

Antonino, P. O., & Trapp, M. (2012). Improving Consistency Checks Between Safety Concepts and View Based Architecture Design. *Probabilistic Assessment and Management*.

Antonino, P., Trapp, M., Barbosa, P., Gurjao, E., & Rosario, J. (2015). The Safety Requirements Decomposition Pattern. *Proceedings of International Conference on Computer Safety, Reliability and Security (SAFECOMP)*. Delft: Lecture Notes in Computer Science, Springer-Verlag.

Ericson, C. A. (2005). *Hazard analysis techniques for system safety.* Fredericksburg: JOHN WILEY & SONS, INC.

IEC 62304. (2006). *IEC 62304 - Medical Device Software—Software Life Cycle.* Assoc. Advancement Medical Instrumentation.

ISO 14971. (2007). *ISO 14971 - Medical Devices - Application of risk management to medical devices.* ISO Tech. Rep.

J. Hatcliff, A. W. (2014). Certifiably safe software-dependent systems: Challenges and directions. *Proceedings of the on Future of Software Engineering, ser. FOSE 2014* (pp. 182-200). New York,NY,USA: ACM.

P. Mader, P. L. (2013, May-June). Strategic traceability for Safety-Critical projects. *IEEE Software, 30*(3), 58-66.

ICARU-FB & FBE: Um Ambiente de Desenvolvimento Aderente à Norma IEC 61499

Leandro I. Pinto[1], Cristiano D. Vasconcellos[1], Roberto S. Ubertino Rosso Jr.[1], Eduardo Harbs[2], Gabriel H. Negri[2]

[1]Departamento de Ciência da Computação
Universidade do Estado de Santa Catarina (UDESC)

[2]Departamento de Engenharia Elétrica
Universidade do Estado de Santa Catarina (UDESC)

{leandro.israel.p, eduardo.harbs, negri.gabriel}@gmail.com
{cristiano.vasconcellos,roberto.rosso}@udesc.br

Abstract. *The IEC 61499 standard provides a development model for industrial automation and control. It defines a visual language that can facilitate the implementation of distributed control systems. In this paper we present a function block editor and a run time environment, both compliant with IEC 61499. The environment allows discrete elements to become low cost smart devices. A virtual machine capable to run IEC 61499 compliant programs in limited platforms such as 8-bit microcontrollers was implemented. This machine has been implemented for the platform Arduino ATmega2560 and also in a version for the Windows operating system.*

http://www.youtube.com/watch?v=VrixtqpgKbE

Resumo. *A norma IEC 61499 provê um modelo de desenvolvimento para automação industrial e controle. Essa norma define uma linguagem visual que pode facilitar a implementação de sistemas de controle distribuídos. Nesse trabalho são apresentados um editor de projetos e um ambiente para a execução, ambos aderentes à norma IEC 61499. O ambiente permite que elementos discretos tornem-se dispositivos inteligentes de baixo custo. Como parte do ambiente de execução foi implementada uma máquina virtual capaz de executar programas aderentes à IEC 61499 em plataformas limitadas como microcontroladores de 8-bits. Essa máquina foi implementada para a plataforma Arduino ATmega2560 e também numa versão para o sistema operacional Windows.*

1. Introdução

Normalmente os sistemas de automação são projetados utilizando Controladores Lógicos Programáveis (CLPs). Sistemas de grande porte possuem um considerável número de CLPs podendo ser caracterizados como sistemas distribuídos. Esses CLPs se comunicam via uma ou mais redes e controlam vários sensores e atuadores, o que geralmente ocasiona dificuldades em futuras modificações e extensões. A dificuldade inerente à implementação de sistemas distribuídos, assim como a incompatibilidade entre dispositivos e ferramentas de diferentes fabricantes, podem dificultar uma ágil adaptação da linha de produção a novos produtos, tornando a empresa totalmente dependente de um único fornecedor.

A norma IEC 61499 [Vyatkin 2012] fornece uma alternativa de implementação para esse tipo de sistema, propondo um modelo de abstração orientado a componentes que facilita a implementação de sistemas distribuídos e que garante a compatibilidade e portabilidade entre os sistemas que adotam esta norma. O uso desses componentes, chamados blocos de funções, possibilita a implementação de sistemas com baixo acoplamento, facilitando o suporte à reconfiguração dinâmica que é a possibilidade de alterar (incluir ou excluir) componentes sem a necessidade da interrupção na execução do sistema [Yoong, Roop e Salcic 2009]. O conceito de blocos de funções é uma extensão do bloco de função definido na norma IEC 61131-3, tornando-o mais adequado ao desenvolvimento de sistemas distribuídos [Chouinard e Brennan 2006]. Reduzir a complexidade de programação para um sistema de controle distribuído (DCS - *Distributed Control System*) é um dos principais objetivos da IEC 61499 [Vyatkin 2009, Zoitl et al. 2005].

A descentralização de um sistema para automação implica em distribuição das tarefas entre os dispositivos; consequentemente, esses dispositivos podem ser bastante simples e com pouca capacidade de processamento e recursos. A maioria dos ambientes, aderentes à norma IEC 61499, disponíveis utiliza vários processos ou *threads* na execução das especificações e demanda uma relativa capacidade de processamento e memória. Dentre esses ambientes destacam-se o FORTE [PROFACTOR 2008] e o FBRT [Christensen 2011] que são iniciativas abertas. Atualmente, a menor plataforma para execução do FORTE é composta por um processador ARM7 [ARM 1994] e o sistema operacional *eCos* [ECOS 2013]. O FBRT é considerado um ambiente de referência para o suporte à norma IEC 61499, nesse ambiente as definições são traduzidas para linguagem Java, sendo necessária a máquina virtual Java (JVM) [Buckley 2013] para a execução. Não parece viável, considerando os custos, utilizar essas plataformas para operar alguns poucos sensores e atuadores, um problema que pode ser resolvido com microcontroladores mais simples de 8-bits.

O principal objetivo desse trabalho é apresentar o ICARU-FB, um ambiente para execução de sistemas aderente à norma IEC 61499 que pode ser executado em hardware como o *Arduino ATmega2560* com um microcontrolador de 8-bits [Arduino 2013]. Como parte desse ambiente foram implementados um compilador e uma máquina virtual, ambos desenvolvidos em linguagem C. O código fonte e exemplos estão disponíveis em [ICARU-FB 2014] sob a licença GPLv3. O estudo de caso, apresentado na Seção 4 e também no vídeo, visa mostrar o suporte à reconfiguração dinâmica.

O editor de blocos de funções (FBE) faz parte do ambiente GASR-FBE, originalmente desenvolvido para edição, simulação e desenvolvimento de software para Controladores Numéricos Programáveis (CNC) aderentes às normas ISO 14649 e IEC 61499 [Harbs 2012, Harbs et al. 2013]. Esse ambiente está disponível em [GASR-FBE 2014]. Nesse trabalho é apresentado apenas o editor FBE (*Function Blocks Environment*) integrado ao ICARU-FB.

2. IEC 61499

Portabilidade, reconfiguração dinâmica e interoperabilidade são os principais requisitos que a norma IEC 61499 pretende atender. Os blocos de funções são a abstração de software definida para atingir esse intuito. Blocos de funções podem ser classificados em três tipos:

Bloco de Funções Básico: Esse tipo de bloco pode encapsular algoritmos e variáveis.

Bloco de Funções para Interface com Serviços: Esse tipo de bloco fornece acesso ao ambiente externo. Através desse tipo de bloco é possível, por exemplo, ler sensores e estabelecer comunicação através de uma rede.

Bloco de Funções Composto: Um bloco de funções composto pode encapsular vários blocos de funções, independente do seu tipo.

A interface de um bloco de funções é definida por um conjunto de eventos e dados, sendo que tais eventos e dados podem ser de entrada ou saída. A representação de um bloco de funções é dividida em cabeçalho e corpo [Vyatkin 2009], como mostrado na Figura 1a. No cabeçalho são definidos os eventos e um *ECC* (*Execution Control Chart*). O corpo contém as declarações de variáveis e a implementação dos algoritmos. A norma IEC 61499 não define uma linguagem específica para implementação dos algoritmos. A princípio, pode-se utilizar qualquer linguagem de programação. Na implementação desse ambiente a opção foi pela linguagem ST (*Structured Text*), uma das linguagens definidas pela norma IEC 61131-3 [Tiegelkamp 2010].

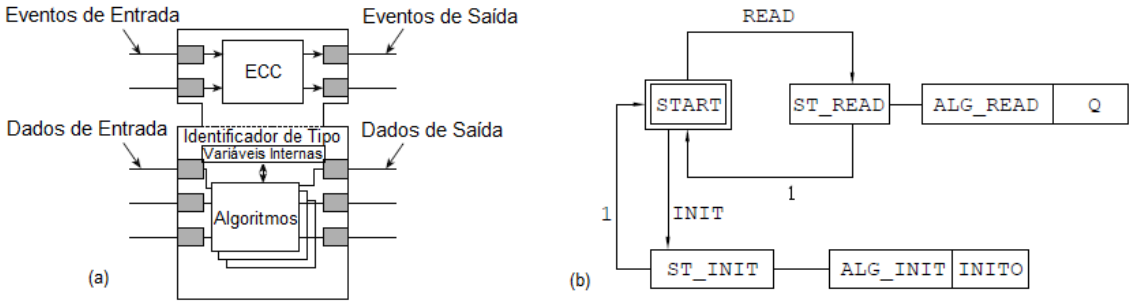


Figura 1. (a) Bloco de Funções (Adaptado de [Lewis 2008]) (b) *ECC*

Uma variável de evento pode assumir dois estados: verdadeiro ou falso. Essas variáveis controlam a execução do bloco de funções. Quando uma variável desse tipo é acionada (ou seja, tem seu valor atribuído como verdade) um dos algoritmos internos é executado. Geralmente o término desse algoritmo aciona um evento de saída para informar o fim da execução. O *ECC* é o elemento responsável por modelar esse fluxo de execução dentro dos blocos de funções. O *ECC* é um tipo de grafo orientado que tem a função de gerenciar a execução do bloco de funções, um exemplo é apresentado na Figura 1b.

As variáveis de dados transportam informações de um bloco para outro. A execução de um algoritmo pode, por exemplo, requerer os valores de variáveis de entrada e atribuir valores às variáveis de saída. Os algoritmos definidos no corpo do bloco de funções têm acesso a todas as variáveis de entrada e saída, com exceção das variáveis de eventos que devem ser utilizadas somente para controle da execução. As variáveis de eventos, declaradas no cabeçalho, não devem ser conectadas às variáveis de dados do corpo. A implementação de uma aplicação é feita através de vários blocos de funções conectados entre si.

3. A Ferramenta

O FBE fornece uma interface gráfica que permite criar, visualizar e editar um sistema implementado através de blocos de funções. O sistema é armazenado como um arquivo XML, conforme definido pela norma IEC 61499. Para a implementar ou alterar os algoritmos associados aos blocos de funções é aberta uma janela para edição de textos. A Figura 2 mostra a interface do FBE, que foi implementado usando a linguagem Lua [Ierusalimschy, Figueiredo e Celes 2006]. Nesse editor também é possível simular a execução dos blocos de funções para depuração.

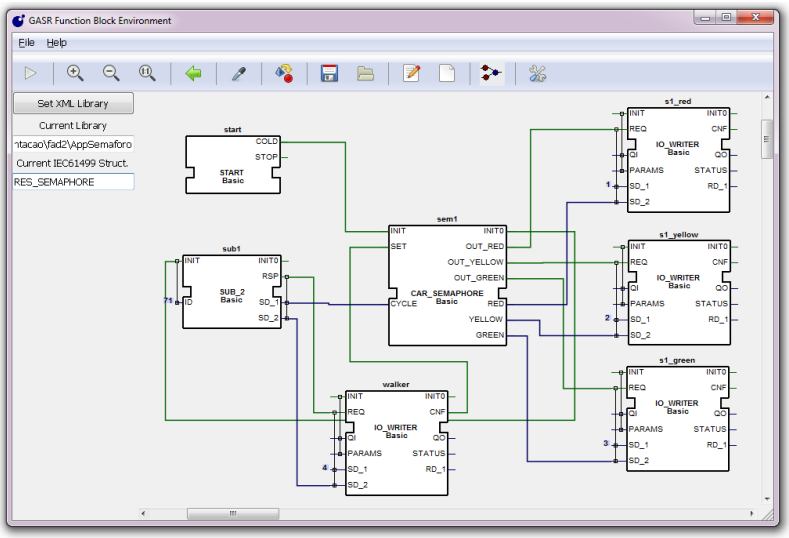


Figura 2. Editor de Blocos de Funções do FBE

O ambiente ICARU-FB é composto por: pré-processador, compilador, ferramenta de atualização e máquina virtual. Conforme pode ser visto na Figura 3, o processo de compilação e transferência dos arquivos para as máquinas virtuais é gerenciado pela ferramenta de atualização, que recebe como entrada dois arquivos XML: um contendo a especificação do sistema (*Sistema.xml*), outro contendo os comandos para envio dos blocos às máquinas virtuais (*Comandos.xml*). O pré-processador é responsável por traduzir a especificação para a linguagem ST e sua principal tarefa é a tradução do ECC. Com relação aos algoritmos, atualmente, o pré-processador aceita apenas implementações em ST, embora a norma IEC 61499 não defina uma linguagem especifica para a implementação.

O compilador é responsável por traduzir o código em linguagem ST para um arquivo binário com o conjunto de instruções da máquina virtual. A opção pela implementação de uma máquina virtual, no lugar do compilador gerar diretamente código em linguagem de montagem ou linguagem C, foi feita para permitir o suporte à reconfiguração dinâmica. Com essa abordagem é possível uma maquina virtual receber comandos para atualização de um ou mais dos seus blocos, enquanto executa outros blocos. Para que a atualização possa ser feita sem interromper a execução dos blocos restantes, a máquina espera que os blocos envolvidos na atualização encerrem seus algoritmos, suspende a execução desses blocos para que as conexões possam ser interrompidas, o bloco é substituído e finalmente as conexões são restabelecidas.

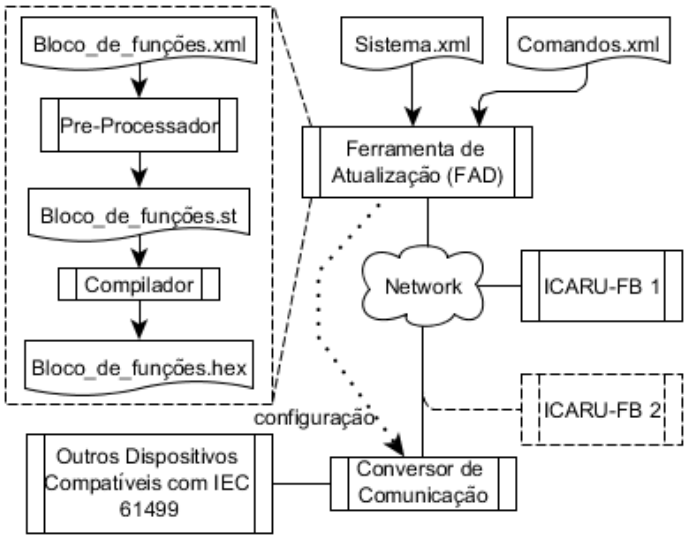


Figura 3. Diagrama da Infraestrutura ICARU-FB & FBE

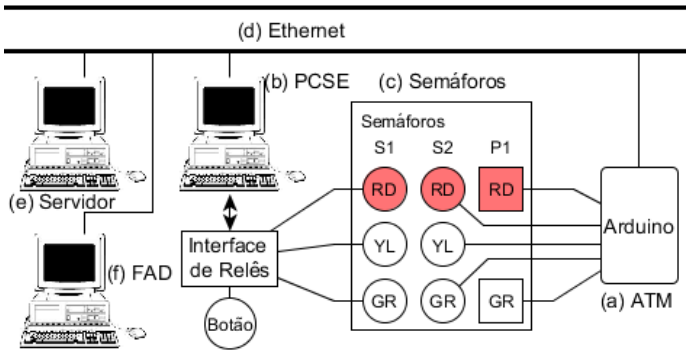


Figura 4. Estudo de Caso: Sistema de Semáforos

Embora a situação ideal fosse que cada dispositivo recebesse as definições dos blocos na forma de um arquivo XML, e o dispositivo fosse responsável pela sua interpretação ou compilação, considerando os escassos recursos do hardware onde a máquina virtual pretende executar, essa solução não é viável. A máquina virtual foi projetada, a exemplo da JVM, como uma arquitetura baseada em pilha, sendo o principal requisito em seu projeto o baixo consumo de recursos. Na versão para *Arduino*, o núcleo da máquina virtual ocupa 13.479 bytes (5,23%) de memória FLASH e apenas 1.643 bytes (20,5%) da memória RAM do microcontrolador. No Arduíno, o arquivo executável, no caso a implementação da máquina virtual, é armazenado na memória FLASH. Esse conteúdo não pode ser alterado com o programa em execução, por isso, os blocos de funções são carregados na memória RAM, permitindo a reconfiguração dinâmica.

4. Experimento

Um dos estudos de caso implementados para validação da ferramenta foi um sistema de semáforos que consiste em dois semáforos para veículos e um para pedestres. Com intuito de verificar a interoperabilidade parte do sistema foi executado em um computador e parte no *Arduino*. A configuração física do sistema é apresentada na Figura 4 e consiste em:

a) Um *Arduino Atmega2560* o qual contém uma versão da máquina virtual;
b) Um computador com interface de saídas para a máquina virtual PCSE;
c) Os semáforos que serão controlados;
d) Rede para comunicação TCP/IP;
e) Um computador para gerenciar a troca de mensagens entre as máquinas virtuais;
f) Um computador para executar a ferramenta de atualização dinâmica.

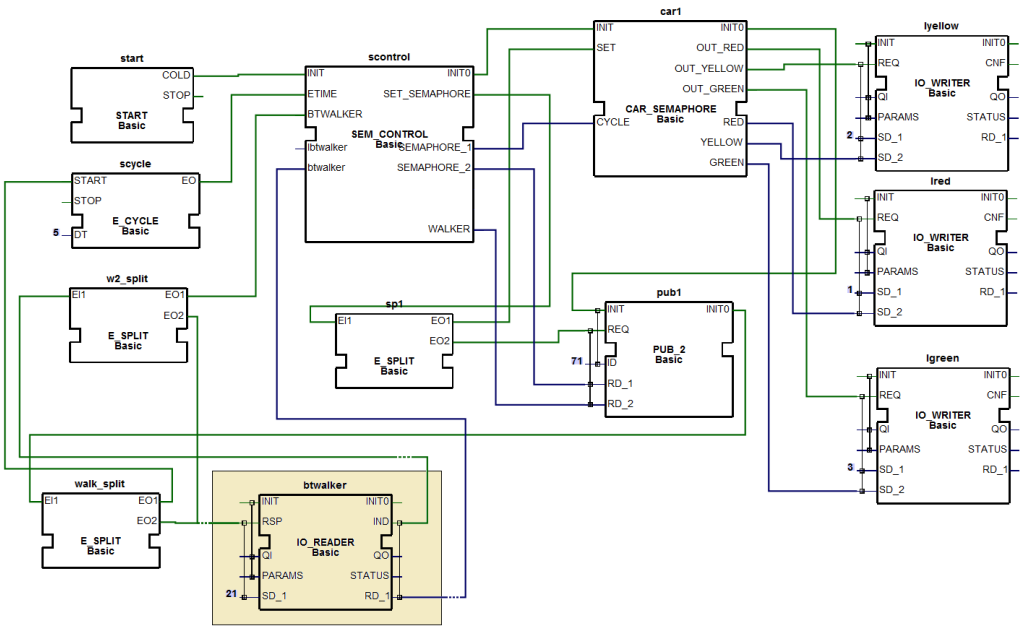


Figura 5. Blocos de Funções executados no Computador

No computador foram executados os blocos de funções responsáveis pela sincronização dos semáforos (*scontrol*), o controle de um dos semáforos para veículos (*car1*) e o botão de acionamento do semáforo para pedestre (*btwalker*). O *Arduino* é responsável por controlar o segundo semáforo para veículos (*car2*) e pelo acionamento do semáforo para pedestre (*walker*). A Figura 5 descreve os blocos que executam no computador e a Figura 6 descreve os blocos que executam no *Arduino*. Os dados para acionamento do semáforo são recebido no *Arduino*, por um bloco do tipo *SUBSCRIBER* (*sub1*), que é um bloco de funções para interface com serviço.

Para verificar o funcionamento da reconfiguração dinâmica o sistema foi colocado em funcionamento em dois passos. Em um primeiro momento apenas os dois semafóros para veículos são colocados em execução. Em um segundo momento, com o sistema em funcionamento, foi adicionado o semáforo para pedestres. As partes em destaque, nas Figuras 5 e 6, indicam os blocos enviados para a máquina virtual em tempo de execução. Note que o bloco *scontrol* da Figura 5 é implementado prevendo a existência de um semáforo para pedestre que irá gerar o evento *btwalker* e tratar o evento *walker*, mas mesmo enquanto os blocos de funções que controlam o semáforo para pedestre não são incluídos os semáforos para carros funcionam normalmente.

Para confirmar que o estudo de caso está em conformidade com a norma IEC 61499 esse exemplo foi também executado no FBRT.

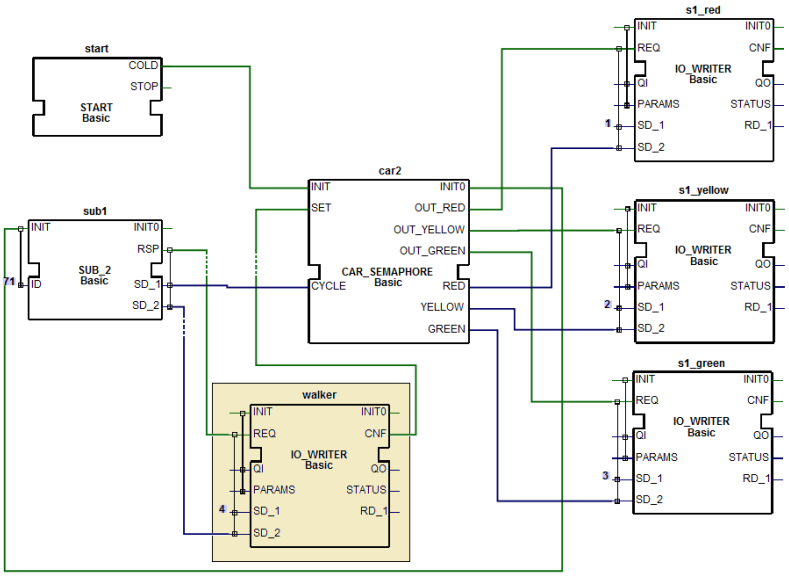


Figura 6. Blocos de Funções executados no *Arduino*

O ambiente também permite a remoção de componentes, mas nesse caso é necessário que o bloco de funções, que será removido, forneça o tratamento de um evento gerado pelo comando de remoção. Ao ocorrer este evento o bloco de funções deve colocar os dispositivos que controla em um estado seguro. Por exemplo, o bloco de funções que controla o semáforo para pedestres pode ser removido apenas se estiver com a luz vermelha acesa. Para remover um bloco que controla um semáforo para carros todos os semáforos do cruzamento devem estar a luz amarela piscando. Esse tratamento é de responsabilidade do programador. O ambiente não fornece qualquer verificação de consistência global do sistema, apenas garante que todos os blocos que serão afetados pela alteração estejam com a execução suspensa e que não serão afetados pelas inclusão ou remoção de ligações até que um novo evento ocorra.

5. Conclusão

No estudo de caso foi possível verificar características como interoperabilidade e portabilidade, uma vez que parte do sistema foi executada em um PC e outra no *Arduino ATmega2560*. Durante a execução foram adicionados blocos e alteradas suas conexões. Essas alterações demostraram o suporte à reconfiguração dinâmica. A aderência à norma IEC 61499 foi verificada através da execução do exemplo também no FBRT. Embora seja necessário um número maior de testes com o ambiente, os resultados obtidos até o momento fornecem uma evidência da viabilidade em se adotar a norma IEC 61499 para implementação de dispositivos inteligentes com baixo custo, ou no desenvolvimento de sistemas mais simples, como por exemplo, sistemas para automação predial ou residencial. É evidente que as limitações do hardware têm consequências, sendo possível a execução de um número reduzido de blocos em cada máquina virtual.

Referências

ARDUINO. <http://www.arduino.cc/>, 2013. Acessado em 04 de Outubro de 2013.

ARM. *ARM: The Architecture for the Digital World*. <http://www.arm.com/>, 1994. Acessado em 26 de Setembro de 2013.

BUCKLEY, T. L. Y. B. A. *The Java Virtual Machine Specification: Java SE 7 Edition*. <http://docs.oracle.com/javase/specs/jvms/se7/html/>: Addison Wesley, 2013.

CHOUINARD, J.; BRENNAN, R. W. Software for Next Generation Automation and Control. In: *IEEE International Conference on Industrial Informatics*., 2006. p. 886–891. ISBN 0-7803-9700-2.

CHRISTENSEN, J. H. *Holobloc, Inc.* <http://www.holobloc.com/doc/fbdk/index.htm>, 2011. Acessado em 04 de Junho de 2013.

ECOS. <http://ecos.sourceware.org/>, 2013. Acessado em 20 de Setembro de 2013.

GASR-FBE. <https://sourceforge.net/projects/gasrfbe/>, 2014. Acessado em 27 de Maio de 2014.

HARBS, E. *CNC-C2: Um Controlador Aderente às Normas ISO 14649 E IEC 61499*. Dissertação (Mestrado em Engenharia Elétrica) — Universidade do Estado de Santa Catarina - UDESC, 2012.

HARBS, E. et al. CNC Servo Acionados Aderentes às Normas ISO 14649 e IEC 61499. In: *VII Congresso Brasileiro de Engenharia de Fabricação*., 2013.

ICARU-FB. <http://sourceforge.net/projects/icarufb/>, 2014. Acessado em 20 de Maio de 2014.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. *Lua 5.1 Reference Manual*. [S.l.]: Lua.org, 2006.

LEWIS, R. *Modelling Control Systems Using IEC 61499: Applying function blocks to distributed systems*. 1th. ed. [S.l.]: Institution of Engineering and Technology, 2008.

PROFACTOR. *Framework for Distributed Industrial Automation and Control*. <http://www.fordiac.org/>, 2008. Acessado em 04 de Junho de 2013.

TIEGELKAMP, K.-H. J. *IEC 61131-3: Programming Industrial Automation Systems*. 2th. ed. [S.l.]: Springer, 2010.

VYATKIN, V. The IEC 61499 Standard and its Semantics. *Industrial Electronics Magazine, IEEE*, v. 3, n. 4, p. 40–48, 2009. ISSN 1932-4529.

VYATKIN, V. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. 2th. ed.: Institution of Engineering and Technology, 2012.

YOONG, L. H.; ROOP, P.; SALCIC, Z. Efficient Implementation of IEC 61499 Function Blocks. In: *IEEE International Conference on Industrial Technology, 2009. ICIT 2009.*, 2009. p. 1 –6.

ZOITL, A. et al. Executing Real-Time Constrained Control Applications Modelled in IEC 61499 with Respect to Dynamic Reconfiguration. In: *INDIN'05. 3rd IEEE International Conference on Industrial Informatics*., 2005. p. 62 – 67.

Restrictifier: a tool to disambiguate pointers at function call sites

Victor Campos[1], **Péricles Alves**[1], **Fernando Pereira**[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Avenida Antônio Carlos, 6627, 31270-010, Belo Horizonte, MG

{victorsc,periclesrafael,fernando}@dcc.ufmg.br

Abstract. *Despite being largely used in imperative languages, pointers are one of the greatest enemies to code optimizations. To circumvent this problem, languages such as C, C++, Rust and Julia offer reserved words, like "restrict" or "unique", which are able to inform the compiler that different function parameters never point to the same memory region. Even with proven applicability, it's still left to the programmer the task of inserting such keywords. This task is, in general, boring and prone to errors. This paper presents Restrictifier, a tool that (i) finds function call sites in which actual parameters do not alias; (ii) clones the functions called at such sites; (iii) adds the "restrict" keyword to the arguments of these functions; and (iv) replaces the original function call by a call to the optimized clone whenever possible.*

Video tutorial: https://youtu.be/1blYSsDV5Qc

1. Introduction

One of the most important characteristics of languages such as C and C++ is the existence of pointers. These have been included in the project of those languages as a way to allow a better control over the memory usage along the life cycle of a program. In spite of providing a broader control over the memory management to the developer, pointers make the operation of optimizing compilers more complex. This is due to, in general, the fact that static languages' compilers aren't able to determine whether two or more pointers may reference the same memory region at some point in the control flow of an application. This limitation blocks several code optimizations [Landi and Ryder 1991]. Aiming the mitigation of this problem, in the last decades there has been a considerable amount of research about efficient techniques of alias analyses [Wilson and Lam 1995, Hind 2001, Whaley and Lam 2004].

In a similar effort, the C standard from 1999 (C99) included the keyword "restrict". This modifier allows the programmer to inform the compiler about pointers that never reference overlapping regions. More specifically, when applied on a pointer argument, the "restrict" word determines the memory region pointed by this parameter cannot be accessed by any other variable inside the function. Despite being available for more than a decade, the usage of "restrict" has been very little among the C community. This phenomenon has been led by two factors: (i) the use of this modifier requires a deep knowledge about memory resources across the program's lifecycle; (ii) by demanding manual intervention from the programmer, the insertion of "restrict" is prone to errors. Our tool aims to move such task to the compiler.

Restrictifier combines code versioning, in the form of function cloning, with static alias analyses to boost the number of disambiguated pointers, which in turn enables more aggressive optimizations on functions that take pointers as arguments. Our method consists in constructing, for each function that receives two or more pointers as arguments, a new version in which these parameters are annotated with "restrict". We then utilize a completely static approach to substitute calls to such functions by calls to their optimized versions. This static approach relies on the different alias analyses implemented by our compiler of choice, LLVM.

2. Overview

```
1  void prefix(int *src, int *dst, int N) {
2    int i, j;
3    for (i = 0; i < N; i++) {
4      dst[i] = 0;
5      for (j = 0; j <= i; j++) {
6        dst[i] += src[j];
7      }
8    }
9  }
10
11 int main() {
12   int N = 100;
13   int A1[N], A2[N];
14   prefix(A1, A2, N);
15 }
```

Figure 1. Example of a target program for Restrictifier.

In this Section, we will use the program in Figure 1 to introduce the technique which Restrictifier incorporates. The function *prefix* receives two pointers as arguments and stores to *dst* the sums of prefixes for each position of the array *src*. If the compiler was able to infer that *src* and *dst* point to completely distinct regions, it could apply more aggressive optimizations on *prefix*, such as Loop Unrolling, Loop Invariant Code Motion, automatic parallelization, and vectorization, making it significantly faster. It seems rather simple, but modern compilers, like LLVM, are not able to optimize fragments of code such as this one.

Our technique starts with the cloning of functions that take two or more pointers as input, applying the "restrict" modifier to the arguments of the cloned version. The clone of *prefix* is the function *prefix_noalias*, shown in Figure 2. Those "restrict" modifiers allow the compiler to assume *src* and *dst* do not overlap. This information permits, for instance, to substitute the assignment to *dst[i]* in the inner loop (Figure 1, line 6) by an assignment to a temporary variable (Figure 2, line 6), executing the store operation only once for each iteration of the outer loop (Figure 2, line 8), thus reducing the number of memory accesses. In this report, we present a tool to promote substitution of function calls by their cloned version, optimized with "restrict", using compiler static analyses.

Next, we present how Restrictifier works on the example in Figure 1.

```
1  void prefix_noalias(int * restrict src, int * restrict dst, int
       N) {
2    int i, j;
3    for (i = 0; i < N; i++) {
4      int tmp = 0;
5      for (j = 0; j <= i; j++) {
6        tmp += src[j];
7      }
8      dst[i] = tmp;
9    }
10 }
11
12 int main() {
13   int N = 100;
14   int A1[N], A2[N];
15   prefix_noalias(A1, A2, N);
16 }
```

Figure 2. Optimized version of *prefix* and the calling context after being processed by Restrictifier tool.

In the context of *prefix*'s function call (Figure 1, line 14), the input arrays clearly do not overlap, since they were allocated separately in the program's stack. For this context, it's possible to substitute the called function by its cloned version using only compilation time analyses, i.e. static alias analyses can detect those memory references are independent. This way, we say this context could be solved by a completely static approach, which defines the presented Restrictifier tool.

