PerfMiner Visualizer: uma ferramenta para análise da evolução do atributo de qualidade de desempenho em sistemas de software

Leo Silva[1,2], **Felipe Pinto**[2], **Uirá Kulesza**[1], **Christoph Treude**[3]

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

[2]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte (IFRN)
Natal – RN – Brasil

[3]School of Computer Science
The University of Adelaide (Australia)

{leo.silva,felipe.pinto}@ifrn.edu.br, uira@dimap.ufrn.br, ctreude@gmail.com

Abstract. *Each application present in society has a base architecture, regardless of its complexity. The lack of monitoring the evolution of this architecture can lead to the degradation of system quality attributes. This paper presents PerfMiner Visualizer, a web tool that uses software visualization techniques to help developers and architects to analyze the evolution of the performance quality attribute, in terms of execution time, for different software versions. In this paper, the tool is shown through its application in two real systems. The most degraded scenarios and the possible causes of performance deviations can be identified through the tool.*

Resumo. *Cada aplicação presente na sociedade possui uma arquitetura base, independente da sua complexidade. A falta de acompanhamento da evolução dessa arquitetura pode levar a sua degradação, em especial dos atributos de qualidade. Este trabalho apresenta PerfMiner Visualizer, uma ferramenta web que lança mão de técnicas de visualização de software para ajudar os desenvolvedores e arquitetos a analisar a evolução do atributo de qualidade de desempenho, em termos de tempo de execução, para as versões de um software. Neste artigo, a ferramenta é mostrada através da sua aplicação em dois sistemas reais. Os cenários mais degradados e as possíveis causas de desvios de desempenho podem ser identificadas através das visualizações exibidas.*

Video. *https://youtu.be/5bU-pXn6ZKo*

1. Introdução

Inevitavelmente, os *softwares* mudam para atender a novos requisitos, adicionar novas tecnologias ou para reparar erros [Diehl 2007]. A evolução de um software é uma tarefa complexa, pois o processo acaba gerando grande quantidade de dados. Mudanças no código-fonte podem causar comportamentos inesperados em tempo de execução como,

por exemplo, o desempenho de partes da aplicação podem ser degradados em uma nova versão em comparação a anterior [Sandoval Alcocer et al. 2013].

Caserta e Zendra [Caserta and Zendra 2011] mencionam que um software se torna complexo quando o seu tamanho aumenta, tornando difícil o seu entendimento, manutenção e evolução. Nesse sentido, sem acompanhamento, os atributos de qualidade, como o desempenho, inicialmente definidos a partir de decisões arquiteturais e de design tomadas durante o processo de desenvolvimento podem deixar de ser bem atendidos.

A visualização de software é uma área da engenharia de software que visa usar representações visuais para melhorar o entendimento de diferentes aspectos de um sistema. Alguns destes incluem arquitetura, processo de desenvolvimento e histórico de código-fonte [Ghanam and Carpendale 2008]. Diehl [Diehl 2007] cita que o objetivo principal é ajudar a compreender sistemas de software e aumentar a produtividade do processo de desenvolvimento. Com relação a visualização de arquiteturas, a adição da variável tempo torna a tarefa mais difícil, passando a ser necessária a representação da sua evolução.

Este trabalho apresenta uma ferramenta, chamada *PerfMiner Visualizer* (PMV), cujo objetivo é aplicar técnicas de visualização de software para ajudar desenvolvedores e arquitetos a analisar a evolução do atributo de qualidade de desempenho, em termos de tempo de execução, ao longo das versões de um software. A ferramenta propõe duas visualizações com escopos e granularidades diferentes e proporciona ao usuário mecanismos de interação para explorá-las. Para avaliá-la, a ferramenta foi aplicada em dois sistemas reais de diferentes domínios: o Jetty e o VRaptor.

O restante deste artigo está organizado como segue: a seção 2 apresenta a ferramenta PMV, destacando as suas principais funcionalidades. A seção 3 mostra a avaliação realizada. Já a seção 4 discute os trabalhos relacionados e a seção 5 conclui este artigo.

2. PerfMiner Visualizer

O *PerfMiner Visualizer* (PMV) é uma ferramenta *web* desenvolvida como uma extensão ao *PerfMiner* (PM) [Pinto et al. 2015]. Ela dispõe de duas visualizações para ajudar desenvolvedores e arquitetos a acompanhar a evolução do atributo de qualidade de desempenho ao longo das versões de um software. Esta seção introduz o PM, a arquitetura da ferramenta proposta, os detalhes das visualizações e apresenta aspectos de implementação.

2.1. PerfMiner

A ferramenta PM provê a avaliação contínua de cenários no tocante a análise de desvios de desempenho, de forma a minimizar a degradação de desempenho em cenários arquiteturalmente relevantes [Pinto et al. 2015]. A sua principal funcionalidade é realizar a análise de desvio - otimizações ou degradações - de desempenho entre duas versões de um sistema, revelando, de maneira automatizada, as potenciais causas para o desvio nos cenários. Neste trabalho, um cenário é definido como um caso de teste automatizado do sistema.