3. Pointer Disambiguation

In this tool, we utilize a static approach to disambiguate pointers. This method uses traditional pointer analyses to determine whether memory regions may overlap or not. Efficiency is its greatest advantage: the pointer disambiguation, being statically performed, does not incur cost in the running time of the program.

As initially mentioned, the purely static method employed by the Restrictifier tool uses alias analyses to distinguish, in compilation time, memory regions. There are several algorithms described in the literature that tackle the pointer aliasing problem. In LLVM, there are six different implementations that complement each other, given that they handle different cases. Thus, our tool uses them in conjunction to maximize effectiveness:

- **Basic AA**: the simplest implementation (yet, one of the most effective) present in LLVM. This analysis uses several heuristics, all based in tests that can be done in constant time to disambiguate pointers.
- **Type-based AA**: based on the C99 property stating that pointers of different types cannot overlap.
- **Globals AA**: based on the fact that global variables whose address is not taken cannot overlap other pointers in the program.

- **SCEV AA**: uses *scalar evolution* of integer variables to disambiguate pointers. A variable's scalar evolution is an expression that describes the possible values it may assume during execution. Such information is extracted from loops. For instance, in the loop `for(i = B; i < N; i += S)`, variable `i` has the following evolution: $e(\texttt{i}) = I \times S + B$, I being the loop iteration. Regions indexed by integer variables overlap if their scalar evolutions have a non-empty intersection.
- **Scoped NoAlias**: preserves aliasing information, found by the compiler frontend, between different scopes in the program. This is rather important to achieve a limited form of inter-procedurality of the alias analyses, which are primarily all intra-procedural in LLVM.
- **CFL AA**: this is the most refined [Zhang et al. 2013] alias analysis among the LLVM implementations. However, it is also the most expensive in terms of computational cost, reaching $O(n^3)$ in the worst case.

The static approach, which we describe in this report, does not generate any test to be executed during runtime. Therefore, its runtime overhead is *zero*.

4. Tutorial

In this Section we show how to use Restrictifier to optimize programs. Our tool can be found in `http://cuda.dcc.ufmg.br/restrictification/`. Restrictifier can be used in two ways: via the web tool or installing it on your computer. The former is more contrived, and it should be used for demonstration purposes. To wholely profit from the optimization opportunities, we recommend installing and using our tool locally.

4.1. Web tool

The web tool is a simpler demonstration of the Restrictifier's features. As such, it accepts only one C or C++ file as input, which can be either uploaded or have its contents inserted in a form. Given that it works on a single file as an independent unit, this file should contain (i) the functions to be optimized, i.e. functions that receive pointer arguments, and also (ii) their calls. Failing to comply to both conditions will naturally undermine our optimizer.

The web interface of Restrictifier produces to its user two assembly files written in the LLVM intermediate program representation. The first is the original version of the program, without any optimization from our tool nor LLVM's. The second one, on the other hand, is the final version after going through the entire optimization chain of Restrictifier and LLVM, which takes advantage of "restrict" keywords automatically inserted to aggressively optimize your program.

Along with these two resulting files, the user can also ask to see statistics of the Restrictifier run, which show information such as how many function calls have been substituted by their optimized clones, how many functions have had an optimized clone created for, and how long it has taken to run our tool. Not only that, but the resulting optimized assembly can also be output into the page, if so desired.

Having the optimized LLVM assembly file, you need to have it integrated to your project. Before linking it with the other modules of your C/C++/Objective-C program, you need to compile it for your specific machine. For this, you ought to have installed

the LLVM binaries. These can be acquired by either building LLVM yourself (our page has directions for this), or by installing them using a package manager of your operating system:

- Ubuntu Linux: `apt-get install clang llvm`
- Mac OS X (you need to have Homebrew installed): `brew install llvm`

After having LLVM binaries installed, you can translate from the LLVM assembly to native assembly:

```
llc optimized.ll -o module.s
```

And finally compile the native assembly to an object file.

```
g++/clang++ module.s -o module.o
```

4.2. Local use

Restrictifier is available as a web interface, so that it can be easily used. However, we also provide a downloadable version of it. Using Restrictifier locally provides a better degree of optimization. For instance, one can optimize the entire program, linked, with our tool. As such, functions that are defined inside one unit, but called from others, can also be candidates for optimization. This is not the case when the web tool is used.

In order to use Restrictifier on your own computer, you need to compile it. Our tool's source code is available at our web page, and it is compatible with LLVM 3.6. To use our tool locally, you are required to build LLVM yourself. A very good guide for this can be found in LLVM's tutorial[1]. We recommend that you also build Clang, the C/C++/Obj-C frontend, alongside with LLVM[2]. After building LLVM and Clang, it's the turn of building Restrictifier itself. This process is the same as building any custom LLVM pass. As such, the LLVM's custom pass tutorial[3] can help you.

Assuming you've correctly built LLVM, Clang and Restrictifier, it's pretty simple to use our tool. You should have the LLVM binaries in your path now.

First, we can translate a C/C++ file into an LLVM bitcode using Clang:

```
clang -c -emit-llvm file.c -o file.bc ${CFLAGS}
clang++ -c -emit-llvm file.cpp -o file.bc ${CXXFLAGS}
```

After all units have been compiled, you need to link them together:

```
llvm-link file1.bc file2.bc file3.bc -o program.bc ${LDFLAGS}
```

Now it's time to optimize your program. As such, use the LLVM optimizer:

```
opt -mem2reg -cfl-aa -libcall-aa -globalsmodref-aa -scev-aa
   -scoped-noalias -basicaa -tbaa
```

[1] `http://www.llvm.org/docs/GettingStarted.html`
[2] `http://clang.llvm.org/get_started.html`
[3] `http://llvm.org/docs/WritingAnLLVMPass.html`

```
-load PATH_TO_Restrictifier/AliasFunctionCloning.(so|dylib)
   -afc -O3 -disable-inlining
-o program.optimized.bc
```

This is quite a long command. Let's break it down:

1. The first line invokes the optimizer and calls every alias analysis that LLVM contains.
2. The second line loads the shared library, calls the Restrictifier, and finally performs the whole suite of optimizations (-O3) taking advantage of Restrictifier's work.

What is left to do is just translating it from LLVM assembly to native assembly:

```
llc program.optimized.bc -o program.s
```

And assemble it:

```
g++/clang++ program.s -o program.exe
```

Now you have an optimized version of your program.

5. Use case: prefix sum

In this section, we present a use case for Restrictifier. In Figure 3, we have a program that computes the prefix sums of a source array and stores them to a destination array. This example was already presented in Section 2. Here, though, we observe the effects of using Restrictifier on the running time of the program.

We chose an array size of 400,000 so that runtimes could be long enough to be properly compared. The source array, *A1*, is filled with random integers before being passed on to the core function of the program, *prefix*.

It is easy to see that, if the compiler has knowledge that `src` and `dst` point to non-overlapping regions, *prefix*'s code can be optimized to reduce the number of memory accesses. Particularly, the statement in Figure 3, line 13, which stores to `dst[i]` at every single iteration of the inner loop, can be transformed into a sum that stores to a register, which would then be stored to memory at position `dst[i]` outside of the inner loop. Such optimization can be seen in Figure 2, page 3.

We applied the Restrictifier onto this example and assessed the results. For this small experiment, we used a computer with an Intel Core i5 1.6 GHz processor running Mac OS X 10.10.3. We have taken the mean of 15 runs.

Version	Mean	Std. deviation
Regular	41.743s	1.642s
Restrictified	13.075s	0.826s

Table 1. Comparison of runtimes between regular and optimized versions.

From the Table 1, we clearly see how much the program earned from using Restrictifier. The addition of "restrict" to the arguments of function *prefix* enabled the reduction of the quantity of memory accesses and also the loop's vectorization. These

```
1  void fill(int *array, int N) {
2    int i;
3    for (i = 0; i < N; i++) {
4      array[i] = rand();
5    }
6  }
7
8  void prefix(int *src, int *dst, int N) {
9    int i, j;
10   for (i = 0; i < N; i++) {
11     dst[i] = 0;
12     for (j = 0; j <= i; j++) {
13       dst[i] += src[j];
14     }
15   }
16 }
17
18 int main() {
19   int N = 400000;
20   int A1[N], A2[N];
21   fill(A1, N);
22   prefix(A1, A2, N);
23 }
```

Figure 3. Use case of Restrictifier: prefix sum.

optimizations, only possible because of our tool's work, permitted our use case program to achieve 3.2x of speedup for this input.

We also state that Restrictifier takes a negligible time to run. To illustrate this, we applied our optimization on the largest benchmark from SPEC CPU2006 [Henning 2006], called `403.gcc`, which contains 235,884 lines of code. Restrictifier took 0.67 seconds to run, which is less than the time taken by Global Value Numbering, 3.7 seconds. The latter is an important optimization commonly performed by compilers to eliminate redundant code.

6. Conclusion

This report described our Restrictifier tool. The technique behind our tool is to automatically insert the "restrict" keyword to function pointer arguments, which indicates that these pointers do not reference overlapping memory regions. Such property enables more aggressive compiler optimizations, hence improving efficiency of programs. We developed a web version of our tool to demonstrate its features and we also provide its source code along with a tutorial teaching users how to deploy Restrictifier.

Restrictifier can be found at:

`http://cuda.dcc.ufmg.br/restrictification/`.

References

Henning, J. L. (2006). SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

Hind, M. (2001). Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE*, pages 54–61. ACM.

Landi, W. and Ryder, B. G. (1991). Pointer-induced Aliasing: A Problem Classification. In *POPL*, pages 93–103. ACM.

Whaley, J. and Lam, M. S. (2004). Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, pages 131–144. ACM.

Wilson, R. P. and Lam, M. S. (1995). Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI*, pages 1–12. ACM.

Zhang, Q., Lyu, M., Yuan, H., and Su, Z. (2013). Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis. In *PLDI*, pages 435–446. ACM.

Etino: Colocação Automática de Computação em Hardware Heterogêneo

Douglas do Couto Teixeira, Kézia Andrade, Gleison Souza, Fernando Pereira

[1] Departamento de Ciência da Computação –
Universidade Federal de Minas Gerais (UFMG)
Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

{douglas,kezia.andrade,gleison.mendonca,fernando}@dcc.ufmg.br

Abstract. *Graphics Processing Units (GPUs) revolutionized high performance computing. However, programming these devices is still a hard task. This paper presents* ***Etino****, a tool that aims at reducing the complexity of programming GPUs by hiding details of the underlying hardware as well as inserting OpenACC annotations automatically in programs. Etino uses profiling information to decide what parts of a program should run in the GPU. By sending functions that have high asymptotic complexity to the GPU and keeping those with low complexity in the CPU, Etino is able to improve substantially the running time of programs. Our tool is available at* `http://youtu.be/HdzsRZuPqJM`*.*

Resumo. *As placas de processamento gráfico (GPUs) revolucionaram a programação de alto desempenho, porém a programção desses dispositivos é ainda um desafio. Este artigo apresenta* ***Etino****, uma ferramenta que visa facilitar a programação desses dispositivos, escondendo detalhes de hardware e distribuindo anotações OpenACC em programas de forma automática. Etino usa informação de perfilamento para decidir quais partes de um programa devem executar na GPU. Ao enviar funções de alta complexidade computacional para a GPU e manter funções de baixa complexidade na CPU, Etino é capaz de melhorar substancialmente o tempo de execução de programas. Nossa ferramenta está disponível em* `http://youtu.be/HdzsRZuPqJM`*.*

1. Introdução

As placas de processamento gráfico (GPUs) revolucionaram a programação de alto desempenho, pois elas reduziram o custo do hardware paralelo. A programação desses dispositivos, contudo, é ainda um desafio, pois programadores ainda não são treinados para escrever código que coordene a atuação simultânea de milhares de threads. A fim de lidar com esse problema, a indústria e a academia vêm introduzindo sistemas de anotações, como OpenMP 4.0, OpenSs e OpenACC, que permitem indicar quais partes de um programa C ou Fortran deveriam executar em GPU ou em CPU. Contudo, a tarefa de semear tais anotações no código ainda cabe ao programador.

Apesar de a inserção de diretivas no código esconder detalhes de hardware, ela não resolve todos os problemas do programador: ele ainda precisa identificar quando será vantajoso executar um dado trecho de código na GPU e quando não. Para que a GPU possa ser utilizada, os dados a serem processados precisam ser transferidos para sua memória. Essa transferência de dados entre CPU e GPU é uma operação muito cara, e

inerentemente linear no tamanho dos dados a serem transferidos. Dessa forma, a GPU passa a ser vantajosa quando o custo do trabalho a ser realizado sobre os dados é maior que o custo de mover esses dados entre dispositivos diferentes. Determinar as situações em que a GPU deve ser utilizada é uma tarefa desafiadora, dada a quantidade de fatores envolvidos nesse cálculo.

Na tentativa de retirar do programador o peso de decidir quais trechos de código devem ser movidos para a GPU, nós desenvolvemos uma técnica para automatizar essa decisão. A idéia chave do presente trabalho é utilizar informação de perfilamento (*profiling*) para decidir em que parte do hardware heterogêneo cada computação deve ser executada. Se a complexidade computacional de um trecho de código é super-linear, nós o enviamos para a GPU, doutro modo, o executamos na CPU. Indicamos tal decisão para o gerador de código via diretivas OpenACC [Standard 2013]. Etino exige somente uma intervenção de usuários: eles precisam indicar quais laços possuem iterações independentes via a anotação OpenACC `#pragma acc loop independent`.

A fim de validar nossas idéias, nós as materializamos em um sistema construído a partir da união de três ferramentas: aprof [Coppa et al. 2012], LLVM [Lattner and Adve 2004] e ipmacc [Lashgar et al. 2014]. A primeira dessas ferramentas, aprof, é um perfilador capaz de coletar informações sobre a complexidade assintótica de programas. Especulamos que funções de complexidade super-linear são boas candidatas para serem enviadas à GPU. LLVM é uma infra-estrutura de compilação que possui um *front-end* para C. Nós utilizamos esse compilador para inserir diretivas OpenACC sobre o programa a ser paralelizado. Finalmente, ipmacc é um compilador fonte-a-fonte que traduz o código C aumentado com as diretivas OpenACC para código escrito em C para CUDA. A ferramenta que esse artigo descreve, a partir deste ponto, será chamada *Etino*. Esse nome advém do fato da molécula de etino possuir composição química C^2H^2, a mesma sigla que adotamos para *Colocador de Computação em Hardware Heterogêneo*.

Nossa abordagem, até o presente momento, é totalmente estática: decidimos onde cada função deve executar durante a compilação do programa, e tal decisão não muda durante a sua execução. Ainda assim, nossos resultados são animadores. Os experimentos realizados mostram que nossa ferramenta foi capaz de indicar corretamente os trechos de código que devem ser executados na GPU, de forma similar ao que seria feito por um programador experiente na área. Como consequência dessa precisão, os programas que produzimos – de forma totalmente automática – são até 50x mais rápidos que os programas originais.

2. Colocação Automática de Computação em Hardware Heterogêneo

Conforme mencionado na Seção 1, o objetivo deste trabalho é determinar, com um mínimo de intervenção do usuário, em qual processador cada parte de uma aplicação paralela deve ser executada. Com tal propósito, foi construída Etino, uma ferramenta publicamente disponível, que está representada na Figura 1. Etino agrupa, em um mesmo pacote, três ferramentas bem conhecidas, a saber: o compilador LLVM, o perfilador *aprof* e o tradutor de código ipmacc. Além de ligar tais aplicações, a construção de Etino demandou a implementação de uma análise de complexidade assintótica, e um anotador de código. A iteração entre esses vários produtos de *software* será explicada no restante da

presente seção.

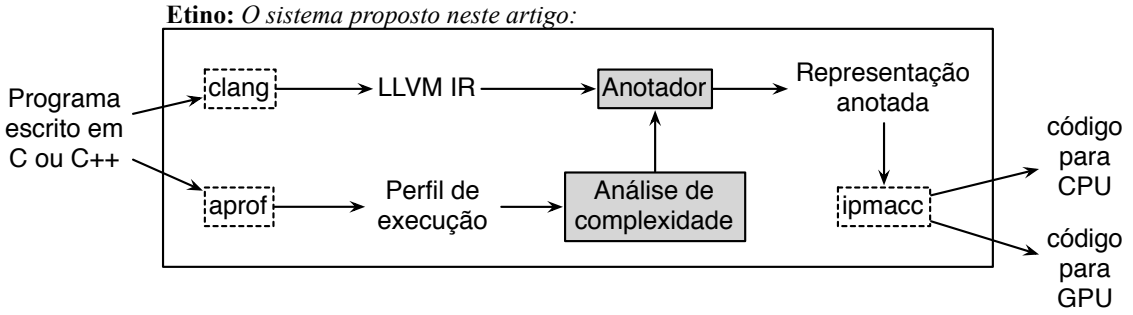


Figure 1. Visão geral do sistema de compilação proposto neste trabalho.

Perfilamento sensível à entrada. A fim de determinar o comportamento assintótico de uma aplicação, Etino produz um perfil de sua execução. Tal perfil é criado por meio da ferramenta *aprof*. Aprof retorna, para cada função de um programa, um gráfico que relata o tamanho da entrada lida com o número de instruções necessário para processar aquela entrada. O tamanho da entrada de um programa é definido como a quantidade de posições de memória que ele lê sem antes escrevê-las. Tal métrica é chamada RMS, sigla de *Read Memory Size* [Coppa et al. 2012].

Uma vez que aprof usa a memória lida como medida de tamanho de entrada, essa ferramenta é robusta o suficiente para determinar a complexidade de funções que usam estruturas de dados esparsas, tais como listas encadeadas e grafos. Essa abordagem, na opinião dos autores desse artigo, é superior às alternativas puramente estáticas de inferência de complexidade [Gawlitza and Monniaux 2012, Gulavani and Gulwani 2008, Gulwani et al. 2009] que foram recentemente introduzidas. A principal desvantagem das abordagens puramente estáticas é o seu curto alcance: elas dependem de entradas de dados regulares e produzem resultados muito conservadores para programas cujo fluxo de controle pode atravessar diversos caminhos. Em face desse tipo de desafio, elas assumem que o comportamento do programa é sempre o pior caso, ainda que tal situação ocorra raramente. Além disso, a experiência dos autores com Loopus, uma ferramenta baseada nas técnicas de Gulwani *et al.* [Gulavani and Gulwani 2008, Gulwani et al. 2009], revela que análises de complexidade puramente estáticas tendem a retornar resultados para uma quantidade relativamente pequena de laços em uma aplicação.

Análise de Complexidade e Anotação de Código. Uma vez produzido um perfil de execução para um programa, Etino passa à fase de análise de complexidade. Nesta fase analisam-se pares $(I \times T)$, onde I é o tamanho da entrada, em RMS, e T é o tempo de execução, medido como o número operações executadas. Enfatiza-se que ambas grandezas são discretas, isto é, um mesmo programa produz sempre o mesmo par $(I \times T)$, para a mesma entrada. A partir de um conjunto de pares, Etino procura encontrar uma curva que melhor se adeque àqueles pontos. Neste estágio de seu desenvolvimento, Etino não utiliza qualquer heurística elaborada: funções cuja complexidade seja super-linear são marcadas para execução na GPU. Considera-se como super-linear qualquer função cujo

coeficiente de regressão linear (R) seja inferior a 0.9, e cuja reta de integração possua inclinação superior a 1.0.

A decisão de para onde enviar cada computação é feita via anotações OpenACC. Etino utiliza uma transformação de código, implementada como um passe LLVM, para inserir tais anotações. Somente laços paralelos são considerados nesta fase. Um laço paralelo é marcado com a diretiva `#pragma acc loop independent`. O usuário de Etino deve indicar quais laços são paralelos: a ferramenta ainda não consegue resolver dependências entre iterações de laços. Cada laço paralelo tem sua complexidade analisada, conforme descrito anteriormente, e aqueles considerados promissores são anotados com a diretiva `#pragma acc kernels`. Etino usa informações de depuração para encontrar os cabeçalhos de laços onde inserir anotações. LLVM adiciona essas informações à sua representação intermediária.

Geração de Código. A geração de código é feita via ipmacc, um compilador de diretivas OpenACC desenvolvido na Universidade de Vitória, Canadá[1]. Ipmacc é um compilador fonte-a-fonte, que, conforme pode ser visto na Figura 1, produz dois tipos de código: funções escritas em C e funções escritas em C para CUDA. O mecanismo de geração de código adotado por ipmacc é simples. Laços que são marcados com a diretiva `kernel` são transformados em funções CUDA. Essa transformação atribui uma *thread* para cada iteração daquele laço. Assim, é vital que não existam dependências entre iterações diferentes, ou o programa resultante dessa transformação pode ficar semanticamente errado. A Figura 2 ilustra essa transformação. O compilador ipmacc é capaz de traduzir o código visto na parte (a) da figura para o código visto na parte (b).

```
void saxpy(int n, float alpha, float *x, float *y) {
 #pragma acc data copyin(x[0:l]) copy(y[0:l]) {
  #pragma acc kernels {
   #pragma acc loop independent {
    for (int i = 0; i < n; i++) {
     y[i] = alpha*x[i] + y[i];
    }
   }
  }
 }
}
```

(a)

```
__global__ void
saxpy(int n, float alpha, float *x, float *y) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < n) {
  y[i] = alpha * x[i] + y[i];
 }
}
```

(b)

Figure 2. (a) Programa escrito com diretivas OpenACC. (b) Programa que ipmacc produz para a entrada vista na parte (a) desta figura.

Execução Completa da Ferramenta. Etino é uma ferramenta que distribui anotações OpenACC em programas de forma automática. Para o uso da ferramenta é necessário executar os seguintes comandos:

1. Geração do binário do programa sequencial: `gcc -O3 matmul.c -o matmul -lm`

[1]Nesse trabalho foi usado a versão de ipmacc de 15 de Março de 2015, disponível neste endereço: `https://github.com/lashgar/ipmacc`

2. Perfilamento: `./inst/bin/valgrind --tool=aprof -single-log=yes matmul`
3. Coleta das principais informações do perfilameto: `./AprofOutput matMulRelatorio.aprof`
4. Geração do código para GPU: `opt -load /path-to-llvm/LLVMInsertPragmas.so -insert-pragmas matmul.ll -stats -debug`
5. Compilação do código para GPU gerado: `ipmacc matmul.c -o matmul`

3. Estudo de Caso

A fim de validar as idéias discutidas neste artigo, esta seção apresenta dois estudos de caso e seus respectivos resultados produzidos com Etino. O hardware utilizado nesses experimentos é descrito a seguir:

CPU: processador Intel Xeon CPU E5-2620 com ciclo de 2.00GHz e 16 GB de memória RAM (DDR2). O sistema operacional usado é Linux Ubuntu 12.04 3.2.0;
GPU: placa gráfica GeForce GTX 670, com 2 GB de memória.

Nesta seção iremos apresentar duas aplicações, escritas em C que foram utilizadas para validar a ferramenta. Os laços independentes de cada programa foram marcados com a diretiva `#pragma loop independent`. Etino não precisa de qualquer outra indicação, além dessa, para adicionar pragmas sobre um programa, indicando que algumas de suas partes deveriam ser usadas na GPU. As aplicações utilizadas são citadas a seguir:

- **mat-mul**: realiza a multiplicação de duas matrizes quadradas de lado n. Complexidade: $O(n^3)$.
- **mat-sum**: realiza a adição de duas matrizes quadradas de lado n. Complexidade: $O(n^2)$.

Perfilamento e Análise de Complexidade O primeiro passo para execução da ferramenta é fazer o perfilamento. A Figura 3 mostra *o perfil de execução* dos dois algoritmos utilizados como caso de uso neste artigo. O perfil de execução de um programa é uma curva que relaciona o tamanho de sua entrada ao tempo necessário para processar aquela entrada. Entradas são medidas em RMS e tempo é medido em número de operações executadas. Junto a cada perfil é mostrado a curva que melhor descreve aquela complexidade, e o coeficiente de correlação (R). Quanto mais próximo de 1.0, mais apropriada é a curva encontrada via regressão polinomial.

Na soma de matrizes na GPU, o custo total será o custo de transferência dos dados da CPU para a GPU ($O(N^2)$) somado ao custo de se realizar esses cálculos na GPU. Em um modelo PRAM [Gibbons 1988] (*Parallel Random Access Memory*) pode-se executar a soma matricial em $O(1)$. Assumimos que essa complexidade é possível em uma GPU, dado o elevado grau de paralelismo desses dispositivos [2]. Assim, o custo total da soma de matrizes na GPU ($O(N^2) + O(1)$) não é inferior ao custo de se realizar essa operação na CPU ($O(N^2)$).

[2]Note que essa é uma aproximação grosseira: uma GPU não é uma máquina PRAM: ela possui uma quantidade limitada de processadores. Porém, para fins práticos, essa suposição serve aos nossos propósitos.

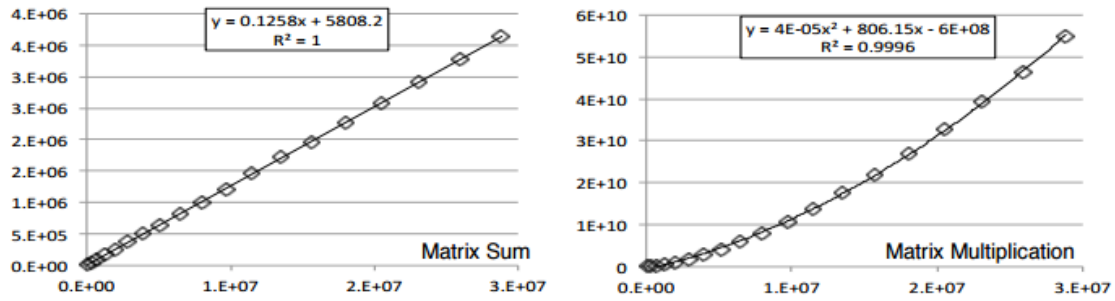


Figure 3. Análise de complexidade dos benchmarks (Eixo X: RMS; Eixo Y: Custo).

Na multiplicação de matrizes o cenário é outro. Nesse caso, o custo computacional inclui o preço de transferência dos dados da CPU Uma implementação mais simples atinge $O(N)$; essa é a complexidade que assumimos para a GPU. Assim, o custo total da multiplicação de matrizes na GPU ($O(N^2) + O(N)$) é inferior ao custo de se realizar essa operação na CPU ($O(N^3)$). Baseado nisso deveríamos mover a multiplicação de matrizes para a GPU e deixar a soma de matrizes na CPU.

Tempo de Execução. A principal meta da ferramenta é permitir que programas aproveitem melhor os recursos disponíveis em hardware heterogêneo. O melhor aproveitamento de recursos, nesse contexto, traduz-se em menor tempo de execução de aplicações. Esta seção compara o tempo de execução de programas na CPU e na GPU. Os programas executados na GPU foram produzidos automaticamente por ipmacc, dadas as pragmas OpenACC que inserimos nos programas. O critério adotado para inserção de pragmas foi o seguinte: Etino foi configurado para marcar como `kernel` todo laço independente, cujo comportamento é superlinear, de acordo com o relatório produzido por *aprof*. Em geral, Etino foi capaz de identificar corretamente os casos em que o código executado no hardware gráfico apresenta ganho de performance em relação ao código executado na CPU.

A Figura 4 mostra os tempos de execução das aplicações citadas na CPU e na GPU, dado o critério acima descrito. A primeira conclusão imediata que obtém-se dos experimentos aqui descritos é que não é vantajoso, em termos de desempenho computacional, mover código linear ou sublinear para a GPU. O custo de mover dados da CPU para o dispositivo gráfico torna irrelevante o ganho em termos de paralelismo. Essa constatação não quer dizer que tais programas jamais deveriam ser enviados para a GPU. Na opinião dos autores deste trabalho, há duas situações em que tal execução é providencial: (i) quando os dados de entrada do algoritmo já estão na GPU, devido a computações prévias e (ii) quando é possível enviar os dados para a GPU em paralelo com processamento útil na CPU. Esse padrão, conhecido como *pipeline de cópia*, é bastante comum entre desenvolvedores CUDA ou OpenCL [Baghsorkhi et al. 2010].

4. Ferramentas Relacionados

O objetivo da ferramenta Etino é decidir, em tempo de compilação, em que parte de um hardware heterogêneo os diferentes laços de um programa devem ser executados. Não existem, até o presente momento, outras ferramentas que têm objetivos similares.

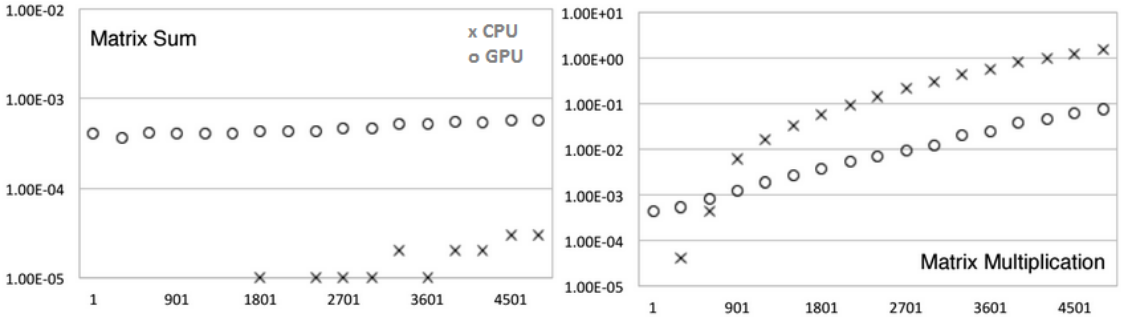


Figure 4. Comparação entre o tempo de execução dos algoritmos na CPU e na GPU (Eixo X: Tamanho de uma matriz quadrada; Eixo Y: Tempo de execução do algoritmo).

Existe, contudo, uma vasta literatura descrevendo diferentes tentativas de reimplementar algoritmos tradicionais em GPUs. Exemplos de tais algoritmos envolvem sequenciamento genético [Sandes and de Melo 2010], ordenação [Cederman and Tsigas 2009], roteamento [Mu et al. 2010], entre outros. Ao contrário desses trabalhos, a técnica de compilação aqui implementada gera código para placas gráficas – sem – a intervenção do usuário. Em outras palavras, a abordagem proposta neste artigo é uma forma automática de reimplementar em C para CUDA código originalmente feito para uma CPU. A implementação automática de código dá, ao desenvolvedor, uma forma rápida de prototipar idéias na placa gráfica.

Embora não se tenha conhecimento de algum trabalho que faça o que a ferramenta proposta faz, existem trabalhos que realizam tarefas semelhantes. Amni *et al.* [Amini et al. 2011], por exemplo, desenvolveram uma técnica para melhorar a cópia de dados entre CPU e GPU re-escrevendo código de forma automática. Contudo, ao contrário do presente trabalho, eles não decidem se computação deve ser executada na GPU. Tal decisão é tomada pelo desenvolvedor de programas.

5. Conclusão

Este artigo apresentou Etino, uma ferramenta que decide em que processadores, GPU ou CPU, cada parte paralela de um programa deve ser executada. Etino exige que seus usuários indiquem, via pragmas OpenACC, quais laços são independentes. Porém, a decisão de onde executar cada computação é automática, e baseia-se tão somente nos resultados produzidos por um perfilador de execução. Os experimentos realizados neste trabalho mostraram que, em geral, é interessante enviar código super-linear para a GPU, enquanto código linear, ou sub-linear, ainda que parelelo, não leva a ganhos de desempenho. O próximo passo no projeto de Etino é adicionar a essa ferramenta a capacidade de reconhecer código divergente [Coutinho et al. 2011]. Esse fenômeno, as divergências, podem acabar com os ganhos de desempenho obtidos ao mover-se código para a GPU.

References

[Amini et al. 2011] Amini, M., Coelho, F., Irigoin, F., and Keryell, R. (2011). Static compilation analysis for host-accelerator communication optimization. In *LCPC*, pages 237–251. Springer.

[Baghsorkhi et al. 2010] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-M. W. (2010). An adaptive performance modeling tool for gpu architectures. In *PPoPP*, pages 105–114. ACM.

[Cederman and Tsigas 2009] Cederman, D. and Tsigas, P. (2009). GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4–24.

[Coppa et al. 2012] Coppa, E., Demetrescu, C., and Finocchi, I. (2012). Input-sensitive profiling. In *PLDI*, pages 89–98. ACM.

[Coutinho et al. 2011] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2011). Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE.

[Gawlitza and Monniaux 2012] Gawlitza, T. M. and Monniaux, D. (2012). Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3).

[Gibbons 1988] Gibbons, A. (1988). *Efficient Parallel Algorithms*. Cambridge University Press.

[Gulavani and Gulwani 2008] Gulavani, B. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, volume 5123 of *LNCS*, pages 370–384. Springer.

[Gulwani et al. 2009] Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009). Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM.

[Lashgar et al. 2014] Lashgar, A., Majidi, A., and Baniasadi, A. (2014). IPMACC: open source openacc to cuda/opencl translator. *CoRR*, abs/1412.1127.

[Lattner and Adve 2004] Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

[Mu et al. 2010] Mu, S., Zhang, X., Zhang, N., Lu, J., Deng, Y. S., and Zhang, S. (2010). IP routing processing with graphic processors. In *DATE*, pages 93–98. IEEE.

[Sandes and de Melo 2010] Sandes, E. F. O. and de Melo, A. C. M. (2010). Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *PPoPP*, pages 137–146. ACM.

[Standard 2013] Standard, O. (2013). The openacc programming interface. Technical report, CAPs.

Multi-Level Mutation Testing of Java and AspectJ Programs Supported by the Proteum/AJv2 Tool

Filipe Gomes Leme[1], **Fabiano Cutigi Ferrari**[2]
José Carlos Maldonado[1], **Awais Rashid**[3]

[1]Computer Systems Department – University of São Paulo (ICMC/USP) – Brazil

[2]Computing Department – Federal University of São Carlos (UFSCar) – Brazil

[3]Computing Department – Lancaster University – United Kingdom

leme.fg@gmail.com, fabiano@dc.ufscar.br,
jcmaldon@icmc.usp.br, marash@comp.lancs.ac.uk

Abstract. *The application of testing techniques and the associated test selection criteria strongly relies on adequate tooling support. This paper describes the evolution of Proteum/AJ, a tool originally conceived to support the mutation testing of aspect-oriented (AO) programs. Proteum/AJ automates the application of AspectJ-specific mutation operators. Its evolution, named Proteum/AJv2, also supports the application of unit mutation operators in both Java (object-oriented) and AspectJ programs through a newly graphical user interface (GUI). We show how the tool architecture and the use of design patterns facilitated the addition of mutation operators as well as the GUI development. We also describe results of a preliminary assessment study that comprised the application of both traditional (unit) and AO-specific mutation operators in a complete mutation testing cycle.*

Demo Video: `http://www.youtube.com/watch?v=v1hxRKifXbc`

1. Introduction

Historically, mutation testing [5] has shown to be an effective test selection criterion to evaluate existing test sets [10]. Furthermore, it also leads to the creation of test sets that are effective to reveal software faults. Using mutation operators, testers can create various slightly modified versions of a program, called mutants. Each mutant contains a single fault and composes the set of test requirements that should be covered by the test set. Note that due to the large number of mutants that testers must handle (*i.e.* create, execute and analyse), this testing criterion strongly relies on automated mechanisms.

In previous research [6], mutation testing was customised to software developed with aspect-oriented programming (AOP), a relatively recent software development technique [11]. The *Proteum/AJ* tool [7] was built to support the application of the approach to programs written in AspectJ, which represents a mainstream AOP technology. The tool automates a set of AspectJ-specific mutation operators [6], and supports all steps of mutation testing, such as mutant generation, test case execution and mutant analysis. Note that such operators target basic AO elements such as pointcuts and advices. As a consequence, they impact on the interfaces between aspects and the base code (*i.e.* their communication points), so that testing using such operators can be classified at the integration level.

This paper describes *Proteum/AJv2*, which is an evolution of *Proteum/AJ* to support the mutation testing of both object-oriented (OO) and aspect-oriented (AO) programs. The novel features of *Proteum/AJv2*, when compared to its predecessor, are: (i) a set of unit mutation operators that can be applied to both Java and AspectJ programs; (ii) a newly developed graphical user interface; (iii) enhanced visualisation and manipulation of mutation targets and mutants; (iv) customised test results reports; and (v) simultaneous management of multiple test projects. To the best of our knowledge, *Proteum/AJv2* is the only tool that

automates the mutation testing of programs developed under these two paradigms. Thus, it represents a step towards the integrated testing of OO and AO programs.

In this paper, Section 2 describes the tool architecture and how it eases the addition of new mutation operators as well as the new GUI development. Section 3 describes the process of creating mutants in *Proteum/AJv2* and how this is supported by the GUI. An evaluation study, together with the obtained results, is presented in Section 4. To conclude, we compare *Proteum/AJv2* with other mutation testing tools (Section 5) and present some final remarks (Section 6).

2. Proteum/AJv2 Architecture

The tool architecture partially implements a reference architecture for software testing tools called *RefTEST* [13]. *RefTEST* is based on the separation of concerns (SoC) principles, on the Model-View-Controller (MVC) and three-tier architectural patterns, and on the ISO/IEC 12207 standard. *Proteum/AJv2* largely benefits from the reuse of domain knowledge contained in *RefTEST* (organisational and supporting modules) and its easy integration with other life-cycle tools (e.g. requirements and implementation), which has guided the tool structuring in terms of functionalities and module interactions.

The *Proteum/AJv2* architecture is depicted in Figure 1. To clarify, *Proteum/AJv2* does not currently include features defined in *RefTEST* related to supporting and organisational activities[1] (e.g. documentation and configuration management). *RefTEST* also requires a set of service tools (e.g. persistence and access control), which is partially implemented with support of the iBATIS framework[2] for data persistence.

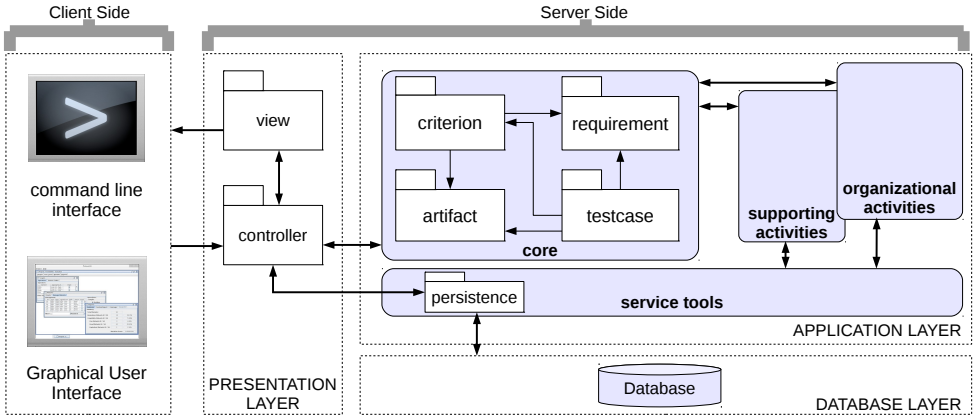


Figure 1: *Proteum/AJv2* Architecture

The *Server Side* includes MVC modules. The alternative user interfaces appears on the *Client Side*. The core comprises the main concepts that should be handled by testing tools, as proposed in *RefTEST*. In *Proteum/AJv2*, *criterion* maps to mutation testing, *artefact* maps to source code, *requirement* maps to mutant, and *test case* maps to the test case itself. For example, the *criterion* module handles the mutant generation, compilation and analysis, while the *test case module* handles test case execution and and evaluation. Changes required to provide support for unit operators and to build the GUI are described next.

The Mutation Engine: Since the tool architecture was designed with special attention to the evolution and maintenance properties, adding support to unit mutation operators has not required drastic changes in the original structure. Figure 2 shows a simplified UML diagram

[1] More details can be found elsewhere [13].

[2] `http://attic.apache.org/projects/ibatis.html` (07/05/2015).

that illustrates some internal details of the *Proteum/AJv2* mutation engine. The classes with grey background represent additions implemented in this version of the tool. Each mutation operator is encapsulated within a specific class. Table 1 lists the unit mutation operators supported by *Proteum/AJv2*. More details about the operators are given in Section 3.

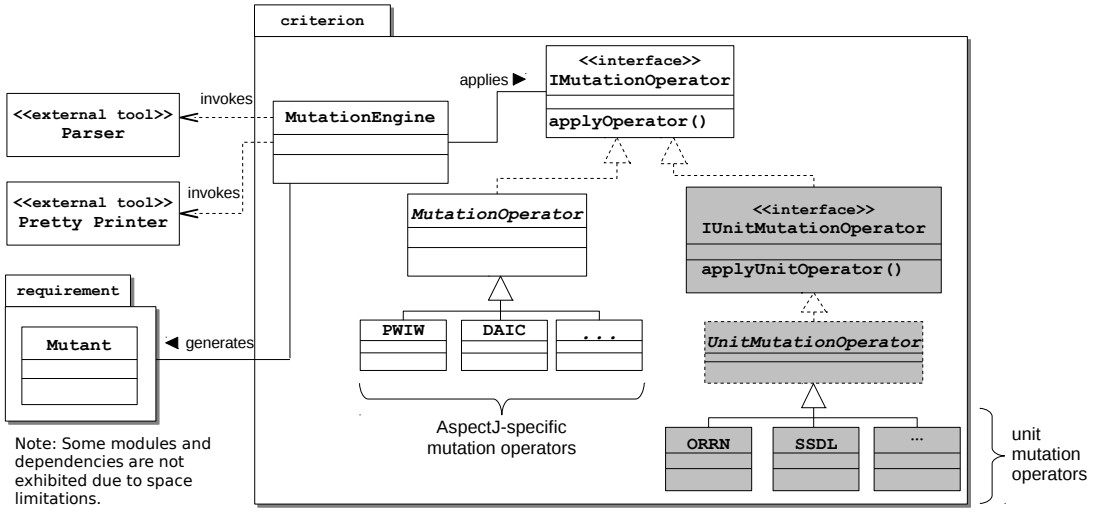


Figure 2: Details of the *Proteum/AJv2*'s mutation engine

The tool allows the selection of specific internal targets (methods and advices) to be mutated. This is realised through the `IUnitMutationOperator` interface, which contains a method whose parameters include a list of internal target names. This enforces client classes to provide such a list. It is implemented by each unit mutation operator, *i.e.* the children of the `UnitMutationOperator` abstract class.

The Graphical User Interface: *Proteum/AJv2* brings a newly developed graphical user interface (GUI). The creation of GUIs helps to increase the adoption of testing tools, which otherwise represents a barrier for both academia and industry [9].

The set of requirements for the *Proteum/AJv2* GUI was derived from existing testing tools such as *Proteum* [4] and *JaBUTi* [16]. These requirements can be divided into: (i) management of test projects; (ii) test case handling (e.g. addition and visualisation); (iii) test requirement generation (mutants); and (iv) report generation. To meet such requirements, we used the *Mediator*, *Observer*, *Command* and *Singleton* design patterns [8]. *Observer* controls the updates of graphical components in the interface. *Command* and *Mediator* encapsulate events related to the graphical components and handle such events. Finally, *Singleton* guarantees the access to the main frame to the other components.

3. Proteum/AJv2 Main Features

This section describes the main features of *Proteum/AJv2* in terms of the mutation process supported by the tool and how this process is enhanced by the addition of the new features listed in Section 1. Note that *Proteum/AJv2* licence is still under definition; the reader can contact the authors for further information.

Running Example: To illustrate the usage of *Proteum/AJv2* and its features, we selected a customised version of the *Telecom* application [2], which is originally distributed together the AspectJ tools. *Telecom* simulates telephony calls and conferences between two or more local and long distance customers.

Figure 3 depicts the classes and aspects included in the application. The main basic classes are `Call`, `Customer` and `Connection` (`Local` or `LongDistance`), which

model the basic entities of the system. The `Billing` and `Timing` aspects implement the billing and timing concerns, respectively, with support of `Timer`, a stopwatch functionality.

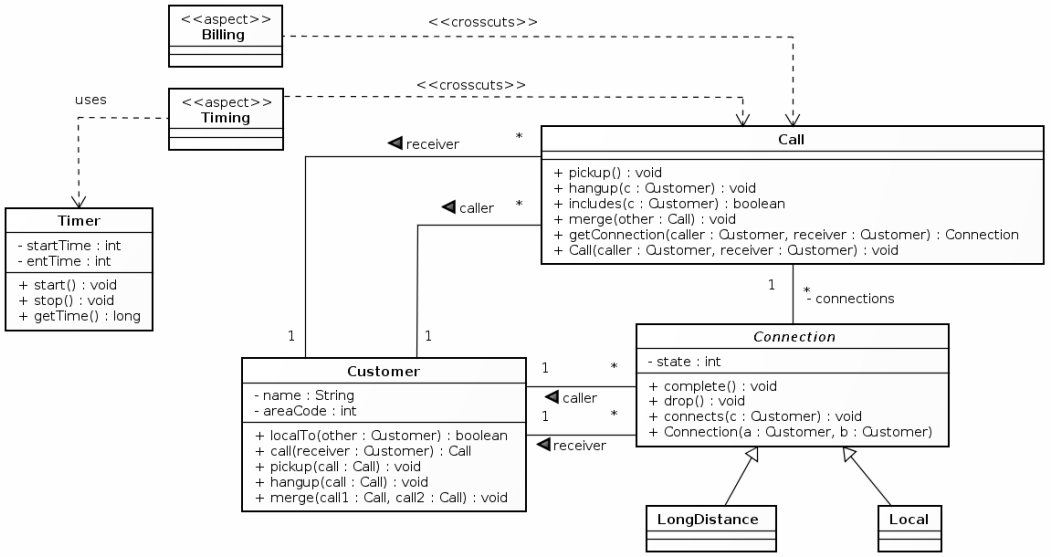


Figure 3: Class diagram of the *Telecom* application

The Mutation Process Supported by Proteum/AJv2: Mutation testing has a high computational cost due to the execution and analysis of numerous mutants. This motivates researchers to spend effort to identify the so-called *sufficient mutation operators* [14], which have low costs (in terms of the number of produced mutants) and high effectiveness in revealing faults [14].

Based on this previous knowledge, the list of unit mutation operators introduced in *Proteum/AJv2* consists of a group of sufficient operators identified for C programs [3, 14] and further validated by Vincenzi [15][3]. They are listed in Table 1[4]. Full descriptions can be found in the original report by Agrawal et al. [1].

Table 1: Unit mutation operators implemented in *Proteum/AJv2*

Operator	Description
CGCR	Constant replacement using global constants
CLCR	Constant replacement using local constants
CGSR	Scalar variable replacement using global constants
CLSR	Scalar variable replacement using local constants
VDTR	Adds a trap function in scalar variables to test if variable will be zero, positive and negative
VTWD	Changes the value of scalar variables for its value predecessor / successor
OASN	Replacement of an arithmetic operator with shift operator
OEBA	Replacement of a plain assignment with bitwise assignment
ORRN	Replacement of a relational operator with other relational operator
SDWD	Statement `do-while` replacement with `while` statement
SMTC	Adds a statement in every loop to force execute at most *n* times
SSDL	Systematically removes each statement

Once the mutants are created, the user can compile the mutants and run *JUnit* test cases, for which the tool will collect the results and compare them to the results from the original application. Note that for some AspectJ-specific mutation operators, the tool is able to automatically detect equivalent mutants through the analysis of join point shadows [7].

[3]According to Vincenzi [15], the OLBN operator [1] – which is included in the sufficient set – is not applicable to Java programs. The OEBA operator, on the other hand, has proved to be efficient in some studies [3], and hence has been included in *Proteum/AJv2*.

[4]The AspectJ-specific mutation operators [6] of *Proteum/AJ* are also available in *Proteum/AJv2*.

The full set of features introduced in *Proteum/AJv2* are not herein described due to space constraints. Amongst them, we highlight: (i) multiple test project creation and management; (ii) selective mutant execution; (iii) individual test case execution; (iv) test case activation/deactivation; (v) mutant visualisation; and (vi) source class/aspect inclusion/exclusion. *Proteum/AJv2* can also be operated through a command line interface, although this paper focuses on the GUI, as next presented.

GUI Features: To create a new test project, the user needs to first create a compressed file (in ZIP format) that includes the full source code of the system under testing (SUT) and also compilation and test execution directives, defined as Apache *Ant*[5] tasks. Once the SUT is submitted to the tool, test project customisation in several ways is facilitated by the GUI (illustrated in Figure 4a).

In the *Targets* window (Figure 4b), the user can select which source files will be tested and also choose the mutation operators to be applied to the SUT. For each source file, the user can select internal elements (methods and advices) to be mutated.

The *Mutants* window exhibits the set of mutants for the current test project. In the *Manage Mutants* tab, the user can check the details of each mutant such as the original and the mutated source files, the differences between these files (Figure 4c), and the mutant status. The user can also filter the mutants by mutation operators. On the *Create* tab, options related to mutant generation (e.g. re-generating all mutants or generating mutants for newly selected mutation operators) are available.

The *Reports* window shows reports regarding the current test project results. For example, it shows the number of generated mutants, the number of anomalous (i.e. non-compilable) mutants, the number of dead mutants and the current mutation score, which is the rate of dead mutants in relation to the total number of non-equivalent, generated mutants. The tool also permits the user to generate an external report with results filtered by mutant status. On the *Coverage* tab, the user can analyse the mutant scores broke down either by applied operators (Figure 4d) or test cases.

Finally, the *Test Cases* window (not shown in Figure 4) allows the selection of test case files to be executed (*Ant* directives are provided with the SUT), inclusion of new test case files and the activation/deactivation of test cases.

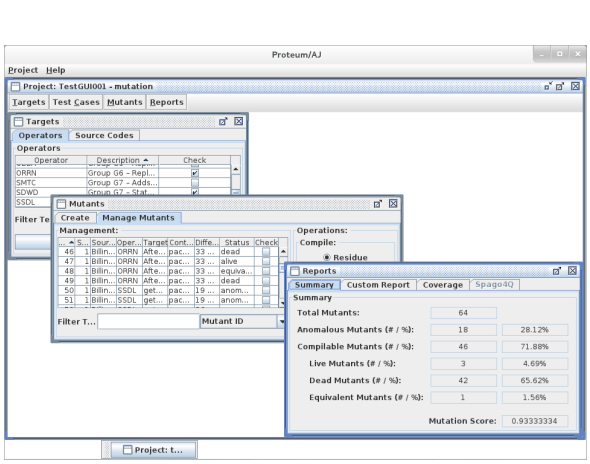
4. Evaluation

This section describes an evaluation study of *Proteum/AJv2*. The study aimed a preliminary assessment of the tool in terms of completeness and correctness. To achieve this goal, we executed a full mutation testing process using *Proteum/AJv2* over two versions of the *Telecom* application, early introduced in Section 3. The first version is purely object-oriented (implemented in Java), while the second includes some concerns modularised with aspects (implemented in AspectJ), as depicted in Figure 3. Note that the OO version of *Telecom* was obtained by inserting aspect-related behaviour into the core classes such as `Call`, `Connection` and `Customer`.

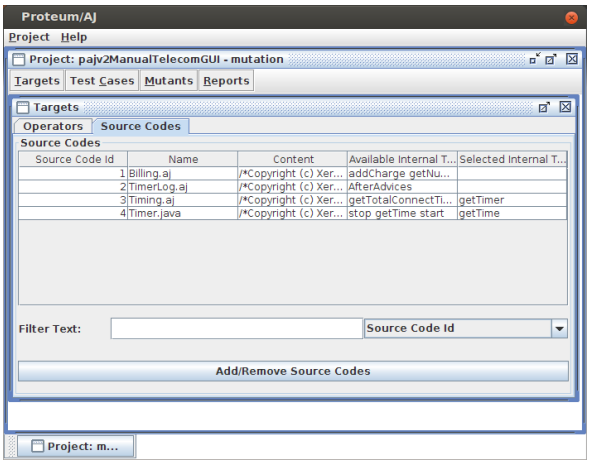
Applied methodology: After setting up the *Telecom* application to be used during the study, we systematically defined functional test cases by analysing and identifying our system inputs and outputs, their valid and invalid equivalence classes and boundary values. Then, we implemented test cases to cover such test conditions using *JUnit* for both projects (*Telecom OO* and *Telecom AO*).

With both applications ready to be tested with *Proteum/AJv2*, we applied the mutation testing considering three different scenarios, as shown in Table 2. Scenarios 1 and 2

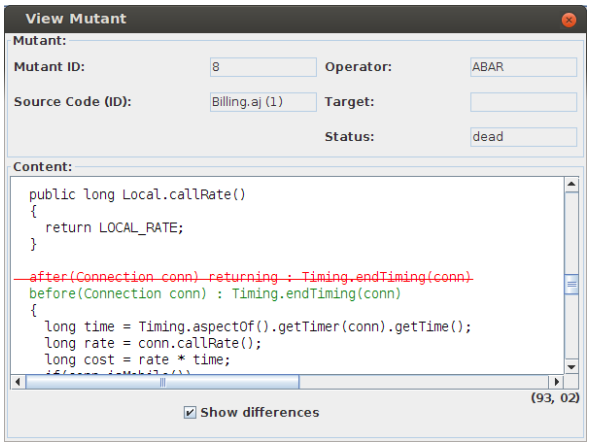
[5] `http://ant.apache.org/` (07/05/2015).



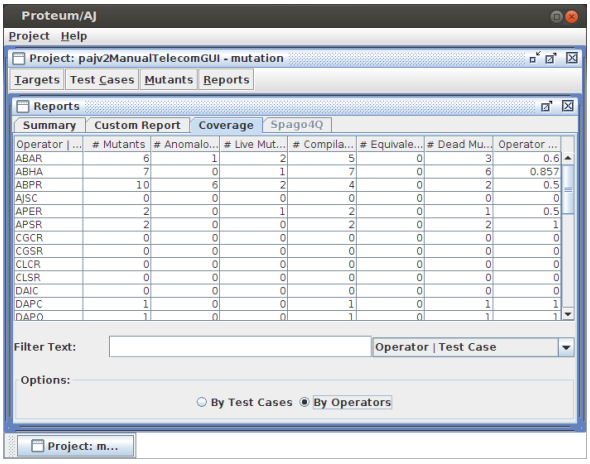
(a) *Proteum/AJv2* GUI overview



(b) Source code selection tab



(c) Mutant difference view



(d) Coverage report by mutation operator

Figure 4: Examples of windows available in the *Proteum/AJv2*'s GUI.

address the newly included OO operators and scenario 3 encompass only AO mutation operators In order to gain confidence on the evaluation results, we followed the same procedure twice and, as expected, results for the two rounds were identical.

To execute each of those scenarios we followed six steps: (A) Creation of a test project; (B) Selection of desired targets including classes, aspects, operations (all methods and advices were selected) and mutation operators; (C) Generation and compilation of mutants; (D) Execution of mutants using the functional-adequate test set; (E) Creation of new test cases for live mutants (mutant adequate set); (F) Classification of equivalent mutants.

Results: After applying the planned steps to all three scenarios, we were able to gather and synthesise the results presented in Table 3. The table shows the evolution of the mutant sets during the last four steps. For example, the number of mutants for the second scenario was 251, in which unit operators were applied to classes and aspects of *Telecom AO*. After running the initial test set (step D), 112 mutants were killed and 71 remained alive. The enhancement of the test set (step E) killed more 41 mutants (153 killed, in total). The remaining mutants were set as equivalent (step F), thus reaching a mutation score of 1.0.

The sizes of the new mutation-adequate test sets for each of the projects (step E) are shown in Table 4. For example, in the second scenario (*i.e.* testing of *Telecom AO* using

Table 2: Selected targets for each scenario

Application	Scenario	Targets		
		Classes	Aspects	Operators
Telecom OO	OO testing (1)	Call, Connection, Customer, Local, LongDistance, Timer	-	CGCR CLCR CGSR CLSR VDTR VTWD OASN OEBA ORRN SDWD SMTC SSDL
Telecom AO	OO testing (2)	Call, Connection, Customer, Local, LongDistance, Timer	Billing, Timing	CGCR CLCR CGSR CLSR VDTR VTWD OASN OEBA ORRN SDWD SMTC SSDL
Telecom AO	AO testing (3)	Call, Connection, Customer, Local, LongDistance, Timer	Billing, Timing	ABAR ABHA ABPR AJSC APER APSR DAIC DAPC DAPO DEWC DSSR PCCC PCCE PCCR PCGS PCLO PCTT POAC POEC POPL PSDR PSWR PWAR PWIW

Table 3: Results

Application	Telecom OO				Telecom AO - OO testing				Telecom AO - AO testing			
Execution Step	C	D	E	F	C	D	E	F	C	D	E	F
Anomalous	80	80	80	80	68	68	68	68	6	6	6	6
Alive	218	81	29	0	183	71	30	0	55	17	8	0
Dead	0	137	189	189	0	112	153	153	0	11	20	20
Equivalents	0	0	0	29	0	0	0	30	0	27	27	35
Total	298	298	298	298	251	251	251	251	61	61	61	61
Mutation Score	0	0.628	0.867	1.0	0	0.612	0.836	1.0	0	0.393	0.714	1.0

only unit operators), we added seven new test cases to achieve full mutation score. Note that the final test set was not revised to be an optimal (*i.e.* minimum) set.

Table 4: Number of tests cases on each evaluation step.

Test case set size	***Telecom OO***	***Telecom AO* - Unit**	***Telecom AO* - Aspect**
Initial functional	22	22	22
Mutation adequate	13	7	1
Final set size	**35**	**29**	**23**

This case study allowed us to verify that *Proteum/AJv2* is a tool capable of supporting the entire mutation testing process: from mutant generation, test case execution to mutant analysis. Also, the new GUI proved to be very helpful for quickly customising the testing targets, specific mutant execution and analysis and for generating test reports.

5. Related Work and Limitations

To the best of our knowledge, *Proteum/AJv2* is the first mutation testing tool that supports unit mutation testing of both Java and AspectJ programs, as well as AspectJ-specific mutations. Other tools like *Jester*[6], $\mu Java$[7] and *PIT*[8] only support the mutation testing of Java programs. In particular, $\mu Java$ supports mutation testing at the unit ant class levels, while the others only address the unit level. Besides them, *AjMutator*[9] is a tool that supports the mutation testing of AspectJ programs based on a subset of the AspectJ-specific mutation operators implemented in *Proteum/AJv2*.

[6] `http://jester.sourceforge.net/` (07/05/2015).
[7] `http://cs.gmu.edu/~offutt/mujava/` (07/05/2015).
[8] `http://pitest.org/` (07/05/2015).
[9] `http://www.irisa.fr/triskell/Software/protos/AjMutator/` (07/05/2015).

However, these tools have limitations specially related to the test project management and results storage. *Proteum/AJv2* leverages previous knowledge on mutation tools from its developers' research group, thus allowing for a wide range of test project configurations, as well as experimental procedures through its interfaces.

One negative aspect of *Proteum/AJv2* is that it requires an infrastructure setup to execute. It includes the installation of the *AspectJ-front* toolkit and database setup. Besides, *Proteum/AJv2* can only be executed in Linux-based environments, due to its dependence on *AspectJ-front*. The integration of *Proteum/AJv2* with the Eclipse IDE and the development of a pure Java parser could resolve that issue.

6. Final Remarks

We described the main features of the *Proteum/AJv2* tool that supports the mutation testing of Java and AspectJ programs. Similar to its predecessor [7], *Proteum/AJv2* leverages previous knowledge on the development of testing tools [4, 16] and on reference architectures [13] to configure an integrated environment for testing Java and AspectJ applications.

The presented version of the tool was used to support a master's study conducted by Lacerda that investigated the cost reduction of mutation testing for AO programs [12]. Feedback from Lacerda and Ferrari allowed us to not only fix some bugs but also implement improvements related to the GUI usability.

We are currently planning a series of improvements in *Proteum/AJv2* to support experimentation. This includes creating parameterised scripts to execute a chain of tasks such as test project creation, mutation generation, test case execution and coverage analysis. The design of customised reports are also included in the improvement plan.

Acknowledgements

We thank the financial support received from CNPq (Universal Grant #485235/2013-7), FAPESP (grants #2011/21515-3 and #05/55403-6), and UFSCar (RTN grant).

References

[1] Agrawal et al., H. (1989). Design of mutant operators for the C programming language. Tech. Report SERC-TR41-P, Purdue University, West Lafayette/IN - USA.

[2] Alves, P., Figueiredo, E., and Ferrari, F. C. (2014). Avoiding code pitfalls in aspect-oriented programming. In *SBLP'14*, pages 31–46 (LNCS v.8771). Springer.

[3] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verif. & Reliab.*, 11(2):113–136.

[4] Delamaro, M. E. and Maldonado, J. C. (1996). Proteum: A tool for the assessment of test adequacy for C programs. In *PCS Conference*, pages 79–95.

[5] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43.

[6] Ferrari, F. C., Maldonado, J. C., and Rashid, A. (2008). Mutation testing for aspect-oriented programs. In *ICST'08*, pages 52–61. IEEE.

[7] Ferrari, F. C., Nakagawa, E. Y., Rashid, A., and Maldonado, J. C. (2010). Automating the mutation testing of aspect-oriented Java programs. In *AST'10*, pages 51–58. ACM.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Pattern, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[9] Horgan, J. R. and Mathur, A. P. (1992). Assessing testing tools in research and education. *IEEE Software*, 9(3):61–69.

[10] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[11] Kiczales et al., G. (1997). Aspect-oriented programming. In *ECOOP'97*, pages 220–242 (LNCS v.1241). Springer.

[12] Lacerda, J. T. S. and Ferrari, F. C. (2014). Towards the establishment of a sufficient set of mutation operators for AspectJ programs. In *SAST'14*, volume 2, pages 21–30. Brazilian Computer Society.

[13] Nakagawa, E. Y., Simão, A. S., Ferrari, F. C., and Maldonado, J. C. (2007). Towards a reference architecture for software testing tools. In *SEKE'07*, pages 157–162.

[14] Offutt et al., J. (1996). An experimental determination of sufficient mutant operators. *ACM TOSEM*, 5(2):99–118.

[15] Vincenzi, A. M. R. (2004). *Object-oriented: Definition, Implementation and Analysis of Validation and Testing Resources*. PhD thesis, ICMC/USP, São Carlos, SP - Brazil.

[16] Vincenzi, A. M. R., Wong, W. E., Delamaro, M. E., and Maldonado, J. C. (2003). Jabuti: A coverage analysis tool for java programs. In *SBES'03*, pages 79–84.

MTControol: Ferramenta de Apoio à Recomendação e Controle de Critérios de Teste em Aplicações Móveis

Juliana P. do Nascimento, Jonathas S. dos Santos, Arilo C. Dias-Neto

Instituto de Computação – Universidade Federal do Amazonas (UFAM) n° 6.200 – 69.007-000 – Manaus – AM – Brasil

{jpn,jss,arilo}@icomp.ufam.edu.br

Abstract. *Nowadays, guaranteeing the quality of mobile applications has become essential because of the high complexity and importance that they have assumed in our lives. In this context, tools that allow the user to ensure the quality of mobile applications in different platforms and devices become excellent alternatives. This work aims to present the features of the MTControol tool, which aims to recommend and manage test cases for mobile applications based on test criteria published by companies responsible by mobile platforms.*

Resumo. *Atualmente, garantir a qualidade de aplicativos móveis se tornou essencial por conta da alta complexidade e importância que eles têm assumido em nosso dia-a-dia. Neste contexto, ferramentas que possibilitem ao usuário garantir a qualidade dessas aplicações em diversas plataformas e dispositivos tornam-se excelentes alternativas. Este artigo apresenta as funcionalidades da ferramenta MTControol, que tem como objetivo recomendar e gerenciar casos de teste para aplicativos móveis baseados em critérios de teste publicados por empresas responsáveis por plataformas móveis.*

Link do vídeo. *https://goo.gl/e0tPGi*

1. Introdução

A construção de aplicações para dispositivos móveis, ou simplesmente *apps*, de qualidade e que atendam a todas as necessidades dos usuários já deixou de ser um diferencial no mercado e tornou-se uma exigência Harisson *et al*. (2013). Uma das formas de aprimorar a qualidade das *apps* é por meio da aplicação de técnicas de teste de software.

Segundo Dantas *et al*. (2009), o teste em *apps* requer novos desafios devido às limitações de recursos existentes nestes dispositivos e a necessidade de interação das *apps* com os demais serviços providos pelo dispositivo, como por exemplo atender chamadas, oscilação de carga da bateria, comunicação por redes *Wi-Fi*, *Bluetooth*, acesso à câmera fotográfica, dentre outros. Além disso, a diversidade plataformas e dispositivos móveis ocasiona um grande esforço para validação das *apps*, visto que seu correto funcionamento precisa ser assegurado nos diversos ambientes para os quais ela foi idealizada.

Como uma tentativa de reduzir o esforço, padronizar e estruturar os testes em *apps*, as empresas responsáveis pelas principais plataformas móveis usualmente publicam critérios que visam indicar um conjunto mínimo de testes a serem realizados em *apps* antes de sua submissão às lojas de *apps*. No entanto, a variedade de características dos dispositivos em uma mesma plataforma, tais como tamanho de tela, disponibilidade de sensores, formas de acesso à rede ou capacidade de processamento, faz com que nem todos os testes previstos nos critérios publicados para uma plataforma sejam instanciáveis a um determinado dispositivo. Assim, um apoio à análise sobre

quais critérios de teste são aplicáveis a cada plataforma/dispositivo contribuiria para o planejamento dos testes e resultaria em mais recurso/tempo para que se foque na parte técnica, que seria o projeto e execução dos testes.

Neste contexto, este artigo apresenta a ferramenta *MTControol* (*Mobile Testing Control Tool*), que visa apoiar na recomendação e controle da execução de casos de teste a partir de critérios de teste publicados para aplicações móveis. A recomendação baseia-se na indicação dos casos de testes mais adequados, por meio de um questionário que visa o mapeamento das possíveis características de uma *app*, específicas de cada plataforma. O controle é realizado por meio do registro dos casos de testes executados, possibilitando um acompanhamento constante para avaliação e comparação de versões de *apps*.

2. Ferramentas de Apoio à Seleção e Controle de Testes em *Apps*

Em um primeiro momento, ferramentas de apoio ao gerenciamento de testes em software podem ser aplicadas para a gestão dos testes em *apps*. Neste sentido, as ferramentas *TestLink* e *Microsoft Visual Studio Test Professional* se apresentariam como alternativas populares na indústria.

TestLink (2008) é uma ferramenta de código aberto para gestão de criação de planos e casos de teste. Ela possibilita o controle dos testes planejados em relação aos testes executados e permite associar um conjunto de casos de teste a um testador e acompanhar os resultados da execução dos testes. A ferramenta ainda oferece um recurso que possibilita registrar e organizar os requisitos do projeto, assim como, associar os casos de teste aos requisitos. No entanto, esta ferramenta não oferece recursos para a seleção de testes para qualquer plataforma, no caso deste trabalho, o foco é em testes de *apps*. Assim, um testador teria um grande esforço para a criação e registro manual e individual dos testes para *apps*, caso deseje seguir as recomendações publicadas nos critérios de teste publicados para as plataformas móveis.

M.V.S. Test Professional (2013) é uma solução proprietária e permite a criação de planos, conjuntos e casos de teste, além da execução de testes manuais. Ela pode ter suas funcionalidades ampliadas através da integração com outras plataformas compatíveis. Integrada com MSDN, permite a realização, armazenamento e repetição de testes manuais, gerenciamento de casos de teste, gerenciamento de ambientes. Suas limitações se assemelham às citadas para *TestLink*. A ferramenta não possui recursos para sua integração com critérios de teste publicados para plataformas móveis, dificultando o planejamento e controle dos testes em *apps*.

Como alternativa de apoio à seleção e controle de testes em *apps*, a *App Quality Alliance* (AQuA), uma organização sem fins lucrativos que visa apoiar a indústria por meio do fornecimento de critérios de testes com foco na melhoria e qualidade de aplicações móveis, publica critérios de teste para as plataformas móveis Android e iOS. Estes critérios descrevem requisitos necessários para os casos de testes em diferentes categorias de *apps*, incluindo testes que complementam o teste funcional realizado pelo desenvolvedor a construção de uma *app* outros tipos de testes não-funcionais, como usabilidade, instalação, desempenho e testes de compatibilidade (AQuA, 2013). A partir desses critérios de testes disponibilizados, o desafio é analisar e escolher quais testes são adequados para uma aplicação específica, plataforma ou dispositivo, visto a imensidão de características dos três fatores citados acima. Neste sentido, a AQuA possui uma ferramenta online (*Online Testing Criteria Tool*) que apoia testadores na aplicação de critérios de testes em suas aplicações, bem como a documentação dos resultados dos

critérios de testes, armazenando os resultados de aprovação/reprovação por meio de relatórios. No entanto, algumas limitações são destacadas, tais como:

- A recomendação de todos os casos de teste para qualquer *app*, independente de possuir adequação com as funcionalidades da aplicação;
- Apenas uma rodada de teste pode ser instanciada por *app*, o que dificulta o gerenciamento dos testes quando a *app* já testada sofre uma evolução;
- Ela não possui suporte para *apps* multiplataforma, obrigando ao testador a criação de uma nova *app* para a execução dos testes em cada a plataforma, o que dificulta o controle de qualidade de uma *app*;
- Por fim, ela possui limitações em relação às plataformas apoiadas, sendo limitada a Android e iOS, desconsiderando a evolução de aplicativos em plataformas diferentes das consideradas tradicionais (ex: Windows Phone ou HTML5).

Assim, com o objetivo de prover uma evolução da ferramenta online proposta pela AQuA em relação às limitações destacadas acima, este artigo apresenta uma ferramenta alternativa de apoio à recomendação e controle de casos de testes para *apps*, chamada *MTControtrool* (*Mobile Testing CONTRol tOOL*), descrita na próxima seção.

3. Descrição da Ferramenta *MTControtrool*

A ferramenta *MTControtrool* visa prover um apoio mais abrangente para a gestão dos testes realizados em *apps*. Seus objetivos principais são a recomendação de testes mais adequados às características, funcionalidades e plataformas de uma *app* durante a fase de planejamento dos testes, e o controle e registro dos casos de teste durante a fase de execução. Ela foi construída na linguagem PHP com o framework Yii, seguindo a arquitetura MVC (*Model-View-Control*) e utiliza o SGBD MySQL. A ferramenta está disponível em http://sistemas.icomp.ufam.edu.br/mtcontrool (login e senha *guest*).

MTControtrool é destinada a desenvolvedores e testadores de *apps* de grandes empresas ou individuais. Ela possui dois perfis de usuários: administrador (mantenedor do sistema e possui acesso irrestrito às suas funcionalidades) ou testador (é quem registra *apps* e instancia novas rodadas de testes para serem controladores pela ferramenta).

Nas próximas seções será apresentado como a ferramenta está estruturada para atender ao seu objetivo proposto.

3.1. Visão Arquitetural

A Figura 1 apresenta uma visão arquitetural de como a ferramenta é estruturada em relação ao seu objetivo, que é prover um conjunto de testes adequados em relação às características de uma *app*. O elemento principal para gestão das informações em *MTControtrool* são os *Critérios de Teste*, que representam categorias/requisitos de teste a serem realizados em *apps* para uma ou mais *Plataformas*. Estes critérios são compostos por *Casos de Teste*, que indicam passos a serem seguidos durante os testes e que estão relacionados a um conjunto de *Características* de uma *App*, tais características são provenientes dos critérios e casos de testes cadastrados.

Atualmente, a ferramenta contém 52 características (35 para Android, 34 para iOS e 8 para Windows Phone), algumas sendo comuns a diferentes plataformas. Além disso, ela possui 26 critérios de teste (137 casos de teste) para Android, 22 critérios (89 casos de teste) para iOS e 9 critérios (38 casos de teste) para Windows Phone, estes

últimos coletados da loja da Microsoft e adaptados ao modelo padrão de casos de teste da AQuA.

Assim, *Rodadas de Teste* podem ser instanciadas para uma determinada *app*. Em cada rodada os critérios serão instanciados e seus casos de testes podem ser controlados à medida que são executados no projeto. Uma *app* pode ser compartilhada entre vários *Usuários* da ferramenta, permitindo que o monitoramento dos testes possa ser realizado por uma equipe.

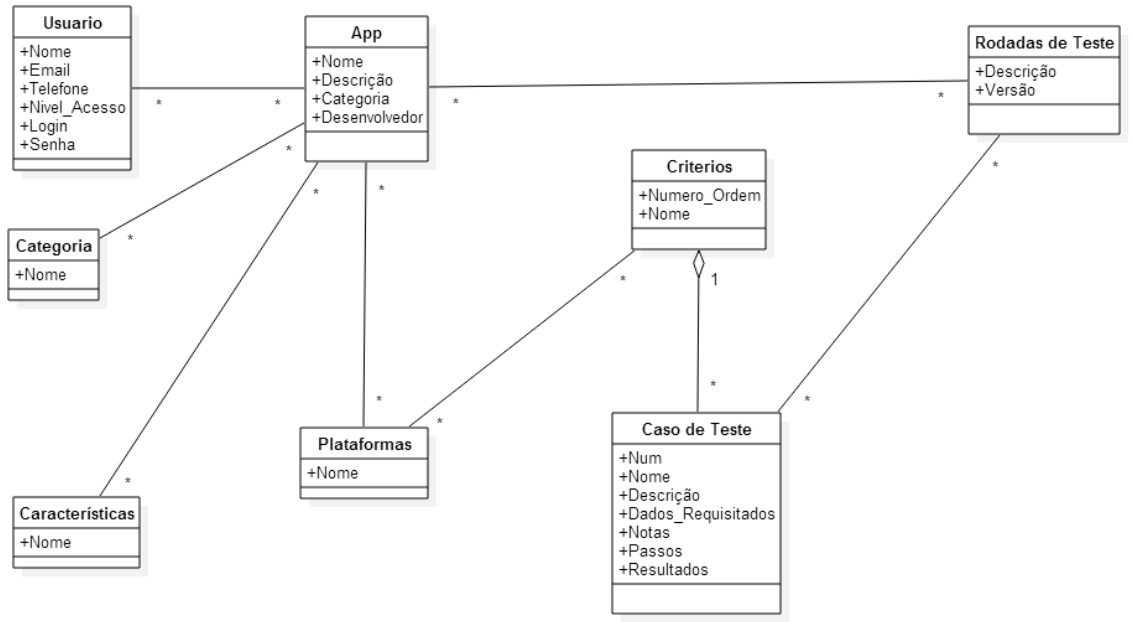


Figura 1. Diagrama de classes da ferramenta *MTControol (* Visão arquitetural *).*

As funcionalidades providas pela ferramenta *MTControol* serão descritas nas próximas seções.

3.2. Funcionalidades de *MTControol*

As funcionalidades que compõem a ferramenta *MTControol* podem ser divididas em duas categorias: (1) configuração da ferramenta e (2) apoio à realização de testes em uma *app*.

As funcionalidades que visam configurar a ferramenta são disponibilizadas a usuários com perfil de administrador e possivelmente serão usadas poucas vezes, apenas para manutenção dos dados manipulados pela ferramenta. São elas:

- **Gerenciar Usuários:** CRUD[1] de usuários que possuem acesso à ferramenta com diferentes perfis (administradores [acesso irrestrito à ferramenta] ou testadores [acesso restrito à gestão de suas *apps*]);
- **Gerenciar Plataformas Móveis:** CRUD de plataformas móveis utilizadas pelos usuários da ferramenta;
- **Gerenciar Critérios de Teste:** CRUD de critérios de teste (por plataforma) de *apps* publicados (ou não) pela AQuA. Os critérios de teste são compostos por um conjunto de casos de teste adequados a cada cenário/característica de uma *app*;

[1] Acrônimo em Inglês referente às operações de *Create-Retrieval-Update-Delete*.

- **Gerenciamento de Características:** CRUD de possíveis características de *apps*. Tais características são associadas ao critério que se deseja mapear, e são utilizadas para a montagem do questionário.
- **Gerenciamento de Casos de Teste:** CRUD que permite gerenciar casos de teste referentes aos critérios disponíveis da ferramenta. Na ferramenta, casos de teste compõem os critérios da plataforma.
- **Gerenciar Categorias de *Apps*:** CRUD que permite o gerenciamento de categorias para as *apps* da ferramenta.

As funcionalidades que visam apoiar à realização de testes em uma *app* são disponibilizadas a usuários com perfil de testador e serão usadas frequentemente visando apoiar o planejamento e controle dos testes para uma (ou várias) *app*(s). São elas:

- **Cadastrar *Apps*:** como entrada de dados, a ferramenta cadastra o aplicativo a ser testado, bem como todas suas propriedades (idiomas, categorias, plataformas, etc.).
- **Compartilhar *App*:** usuários podem compartilhar *apps* entre si, permitindo o gerenciamento e criação de rodadas de testes por ambos.
- **Criar Rodada de Teste:** criação de uma nova instância de teste para uma *app* já gerenciado pela ferramenta. Isso pode ser feito a cada nova versão ou plataforma do aplicativo. Para isso, a ferramenta analisa as características e plataformas da *app* para selecionar quais casos de teste dos critérios cadastrados na ferramenta se adequam à *app;*
- **Controlar Testes:** uma vez instanciada uma nova rodada de teste, *MTControol* oferece uma funcionalidade de registro e controle da execução dos testes no aplicativo, de forma a visualizar o status atual dos testes (em percentual) em geral, testes que foram aceitos e testes que falharam, disponibilizando sempre um relatório do resultado dos testes realizados na *app;*
- **Visualizar Histórico:** o usuário pode visualizar a qualquer momento os testes em andamento ou já realizados em um *app* por meio do histórico das rodadas. A ferramenta permite a comparação de resultados em diferentes rodadas de testes;

4. Funcionamento da *MTControol*

O principal diferencial da ferramenta é o fato de indicar testes a partir de características da *app* mapeadas nos critérios da(s) plataforma(s) na(s) qual(is) foi construída. Tais características são provenientes dos critérios e casos de testes cadastrados. Isso permite ao usuário da ferramenta gerenciar testes para todas as plataformas da *app*. Ao longo desta seção serão descritas as principais funcionalidades de *MTControol*.

4.1. Cadastro de uma *app*

Para iniciar o uso da ferramenta, um usuário com perfil de testador precisa inicialmente cadastrar uma *app* que terá seus testes gerenciados (Figura 2). Nesse momento, o usuário informa as principais propriedades da aplicação (ex: descrição, idiomas, categorias, plataformas). Uma *app* pode ser associada a diversas plataformas, não necessitando a criação de várias instâncias de uma mesma *app* para várias plataformas.

4.2. Criação de uma rodada de teste

Uma vez que a *app* a ser testada já está cadastrada, rodadas de teste podem ser

instanciadas. Ao optar pela criação de uma nova rodada de teste, a ferramenta disponibiliza um questionário (Figura 3) com o objetivo de coletar as características da aplicação. Tal questionário é apresentado com as perguntas agrupadas por critérios da plataforma. Cabe ao usuário marcar as respostas, baseando-se no que a *app* possui como característica (como exemplificado na seção 3.1). Estas características foram extraídas inicialmente dos critérios de teste publicados pela aliança AQuA para classificação de *apps* e servirão como entrada para o processo de recomendação de casos de teste.

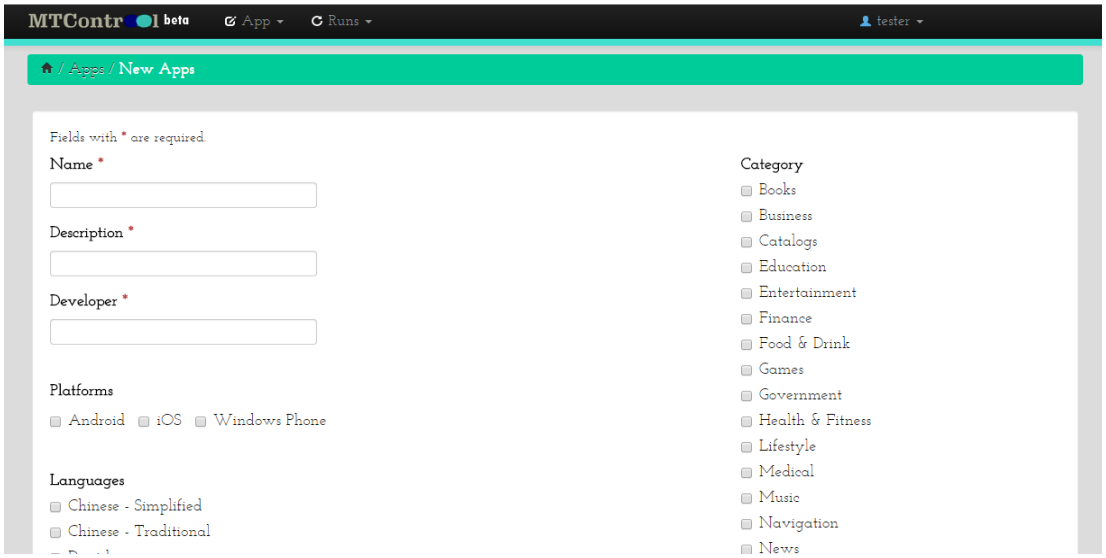


Figura 2. Tela de cadastro de uma *app*.

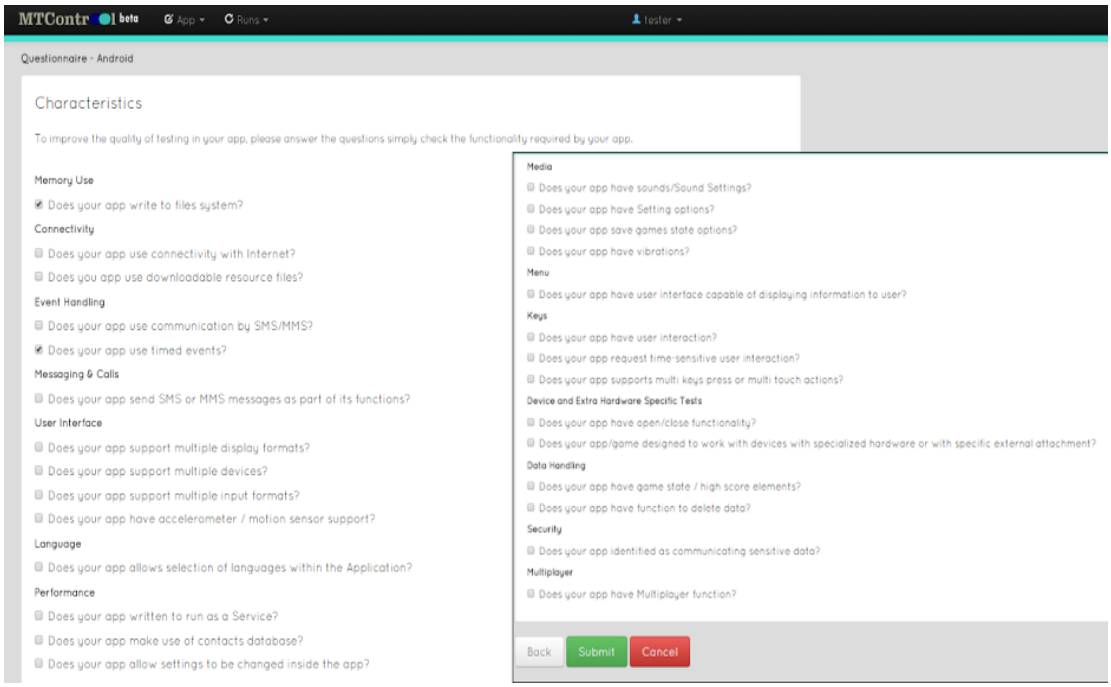


Figura 3. Questionário de coleta de características da aplicação.

Um ponto importante é que a ferramenta possibilita a criação de novas rodadas para uma mesma aplicação. Com isso, a ferramenta pretende apoiar possíveis novas *releases* da *app*, fazendo com que o usuário possua um controle do que já foi testado.

4.3. Controle dos testes de uma *app*

Após o preenchimento do questionário e instanciação de uma nova rodada de teste, o testador pode gerenciar a execução dos testes. Esta tela, chamada de *Dashboard*, apresenta os casos de teste recomendados para a *app*, dispostos em uma lista, acompanhada de alguns indicadores gerais na parte superior da tela (percentual de testes concluídos [área azul], total de testes que falharam [área vermelha], plataforma a ser tesada na rodada [área verde], versão da *app* sob teste [área laranja]), como apresentado na Figura 4.

À medida em que os casos de teste vão sendo executados, seus respectivos resultados podem ser registrados como Aprovado (*passed*) ou Reprovado (*failed*). Neste momento, a ferramenta atualiza automaticamente a porcentagem de testes concluídos, o total de testes que falharam/foram aceitos, assim como os testes pendentes.

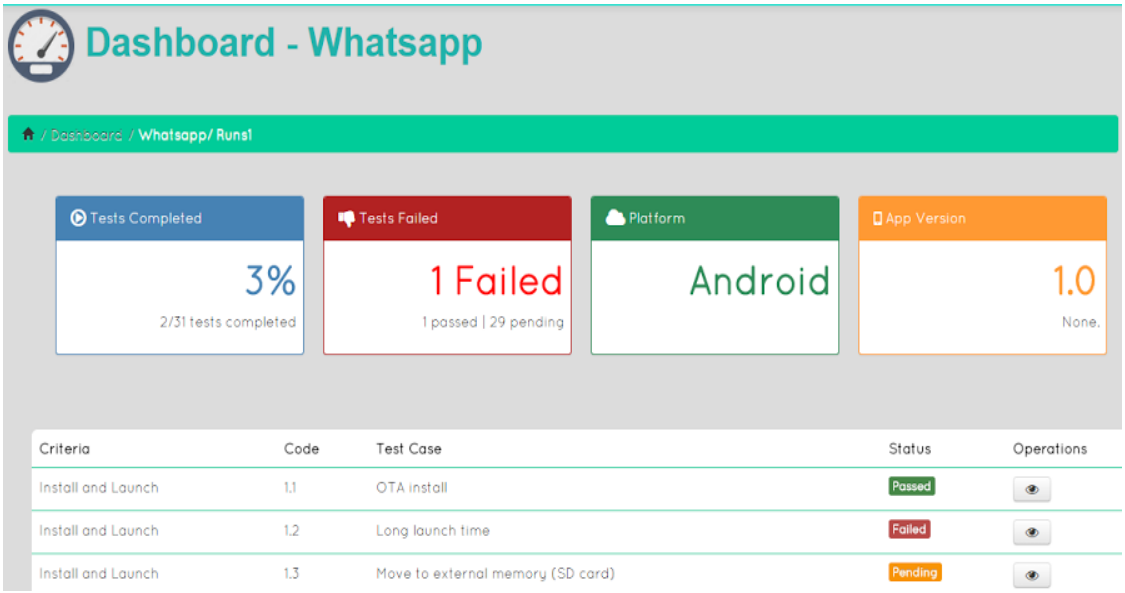


Figura 4.Tela de Controle de Testes.

O controle dos testes pode ser acessado para todas as rodadas de uma determinada *app* por meio da tela de gerenciamento de *apps*. Desta forma, a equipe de testes pode ter informação sobre o estado atual das rodadas de teste em execução para uma app, além dos resultados dos testes anteriores (Figura 5).

5. Conclusões e Trabalhos Futuros

Neste artigo foi apresentada uma ferramenta que permite a recomendação e o controle de casos de teste para *apps* a partir de critérios de teste publicados para diferentes plataformas. Com isso, espera-se contribuir com a qualidade das mesmas, fazendo com que atendam os critérios de qualidade das plataformas exigidos pelos fabricantes.

Espera-se que a recomendação baseada na indicação dos casos de testes mais adequados, por meio das possíveis características de uma *app*, auxilie testadores, visto que eles não teriam a obrigação de conhecer todos os critérios de teste para uma plataforma, podendo focar no desenvolvimento da própria *app*.

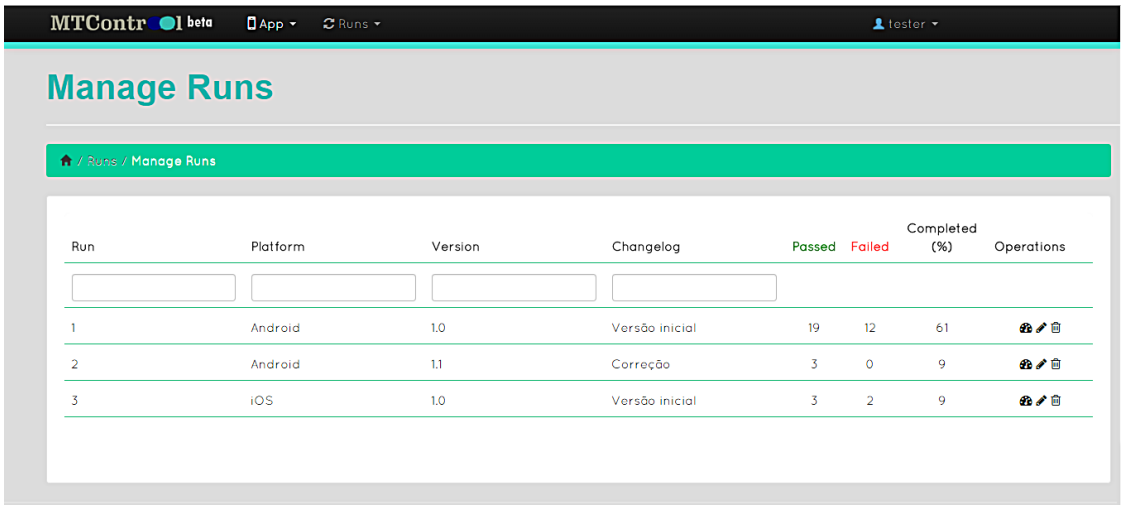


Figura 5.Tela de Gerenciamento de Rodadas de Testes.

Como trabalhos futuros, pretende-se integrar a proposta de Villanes *et al*. (2015), que visa automatizar a execução dos critérios de teste para aplicações móveis por meio de um serviço de testes baseados na instanciação de emuladores a ser disponibilizado na nuvem. Com isso, a *MTControol* permitiria não apenas o gerenciamento de testes baseados em critérios, mas também a sua execução.

E a realização de um experimento controlado com projetos reais na academia e indústria, tendo em vista que a MTControol foi desenvolvida no contexto de projeto de cooperação com uma grande empresa de plataforma móvel, assim está prevista sua implantação em projetos de aplicações desenvolvidos por esse parceiro, para avaliar a viabilidade e uso da ferramenta.

Agradecimentos

Os autores agradecem o apoio concedido pela FAPEAM, através dos projetos número 01135/2011 (PRONEX), 582/2014 (PRO-TI-PESQUISA), e ao INDT.

6. Referências

Harrison, R., Flood, D., Duce, D., (2013) “Usability of mobile applications: literature review and rationale for a new usability model” In: Journal of Interaction Science (1), pp. 1-16.

Dantas, V. L. L. (2009) “Requisitos para Testes de Aplicações Móveis”. Dissertação (Mestrado em Ciência da Computação) - Departamento de Computação, Universidade Federal do Ceará, Fortaleza.

TestLink. TestLink developers Community, v. 1.8 (2008). Disponível em: http://www.teamst.org/. Acesso em: 26/05/2015.

Microsoft Visual Studio Test Professional (2013). Disponível em: https://www.microsoft.com/en-us/download/details.aspx?id=40786. Acesso em: 26/05/2015.

AQuA. (2013) “Essentials of Mobile App Testing” - Disponível em: http://www.appqualityalliance.org/AQuA-essentials-of-mobile-app-testing Acesso em: 26/05/2015.

Villanes, I., Costa, E., Dias-Neto, A. (2015) “Automated Mobile Testing as a Service (AM-TaaS)”. In: IEEE 11th World Congress on Services (SERVICES 2015).

Asymptus - A Tool for Automatic Inference of Loop Complexity

Junio Cezar, Francisco Demontiê, Mariza Bigonha and Fernando Pereira

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

{juniocezar,demontie,mariza,fernando}@dcc.ufmg.br

Abstract. *Complexity analysis is an important activity for software engineers. Such an analysis can be specially useful in the identification of performance bugs. Although the research community has made significant progress in this field, existing techniques still show limitations. Purely static methods may be imprecise due to their inability to capture the dynamic behavior of programs. On the other hand, dynamic approaches usually need user intervention and/or are not effective to relate complexity bounds with the symbols in the program code. In this paper, we present a tool which uses a hybrid technique to solve these shortcomings. Statically, our tool determines: (i) the inputs of a loop, i.e., the variables that control its iterations; and (ii) an algebraic equation relating the loops within a function. We then instrument the program to output pairs relating input values and number of operations executed. By running the program over different inputs, we generate sufficient points for a polynomial interpolator in order to precisely determine a complexity function for loops. In the end, the complexity function for each loop is combined using an algebra of our own craft. We have implemented this tool using the LLVM compilation infrastructure. We correctly analyzed 99.7% of all loops available in the Polybench benchmark suite. These results indicate that our technique is an effective and useful way to find the complexity of loops in high-performance applications.*

Demo URL: https://youtu.be/pzfrIDfoCEc

1. Introduction

Complexity analyses show how algorithms scale as a function of their inputs. The importance of complexity analysis stems from the fact that such a technique helps program developers to uncover performance bugs which would otherwise be hard to find. In addition to this, complexity analysis supports the decision of offloading or not computation to the cloud or GPU. Finally, this kind of technique has implications to the theoretical computer science community, as it provides data that corroborate the formal asymptotic analysis of algorithms. Given this importance, it comes as no surprise that, since the 70s [Wegbreit 1975], large amounts of effort have been spent in the design and improvement of empirical methodologies to infer the complexity of code.

Over the time, different static approaches were proposed to analyze programs in functional [Wegbreit 1975, Le Métayer 1988, Rosendahl 1989, Debray and Lin 1993] and imperative [Gulavani and Gulwani 2008, Gulwani et al. 2009b, Gulwani et al. 2009a] languages. Although the static approaches have the benefit of running fast and may give correct upper bounds, this methodology has shortcomings.

Static analyses may yield imprecise – or even incorrect – results. This imprecision happens due to the inherently inability of purely static approaches to capture the dynamic behavior of programs. In order to circumvent this limitation of static approaches, the programming language community has resorted to profiling-based methodologies [Goldsmith et al. 2007, Zaparanuks and Hauswirth 2012, Coppa et al. 2012]. However, even these dynamic techniques are not free of limitations.

The main drawback of a profiling-based complexity analysis is the fact that it is usually ineffective to relate the symbols in the program text to the result that it delivers. For instance, the state-of-the-art tool in this field is `aprof` [Coppa et al. 2012]. Aprof furnishes programmers with a table that relates input sizes with the number of operations performed. This modus operandi has two problems, in our opinion. First, the input is provided as a number of memory cells read during the execution of a function. This number may not be meaningful to the programmer, as we will clarify in Section 2. Second, it works at the granularity of functions. However, developers are often more interested in knowing the computational complexity of small regions within a function. Such regions can be, for instance, performance-intensive loops. This paper addresses these two limitations of input sensitive profiling.

The main contribution of our work is a tool named Asymptus, which uses a novel hybrid technique to perform complexity analysis on imperative programs. Our technique is hybrid because it combines static analysis with dynamic profiling. First, we use static analysis to determine loop inputs and to find algebraic relations between these loops. Then, we use a dynamic profiler, plus polynomial interpolation, to infer the complexity of each loop in a function. Our technique is capable of generating symbolic expressions that denote the complexity of each loop, instead of the whole function. Furthermore, we combine and simplify these expressions to make them even more meaningful to the software engineer. We believe that this granularity can help developers to have a deeper understanding of a function's behaviour; hence, it provides them with the means to detect and solve performance bugs more efficiently. We also show that our technique is simpler than previous work while producing more useful results.

We have designed, tested, and implemented a tool on top of the LLVM compilation infrastructure [Lattner and Adve 2004] to infer, automatically, the complexity of loops within programs. We ran our tool over the Polybench [Pouchet 2012] and Rodinia [Che et al. 2009] benchmark suites. Our results indicate that we are capable of correctly inferring the complexity of 99.7% of the Polybench loops and 69.18% of the Rodinia loops. All the equations that we output, as explained in detail in Section 2, are written as functions of the symbols, i.e., variable names, present in the program code – that is an improvement on top of `aprof` and similar tools. Moreover, we have found that 38% of all functions in the benchmarks that we analyzed have at least two independent loops. In this case, tools that only report complexity information for entire functions may miss important details about the asymptotic behaviour of smaller regions of code.

2. Overview

In this section we give an overview of the challenges Asymptus addresses. Figure 1 shows the example we will use to illustrate our technique. Function *multiply* is a routine that performs matrix multiplication of two square matrices. For pedagogical purposes,

```
1: void multiply(int **matA, int **matB, int n){
2:     int i, j, k, sum;
3:     int **result = (int**) malloc(n * sizeof(int*));
4:     for (i = 0; i < n; i++)
5:       result[i] = (int*) malloc(n * sizeof(int));
6:
7:     for (i=0; i < n; i++) {
8:       for (j=0; j < n; j++) {
9:         sum = 0;
10:        for (k=0; k < n; k++) {
11:          sum += matA[i][k] * matB[k][j];
12:        }
13:        result[i][j] = sum;
14:      }
15:    }
16:
17:    j = 0;
18:    for (i = 0; i < n;) {
19:      if (j >= n) {
20:        j = 0;
21:        i++;
22:        printf("\n");
23:      } else {
24:        printf("%8d", result[i][j++]);
25:      }
26:    }
27:    printf("\n");
28: }
```

Figure 1. Matrix multiplication – the running example that we shall use to explain our contributions.

our function does not return the resulting matrix; instead, it prints the result. We chose to implement the function in such a way to show how our technique behaves on functions with multiple loops.

As developers, we would like to know the computational cost to execute this function. For instance, knowing the complexity of each part of the target function, we can find out performance bottlenecks and improve its implementation. Looking at the *multiply* function we can easily identify the linear behavior of the loop on line 4 and the cubic behavior of the nested loops beginning at line 7. However, a quick visual inspection on the loop at line 18 may not capture its quadratic complexity.

We can use profilers to find out where the program is spending most of its resources. However, traditional tools lack the ability to show how the program scales as a function of its inputs. For instance, Figure 2 shows the output that Gprof [Graham et al. 1982] – the most well-known profiler in the Unix systems – produces for our example. This profiler does not give us any information regarding the asymptotic complexity of the program in Figure 1. Instead, it produces a table describing

```
index % time    self  children   name
[1]    100.0    0.00    0.03     main [1]
                0.03    0.00     multiply(int**, int**, int) [2]
                0.00    0.00     initArray(int**, int, int) [9]
                0.00    0.00     free_all(int**, int**, int) [10]
-----------------------------------------------
```

Input size	Average Cost
138	1111
174	3324
286	12007
367	18576
463	26694
575	36394
701	47749
846	60781
1007	75587

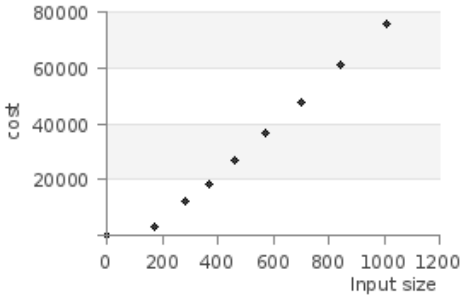


Figure 3. The output produced by the aprof input sensitive profiler.

where the program spends more time during its execution.

There exist profilers that have been designed specifically to provide developers with an idea about the asymptotic complexity of programs [Goldsmith et al. 2007, Zaparanuks and Hauswirth 2012, Coppa et al. 2012]. Nevertheless, `aprof` [Coppa et al. 2012], the state-of-the-art approach in this field, is also not very useful in this example. For instance, only looking at Figure 3, which shows `aprof`'s results for the function *multiply*, the user may not fully understand the function behaviour: this table shows numbers, but do not relate these numbers with symbols in the program text. Moreover, the complexity curve seems to be linear, since `aprof` considers the whole matrices as inputs (n^2) – usually, developers describe asymptotic complexity in terms of the matrices dimentions (n). Finally, the result generated by `aprof` describes the whole function. We believe that this granularity is too coarse, because it makes it very difficult for the user to verify the behavior of particular parts of the function.

We can do better: the technique that we describe in this paper produces one polynomial for each loop in the function. These polynomials range on symbols defined in the program text, e.g., the names of variables. Therefore, we claim that our output is clearer to the developer. For instance, considering the loop in line 7, we will state – automatically – that its complexity is:

$$n + 1$$

Furthermore, considering the loop nest starting in line 18, we produce the following equa-

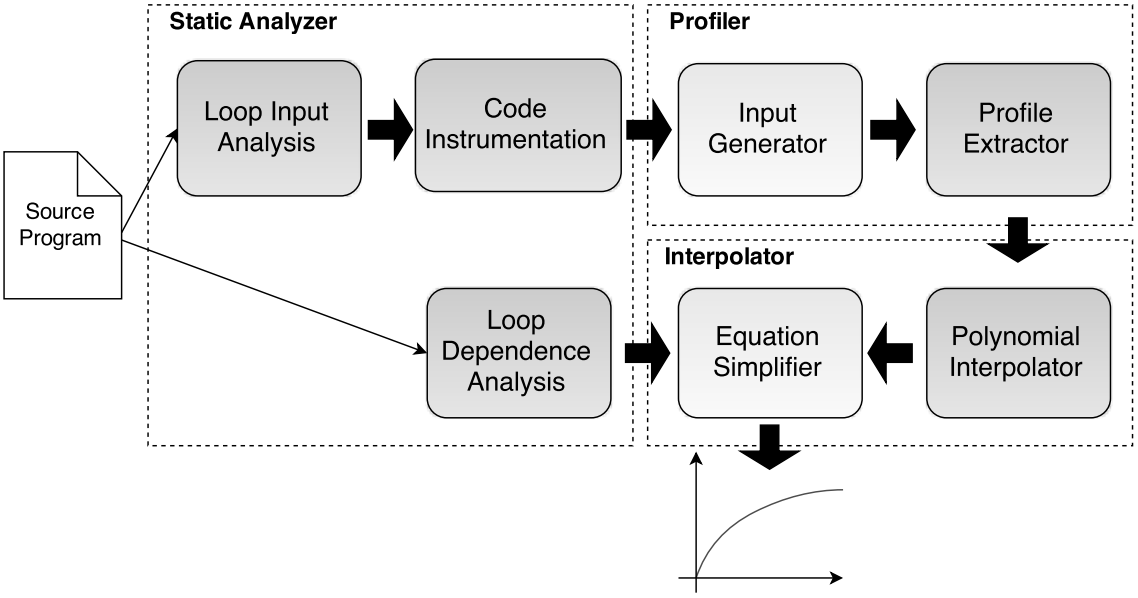


Figure 4. The Asymptus architecture. The arrows represent data flow. This diagram shows how the components of the tool work together.

tion to denote its complexity:

$$n^2 + n + 1$$

Our result is on a finer granularity, so we can combine them to generate an equation that expresses the asymptotic behavior of the whole target function. For the function in the Listing 1, our approach generates the following simplified equation, in big O:

$$O(n^3)$$

We claim that this notation, which uses the names of variables present in the program, is more meaningful to the application developer than the output produced by traditional profilers, such as *gprof* or `aprof`.

3. The Asymptus Tool

Asymptus, the tool that we describe in this paper, consists of four main steps: (1) static analysis, (2) code instrumentation, (3) dynamic information extraction and (4) polynomial interpolation. In this section we describe each one of these steps. Figure 4 shows a simple architecture of Asymptus.

3.1. Input Analysis

We start the process of inferring the complexity of code with a static analysis phase. The static analysis determines the inputs of each loop in the function. We qualify as *loop input* any data that: (i) influences the stop condition of the loop; and, (ii) is not defined within the loop. For instance, the loop at line 7 in Figure 1 is controlled by $i < n$. Variable i has two definitions: one outside the loop, which we shall call i_0, and another inside, which we shall call i_1. The former is initialized with the constant zero, which is thus considered a loop input. Variable n is a parameter of the function; hence, it is considered a symbolic

input. Therefore, the two inputs of the loop that exists at line 7 are $\{0, n\}$. Concretely, we detect inputs through a *backward* analysis, that starts at the variables used in the loop's stop condition, and ends at the definitions of variables that lay outside the loop body.

3.2. Loop Dependence Analysis

Our profiler outputs the complexity of all the loops within a program. We must combine this information to have a snapshot of the program's complexity. For doing this, we statically analyze control dependency relations between the loops within a function. We use these relations to determine an equation based on the big-O notation. For example, if two loops are in the same level and they are control equivalent, we can sum their complexities. As if we have a loop nest, the complexities can probably be multiplied. After we fill this equation with the loop complexities, the equation is simplified using an operators algebra based on the big-O semantics. For instance, for the function in Figure 1 our tool generates the following equation: $C(multiply) = C(L_{4-5}) + C(L_{7-15}) \times C(L_{8-14}) \times C(L_{10-12}) + C(L_{18-26})$. After the profiling phase, this equation is replaced by $O(n + n * n * n + n^2)$. It is easy to see that we can simplify this equation and have $O(n^3)$ as the resulting complexity.

3.3. Code Instrumentation

We infer the complexity of code by analyzing profiling data. We produce this data through code instrumentation. To be able to extract dynamic information, we instrument the target program to output: (i) the values of the loop inputs immediately before the loop execution and (ii) the number of operations performed by each loop. Loop inputs are determined by the analysis seen in Section 3.1. The execution cost is measured in terms of instructions executed. We have implemented this instrumentation framework within the LLVM compiler infrastructure. Care must be taken with regard to loops with multiple paths. Different paths may yield different costs, a fact that could hinder our interpolator from finding a perfect polynomial fit. To avoid this problem, we consider that the cost of a loop is determined by its path of highest cost, which we estimate statically. To obtain a conservative estimate of this path, we resort to a modified version of Dijkstra's algorithm, to solve the single-source largest path problem for an acyclic graph with non-negative weights assigned to edges [Dijkstra 1959]. Once we have instrumented the program, we execute it. As mentioned before, each execution of an instrumented program outputs the values of each loop input, together with the number of operations executed within that loop.

3.4. Polynomial Interpolation

We log the output of our profiler and parse it to extract pairs: input value $\times$ execution cost. With these points, we execute a polynomial interpolation method to find the curve that best fits into this set. For example, the program seen in Figure 1 has two blocks of loops; thus, we produce two polynomials. Let us take a deeper look into the polynomial that we produce for the loop that exists at lines 18-25 of Figure 1. In this example, we have obtained, after profiling the program with eight different inputs, the following pairs of size $\times$ cost: (13, 183), (50, 2,551), (72, 5,257), (80, 6,481), (98, 9,704), (115, 13,341), (139, 19,461). Our interpolator produces the polynomial $n^2 + n + 0.8$ The complexity of the loop is then $O(n^2)$, where n is the only symbolic input of the loop under analysis, as

we have explained in Section 3.1. We perform similar process to discover the polynomial that characterizes the loop nest at lines 7-15 of Figure 1.

3.5. Usage

Since Asymptus was implemented using the LLVM compilation infrastructure, it runs over, either C/C++ source files, or LLVM bytecodes. We give the user the possibility to specify the inputs for the target program in three ways: (i) providing real data as inputs for the program, (ii) letting Asymptus generate random values or (iii) a mix of real data and random values. To let Asymptus to generate random values, the user have to use the option `--args` specifying the type of each command line argument of the program. The types may be one of the following: int, long, float, double, num (a numeric value without specifying the specific type), char or string. To mix concrete and random values, there is the option `--mix`. In this case, the types for the random values have to be specified within '{}'. For example, `--mix concrete1 {int} concrete2 {string}`. To execute the program with only concrete data, the user has to use the option `--man`. When using this option, Asymptus asks the user for the input values for each execution. An empty line marks the end of the inputs. By default, Asymptus shows results in the function level. To show the results for each loop within the program, the user has to specify the option `-v`. Figure 5 shows an output example. The function *multiply* is the one of Figure 1. Function *init* creates an array with dimensions of size passed as argument.

```
Function 'multiply(int**, int**, int)':
    Loop at line 6: 1.00 * n + 1.00
    Loop at line 9: 1.00 * n + 1.00
    Loop at line 10: 1.00 * n + 1.00
    Loop at line 12: 1.00 * n + 1.00
    Loop at line 20: 1.00 * n^2 + 1.00 * n + 1.00

    Complexity: O(n^3)

Function 'init(int)':
    Loop at line 38: 1.00 * size + 1.00
    Loop at line 42: 1.00 * size + 1.00
    Loop at line 43: 1.00 * size + 1.00

    Complexity: O(size^2)
```

Figure 5. Asymptus output example using verbose mode and random inputs.

4. Conclusion

This paper presented Asymptus, a publicly available tool which uses a new technique based on a combination of profiling and static analysis, to infer the complexity of code. Static analysis gives us the names of variables that bound the trip count of loops. Profiling lets us associate these variables with the number of operations in the loops that they control. We believe that our approach, whenever applicable, yields results that are more meaningful to the application developer than the state-of-the-art tools that are currently available. Asymptus has three main limitations: (i) it can only analyze functions reached by the execution flow of the program, (ii) it only generates results for a function if it can find enough data-points (i.e. if the function is executed with different inputs) and (iii) only work for loops with polynomial complexities. There are several ways in which such a tool can be employed. Our immediate goal is to use it to help in the automatic placement of code in non-uniform memory access architectures. In this scenario, it is worthwhile to

migrate processes of high computational cost closer to the memory banks that contain the data that said processes use. A totally static solution has been devised to this problem by Piccoli *et al.* [Piccoli et al. 2014]. Our intention is to add to this solution a dynamic component based on this paper's ideas, in hopes to increase its precision.

References

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE.

Coppa, E., Demetrescu, C., and Finocchi, I. (2012). Input-sensitive profiling. In *PLDI*. ACM.

Debray, S. K. and Lin, N.-W. (1993). Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *FSE*, pages 395–404. ACM.

Graham, S. L., Kessler, P. B., and McKusick, M. K. (1982). gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pages 49–57.

Gulavani, B. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, volume 5123 of *LNCS*, pages 370–384. Springer.

Gulwani, S., Jain, S., and Koskinen, E. (2009a). Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385. ACM.

Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009b). SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Le Métayer, D. (1988). Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266.

Piccoli, G., Santos, H., Rodrigues, R., Pousa, C., Borin, E., and Pereira, F. M. Q. (2014). Compiler support for selective page migration in NUMA architectures. In *PACT*, pages 369–380. ACM.

Pouchet, L.-N. (2012). Polybench: The polyhedral benchmark suite. URL: http://www.cs.ucla.edu/ pouchet/software/polybench/. Last access: April, 2015.

Rosendahl, M. (1989). Automatic complexity analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 144–156, New York, NY, USA. ACM.

Wegbreit, B. (1975). Mechanical program analysis. *Commun. ACM*, 18(9):528–539.

Zaparanuks, D. and Hauswirth, M. (2012). Algorithmic profiling. In *PLDI*, pages 67–76. ACM.

FlowTracker - Detecção de Código Não Isócrono via Análise Estática de Fluxo.

Bruno R. Silva[1], **Leonardo Ribeiro**[2], **Diego Aranha**[3], **Fernando M. Q. Pereira**[1]

[1]Dep. de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

[2]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)

[3]Inst. de Computação - Universidade Estadual de Campinas (UNICAMP)

{brunors, fernando}@dcc.ufmg.br, dfaranha@ic.unicamp.br

leonardofribeiro@gmail.com

Resumo. *FlowTracker é uma ferramenta disponível para uso online, capaz de analisar programas escritos em linguagem C ou C++ e reportar traços de instruções vulneráveis à ataques por análise de variação de tempo. Esses ataques são comuns em implementações de primitivas criptográficas e são possíveis devido a presença de canais laterais nas respectivas implementações. Um canal lateral ocorre quando bits da chave criptográfica ou de outra informação sigilosa acabam por influenciar a execução de desvios condicionais, alterando o fluxo de controle da aplicação e consequentemente variando o seu tempo de execução. Eles também ocorrem quando esses mesmo bits são utilizados como índices de acesso à memória, o que também pode afetar o tempo de execução devido às falhas de cache. FlowTracker foi testado em duas famosas e reconhecidas bibliotecas criptográficas: OpenSSL e NaCl. Foi possível validar a ausência de canais laterais em NaCl, e detectar centenas de trechos vulneráveis em OpenSSL.*

Vídeo disponível em https://youtu.be/e2AEZlimNas

1. Introdução

Algoritmos criptográficos robustos e consistentes são essenciais para garantir a segurança de aplicações modernas. Entretanto, a correção de um algoritmo criptográfico não está somente em seu projeto abstrato, mas também em sua implementação concreta [Fan et al. 2010, Kocher 1996, Standaert 2010]. Uma implementação é dita insegura quando ela apresenta *canais laterais*. Canais laterais são falhas de implementação que permitem a um adversário conhecer informações sigilosas de um algoritmo criptográfico. A literatura especializada em segurança já apresentou vários exemplos de tais ataques. Em alguns desses exemplos, adversários necessitam de acesso físico ao sistema, para nele inserir falhas [Biham and Shamir 1997] ou recuperar dados residuais da memória [Halderman et al. 2009]. Há casos onde o simples acesso a algum componente elétrico do hardware criptográfico, tal como os pinos das portas de I/O, pode ser suficiente para a recuperação de dados sigilosos [Genkin et al. 2014]. Porém, ataques menos invasivos também têm despertado o interesse da comunidade científica. A literatura da área descreve situações em que atacantes são capazes de detectar diferenças ínfimas em aspectos mensuráveis do comportamento de programas, tais como seu tempo

de execução [Kocher 1996], seu consumo de energia [Kocher et al. 1999], seu campo magnético [Quisquater and Samyde 2001] ou ruídos que ele produz [Genkin et al. 2014]. De posse de tais informações, adversários podem usá-las para inferir informação sigilosa do algoritmo criptográfico.

No contexto dos ataques que exploram a variação de tempo, garantir que informações sigilosas não influenciem o fluxo de controle da implementação é uma forma natural de eliminar canais laterais. Isso pode ser feito manualmente pelo programador ao evitar que dados oriundos das variáveis secretas controlem o resultado de desvios condicionais. Obviamente essa é uma tarefa árdua, que exige atenção e profundo conhecimento da linguagem de programação utilizada. Entretanto, mesmo um algoritmo implementado com tais cuidados necessita ser verificado após a compilação, pois otimizações no nível de linguagem intermediária podem inserir canais laterais. Verificações nesse nível usualmente requerem análises complexas de código por profissionais experientes, que tenham pleno conhecimento das características específicas da arquitetura de computador utilizada. Apesar de existirem ferramentas que dão suporte a essa verificação manual [Almeida et al. 2013, Chen et al. 2014], não existe atualmente técnica automática incorporada ao compilador capaz de garantir a isocronia de programas. Em outras palavras, compiladores de uso geral não proveem garantias de que eles não estão introduzindo vazamento de informação por variação de tempo no código que eles geram.

Neste trabalho, apresenta-se uma solução capaz de detectar um código não isócrono. Chamamos **não isócrono** o código que executa em tempo variável de acordo com os bits da informação que se deseja proteger. De forma análoga, chamamos de **isócrono** um programa criptográfico em que os bits da informação sigilosa não influenciam seu tempo de execução. Pode-se afirmar que um código isócrono é ausente de canais laterais e portanto não passível de ataques por variação de tempo. Diferentemente de outras abordagens descritas na literatura, que buscam detectar e reparar a ocorrência de canais laterais em uma determinada linguagem de programação [Lux and Starostin 2011], a solução aqui proposta atua sobre a representação intermediária do compilador. Além disso, a técnica proposta neste artigo é capaz de lidar com programas não estruturados, isto é, aqueles que utilizam instruções do tipo `goto`. Para tanto, um grafo de dependências é construído de acordo com a relação de dependências de dados e de controle entre as variáveis do programa. Tal grafo permite a detecção de todos os caminhos que conectam informações sigilosas a predicados de instruções de desvio. Esses caminhos são então reportados ao desenvolvedor da aplicação para que este tenha o devido cuidado de modificar o programa a fim de eliminar esses segmentos de código não isócronos.

A fim de validar as ideias apresentadas neste artigo, um detector de código não isócrono foi implementado sobre o compilador LLVM [Lattner and Adve 2004]. Essa implementação, batizada de *FlowTracker*, consiste em duas partes: (i) um grafo de dependências; e (ii) um detector de isocronia. O grafo de dependências descreve as relações entre as variáveis de um programa. O detector de isocronia usa esse grafo para verificar se informação sigilosa pode influenciar o tempo de execução do programa. Em caso afirmativo, cada traço estático vulnerável é reportado via um conjunto de arquivos de saída no formato `.png` e `.txt`. Esses módulos foram testados em duas famosas e reconhecidas bibliotecas criptográficas: *OpenSSL* e *NaCl* [Bernstein et al. 2012]. Foi possível validar a isocronia de *NaCl*, e detectar centenas de trechos não-isócronos em *OpenSSL*. *FlowTrac-*

```
0: int compVar(char* pw, char* in) {
1:   int i = 0;
2:   for (; i < 7; i++) {
3:     if (pw[i] != in[i])
4:       return 0;
5:   }
6:   return 1;
7: }
```

Figura 1. Exemplo de código não isócrono.

ker está disponível para uso *online*[1], não necessitando a instalação de qualquer programa no computador do desenvolvedor. Entretanto, o código fonte da ferramenta está publicamente disponível para aqueles que desejarem instalação *offline*. O fato de todas essas detecções terem sido realizadas de forma totalmente automática indica que a técnica apresentada neste artigo é um mecanismo efetivo e útil para melhorar a qualidade de sistemas de criptografia.

2. Ataque por variação de tempo

Ataques por variação de tempo visam obter o conhecimento de informações sigilosas tais como uma chave criptográfica ou um número aleatório. Eles se baseiam na execução do programa com dados de entradas pré-determinados e medições do tempo de execução, permitindo ao adversário inferir com grande precisão alguns ou todos os bits da informação secreta. Tais ataques podem ocorrer quando esses dados sigilosos influenciam os predicados, isto é, as estruturas condicionais das instruções de desvio e portanto determinam quais partes do código serão executadas, afetando o tempo de execução do programa.

Para ilustrar esse problema, a Figura 1 a) apresenta uma função simples de comparação de *strings*. Essa função recebe uma senha pw que será comparada com uma entrada do usuário in. Visto que o usuário tem o total controle do que será definido como in, considera-se que essa variável pode estar contaminada com dados maliciosos a fim de ativar/desativar partes pré-determinadas do código fonte. Um adversário pode monitorar quanto tempo a função `compVar` leva para retornar. Um retorno antecipado indica que a comparação na linha 3 não obteve sucesso nos primeiros caracteres. Portanto, variando, em ordem lexicográfica, o conteúdo da *string* in, o adversário pode reduzir de exponencial para linear a complexidade de busca pelo conteúdo de pw.

Esse tipo de ataque é facilitado quando o usuário tem acesso ao código fonte da implementação, mas também é completamente viável mesmo sem a divulgação de detalhes da implementação, bastando o conhecimento prévio do algoritmo criptográfico utilizado, o qual é comumente de conhecimento público.

3. Detecção de código não isócrono

FlowTracker é capaz de detectar e reportar ao usuário o problema apontado na Figura 1. Para tanto, a primeira tarefa é a construção de um grafo de dependências que armazena toda informação relacionada aos fluxos explícitos e implícitos do programa, que estão definidos logo abaixo:

[1] http://cuda.dcc.ufmg.br/flowtracker

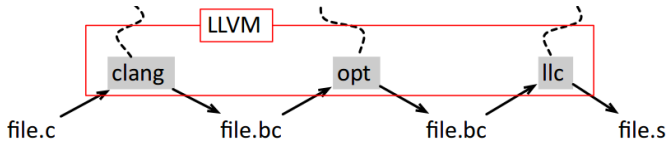


Figura 2. Exemplo de uso do compilador LLVM

1. Fluxos explícitos estão relacionados às dependências de dados. Se um programa contém uma instrução que define a variável v, e usa a variável u, tal como $v = u + 1$, então existe um fluxo explícito de informação de u para v.
2. Fluxos implícitos estão relacionados ao fluxo de controle do programa. Se o programa contém um desvio tal como $if\ p = 0\ then\ v = u + 1\ else\ v = u - 1$, então existe um fluxo implícito de informação de p para v, pois o valor atribuído ao último depende do primeiro.

FlowTracker detecta esses fluxos, cria um grafo de dependências do programa e procura nesse grafo caminhos entre informações sigilosas e instruções de desvio e índices de memória. No grafo de dependências, os vértices representam operandos, acessos à memória ou operações. As arestas são classificadas em arestas de dados, que representam um fluxo explícito entre dois vértices ou arestas de controle que representam fluxo implícito entre um predicado e um outro vértice qualquer.

Uma vez construído o grafo de dependências, *FlowTracker* inicia a busca por caminhos que conectam vértices que representam informações sigilosas à vértices sorvedouros, que representam predicados de instruções de desvio ou indexação de memória. Estas informações sigilosas devem ser informadas pelo usuário através de uma entrada para *FlowTracker* em formato XML que será descrita na próxima Seção.

Cada caminho encontrado é reportado ao usuário que deverá por sua vez identificar tal traço de instruções em seu programa e corrigir o problema, basicamente modificando sua implementação a fim de quebrar a cadeia de dependências entre o sorvedouro e a informação sigilosa. *FlowTracker* executa em tempo linear sobre o tamanho do programa. Vale mencionar que a escalabilidade de *FlowTracker* foi colocada à prova durante testes sobre a coleção de *benchmarks* SPEC CPU INT 2006 que contém programas como `GCC` com mais de 600 mil linhas de código. Nesse cenário, *FlowTracker* foi capaz de analisar todos esses programas em poucos segundos.

4. Exemplo de uso

FlowTracker é uma ferramenta implementada na forma de um módulo, também conhecido como *pass*[2], para o compilador industrial LLVM[Lattner and Adve 2004], bastante utilizado também em projetos de pesquisa envolvendo otimizações no nível do compilador. Basicamente LLVM corresponde à um *front end* e um conjunto de ferramentas a fim de gerar código compilado com possibilidade de realizar diversas otimizações.

Conforme mostra a Figura 2, utiliza-se o *front end clang* para transformar o programa C em *bytecodes* [3] LLVM - uma representação intermediária em formato *SSA - Static single assignment form* [Cytron et al. 1989] - gerando o arquivo *file.bc* que por sua vez

[2]Um *pass* é uma transformação ou análise que um compilador aplica sobre o programa.
[3]http://llvm.org/docs/LangRef.html

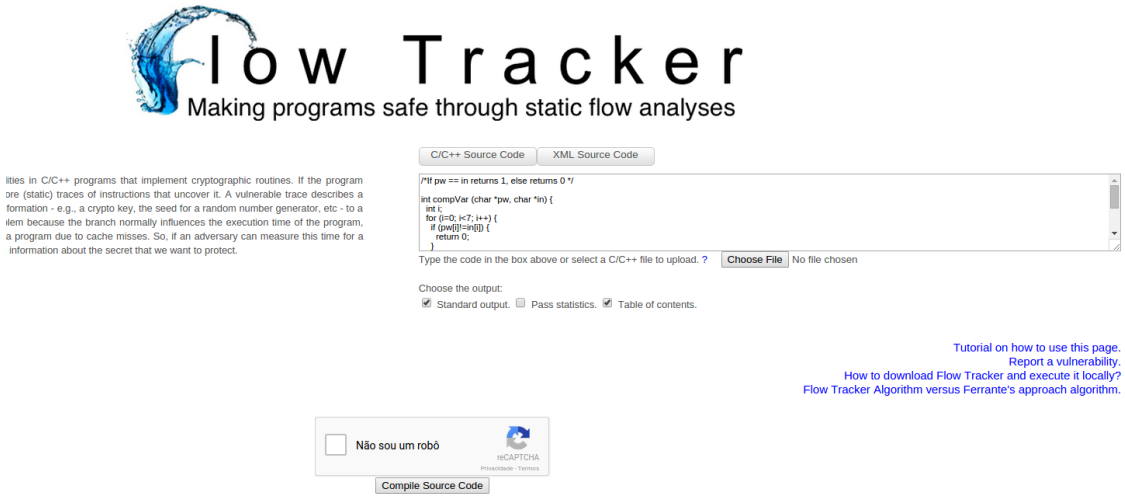


Figura 3. Interface Web de *FlowTracker*.

pode ser otimizado pelo programa *opt* também pertencente ao conjunto de ferramentas LLVM. *FlowTracker* é um analisador estático de código que é então carregado dinamicamente por *opt*. Uma vez que a representação intermediária é analisada por *FlowTracker* durante a execução de *opt* e concluído a não existência de canais laterais, pode-se então gerar o código final na linguagem assembly da arquitetura alvo, utilizando a ferramenta *llc* também disponível no conjunto de ferramentas do LLVM.

Entretanto, com o objetivo de facilitar ao máximo o uso de *FlowTracker* por aqueles sem muito conhecimento de como LLVM deve ser instalado e utilizado, desenvolveu-se uma interface web[4] que agrupa todas as etapas listadas acima, com exceção da geração final do código na linguagem alvo. Nesse caso, o usuário deverá apenas especificar um arquivo contendo o código fonte que será analisado e um arquivo XML contendo informações sobre quais dados do respectivo código, deverão ser considerados sigilosos por *FlowTracker*. O usuário também tem a possibilidade de digitar diretamente em caixa de texto, essas informações e o programa em linguagem C ou C++.

4.1. Entradas

A Figura 3 exibe a interface de usuário disponibilizada por FLow Tracker. Na aba *C/C++ Source Code* deve-se digitar o programa em linguagem C/C++. O usuário poderá optar por fazer o *upload* do arquivo com o código fonte clicando no botão *Choose File*. Na aba *XML Source Code*, o usuário poderá digitar as informações de quais dados serão considerados como sigilosos por *FlowTracker*. Obviamente também existe a opção de fazer *upload* de um arquivo `.xml` contendo tais informações.

O Formato do arquivo XML é descrito na Figura 4. Basicamente ele é composto de um preâmbulo fixo contendo as *tags* `<functions></functions>` e `<sources></sources>`. Para cada função que recebe ou gera uma informação considerada sigilosa tal como uma chave criptográfica ou um número aleatório, deve-se iniciar uma *tag* `<function></function>` contendo o nome da função. No exemplo da Figura 4 essa função é a `compVar`. Finalmente deve-se iniciar uma

[4]http://cuda.dcc.ufmg.br/flowtracker

```
<functions>
  <sources>
  <function>
    <name>compVar</name>
    <parameter>1</parameter>
  </function>
  </sources>
</functions>
```

Figura 4. Arquivo XML informando à *FlowTracker* o(s) dado(s) considerado(s) sigiloso(s).

tag `<parameter></parameter>`, contendo o valor 1 caso o primeiro parâmetro seja sigiloso, 2 caso o segundo e assim por diante. No caso de `compVar`, apenas o primeiro parâmetro (*pw*) deverá ser considerado sigiloso. Caso, o valor de retorno dessa função também seja considerado sigiloso, deve-se iniciar uma nova *tag* `<parameter></parameter>` informando o parâmetro 0.

4.2. Saídas

A Figura 5 mostra a saída de *FlowTracker* para as entradas correspondentes às Figuras 1 e 4. No sumário, percebe-se que foi encontrado: 1 informação sigilosa (parâmetro `pw` de `compVar`); 10 instruções de desvio e/ou índices de memória e 4 subgrafos vulneráveis. Cada subgrafo corresponde a um traço de instruções *assemby* correspondentes à linguagem intermediária de LLVM. Para cada traço, são gerados 3 arquivos de saída. Dois deles são mais adequados a usuários que conhecem um pouco da linguagem intermediária do compilador LLVM. São eles: `subgraphLLVM.dot` e `subgraphLines.dot`. Para visualizá-los pode-se usar qualquer ferramenta de visualização de arquivos `.dot` como `xdot` por exemplo. É possível também, clicar no ícone ao lado de cada arquivo para obter a versão em formato `.png`.

Tanto `subgraphLLVM.dot` quando `subgraphLines.dot` apresentam um subgrafo originado do grafo de dependências completo (arquivo `fullGraph.dot`), de forma que é possível visualizar 1 ou mais caminhos que conectam a informação sigilosa à uma instrução de desvio ou índice de memória. `subGraphLines.dot` apresenta como rótulo de cada vértice, a linha na qual ele é originado. Enquanto que `subGraphLLVM.dot` apresenta como rótulo de cada vértice, um *opcode* ou um operando. Arestas contínuas representam fluxo explícito e arestas tracejadas representam fluxo implícito.

Para cada subgrafo vulnerável é gerando também um terceiro arquivo de saída, chamado `subgraphASCIILines.txt`, o que informa em formato ASCII as linhas do programa C/C++ que correspondem ao respectivo subgrafo e representam um caminho no program entre informação sigilosa e instrução de desvio ou índice de memória. Neste caso, o usuário poderá localizar o traço vulnerável em seu código e corrigi-lo adequadamente à fim de eliminar o canal lateral. Basicamente, as correções se dão pela interrupção do caminho.

5. Trabalhos Relacionados

Lux *et al.* [Lux and Starostin 2011] implementou uma ferramenta que detecta vulnerabilidades de ataques por variação de tempo em programas Java. A principal diferença entre *FlowTracker* e a abordagem de Lux *et al*'s é o fato que eles operam na linguagem

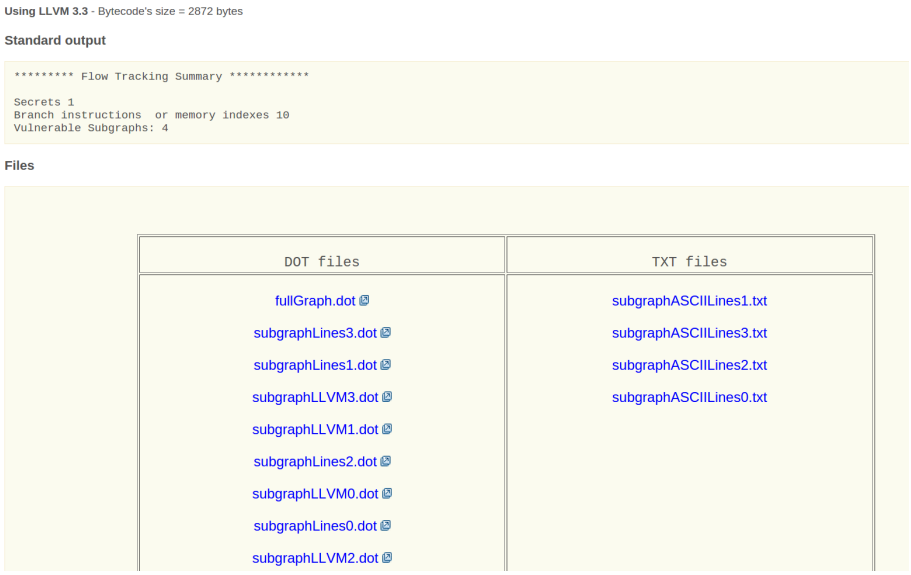


Figura 5. Exemplo de saída de *FlowTracker*.

de programação em alto nível, usando um conjunto de regra de inferência. Porém, a abordagem de *FlowTracker* tem algumas vantagens, porque ele trabalha diretamente na representação intermediária do compilador. Apesar de *FlowTracker* ser uma ferramenta para análise de código C e C++, seu algoritmo pode lidar com diferentes linguagens de programação bastando utilizar a ferramenta *llc* disponível na coleção de programas LLVM, que seja adequada à plataforma de *hardware* alvo. Além disso, otimizações do compilador podem ser realizadas antes da análise estática de *FlowTracker*, garantido que nenhum canal lateral possa ser introduzido no código executável. Finalmente, do ponto de vista de projeto, a abordagem de *FlowTracker* é substancialmente diferente de Lux *et al.*'s, porque pode-se analisar inclusive programas não estruturados, isto é, aqueles que utilizam instruções do tipo `goto`.

Existem também abordagens similares à de *FlowTracker*, mas que objetivam detectar outro tipo de canal lateral: variação de potência. Por exemplo, Moss et al. [Moss et al. 2012] apresenta um sistema de tipos que também opera na linguagem intermediária e detecta seguimentos de código vulneráveis à ataques por análise de variação de potência. Similarmente, Agosta *et al.* [Agosta et al. 2013], recentemente lançaram um passe LLVM que detecta quais instruções são dependentes de chaves criptográficas. As duas principais diferenças entre o trabalho de Agosta e *FlowTracker* são: (i) o fato que *FlowTracker* considera não somente o fluxo de dados, mas também o fluxo implícito de informação, isto é, aquele devido às dependências de controle; e (ii) o fato que *FlowTracker* está lidando com canal lateral relacionado a tempo e não potência.

6. Comentários finais

Acredita-se que *FlowTracker* seja a primeira ferramenta capaz de certificar que um programa possui um comportamento isócrono no nível do compilador. Ela é capaz de lidar com programas não estruturados pois foi concebida com um algoritmo simples e que executa em tempo linear, sendo capaz de representar em um grafo as relações de dependências de controle e de dados entre as variáveis do programa.

Referências

Agosta, G., Barenghi, A., Maggi, M., and Pelosi, G. (2013). Compiler-based Side Channel Vulnerability Analysis and Optimized Countermeasures Application. In *Proceedings of DAC*, New York, NY, USA. ACM.

Almeida, J. B., Barbosa, M., Pinto, J. S., and Vieira, B. (2013). Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796–812.

Bernstein, D. J., Lange, T., and Schwabe, P. (2012). The security impact of a new cryptographic library. In *Progress in Cryptology – LATINCRYPT*, pages 159–176. Springer.

Biham, E. and Shamir, A. (1997). Differential fault analysis of secret key cryptosystems. In *CRYPTO*, pages 513–525. Springer.

Chen, Y.-F., Hsu, C.-H., Lin, H.-H., Schwabe, P., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y., and Yang, S.-Y. (2014). Verifying Curve25519 software. In *Proceedings of CCS*, pages 299–309. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, pages 25–35.

Fan, J., Guo, X., Mulder, E. D., Schaumont, P., Preneel, B., and Verbauwhede, I. (2010). State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In *HOST*, pages 76–87.

Genkin, D., Shamir, A., and Tromer, E. (2014). RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, pages 444–461. Springer.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2009). Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98.

Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer.

Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113. Springer.

Lattner, C. and Adve, V. (2004). The llvm compiler framework and infrastructure tutorial. In *LCPC Mini Workshop on Compiler Research Infrastructures*.

Lux, A. and Starostin, A. (2011). A tool for static detection of timing channels in java. *Journal of Cryptographic Engineering*, 1(4):303–313.

Moss, A., Oswald, E., Page, D., and Tunstall, M. (2012). Compiler assisted masking. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer.

Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer.

Standaert, F.-X. (2010). Introduction to Side-Channel Attacks Secure Integrated Circuits an d Systems. In *Integrated Circuits and Systems*, chapter 2, pages 27–42. Springer.

WPTrans: Um assistente para Verificação de Programas no Frama-C*

David Déharbe[1]**,Vitor A. Almeida**[1]**,Richard Bonichon**[1]

[1] Departamento de Matemática e Informática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN) – Natal, RN – Brasil

david@dimap.ufrn.br, vitoralcantara@ppgsc.ufrn.br, richard.bonichon@gmail.com

Abstract. *This article describes the early stage of WPTrans: an extension to Frama-C and WP. This extension enable to manually help automated theorem provers by manipulating generated proof obligations. This gives yet another bullet to prove program correctness.*

Resumo. *Este artigo descreve a fase inicial de uma extensão para Frama-C e WP: o WPTrans. Esta extensão permite ajudar manualmente provadores automáticos de teoremas, através da manipulação das obrigações de prova geradas. Dando, ainda, uma nova opção para provar a corretude de programas.*

`https://youtu.be/lpCvw6huenk`

1. Introdução

O Frama-C [Cuoq et al. 2012] é uma plataforma aberta e extensível de análise de programas em linguagem C, auxiliando na verificação formal de sistemas baseados nesta linguagem de programação. Possui um conjunto de *plug-ins*, provendo, notadamente, análise estática e verificação formal de código C.

Tanto uma função ou um projeto completo em C pode ser especificado pelo Frama-C. Para tal, devem ser inseridas anotações lógicas no código fonte. Escritas usando a linguagem ACSL [Baudin et al. 2015], estas anotações são usadas pelos *plugins* do Frama-C em suas análises.

A figura 1 contem um exemplo de função em C com uma especificação em ACSL. Toda anotação ACSL é inserida em forma de comentários, com o símbolo `@` logo após `/*` e `//`. Neste exemplo, a especificação é composta por uma pré-condição (`requires`) e uma pós-condição (`ensures`). A pré-condição é que os ponteiros `a` e `b` apontam para regiões válidas de memória. A pós-condição é que o valor final no endereço de memória apontado por `a` é o valor que estava inicialmente no endereço apontado por `b`, e vice-versa.

O *plugin* WP tem como papel realizar a verificação formal de que a implementação satisfaz o seu contrato. Seu nome é uma sigla para *Weakest Precondition*, pois o WP se utiliza da técnica de pre-condição mais fraca para provar as propriedades funcionais, uma técnica baseada nos trabalhos de Hoare[Hoare 1969], Floyd[Floyd 1967] e Dijkstra[Dijkstra 1968] sobre corretude de programas [Cuoq et al. 2012]. O WP pode verificar tanto as propriedades funcionais no código (os resultados das funções condizem

*Este trabalho foi parcialmente suportado pela CAPES e CNPq processo 573964/2008-4 (Instituto Nacional de Ciência e Tecnologia para Engenharia de Software, `www.ines.org.br`.)

```
/*@ requires \valid(a) && \valid(b);
    ensures (*a == \old(*b) && *b == \old(*a));*/
void swap(int *a, int *b){
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Figure 1. Um exemplo de função C anotada com um contrato em ACSL.

com a especificação), assim como verificar sua segurança (*safety*) através da verificação de propriedades genéricas: acessos de memória são válidos, ausência de estouro aritmético (*arithmetic overflow*), de divisão por zero, etc.

Para provar as propriedades de um programa, o WP analisa o código C e as anotações ACSL correspondentes, gerando Obrigações de Provas (chamadas a partir deste ponto de OPs) para averiguar se a execução do sistema efetivamente seguirá à risca as condições exigidas pela anotação.

Existem três tipos de OPs geradas pelo WP:

Lemma: Esse tipo de OP é gerado a partir de lemas inseridos pelo usuário nas anotações do código e possui apenas um predicado que deve ser provado.

Annot: Esse tipo de OP é gerado para verificar as demais propriedades funcionais das anotações e do código. Cada OP deste tipo possui um conjunto de hipóteses $\{H_1, \cdots, H_n\}$ e um objetivo O, no qual se deve provar $\bigwedge_{i \in 1..n} H_i \rightarrow O$.

Check Esse tipo de OP ainda não foi considerado, pois não foram encontrados exemplos de seu uso, nem foi identificado em que situação o mesmo é produzido,

Todas as fórmulas presentes nas OPs supracitadas são da lógica de primeira ordem tipada. Além disso, a geração de OPs tem como parâmetros o *modelo aritmético* e o *modelo de memória*.

O modelo de memória diz respeito a como as variáveis C serão representados nas OPs. As variáveis em C podem ser mapeadas diretamente em variáveis lógicas, traduzidas para vetores de tipos atômicos (inteiro, ponteiro, *float*) ou pode ser representado diretamente em vetores de *bytes*. O primeiro caso é o modelo de Hoare, que produz OPs mais simples, porém não permite tratar ponteiros. O segundo caso é o modelo *Typed*, que produz OPs mais complexas, pode tratar de ponteiros, mas não trata conversões explícitas para ponteiros de tipos diferentes. E o terceiro é o modelo *Bytes*, que pode representar todos os detalhes do programa, mas possui OPs mais complexas e ainda não se encontra implementado até a versão 0.9 do WP.

O modelo aritmético diz respeito a como representar inteiros e *floats* em C: representá-los como inteiros e reais de máquina, com limites numéricos e arredondamentos, ou representá-los matematicamente (modelo dito *natural*).

A figura 3 ilustra uma OP gerada pelo WP a partir do algoritmo 1 com o modelo de memória *Typed* e representação natural de inteiros e reais. As funções region e base acessam posições do ponteiro e a função valid_rw verifica se o ponteiro é válido. Suas

variáveis e tipos são mostrados na figura 2.

$$
\begin{array}{lcl}
a_0 & : & \text{data} \\
\mathsf{Mint}_0 & : & \text{array[int]} \\
b_0 & : & \text{data} \\
\mathsf{Malloc\_0} & : & \text{array[int]}
\end{array}
$$

Figure 2. Variáveis presentes no Exemplo do OP

$$
\begin{array}{rl}
H_0: & \text{let } x_0 = \mathsf{Mint}_0[a_0] \text{ in} \\
& \text{let } x_1 = \mathsf{Mint}_0[b_0] \text{ in} \\
& \mathsf{is\_sint32}(x_0) \wedge \mathsf{is\_sint32}(x_1) \wedge \mathsf{is\_sint32}(\mathsf{Mint}_0[a_0 \mapsto x_1][b_0 \mapsto x_0][a_0]). \\
H_1: & \mathsf{linked}(\mathsf{Malloc\_0}) \wedge \mathsf{region}(\mathsf{base}(a_0)) \leq 0 \wedge \mathsf{region}(\mathsf{base}(b_0)) \leq 0 \\
H_2: & \mathsf{valid\_rw}(\mathsf{Malloc\_0}, a_0, 1) \wedge \mathsf{valid\_rw}(\mathsf{Malloc\_0}, b_0, 1) \\
\hline
O: & \text{let } x_0 = \mathsf{Mint}_0[b_0] \text{ in } x_0 = \mathsf{Mint}_0[a_0 \mapsto x_0][b_0 \mapsto \mathsf{Mint}_0[a_0]][a_0]
\end{array}
$$

Figure 3. Exemplo de OP

Cada OP pode ser provada diretamente pelo WP, usando reescrita de fórmulas e tentativas de provas triviais, através de provadores SMT, como o Alt-Ergo [Bobot et al. 2008], assistentes de provas, como o Coq [Bertot and Castéran 2004] e gerenciadores de provas, como o Why3 [Filliâtre and Paskevich 2013].

As formas mais rápidas de se validar uma OP são através do próprio WP, nos casos em que a prova possa ser trivial, ou pelos provadores SMT. Porém, em ambos os casos, pode não haver sucesso, sendo duas as razões possíveis para tal: as anotações não condizem com a implementação; ou a OP é bastante complexa para ser resolvida nas restrições de tempo e espaço definidas pelo usuário.

No primeiro caso, é preciso verificar e corrigir manualmente as anotações e/ou o código. Já na segunda situação, o usuário deve usar um assistente dedicado a fim de provar interativamente as OPs. Esse último caso pode ser complexo pois o usuário se encontrará em um contexto produzido automaticamente e, geralmente, com uso de ferramentas que trabalham em uma lógica de ordem superior. Desta forma, exige-se experiência do usuário, tanto no manuseio, quanto na estratégia de prova de OPs.

Ademais, dependendo do assistente de provas, pode ser complexa, ou mesmo inexistente, sua comunicação com provadores automáticos. Na segunda condição, os provadores não poderiam ser utilizados para concluir a prova, cabendo ao usuário apenas a opção de prová-la manualmente.

2. WPTrans

Propõe-se então uma terceira alternativa: WPTrans, uma extensão do WP que forneça uma interface gráfica para provar interativamente as OPs e submetê-las a provadores SMT em qualquer etapa da prova. A interação permite aplicar transformações, cada

transformação sendo uma instância de um conjunto pré-estabelecido de táticas. Dá-se então ao usuário a possibilidade de simplificar as OPs até que elas se encontrem tratáveis por provadores automáticos (ou seja evidenciado que não são válidas).

Como guia para escolher quais táticas fornecer através do WPTrans, fez-se um estudo de:

- as táticas Coq usadas para verificar interativamente OPs geradas pelo WP a partir dos exemplos fornecidos no *ACSL By Example* [Burghardt and Gerlach 2015];
- as táticas do provador interativo do Atelier-B, uma plataforma de projeto formal de componentes de software com o método B [Mentré et al. 2012];
- as táticas do provador interativo do Rodin, uma plataforma Eclipse de especificação formal de sistemas em Event-B [Abrial et al. 2010].

As táticas já implementadas são:

Enviar hipótese ao objetivo: Esta tática envia uma hipótese selecionada ao objetivo, sendo adicionado através de uma implicação.

Substituir termo: Ao receber dois argumentos $t1$ e $t2$, no qual ambos são termos, esta tática substitui todas as ocorrências de $t1$ por $t2$ no objetivo da OP selecionada, e cria uma nova OP com as mesmas hipóteses da OP anterior mas com o objetivo $t1 = t2$.

Remover hipóteses: Esta tática permite tanto remover hipóteses selecionadas da OP, como selecionar um subconjunto das mesmas, removendo as demais.

Reescrever termo: Recebendo como argumentos uma hipótese ou um lema com a forma $\forall (x_1 : A_1) \cdots (x_n : A_n) \cdot p_1 \implies \cdots \implies p_n \implies t_1 = t_2$, esta tática substitui todas as ocorrências de t_1 por t_2 ou vice-versa, após realizar o casamento de um subtermo do objetivo com t_1 ou t_2 dependendo da ordem escolhida pelo usuário.

Indução natural: Esta tática aplica uma indução natural no objetivo da OP, digamos $P(x)$, onde x é uma variável inteira positiva, e gera duas OPs com as mesmas hipóteses e como objetivos $P(0)$ e se $P(x) \implies P(x + 1)$.

Substituir função por sua definição: Com esta tática, as aplicações de uma função são substituídas por sua definição, realizando as substituições dos argumentos da função pelos subtermos da OP.

Asserção: Com esta regra uma nova hipótese pode ser adicionada à OP, dando mais um argumento para prová-la. Esta hipótese deve ser provada válida também.
Esta regra se assemelha à regra do corte.

Dividir OP: Se o objetivo da OP for uma conjunção de n termos, esta tática gera n OPs, sendo o objetivo de cada nova OP um dos termos, sem repetição de objetivo.

Dividir conjunção em hipótese: Se a hipótese selecionada H for uma conjunção de n termos, n novas OPs são geradas, sendo a fórmula de H substituída por um dos termos em cada nova OP.

Remover termo em disjunção: Se o objetivo da OP for uma disjunção de termos, a tática remove um dos termos da disjunção, gerando uma OP com o novo objetivo.

Dividir disjunção em hipótese: Se a hipótese selecionada H for uma disjunção de n termos, n novas OPs são geradas, sendo H substituída por um dos termos em cada nova OP, sem repetição de termo.

Contraposição: Esta regra aplica a lei da contraposição a um objetivo ou hipótese.

Reescrever `if-then-else`: As hipóteses ou objetivo podem possuir o formato `If` A `then` B `else` C. Sendo este o caso, a fórmula é substituída por $(A \wedge B) \vee (\neg A \wedge C)$ na OP gerada.

Eliminar quantificador existencial: Esta tática pode ser aplicada tanto em uma hipótese quanto no objetivo, desde que tenha o formato $\exists x \cdot P$, no qual P é um termo.

- Termo selecionado é uma hipótese: é efetuada uma *skolemização*, substituindo todas as ocorrências de x em P por um novo símbolo v, retornando uma nova OP com o objetivo modificado.
- Termo selecionado é o objetivo: é realizada uma instanciação existencial, no qual o quantificador é removido, e x substituído por uma variável livre da OP à escolha do usuário.

Prova por casos: Dado um termo T fornecido pelo usuário e uma OP com o objetivo O, esta regra cria duas OPs, sendo o objetivo da primeira $(\neg T) \implies O$ e o objetivo da segunda $T \implies O$.

Introdução de objetivo: Esta tática é aplicável em dois casos. Caso o objetivo tenha o formato $A \implies B$, no qual A e B são termos quaisquer, é gerada uma nova OP com uma hipótese adicional cuja fórmula é A e cujo objetivo é B.
Se o objetivo for uma quantificação universal, é realizada uma instanciação universal, adicionando uma nova variável livre à OP.

Modus ponens: Uma vez que o usuário selecione uma hipótese com um termo P, a extensão irá procurar, em todas as demais hipóteses da mesma OP, uma hipótese cujo termo possua o formato $P \rightarrow G$, no qual G é um termo qualquer. Caso esta hipótese seja encontrada, uma nova OP é criada com as seguintes condições:

- Uma hipótese com a fórmula G é adicionada.
- A hipótese $P \rightarrow G$ é removida.

3. Interface e Organização do WPTrans

A figura 4 ilustra a janela principal da extensão, mostrando, na coluna central, as hipóteses e objetivo da OP a ser provada. Na coluna à esquerda são listadas, na região superior, as OPs do projeto, e, na região inferior, as variáveis da OP selecionada.

Na coluna mais à direita se encontra, na região superior, o histórico de táticas aplicadas assim como as definições de funções e lemas aplicáveis a OP selecionada. O *widget* textual no meio é o console, por onde o usuário pode interagir com as OPs, e as duas abas na região inferior são, respectivamente, o resultado da execução dos comandos no console, e o resultado dos diferentes provadores aplicados na tentativa de prova da OP.

As regras de modificação e os demais comandos possíveis (desfazer táticas, executar um provador SMT, etc.) são executados no console. Outra opção para executar as regras de modificação é através do clique com o botão direito do *mouse* nas hipóteses e no objetivo, na qual são listadas todas as táticas disponíveis para a(s) fórmula(s) selecionada(s).

Realizamos toda a implementação modificando o código original do WP presente na distribuição *Sodium* do Frama-C[1]. A interface gráfica foi implementada com lablgtk[2] e utilizamos apenas módulos e funções da biblioteca base do OCaml e fornecidos pelo próprio Frama-C. Implementamos o WPTrans na linguagem OCaml versão 4.02.1.

[1] `http://frama-c.com/download.html`
[2] `http://lablgtk.forge.ocamlcore.org/`

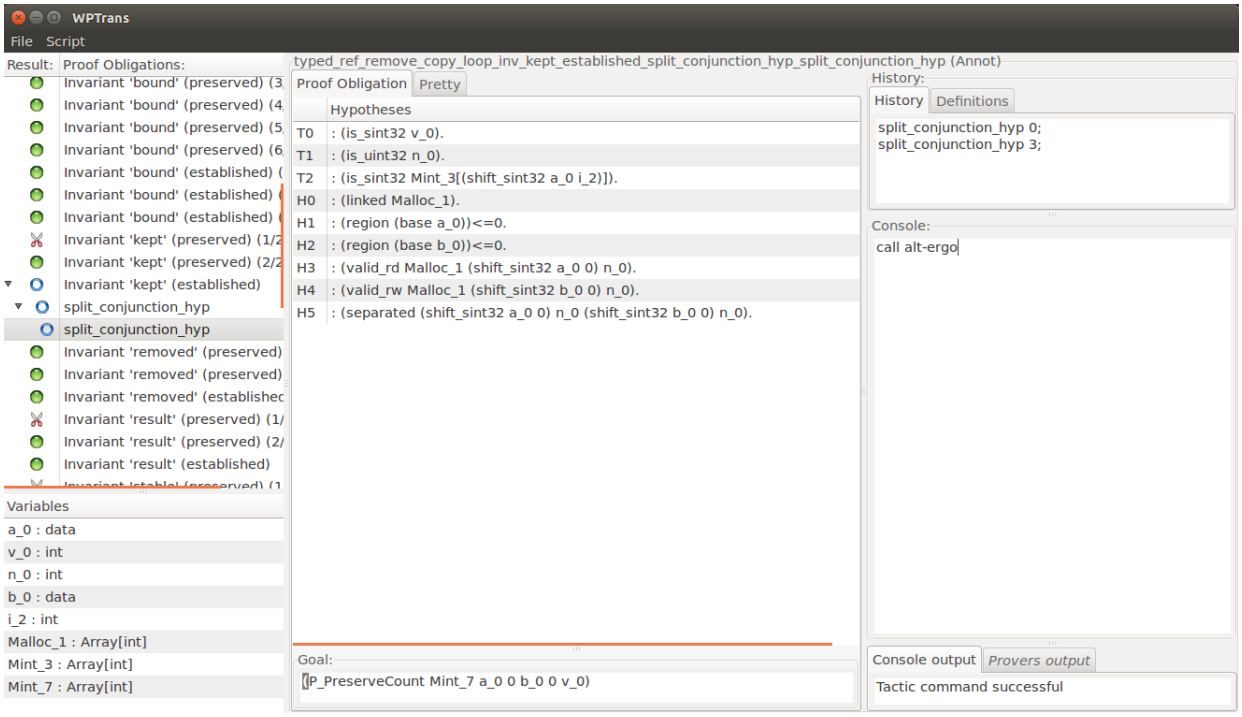


Figure 4. Janela principal do WPTrans

4. Análise experimental

A figura 5 é uma OP com modelo *Typed* e inteiros naturais gerada a partir do algoritmo `remove_copy`, versão estável, disponível em [Burghardt and Gerlach 2015][3].

$$\frac{}{\begin{array}{l}\text{let } x_0 = \mathsf{L\_Count}(\mathsf{Mint}_0, a_0, n_0, v_0) \text{ in} \\ 0 \leq n_0 \rightarrow \mathsf{is\_sint32}(v_0) \rightarrow (0 \leq x_0 \wedge x_0 \leq n_0)\end{array}}$$

Figure 5. Exemplo de OP

Os autores deste exemplo não obtiveram sucesso com os solucionadores SMT Alt-Ergo e CVC4, e tiveram que elaborar uma prova em Coq, utilizando um total de oito (8) táticas diferentes. Repetimos as tentativas de prova com a mesma configuração de solucionadores SMT localmente, porém sem sucesso. Entretanto, ao utilizar o WPTrans, esta OP foi validada utilizando os seguintes comandos em ordem:

1. intros (introdução de objetivo)
2. nat_ind n_0 (Indução natural sobre n_0)
3. call alt-ergo (Alt-Ergo é chamado para a conclusão da prova)

Com WPTrans, apenas dois passos de interação são necessários para chegar então a um estado onde o provador SMT pôde concluir a prova a partir daquele ponto. As versões usadas de Alt-Ergo e CVC4 nas tentativas locais de provas foram, respectivamente, 0.99.1 e 1.4, e o *timeout* definido para os provadores foram de 10 milissegundos, sendo mantido padrão os demais valores dos provadores.

[3]Os autores mantiveram um repositório com a implementação dos exemplos em `https://gitlab.fokus.fraunhofer.de/`

Além deste exemplo, [Burghardt and Gerlach 2015] possui outras oito OPs validadas apenas com o uso de Coq, isto é, provas manuais com auxílio de computador. Destas, duas também foram provadas pelo WPTrans. O estudo das provas interativas com Coq para as OPs remanescentes fornecerá subsídios para determinar quais táticas adicionais devem ser implementadas no WPTrans.

Além destes exemplos, pretendemos consolidar e validar WPTrans a partir dos arquivos-fontes em C com anotações do sítio de tutoriais do Frama-C[4], e do sítio do Toccata[5].

5. Conclusão e trabalhos futuros

O WPTrans surge como uma alternativa com baixo custo de aprendizagem para assistir nas tentativas de prova, fornecendo um ponto intermediário entre provadores totalmente automáticos e assistentes de prova. É uma ferramenta com objetivos similares a TLAPS [Chaudhuri et al. 2008], um sistema de prova para TLA+ e às interfaces de prova interativa encontrados em IDEs como Rodin [Abrial et al. 2010] e Atelier-B [Mentré et al. 2012]. Com esta extensão, não é necessário que o usuário manipule a OP até que encontre uma prova. Basta aplicar táticas simples para reduzir a OP inicial em OPs mais simples onde provadores automáticos têm possibilidade de êxito. Este diferencial é o que nos motiva a realizar esta extensão.

A próxima etapa será introduzir novas regras de acordo com a realização de testes como descrito na seção anterior. Uma outra funcionalidade também planejada é a obtenção de contra-exemplos nos casos em que os provadores refutem uma OP. As atuais alternativas de provadores no Frama-C não disponibilizam muitas informações além do resultado, sendo esta uma forma de permitir uma maior comunicação com os provadores. E será executado, conjuntamente, um aprimoramento da interface gráfica, permitindo o uso para avaliação e testes por outros usuários.

Uma questão importante é: as provas desenvolvidas com WPTrans são corretas? Responder de forma positiva a esta questão necessitará:

1. Comprovar que as regras de inferência implementadas são corretas: isto passa por uma formalização do arcabouço de raciocínio e verificação das regras neste arcabouço. Pode ser realizado utilizando um assistente de prova como Coq.
2. Verificar que a implementação das regras de inferência é correta: é uma instância do problema geral de verificação de corretude de programas que pode ser solucionada utilizando ferramentas como o próprio Frama-C. Alternativamente, poderia-se utilizar a prova de corretude das regras de inferência mencionados no item anterior, para gerar automaticamente uma implementação correta, utilizando a funcionalidade de geração de código do Coq, por exemplo.
3. Ter um mecanismo de verificação dos resultados obtidos através dos solucionadores SMT: como vários solucionadores SMT possuem geradores de prova, uma solução é incluir um verificador de prova no processo de validação do resultado. Idealmente este verificador de prova seria integrado ao mesmo arcabouço formal daquele usado para verificar as regras de inferência.

[4] `https://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:tutorial`

[5] `http://toccata.lri.fr/gallery/jessieplugin.en.html`

Enfim, constatando-se a eficácia do WPTrans, planejamos discutir com os desenvolvedores do Frama-C e do WP a possibilidade de embutir integralmente esta extensão nas próximas distribuições do programa e do *plug-in*, com o intuito de agilizar a especificação formal, bem como diminuir custos no desenvolvimento de sistemas em C.

References

Abrial, J., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466.

Baudin, P., Filliâtre, J. C., Cuoq, P., Marché, C., Monate, B., Moy, Y., and Prevosto, V. (2015). *ACSL: ANSI C Specification Language*.

Bertot, Y. and Castéran, P. (2004). *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York.

Bobot, F., Conchon, S., Contejean, E., and Lescuyer, S. (2008). Implementing Polymorphism in SMT Solvers. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 1–5, New York, NY, USA. ACM.

Burghardt, J. and Gerlach, J. (2015). ACSL By Example: Towards a Verified C Standard Library. Acessado em: 14 maio 2015.

Chaudhuri, K., Doligez, D., Lamport, L., and Merz, S. (2008). A TLA+ proof system. In Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R. A., and Schulz, S., editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg. Springer-Verlag.

Dijkstra, E. (1968). A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186.

Filliâtre, J.-C. and Paskevich, A. (2013). Why3 - Where Programs Meet Provers. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer Berlin Heidelberg.

Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Mentré, D., Marché, C., Filliâtre, J., and Asuka, M. (2012). Discharging Proof Obligations from Atelier-B Using Multiple Automated Provers. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 238–251.

JSClassFinder: A Tool to Detect Class-like Structures in JavaScript

Leonardo Humberto Silva[1], Daniel Hovadick[2], Marco Tulio Valente[2],
Alexandre Bergel[3],Nicolas Anquetil[4], Anne Etien[4]

[1]Department of Computing – Federal Institute of Northern Minas Gerais (IFNMG)
Salinas – MG – Brazil

[2]Department of Computer Science – Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

[3]Department of Computer Science – Pleiad Lab
University of Chile – Santiago – Chile

[4]RMoD Project-Team – INRIA Lille Nord Europe – France

leonardo.silva@ifnmg.edu.br, {dfelix,mtov}@dcc.ufmg.br,
abergel@dcc.uchile.cl, {nicolas.anquetil,anne.etien}@inria.fr

Abstract. *With the increasing usage of JavaScript in web applications, there is a great demand to write JavaScript code that is reliable and maintainable. To achieve these goals, classes can be emulated in the current JavaScript standard version. In this paper, we propose a reengineering tool to identify such class-like structures and to create an object-oriented model based on JavaScript source code. The tool has a parser that loads the AST (Abstract Syntax Tree) of a JavaScript application to model its structure. It is also integrated with the Moose platform to provide powerful visualization, e.g., UML diagram and Distribution Maps, and well-known metric values for software analysis. We also provide some examples with real JavaScript applications to evaluate the tool.*

Video: `http://youtu.be/FadYE_FDVM0`

1. Introduction

JavaScript is a loosely-typed dynamic language with first-class functions. It supports object-oriented, imperative, and functional programming styles. Behaviour reuse is performed by cloning existing objects that serve as prototypes [Guha et al. 2010]. ECMAScript [ecm 2011] is a scripting language, standardized by ECMA International, that forms the base for the JavaScript implementation. ECMAScript 5 (ES5) is the version currently supported by most browsers. Version 6 of the standard is planned to be officially released around mid 2015[1].

Due to the increasing usage of JavaScript in web applications, there is a great demand to write JavaScript code that is reliable and maintainable. In a recent empirical study, we found that many developers emulate object-oriented classes to implement parts of their systems [Silva et al. 2015]. However, to the best of our knowledge, none of the

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript, verified 05/18/2015

existing tools for software analysis of JavaScript systems identify object-oriented entities, such as classes, methods, and attributes.

In this paper, we propose and describe the JSClassFinder reengineering tool that identifies class-like structures and creates object-oriented models based on JavaScript source code. Although ES5 has no specific syntax for class declaration, JSClassFinder is able to identify structures that emulate classes in a system. Prototype-based relationships among such structures are also identified to infer inheritance. The resulting models are integrated with Moose[2], which is a platform for software and data analysis. The main features of the proposed tool are:

- Identification of class-like entities to build an object-oriented model of a system.
- Integration with a complete platform for software analysis.
- Graphical visualization of the retrieved class-like structures, using UML class diagrams, distribution maps, and tree view layouts.
- Automatic computation of widely known source code metrics, such as number of classes (NOC), number of methods (NOM), number of attributes (NOA), and depth of inheritance tree (DIT).

This tool paper is organized as follows. Section 2 describes JSClassFinder's architecture. Section 3 uses one toy example and two real applications to demonstrate the tool. Section 4 describes exceptions that are not covered by the tool. Section 5 presents related work and Section 6 concludes the paper.

2. JSClassFinder in a Nutshell

The execution of JSClassFinder is divided into two stages: preprocessing and visualization. The preprocessing is responsible for analyzing the AST of the source code, identifying class-like entities, and creating an object-oriented model that represents the code. In the visualization stage, the user can interact with the tool to visualize the model and to inspect all metrics and visualization features that the software analysis platform provides.

JSClassFinder is implemented in Pharo[3], which is a complete Smalltalk environment for developing and executing object-oriented code. Pharo also offers strong live programming features such as immediate object manipulation, live update, and hot recompilation. The system requirements to execute JSClassFinder are: (i) the AST of a JavaScript source code in JSON format; (ii) A Pharo image with JSClassFinder. A ready-to-use Pharo image is available at the JSClassFinder website[4]. Figure 1 shows the architecture of the JSClassFinder, which includes the following modules:

AST Parser: This module receives as an input the AST of a JavaScript application. Then it creates a JavaScript model as part of the preprocessing stage. Currently, we are using Esprima[5], a ECMAScript parser, to generate the AST in JSON format.

Class Detector: This module is responsible for identifying class-like entities in the JavaScript model. It is the last step of the preprocessing stage, when an object-oriented model of an application is created and made available to the user.

[2]http://www.moosetechnology.org/, verified 05/18/2015
[3]http://pharo.org/, verified 05/18/2015
[4]http://aserg.labsoft.dcc.ufmg.br/jsclasses/, verified 05/18/2015
[5]http://esprima.org/, verified 05/18/2015

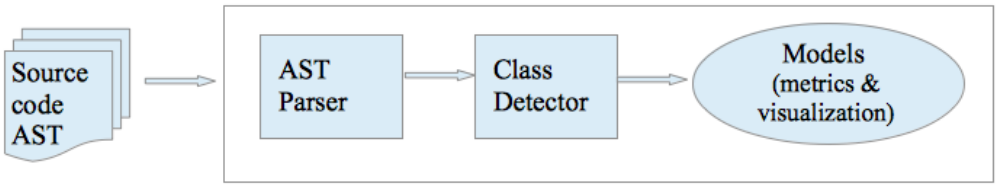


Figure 1. JSClassFinder's architecture

Models (metrics & visualization): This module provides visualizations for an user to interact with the tool and to "navigate" by the application's model. All information about classes, methods, attributes, and inheritance relationships is available. The main visualizations provided are: UML class diagram, distribution maps [Ducasse et al. 2006], and tree views. For the metrics, the tool provides the total number of classes (NOC) and, for each class: number of methods (NOM), number of attributes (NOA), number of children (subclasses) and depth of inheritance tree (DIT).

Figure 2 shows the main browser of JSClassFinder's user interface. In the top menu, the user can load a new JavaScript application or open one existing model, previously loaded. The only information required to load new applications are: (i) application's name and (ii) root directory where the JSON files with the target system's AST are located. After the preprocessing stage, the tool opens a new model inside a panel, where the user can navigate through class entities and select any graphical visualizations or metrics available. Section 3 presents some examples of what the user can do once they have a model created.

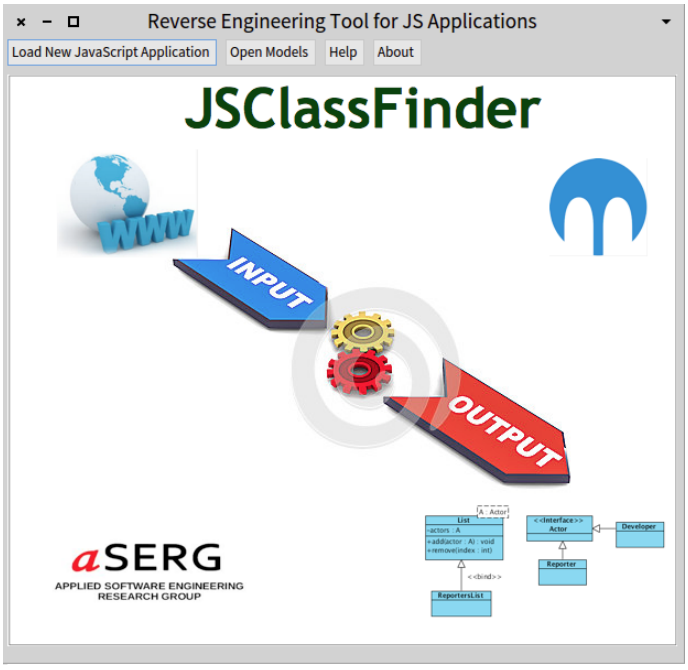


Figure 2. JSClassFinder initial user interface

2.1. Strategy for class detection

In this section, we describe the strategy used by JSClassFinder to represent the way classes are created in JavaScript and the way they acquire fields and methods. A detailed description is provided in a conference paper [Silva et al. 2015].

Definition #1: A class is a tuple $(C, \mathcal{A}, \mathcal{M})$, where C is the class name, $\mathcal{A} = \{a_1, a_2, \ldots, a_p\}$ are the attributes defined by the class, and $\mathcal{M} = \{m_1, m_2, \ldots, m_q\}$ are the methods. Moreover, a class $(C, \mathcal{A}, \mathcal{M})$, defined in a program P, must respect the following conditions:

- P must have a function with name `C`.
- P must include at least one expression of type `new C()` or `Object.create (C.prototype)`.
- For each $a \in \mathcal{A}$, the function `C` must include an assignment `this.a = Exp` or P must include an assignment `C.prototype.a = Exp`.
- For each $m \in \mathcal{M}$, function `C` must include an assignment `this.m = function {Exp}` or P must include an assignment `C.prototype.m = function {Exp}`.

Definition #2: Assuming that $(C1, \mathcal{A}1, \mathcal{M}1)$ and $(C2, \mathcal{A}2, \mathcal{M}2)$ are classes in a program P, we define that $C2$ is a subclass of $C1$ if one of the following conditions holds:

- P includes an assignment `C2.prototype = new C1()`.
- P includes an assignment `C2.prototype = Object.create(C1.prototype)`.

3. Examples

This section shows examples of usage of JSClassFinder to analyze one toy example and two real JavaScript applications extracted from GitHub.

3.1. Toy Example

This example includes two simple classes, `Mammal` and `Cat`, to illustrate how classes can be emulated in JavaScript. Listing 1 presents the function that defines the class `Mammal` (lines 1-3), which includes an attribute `name`. This class also has a method named `toString` (lines 4-6), represented by a function which is associated to the prototype of `Mammal`. Line 7 indicates that the class `Cat` (lines 9-11) inherits from the prototype of `Mammal`. The usage of variables `animal` and `myPet` demonstrate how the classes can be instantiated and used (lines 12-13).

Listing 1 represents one way of defining classes and instantiating objects. There are some variations and customized implementations, as we can find in [Flanagan 2011, Crockford 2008, Gama et al. 2012]. JSClassFinder supports different types of class implementation, as reported in [Silva et al. 2015].

3.2. Algorithms.js

`Algorithms.js`[6] is an open source project that offers traditional algorithms and data structures implemented in JavaScript. We analyzed version 0.8.1, which has 3,263 LOC.

[6]https://github.com/felipernb/algorithms.js/ verified 05/18/2015

```
function Mammal(name) {
  this.name=name;
}
Mammal.prototype.toString=function(){
  return '['+this.name+']';
}
Cat.prototype = Object.create(Mammal.prototype);  // Inheritance
...
function Cat(name) {
  this.name='"meow" ' + name;
}
var animal = new Mammal('Mr. Donalds');
var myPet = new Cat('Felix');
```

Listing 1. Class declaration and object instantiation

Figure 3 shows the class diagram of Algorithms.js, generated automatically by JSClassFinder, after the preprocessing stage. The classes represent the data structures that are supported by Algorithms.js, as we can check in the project's documentation webpage[7]. The main algorithms, e.g., Dijkstra, EulerPath, Quicksort, that the application offers in its API, are implemented as JavaScript global functions, not classes.

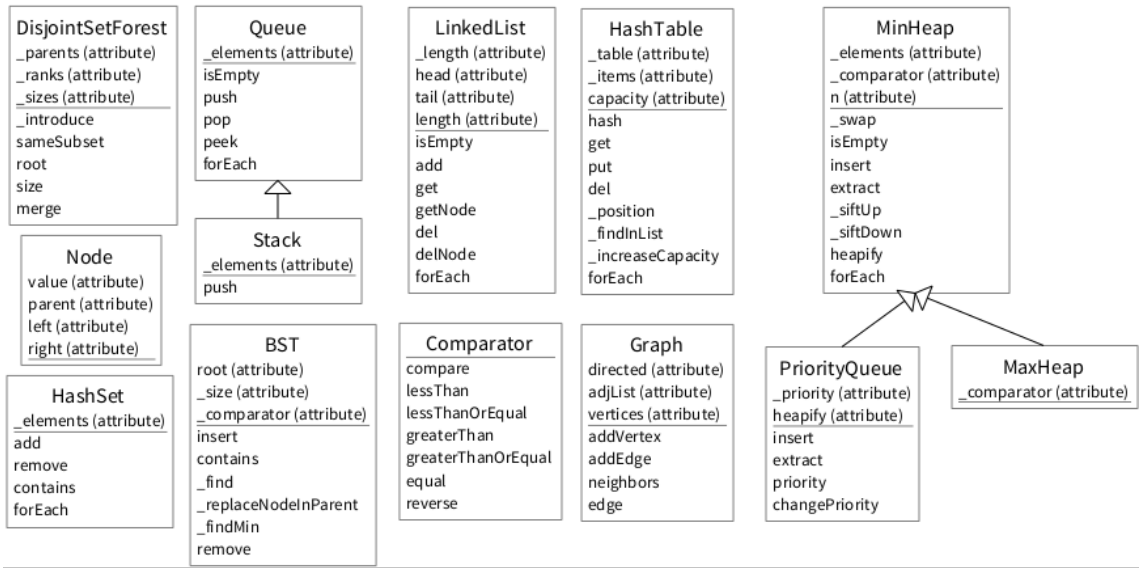


Figure 3. UML class diagram for Algorithms.js

3.3. PDF.js

PDF.js[8] is a Portable Document Format (PDF) viewer that is built with HTML5. It is a community-driven project supported by Mozilla Labs. We analyzed version 1.1.1, which has 57,359 LOC, 182 classes, 947 methods, and 876 attributes. Users can interact with the model to access all visualization features and metric values. It is also possible to use drill-down and drill-up operations when one entity is selected. For example, when the user performs a click on the number of classes ("All classes") metric, the panel will drill-down to show a list with all the classes. If the user performs a click on one of the classes,

[7] http://algorithmsjs.org/ verified 05/18/2015
[8] https://github.com/mozilla/pdf.js verified 05/18/2015

the tool will show all information related to the class, i.e., methods, attributes, subclasses, and superclasses. If the user performs a click on “All methods”, the panel will drill-down again to show a list with all the methods, etc. A menu with all visualization options and diagrams is shown when the user performs a right click on a given element.

JavaScript does not define language constructs for modules or packages. Therefore, a module can be a single file of JavaScript code that might contain a class definition, a set of related classes, a library of utility functions, or just a script of code to execute [Flanagan 2011]. JSClassFinder uses source code files as packages to allow the visualization of distribution maps per packages, like in the example shown in Figure 4. When a user selects the distribution map option, it is possible to choose which parameter will be exposed in the diagram. In this case, packages are represented by external rectangles and the small blue squares are classes. It is also possible to change the colors and to establish a valid range to be considered. For example, users can inform that the diagram should use red squares and consider only classes with more than five methods. This feature can be used, for example, to easily locate the biggest classes in a system.

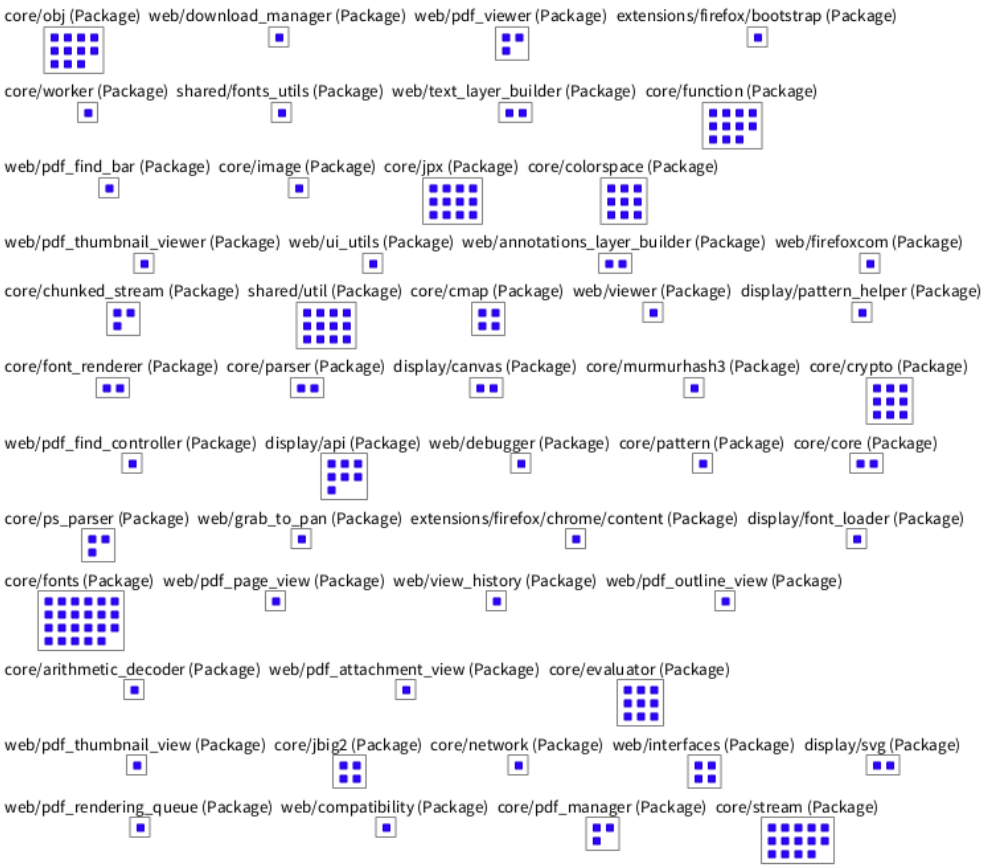


Figure 4. Distribution map for PDF.js (small squares are classes)

4. Limitations

As mentioned before, there are different ways to emulate classes and inheritance in JavaScript. The following specific cases are not covered by our current implementation:

Use of third-party libraries with customized object factories. For example, ClazzJS[9] is a portable JavaScript library for class-style OOP programming. It provides a DSL for defining classes. Our tool is not able to detect classes nor inheritance relationships implemented with this particular DSL. One idea to make JSClassFinder more robust against this kind of limitation is to gather the most common libraries that provide such service and implement specific strategies to identify them.

Use of singleton objects. Objects implemented directly, without using any class-like constructor functions, are not considered classes. Listing 2 shows the implementation of one singleton. Although this kind of object is not considered a class, it can be used to compose and clone other objects.

```
var person = {  firstName:"John",
        lastName:"Doe",
        birthDate: "01-01-2000",
        getAge: function () { ... }
      };
```

Listing 2. Singleton object example

Use of properties bound to variables that are functions. If a class constructor has a property that receives a variable, it is identified as an attribute, even if this variable holds a function. It occurs because JavaScript is a dynamic and loosely typed language, and JSClassFinder relies on static analysis.

5. Related Work

There is an increasing interest on JavaScript software engineering research. For example, JSNose is a tool for detecting code smells based on a combination of static and dynamic analysis [Fard and Mesbah 2013]. One of the code smells detected by JSNose, Refused Bequest, refers to subclasses that use only some of the methods and properties inherited from its parents. Differently, JSClassFinder provides models, visualizations, and metrics about the object-oriented portion of a JavaScript system, including inheritance relationships. Although JSClassFinder is not specifically designed for code smells detection, information provided by our tool can be used for this purpose.

Clematis [Alimadadi et al. 2014] and FireDetective [Zaidman et al. 2013] are tools for understanding event-based interactions, based on dynamic analysis. Their main goal is to reveal the control flow of events during the execution of JavaScript applications, and their interactions, in the form of behavioral models. In contrast, JSClassFinder aims the production of structural models.

ECMAScript definition, in its next version ES6 [ecm 2014], provides support for class definition. ES6 offers a proper syntax for creating classes and inheritance, similar to the syntax used in some traditional object-oriented languages, such as Java. Since ES6 uses specific keywords in the new syntax, it will be simple to adapt JSClassFinder once the new standard is released. Moreover, JSClassFinder could help on the migration of legacy JavaScript code to the new syntax supported by ECMAScript 6.

[9]https://github.com/alexpods/ClazzJS verified on 05/18/2015

6. Conclusions and Future Work

In this paper we proposed a reengineering tool that supports the identification of class-like structures and the creation of object-oriented models for JavaScript applications. The users do not need to have any prior knowledge about the structure of a system in order to build its model. It is possible to interact with the tool to obtain metric data and visual analysis about the object-oriented portion of JavaScript systems. As future work, we plan to extend JSClassFinder with three major features: (i) support for the new ECMAScript 6 standard, (ii) support to coupling information between classes, (iii) support to the computation of metric thresholds for JavaScript class-like structures [Oliveira et al. 2014].

JSClassFinder is publicly available at: **http://aserg.labsoft.dcc.ufmg.br/jsclasses/**.

Acknowledgment: This work was supported by FAPEMIG, CAPES and INRIA.

References

(2011). European association for standardizing information and communication systems (ECMA). ECMA-262: ECMAScript language specification. Edition 5.1.

(2014). European association for standardizing information and communication systems (ECMA). ECMAScript language specification, 6th edition, draft October.

Alimadadi, S., Sequeira, S., Mesbah, A., and Pattabiraman, K. (2014). Understanding JavaScript event-based interactions. In *International Conference on Software Engineering (ICSE)*, pages 367–377.

Crockford, D. (2008). *JavaScript: The Good Parts*. O'Reilly.

Ducasse, S., Gîrba, T., and Kuhn, A. (2006). Distribution map. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 203–212.

Fard, A. and Mesbah, A. (2013). JSNose: Detecting JavaScript code smells. In *13th Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Flanagan, D. (2011). *JavaScript: The Definitive Guide*. O'Reilly.

Gama, W., Alalfi, M., Cordy, J., and Dean, T. (2012). Normalizing object-oriented class styles in JavaScript. In *14th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 79–83.

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2010). The essence of JavaScript. In *24th European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150.

Oliveira, P., Valente, M. T., and Lima, F. (2014). Extracting relative thresholds for source code metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263.

Silva, L. H., Ramos, M., Valente, M. T., Bergel, A., and Anquetil, N. (2015). Does JavaScript software embrace classes? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 73–82.

Zaidman, A., Matthijssen, N., Storey, M. D., and rie van Deursen (2013). Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218.

SORTT - A Service Oriented Requirements Traceability Tool

Arthur Marques[1] **, Franklin Ramalho**[1] **, Wilkerson L. Andrade**[1]

[1] Federal University of Campina Grande – Campina Grande, Brazil
Systems and Computing Department

arthur.marques@ccc.ufcg.edu.br

{franklin,wilkerson}@computacao.ufcg.edu.br

Abstract. *Requirements traceability has been acknowledged as a valuable activity in the software development process. Its importance is reflected in different quality standards, which dictates that requirements should be traceable through the software development life-cycle. However, poor tool support is perhaps one of the biggest challenges to the implementation of traceability. COTS traceability tools only work if the project methodology is based around them. Also, such tools suffer problems with poor integration and flexibility. In order to address these issues, recent researches discuss how to identify common aspects of the traceability process and how to promote its portability. As a possible solution, traceability contracts have been proposed as means by which the traceability process' activities can exchange information. In light of this solution, we propose SORTT, a tool which integrates different activities of the requirements traceability process by providing services that comply with the established traceability contracts. SORTT's provided services can be (de)attached into the tool's core module according to organization's needs. Thus, the tool's purpose is to provide a flexible and integrated traceability environment.*

Tool's demonstration at `https://goo.gl/SJ1bHe`

1. Introduction

Requirements traceability (RT) refers to the ability to describe and follow the life of a requirement, throughout the software development life-cycle [Gotel and Finkelstein 1994]. In order to achieve traceability, a traceable environment must be established. Such environment is composed of procedures, methods, techniques and tools to accomplish the traceability process [Pinheiro 2004].

As a consequence of establishing a RT process, an organization may take benefit from its support in different areas, such as verifying whether a system complies with its requirements and that they have been implemented accordingly [Spanoudakis and Zisman 2004]; or determining what requirements, code, test cases, and other requirement related artifacts need to be updated to fulfill a change request [Egyed and Grunbacher 2005].

Regarding the traceability process, it can be decomposed into a traceability model with three major phases: the definition, production and extraction of trace links [Pinheiro 2004]. The *definition* elucidates what type of artifacts should be traced and how traces are represented; the *production* produces traces according to defined strategies/techniques; whereas, the *extraction* provides different and flexible ways to retrieve the produced

traces. Considering such model, one can have a holistic view of the traceability process and how its phases interoperate. It is important to emphasize that the traceability processes are organizational specific. Even though, requirement traceability COTS tools are usually generic and address overall traceability needs [Dömges and Pohl 1998]. For instance, despite being used in different contexts, the DOORS[1] tool input requirements and output artifacts need to be maintained within the tool environment, thus an organization's RT process needs to be based on the tool itself. If the organization traceability needs diverge from the ones provided by the tool, RT may turn into a cost, time-consuming, and error-prone activity [Ramesh 1998]. On the other hand, if the tool does not exploit existing traceability techniques and/or approaches, its costs, time, and effort will be equally high [Kannenberg and Saiedian 2009].

Identifying and standardizing key aspects of RT processes are among the grand challenges of requirements traceability [Gotel et al. 2012]. In such context, despite standardizing key aspects, customizing their own processes based upon the composition of building blocks and services are also key capabilities in order to accomplish organization's specific needs [Gotel et al. 2012]. Therefore, in this work we consider *(i)* a common trace link representation (TRL) [Marques et al. 2015b], and *(ii)* a generic traceability process [Marques et al. 2015a], and thus, we present SORTT – a Service Oriented Requirements Traceability Tool – as a tool which supports a requirement traceability process that addresses overall and specific needs.

SORTT considers TRL as an abstraction to traceable information and automates part of the activities of the requirements traceability process proposed by [Marques et al. 2015a]. Such process is based on the traceability model and it proposes traceability contracts as means by which process' phases/activities can exchange information. Hence, SORTT relies on the defined process contracts and provides a series of services, which can be integrated into a traceability environment. Though SORTT, one organization can use its existing services or (de)attach new ones in order to address its specific needs. Therefore, the tool can be configured such that it fits into the organization RT process.

2. Background

Considering the traceability process and its phases, different researches address key aspect of such process [Cleland-Huang et al. 2014, Marques et al. 2015a]. By defining key activities of the traceability process, which are common to different organizational environments, one can foster portable traceability and provide means by which different organizations can exchange traceable information [Gotel et al. 2012]. Nonetheless, there must be a common trace link definition, which represents a data structure containing necessary information which traverse through process' phases and activities.

Regarding the necessity of a common trace link definition, TRL has been proposed as a language to abstract traceable information and to represent trace links [Marques et al. 2015b]. Through language's constructions one can declare requirements, traced artifacts and their types, as well as the semantic of the trace link's relationships. Moreover, the language also supports the declaration of queries, hence providing mechanisms in order to search and retrieve trace links.

[1] `http://www-03.ibm.com/software/products/pt/ratidoor`

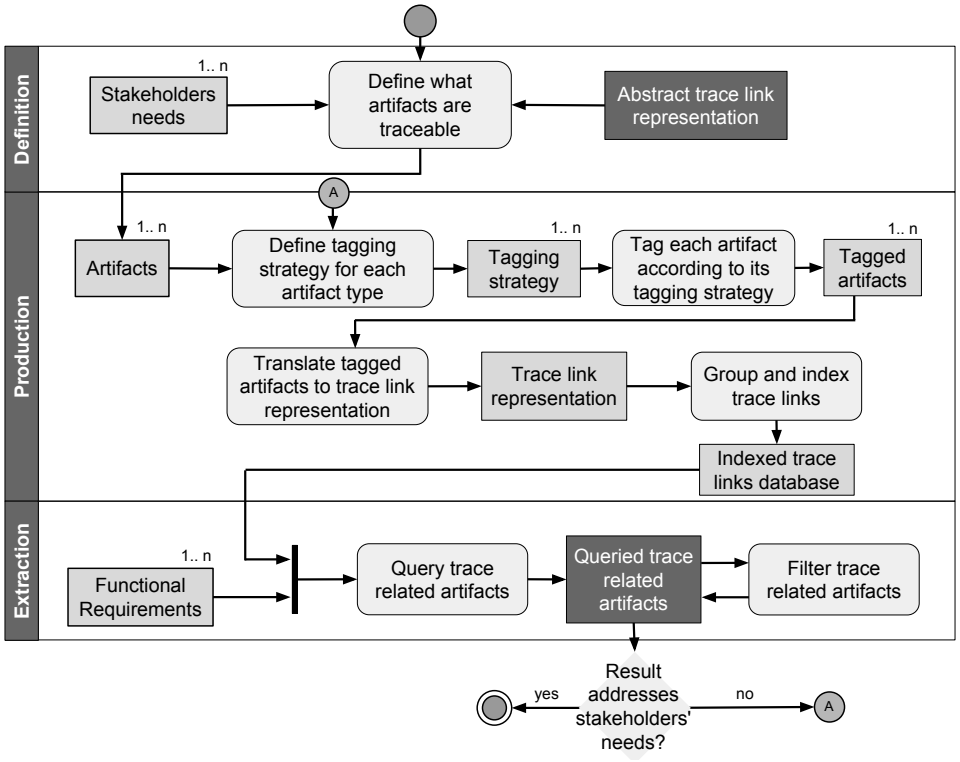


Figure 1. Traceability Process Workflow [Marques et al. 2015a]

In order to propose a reusable traceability process, TRL is exploited by a requirement traceability process, centered on the traceability model [Marques et al. 2015a]. The proposed process, summarized in Figure 1, encompasses the three phases of the traceability model. In the definition phase, the process considers the TRL language specification as an input and the set of artifacts to be traced is defined according to stakeholders' needs. Then, in the production phase, artifacts are analyzed and tagged. Hence, trace links are produced (according to language's constructions) and grouped for later query and also maintenance. Finally, in the extraction phase, requirements are queried (through language's queries) and requirement related artifacts are retrieved. Notice that process' activities and their cost are incorporated throughout the software development life-cycle. For instance, the production phase is related to both the design and the software development phases. Hence, in addition to normal activities of this phase, one should add an activity to tag, or relate artifacts to their requirements.

In light of the aforementioned traceability process, the process' reusability relies on traceability contracts. They provide interfaces to the traceability activities and establish means of communication through the different phases of the proposed process. Table 1 details the defined traceability contracts, its major services as well as service's pre and post-conditions. In summary, the contracts define services to the activities of *(1)* parsing, *(2)* translating, *(3)* indexing, *(4)* searching and *(5)* rendering trace links. The parsing, translating and indexing contracts consider the production phase of the traceability process. They provide interfaces to the activities of parsing and translating artifacts into the adopted trace link data structure as well as grouping and indexing them for later query and also maintenance. On the other hand, the searching and rendering contracts encompass

Table 1. Traceability Contracts

Parsing Contract	
parse	**pre:** Traceable artifacts are tagged according to the defined strategy **post:** All tagged artifacts are extracted as trace links
Translating Contract	
generate	**pre:** Trace links are extracted **post:** The set of parsed trace links is written according to the adopted trace link representation (TLR)
Indexing Contract	
index	**pre:** Storage system is available **post:** Trace links are grouped and indexed into the storage system
Searching Contract	
query	**pre:** Storage system is available, elements can be queried, queries can be described according to the adopted TLR **post:** Trace links that conform to query's parameters are retrieved
Rendering Contract	
render	**pre:** Existence of a data structure is available to represent trace links **post:** Trace links are represented according to the selected data structure

the extraction phase. They provide interfaces to the activities of querying and filtering trace links as well as rendering the search/filter output. More details of the traceability process and its contracts are presented in [Marques et al. 2015b].

3. SORTT

SORTT is a tool that automates part of the activities of the requirements traceability process proposed by [Marques et al. 2015a]. By providing services that comply with the requirements traceability contracts presented in Section 2, SORTT automates the activities of *(1)* producing trace links, *(2)* translating produced trace links, *(3)* indexing trace links, *(4)* searching and filtering trace links, and *(5)* rendering the extracted trace links.

SORTT relies on the fact that different organizations can exploit the services provided by the tool. If one or more services do not address some organization's needs, they can be detached from SORTT, then specific services can be integrated into the tool. To this extend, a core module interacts with other modules through their provided services. Therefore, one can trace requirement related artifacts through SORTT's functionalities.

SORTT's architecture, comprising modules and their required and offered services, is presented in Figure 2. SORTT's core module, the extractor, integrates the renderer module and the storage systems module as well as the translator module, which communicates with the parser module. Considering such architecture, the extractor is a facade between SORTT's provided services, hence delegating service's calls as well as providing service's input and receiving their output data.

Tool's traceability process is executed through interactions between its modules. First, the set of traceable artifacts is sent to the parser module, which will read the traceable artifacts and identify their related requirements. Then, artifacts and their related requirements, *i.e.* trace links, are the input of the translator module, which will produce them according to the adopted trace link data structure (TRL). Through the indexer module, the translator's output is grouped and indexed in the storage system. Additionally, translated trace links are represented into the extractor according to the TRL format, hence they can be visualized and manipulated. In order to do so, one must specify queries according to TRL's query constructions. Thus, the specified queries are parsed and sent to the querier module, which will execute them, according to query's parameters. Finally,

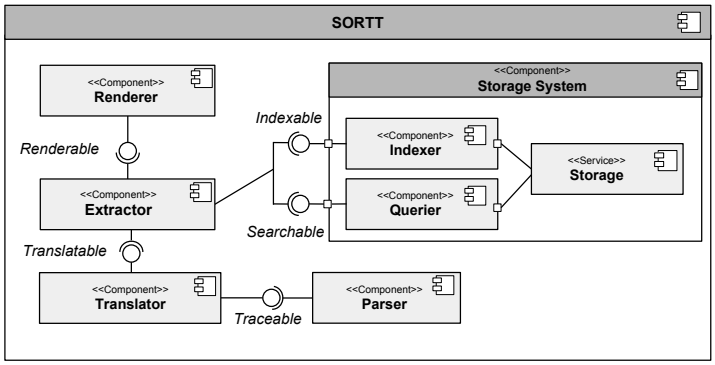


Figure 2. SORTT - Architecture Overview

the result of an executed query is returned to the extractor and the renderer module will display it according to a visualization data structure.

On its current JAVA implementation[2], SORTT has services to parse and translate trace links between requirements and test cases specified on TestLink as well as services to translate the trace links from benchmarks of the Center of Excellence for Software Traceability. Its underlying storage system is implemented using Apache Solr and, consequently, the indexer and querier services are implemented through its API. Finally, the renderer service is implemented using the JTree API, thus trace links are displayed in a set of hierarchical trees of nodes.

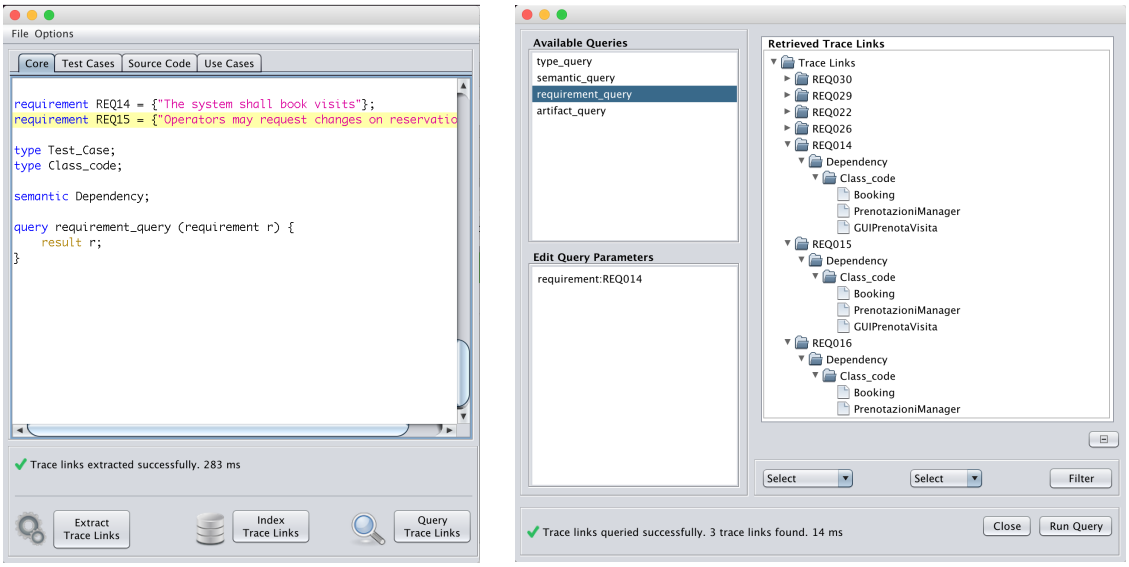
4. Case Study

SORTT has been successfully used in two empirical experiments. In the first experiment, the tool was used to help the comparison of an *ad hoc* process and the one proposed by [Marques et al. 2015a], whereas in the second experiment, SORTT was used for the comparison of TRL [Marques et al. 2015b] against two other traceability languages: TracQL [Tausch et al. 2012] and TQL [Maletic and Collard 2009]. The tool was a key factor in the experiments setup and execution. It allowed the configuration of different services to address each experiments' particularity. For instance, in order to compare TRL, TracQL and TQL, we customized SORTT's translator component to present the extracted trace links according to each language's constructions.

Regarding the conducted experiments, it is important to emphasize that we considered validated trace links from existing benchmarks. The projects and their traced artifacts were extracted from benchmarks provided by the CoEST[3]. Furthermore, one of the experiments also considered a industrial project under development for the Federal Police of Brazil, thus we also evaluated the tool considering real traceability needs. Since each project had different traceability needs, such as tracing client's requirements to developer's requirements or requirements to source code and test cases artifacts, we had to configure the tool and provide different services according to each project particularity. Furthermore, each project had different numbers of requirements, traced artifacts and trace links, thus we could evaluate tool's scalability, *i.e.* if it could support increasing

[2]Distributed under GPLv2 and available at `https://goo.gl/Q4mfFi`

[3]Available at `http://www.coest.org/index.php/resources/dat-sets`



A. Main Screen B. Query Screen

Figure 3. SORTT - Screenshots

number of requirements, artifacts, and so forth. Therefore, it was possible to evaluate the tool's usage in different contexts.

As a case study, we illustrate the tool's usage in the EasyClinic project (a medical ambulatory management application), considering its configuration for the comparison of TRL and TracQL languages. Set up procedures require that one must configure SORTT properties file according to its desired services. Hence, its components are packaged and then, it can be executed.

Figure 3 summarizes SORTT's three main functionalities, which are executed sequentialy: *(1)* producing the trace links; *(2)* indexing them; and also *(3)* querying requirement related artifacts. Notice that, in **A**, extracted and indexed trace links are represented according to the TRL format. Although, they could also be represented according to the TracQL format. To this extent, another translator module was implemented and attached to the tool during its packaging, providing thus flexibility to configure the tool according to each language and its trace link representation. As drawback, we must mention that the implementation of new modules was required. Hence, as more customizable components are needed, the cost and effort of adopting the tool may increase.

Once queries are declared, trace links may be extracted through the query menu, as presented in Figure 3 **B**. Once query's parameters are edited, a given query can be run through a request to the querier module. Then, the query's output is structured according to the renderer service. As an example, in Figure 3 **B**, trace links are displayed as a set of hierarchical tree of nodes. Though trace links may be displayed as a tree of nodes in either TRL or TracQL, one could present trace links in different manners by attaching new rendered modules.

5. Related Work

IBM Rational DOORS stands a major tool supporting the requirements traceability process. However, DOORS does not support adaptability to project-specific needs, *i.e.* a

traceability environment must be established centered on the tool and its functionalities, when in fact the tool should be used according to existing traceability processes of one organization [Kannenberg and Saiedian 2009].

DOORs is an example of a commercial off-the-shelf (COST) traceability tool and, as observed in [Dömges and Pohl 1998], it does not address adaptability to project-specific needs. In order to overcome this hindrance, it is necessary to provide tools that encompass both general needs as well as project-specific ones [Kannenberg and Saiedian 2009, Gotel et al. 2012]. In such context, SORTT is a vanguard tool. It considers common aspects that are likely to arise in the traceability process and automates them through its services. General needs are addressed by using tool's existing services, whereas specific needs by providing new ones.

Lastly, SORTT query mechanism could be compared to the ones of traceability query languages, such as TracQL [Tausch et al. 2012] or TQL [Maletic and Collard 2009]. However, these query languages do not provide a downloadable tool and, as a consequence, we did not have means by which we could compare them.

6. Conclusions

In this paper we have presented SORTT, a service oriented requirements traceability tool, which automates different activities of a proposed requirements traceability process. Considering common aspects that are likely to arise in a traceability process, SORTT relies on traceability contracts, which provide means of communication throughout process' phases and activities. Through the provided contracts and their implementing services, SORTT can address both general traceability needs or project-specific ones, *i.e.* existing traceability tools, techniques and approaches can (de)attached to the tool or new ones can be developed, according to project's needs. Nevertheless, it is important to highlight that a common trace link representation is still required as a data structure that must be shared among SORTT services (or the traceability contracts). Therefore, tool's adoption may be hindered by the lack of a standardized trace link representation [Gotel et al. 2012].

As future work, we plan to integrate more services that comply with the traceability contracts into SORTT as well as to identify means to integrate the tool with other industrial software development tools. As examples, it is possible to implement information retrieval approaches [Hayes et al. 2003] or rule based ones [Spanoudakis et al. 2004] as services for the parsing contract, or integrate the tool and its services in a continuous build approach. Furthermore, we plan to (1) evaluate the tool's usage in other industrial projects, thus we could identify which tool's services and functionalities are more exploited; and (2) evaluate it with requirements engineers and specialists, in order to gather feedback to improve SORTT.

Acknowledgments: This research is sponsored by the agreement Nº 754664/2010 between the Federal University of Campina Grande and the Federal Police Department. Third author is partially supported by the cooperation between UFCG and Ingenico do Brasil LTDA, stimulated by the Brazilian Informatics Law n. 8.248, 1991.

References

[Cleland-Huang et al. 2014] Cleland-Huang, J., Gotel, O. C. Z., Huffman Hayes, J., Mäder, P., and Zisman, A. (2014). Software traceability: Trends and future directions. In

Proceedings of the on Future of Software Engineering, FOSE 2014, pages 55–69, New York, NY, USA. ACM.

[Dömges and Pohl 1998] Dömges, R. and Pohl, K. (1998). Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62.

[Egyed and Grunbacher 2005] Egyed, A. and Grunbacher, P. (2005). Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):783–810.

[Gotel et al. 2012] Gotel, O., Cleland-Huang, J., Hayes, J., Zisman, A., Egyed, A., Grunbacher, P., Dekhtyar, A., Antoniol, G., and Maletic, J. (2012). The grand challenge of traceability. In *Software and Systems Traceability*. Springer.

[Gotel and Finkelstein 1994] Gotel, O. C. Z. and Finkelstein, A. C. W. (1994). An analysis of the requirements traceability problem. In *Proceedings of the 1st IEEE RE*.

[Hayes et al. 2003] Hayes, J., Dekhtyar, A., and Osborne, J. (2003). Improving requirements tracing via information retrieval. In *Proceedings of the 11th IEEE RE*, pages 138–147.

[Kannenberg and Saiedian 2009] Kannenberg, A. and Saiedian, H. (2009). Why software requirements traceability remains a challenge. *The Journal of Defense Software Engineering*.

[Maletic and Collard 2009] Maletic, J. and Collard, M. (2009). Tql: A query language to support traceability. In *5th ICSE Workshop on TEFSE*.

[Marques et al. 2015a] Marques, A., Ramalho, F., and Andrade, W. L. (2015a). Towards a requirements traceability process centered on the traceability model. In *Proceedings of the 30th ACM/SIGAPP Symposium On Applied Computing*, New York, NY, USA. ACM.

[Marques et al. 2015b] Marques, A., Ramalho, F., and Andrade, W. L. (2015b). TRL – a traceability representation language. In *Proceedings of the 30th ACM/SIGAPP Symposium On Applied Computing*, New York, NY, USA. ACM.

[Pinheiro 2004] Pinheiro, F. A. (2004). Requirements traceability. In Prado Leite, J. C. S. and Doorn, J. H., editors, *Perspectives on Software Requirements*, volume 753 of *The Springer International Series in Engineering and Computer Science*, pages 91–113. Springer US.

[Ramesh 1998] Ramesh, B. (1998). Factors influencing requirements traceability practice. *Commun. ACM*.

[Spanoudakis and Zisman 2004] Spanoudakis, G. and Zisman, A. (2004). Software traceability: A roadmap. In *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing.

[Spanoudakis et al. 2004] Spanoudakis, G., Zisman, A., Pérez-Minana, E., and Krause, P. (2004). Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105 – 127.

[Tausch et al. 2012] Tausch, N., Philippsen, M., and Adersberger, J. (2012). Tracql: A domain-specific language for traceability analysis. In *10th Working IEEE WICSA Conference*.

Insight: uma ferramenta para análise qualitativa de dados apoiada por mineração de texto e visualização de informações

Rafael Gastaldi[1], Cleiton Silva[1], André Di Thommazo[2], Elis Hernandes[2], Denis Bittencourt[2], Sandra Fabbri[1]

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

[2]Instituto Federal de Educação, Ciência e Tecnologia do Estado de São Paulo (IFSP)
São Carlos – SP – Brasil

{rafael.gastaldi, cleiton.silva, sfabbri}@dc.ufscar.br, {andredt, elis.hernandes, denis.bittencourt}@ifsp.edu.br

Resumo. *A engenharia de software baseada em evidência envolve a análise de dados quantitativos e qualitativos. Na análise dos dados qualitativos, em que se aplica, geralmente, a técnica Coding, apesar de algumas ferramentas de apoio, há a dificuldade de manuseá-los de forma conjunta para que a análise seja facilitada e promova melhor consistência. Assim, o objetivo deste artigo é mostrar a ferramenta Insight, que dá suporte ao Coding com o uso de visualização e mineração de texto. A metáfora visual permite enxergar todo o conjunto de dados na tela, os quais podem ser arranjados conforme a conveniência do pesquisador e a mineração de texto permite encontrar semelhanças entre os dados, de diferentes formas, as quais são retratadas na metáfora visual. Os resultados do uso da Insight têm mostrado que os recursos usados na ferramenta ajudam na análise dos dados qualitativos, facilitando a extração das informações necessárias.*

https://www.youtube.com/watch?v=ixCXf6hN0g0

1. Introdução

Durante o processo de desenvolvimento de software é comum o registro de ideias e dados na forma de texto. Isso ocorre em diversas fases do processo de desenvolvimento, nos artefatos produzidos, como documento de requisitos, documentação de código fonte, artefatos de testes, reuniões de inspeção, avaliações qualitativas feitas por usuários do software, dentre outros. Como benefício ao registro em formato texto destaca-se, principalmente, a riqueza dos dados qualitativos. Por outro lado é difícil a padronização dos dados, podendo ocorrer inconsistências no entendimento da informação extraída. Para evitar este entendimento equivocado, o analista deve ser cuidadoso na condução da análise dos dados qualitativos a fim de extrair informações relevantes e consistentes.

A análise de dados qualitativos é relevante e leva a informações não quantificáveis. Porém, encontrar padrões de conteúdo e transformar os dados qualitativos em informação para tomada de decisão não é uma tarefa simples. Uma das técnicas utilizadas para isso é a técnica *Coding* (Hancock, Ockleford & Windridge, 1998), que

consiste em extrair trechos relevantes dos dados, categorizá-los e correlacioná-los. De acordo com Seaman (2008), a técnica é estruturada em três etapas: (i) *Open Coding*: o material deve ser lido a fim de identificar trechos relevantes (*quotations*), aos quais são atribuídos rótulos (denominados *codes* ou códigos). Os rótulos são utilizados para agrupar dados que possuem enfoque similar e que, posteriormente, serão correlacionados para dar origem à informação em si; (ii) *Axial coding:* os códigos são agrupados em categorias que promovam melhor entendimento dos mesmos; (iii) *Selective Coding:* os códigos e categorias obtidas devem ser analisados novamente, verificando relações entre eles a fim de sintetizar os dados e informações relevantes.

Dada a importância da análise qualitativa e da quantidade de dados que, em geral, devem ser processados, o suporte de ferramentas computacionais é essencial. Apesar de existirem algumas ferramentas para isso, um dos problemas persistentes é que não se consegue ter uma visão de todos os documentos (que contêm os dados qualitativos) ao mesmo tempo (na mesma tela). Isso dificulta a interpretação dos dados pois, pelo fato de serem em grande quantidade, em geral, não se consegue processá-los no mesmo momento. Dessa forma, este artigo tem por objetivo apresentar a ferramenta Insight, cujo diferencial em relação às outras, está no uso de visualização de informações e mineração de texto, para ajudar na extração das informações relevantes.

Os recursos de visualização e mineração de texto, quando utilizados em conjunto, possibilitam a análise de múltiplos documentos simultaneamente. A mineração de texto abstrai os dados em informações que possam ser relevantes nos documentos e, com auxílio da visualização, esses dados são salientados na tela para que o analista possa processá-los. Isso facilita a padronização no tratamento de dados qualitativos.

A Insight pode dar suporte em vários momentos do desenvolvimento de software, como para verificar documentos de requisitos a fim de identificar dependências entre eles, processar listas de defeitos geradas por inspetores após uma atividade de inspeção, identificar relacionamentos entre defeitos de software, entre outros. Além disso, também dá suporte à aplicação da análise temática (Boyatzis, 1998) no contexto de estudos secundários (Kitcheham, 2004). Neste último caso a Insight está integrada à *StArt* (Fabbri et al, 2012), ferramenta que apoia todo o processo de revisão sistemática, possibilitando a realização de análise temática na etapa de extração e sumarização dos dados.

Este artigo está organizado da seguinte maneira: a Seção 2 apresenta as principais funcionalidades da ferramenta e o processo utilizado para análise, com destaque para a visualização e mineração de texto. Na Seção 3 apresentam-se aspectos de implementação da ferramenta. A Seção 4 comenta outras ferramentas voltadas para análise qualitativa de dados. Por fim, a Seção 5 apresenta a conclusão deste artigo.

2. Aspectos funcionais da ferramenta Insight

A Insight é uma ferramenta para análise de dados qualitativos que auxilia o pesquisador na extração de dados e sumarização de informações. Para realização dessa análise destacam-se as seguintes funcionalidades:

- Suporte a dados textuais e planilhas Microsoft Excel. Deste modo, possibilita integração com diversas ferramentas, entre elas ferramentas de geração de casos

de teste, rastreabilidade de requisitos, detecção de defeitos e outras; basta exportarem os dados nos formatos aceitos pela *Insight*.

- Extração dos dados com auxílio da técnica *Coding*.
- Integração com a ferramenta *StArt*, de apoio a revisões sistemáticas, apoiando a técnica de análise temática em revisões sistemáticas.
- Visualização dos dados para abstrair informações relevantes; e hierarquização e agrupamento de documentos com base em um ou mais atributos escolhidos pelo usuário. Esta funcionalidade será detalhada na seção 2.2.
- Mineração de texto por meio de algoritmos de processamento de texto sensíveis ao contexto. Esta funcionalidade será detalhada na seção 2.3.
- Análise de múltiplos documentos simultaneamente - a visualização dos dados em conjunto com a mineração de texto, permite realizar a análise de múltiplos documentos simultaneamente, adaptando as visualizações ao contexto tratado e identificando trechos similares por meio dos algoritmos de mineração de texto.
- Exportação das informações extraídas com suporte da *Insight*, para importação em software de geração de mapas mentais, como Mindmeister, e dos relatórios da análise realizada e informações obtidas em planilha do Microsoft Excel.

A Figura 1 ilustra a tela principal da ferramenta com os principais componentes rotulados para destaque. O rótulo A representa a guia de navegação entre as etapas da análise. O rótulo B corresponde à área de codificação e análise do documento corrente. O rótulo C corresponde à área de visualização de informações na qual são representados todos os documentos de interesse, sendo que cada quadrado corresponde a um documento. O rótulo D representa a área em que podem ser aplicados os filtros, controle de hierarquias, legendas e demais opções da ferramenta.

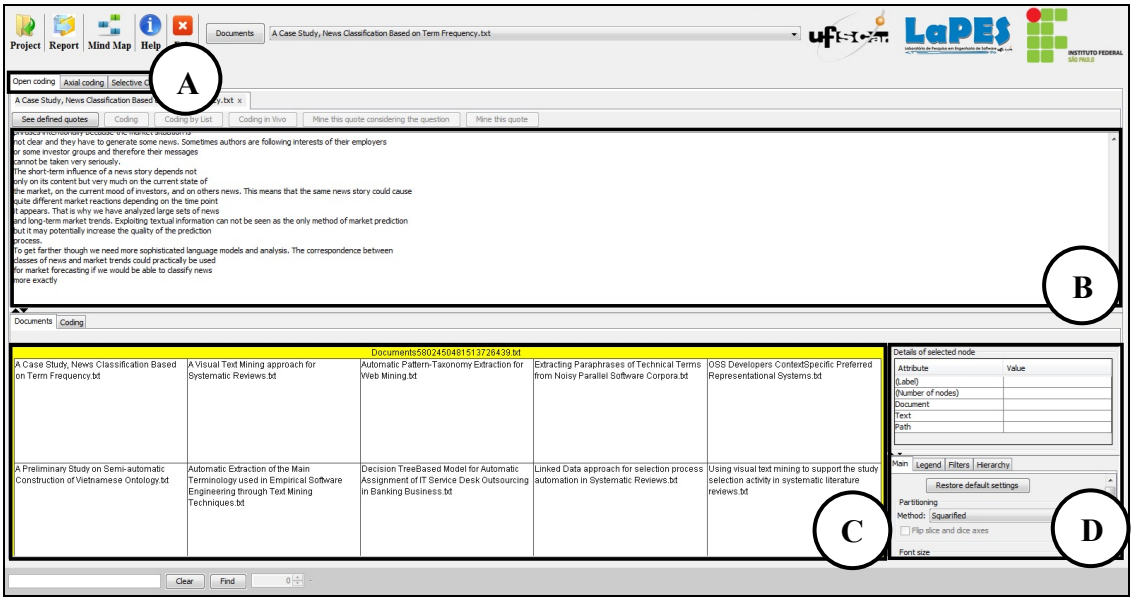


Figura 1 - Interface principal da ferramenta Insight

2.1. Funcionalidades relativas à análise de dados na *Insight*

As etapas propostas para a análise consistem da aplicação da técnica *Coding* apoiada pela utilização da metáfora visual e mineração de texto para abstração das informações. Os seguintes passos são aplicados:

1) Importação dos dados para análise: Os dados devem ser importados na ferramenta, sendo aceitos os formatos texto e planilhas Microsoft Excel.
2) Agrupamento e categorização dos dados: O usuário pode aplicar diferentes visualizações para abstrair informações dos dados em conjunto com os recursos de mineração de texto. Tais funcionalidades, detalhadas nas Seções 2.2 e 2.3, possibilitam a análise de múltiplos documentos simultaneamente.
3) Extração de informações: Nessa etapa o analista deve analisar os dados a fim de gerar informação relevante com auxílio da técnica *Coding*.
4) Conclusão: O analista pode utilizar o editor de texto da ferramenta para sintetizar o tópico de interesse ou exportar relatórios com as informações extraídas e um arquivo para geração de mapa mental.

Apesar de ser estabelecido um fluxo básico composto por etapas, a análise realizada com a ferramenta é iterativa, ou seja, o usuário executa as etapas descritas em conjunto.

2.2. Utilizando visualização na análise de dados

Um dos diferenciais da *Insight* que auxilia o pesquisador a abstrair informações dos dados textuais é a metáfora visual. Visualização é o processo de transformar dados, informações e conhecimento em forma visual, fazendo uso da capacidade visual natural dos seres humanos e fornecendo uma interface entre dois sistemas de tratamento da informação: a mente humana e o computador (Gershon et al, 1998). Existem diversas formas de visualização que podem auxiliar a interpretação: mapas mentais, gráficos, fluxogramas, figuras, entre outros. Como a ferramenta *Insight* reutiliza a visualização da ferramenta Treemap (Bederson et al, 2002), desenvolvida na Universidade de Maryland, a forma de visualização disponível ao usuário é a *tree-map*. A visualização das informações na ferramenta permite, por exemplo:

- Classificar e agrupar os documentos em hierarquias por meio da seleção de um atributo que consta nos documentos a serem analisados (exemplo: tipo do requisito, código do erro, ferramenta utilizada, entre outros). Uma vez selecionado o atributo, a ferramenta agrupa os documentos que tratam de informações similares tornando mais simples a recuperação de dados sobre um mesmo tema e comparação em relação ao todo.
- Filtrar os dados para selecionar e destacar dados de interesse.

Para exemplificar a visualização considere, por exemplo, a atividade de análise das listas de defeitos provenientes de uma atividade de inspeção, conforme utilizado por Hernandes (2012). Listas de defeitos foram importadas na *Insight* para efetuar a comparação e análise dos defeitos da lista original (oráculo) com os defeitos relatados pelos inspetores. Na Figura 2, cada caixa corresponde a um defeito com sua descrição e cada tom de cor corresponde a um inspetor, conforme legenda à direita da imagem. Agrupando-se os defeitos por linha de código e aplicando-se a opção de hierarquia da

ferramenta, as regiões A e B (com borda grossa na cor preta) da Figura 2 destacam dois desses agrupamentos. Dessa forma, é possível observar duas situações nessas regiões:

A) Três inspetores relataram defeitos em uma mesma linha de código na qual também há um defeito relatado no oráculo;

B) Oito inspetores, conforme legenda do lado direito da Figura 3, relataram defeitos em uma mesma linha de código, sendo que na lista original não há defeito relatado nessa linha. Dessa forma, esses relatos correspondem a falso-positivos ou a defeitos novos que não existiam no oráculo.

Figura 2 - Análise de defeitos provenientes de atividade de inspeção com a ferramenta Insight

2.3. Mineração de texto e sua aplicação

Outro importante recurso para auxiliar a análise de dados textuais é a mineração de texto que, em conjunto com a visualização, permite ao usuário realizar a análise de múltiplos documentos simultaneamente. A mineração de texto é uma subárea da mineração de dados, amplamente utilizada e pesquisada para diversos fins. Pode ser definida como um processo de conhecimento intensivo em que o usuário interage com uma coleção de documentos por meio de um conjunto de ferramentas de análise (Feldman & Sanger, 2007).

A mineração de texto na Insight permite que, a partir de um trecho de texto selecionado, identifique-se a incidência desse trecho em todos os documentos, por meio do algoritmo da janela deslizante (Hernandes, 2014). Para isso é calculado o índice de similaridade desse trecho entre os diversos documentos, classificando-os em faixas de valores de 0 a 100, conforme legenda na parte inferior da Figura 3. O índice de similaridade é calculado pelo método do vetor de frequências e similaridade do cosseno (Salton & Allan, 1994) e é sensível ao contexto (atualmente a ferramenta dá suporte aos idiomas português e inglês).

Identificar que uma expressão de interesse ocorre em vários documentos pode mostrar que o tópico é recorrente e, portanto, deve ser analisado com atenção. Caso ocorra a situação oposta, deve-se analisar o motivo de tal: pode tanto ser uma informação a ser desconsiderada, pois se pretende analisar o contexto do todo e não aspectos individuais. Também se pode considerá-la digna de análise cuidadosa, pois levanta um tópico único, não encontrado em outros documentos. Por exemplo, no desenvolvimento de aplicativos móveis pode ser relatado mau funcionamento de um recurso - a falha no recurso pode estar diretamente relacionada à versão do sistema operacional do aparelho. Com auxílio da mineração de texto é possível verificar se aquele problema ocorre também em outras versões do sistema operacional, se o problema é recorrente nas outras avaliações e, até mesmo, se é um problema ocasionado por mau uso do usuário.

No contexto de análise temática em revisões sistemáticas a mineração de texto também é importante. A Insight trabalha em conjunto com a StArt permitindo a utilização da técnica de análise temática, auxiliando na extração de dados e sumarização dos estudos primários. Nesse processo, o planejamento, a aquisição dos estudos primários e a seleção inicial dos estudos são apoiados pela StArt. Após a seleção inicial, os dados são exportados para a Insight em um arquivo compactado com todo o material a ser analisado, permitindo a extração e sumarização dos dados. A Figura 3 corresponde a uma revisão sistemática utilizando a análise temática para extração dos dados. Cada caixa da área de visualização corresponde a um estudo primário em análise.

Parte relevante para destaque na Figura 3 é a categorização inicial gerada. Uma vez definido o tópico de interesse, calculou-se o índice de similaridade da expressão para categorizar os estudos primários pela similaridade com o tópico tratado (conforme legenda na parte inferior da ferramenta).

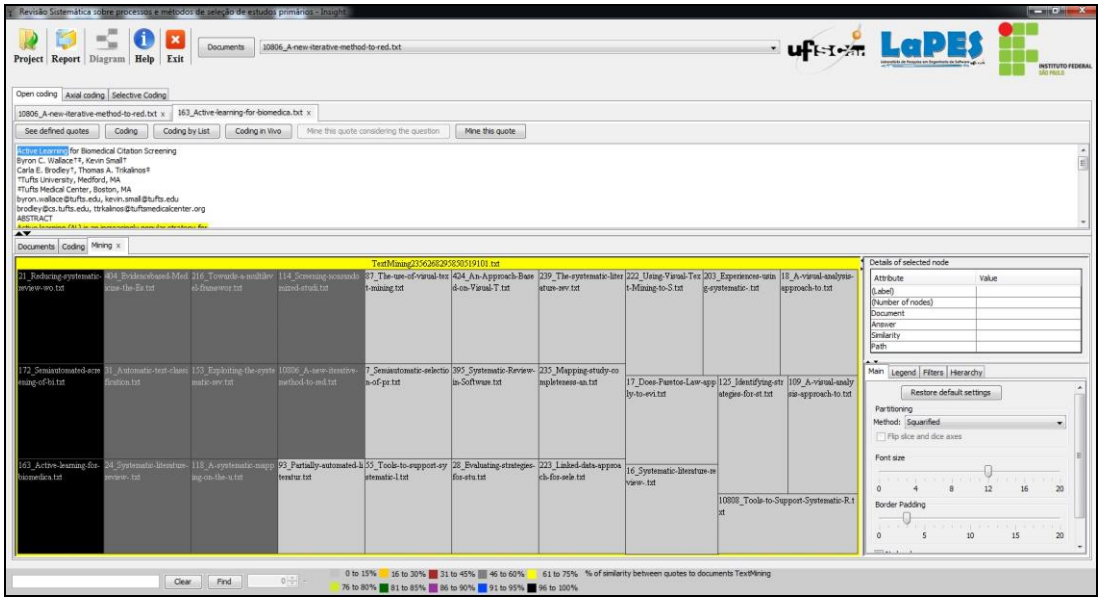


Figura 3 - Classificação baseada na similaridade da expressão selecionada

3. Aspectos de implementação

A ferramenta Insight foi desenvolvida utilizando a linguagem Java devido à portabilidade entre sistemas operacionais. Para lidar com dados advindos de outras

ferramentas, importa dados no formato texto ou planilha Microsoft Excel, tornando-a flexível para ser utilizada em conjunto com outras ferramentas. A interface gráfica é baseada na biblioteca Swing do Java. A arquitetura do sistema está baseada em práticas recomendadas do *Domain Driven Design*, além de fazer uso do padrão *Model-View-Controller* (MVC) para estruturação da comunicação da camada de interface de usuário (UI) com a camada de aplicação e demais componentes do sistema. A persistência é feita com *Java Persistence API* (JPA) utilizando o banco de dados *Hyper Structured Query Language Database* (HSQLDB) em modo arquivo. O modo arquivo do HSQLDB foi escolhido pela flexibilidade para armazenamento de informações de um projeto Insight, isto é, o formato de arquivo de projeto da ferramenta é baseado no formato arquivo do HSQLDB.

Outros aspectos relevantes: a implementação da Treemap, feita com a integração de um módulo (aplicativo) fornecido pela universidade de Maryland ao LaPES, e o uso de uma biblioteca desenvolvida internamente no LaPES para reconhecimento de similaridade de palavras nos idiomas português e inglês. Outros aspectos com menor relevância: biblioteca Apache POI para interpretar planilha de dados como entrada de dados para um projeto Insight, biblioteca Jasper Reports para geração de relatórios sobre análise de dados, entre outros.

4. Ferramentas relacionadas

Diversas áreas do conhecimento realizam análise qualitativa de dados. Algumas ferramentas computacionais que oferecem apoio a esse tipo de análise são Nvivo[1], desenvolvida pela QSR International, Atlas.ti[2], desenvolvida pela Universidade de Berlin, e HyperResearch[3], software proprietário da ResearchWare. Entre os recursos que disponibilizam estão as opções de trabalhar com textos, imagens e vídeos, bem como inserir anotações nos documentos em análise. Essas ferramentas, assim como a *Insight*, utilizam a técnica *Coding* como base para extração dos dados e recuperação das informações. Um dos diferenciais da *Insight* em relação a elas, além de ser gratuita, são os recursos de mineração de texto utilizando, em particular, o algoritmo da janela deslizante, em conjunto com visualização de informações por meio da metáfora *tree-map*. Esses recursos possibilitam a análise de múltiplos documentos simultaneamente, facilitando a abstração das informações.

5. Conclusão

Este artigo apresentou a ferramenta Insight, que tem por objetivo apoiar a análise de dados qualitativos utilizando mineração de texto e visualização para abstrair e extrair informações. Com os estudos conduzidos, verificamos que as técnicas e recursos disponíveis são importantes para auxiliar o analista na análise de dados oriundos de diversas atividades frequentemente empregadas na engenharia de software. Além das aplicações nessa área, a ferramenta é utilizada em outras áreas do conhecimento. Pesquisadores de áreas como educação, medicina, enfermagem e ciência política – áreas

[1] https://www.qsrinternational.com/

[2] http://atlasti.com/

[3] http://www.researchware.com/products/hyperresearch.html

onde a análise de dados qualitativos é prática frequente e consolidada - também realizaram estudos com a ferramenta e forneceram avaliações positivas quanto aos recursos disponíveis para auxiliar a análise. Demais dados a respeito da validação e estudos conduzidos com a ferramenta estão descritos em Hernandes, 2014.

Agradecimentos

Agradecemos ao CNPq e à CAPES pelo apoio financeiro.

Referências

Bederson, B.B., Shneiderman, B., and Wattenberg, M. Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies ACM Transactions on Graphics (TOG), 21, (4), October 2002, 833-854.

Boyatzis, R. E. Transforming qualitative information: thematic analysis and code development. London: Sage, 1998.

Coleman, G.; O'Connor, R. Using grounded theory to understand software process improvement: a study of Irish software product companies. Information and Software Technology, v. 49, p. 654–667, 2007.

Fabbri, S., Hernandes, E., Di Thommazo, A., Belgamo, A., Zamboni, A., Silva, C. (2012). Managing literature reviews information through visualization. In International Conference on Enterprise Information Systems.14th. ICEIS, Wroclaw, Poland, Jun 2012. Lisbon: SCITEPRESS.

Feldman, R., Sanger, J. The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge: Cambridge University Press, 2007. 387p.

Gershon, N., Eick, S.G., Card, S., Information visualization interactions. ACM Interactions, Nova Iorque, v. 5, n. 2, p. 9-15, Mar/Apr. 1998.

Hancock, B.; Ockleford, E.; Windridge, K. An introduction to qualitative research. Nothingham: Trent focus group Nottingham, 1998.

Hernandes, E.M., Teodoro, E., Thommazo, A.D., Fabbri, S.C.P.F.; Using Visualization and Text Mining to Improve Qualitative Analysis. In ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems, Volume 2, Lisbon, Portugal, 27-30 Abril, 2014.

Hernandes, E.C.M. Abordagem orientada à informação para análise qualitativa com suporte de visualização e mineração de texto. 2014. 180f. Tese (Doutorado em Ciência da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Paulo. 2014.

Kitcheham, B.A.; Procedures for Performing Systematic Reviews; Keele University Joint Technical Report TR/SE-0401; ISSN: 1353-7776 and National ICT Australia Ltd. NICTA Technical Report 0400011T.1 Jul. 2004.

Salton, G., Allan, J. 1994. Text Retrieval Using the Vector Processing Model, In 3rd Symposium on Document Analysis and Information Retrieval, Mar/1994. Las Vegas.

Seaman, C. Qualitative methods. In:SHULL, F.;SINGER, J.; SJøBERG, D. (Ed.). Guide to Advanced Empirical Software Engineering. Springer London,2008. p.35–62.