Os seguintes passos resumem o funcionamento do PM [Pinto et al. 2015]: (i) *análise dinâmica*: executa os cenários através dos testes automatizados e armazena informações sobre os *traces* de execução do sistema; (ii) *análise de desvio*: compara, para duas

versões do sistema, os dados extraídos no passo anterior resultando em um artefato com os métodos e cenários com desvios de desempenho; e (iii) *mineração de repositório*: minera os repositórios de controle de versões e tarefas em busca de *commits* e tarefas que alteraram os métodos identificados na fase anterior. Após a execução do PM, são gerados os seguintes artefatos de saída: (i) arquivos de texto contendo os cenários e métodos com desvios de desempenho e (ii) modelos de análise dinâmica com informações sobre os *traces* de execução do sistema, salvos em um banco de dados relacional.

2.2. Visão Geral do PerfMiner Visualizer

A ferramenta PMV utiliza os artefatos de saída do PM para gerar as visualizações. A Figura 1 ilustra o fluxo de informações para o funcionamento da ferramenta PMV.

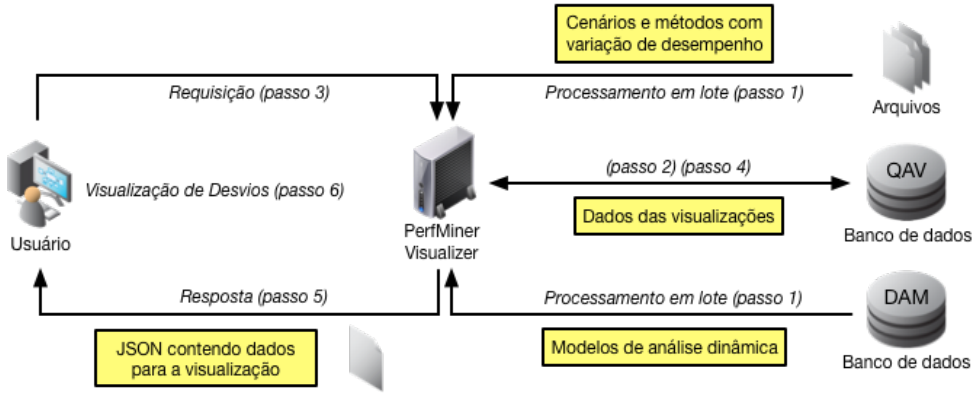

Figura 1. Funcionamento do *PerfMiner Visualizer*.

Antes do usuário acessar as visualizações, é realizado um processamento em lote para que os dados provenientes do PM possam ser computados e, então, são gerados os dados que dão suporte às visualizações. De posse dos artefatos de saída do PM mencionados anteriormente, é realizado um upload dos arquivos de texto, para ambas as versões, para que o processamento em lote (passo 1) se inicie. Depois disso, a ferramenta PMV recupera os modelos de análise dinâmica do banco de dados DAM (*Dynamic Analysis Model*) para as versões desejadas e o processamento continua até que os dados das visualizações sejam determinados. Tais dados resultantes são gravados no banco de dados QAV (*Quality Attribute Visualization*) no passo 2. O processamento em lote é executado apenas uma vez para duas versões de um sistema, sendo necessário para gerar os dados que possibilitam ao usuário um rápido tempo de resposta na requisição das visualizações.

Após esse processamento, os usuários podem realizar as requisições (passo 3) de acesso às visualizações de desvio de desempenho oferecidas pela PMV. No passo 4, os dados são obtidos do banco de dados QAV e, então, retornados em formato JSON (*JavaScript Object Notation*) para o usuário na resposta (passo 5). Por fim é realizado um processamento no *browser* do usuário utilizando os dados recebidos pelo arquivo JSON (passo 6). Esse processamento monta a visualização e a exibe para o usuário.

Embora seja possível realizar a análise para versões não sequenciais, isso pode levar a visualizações grandes e complexas por existirem, possivelmente, muitas mudanças e, consequentemente, desvios de desempenho. O ideal é que a análise seja feita em versões subsequentes para se beneficiar da baixa granularidade apresentada pela ferramenta, em especial o grafo de chamadas.

2.3. Visualização da Sumarização de Cenários

Esta visualização sumariza os cenários com desvios de desempenho entre duas versões do sistema. Para tanto, é utilizada uma variação do gráfico de rosca. No gráfico, a rosca é composta de todos os cenários com desvios para as versões analisadas, e cada fatia representa um cenário com suas características.

A Figura 2-B exemplifica essa visualização. No gráfico de rosca exibido, estão dispostos 5 cenários: na cor marrom claro, podem ser vistos 4 cenários de degradação, e na cor verde 1 cenário de otimização. Complementando as informações textuais destacadas em azul na figura: (i) quanto mais larga a fatia, maior foi o desvio de desempenho; (ii) quanto mais alta a área preenchida da fatia, maior o tempo de execução; e (iii) a otimização ou degradação se dá da versão atual para a anterior.

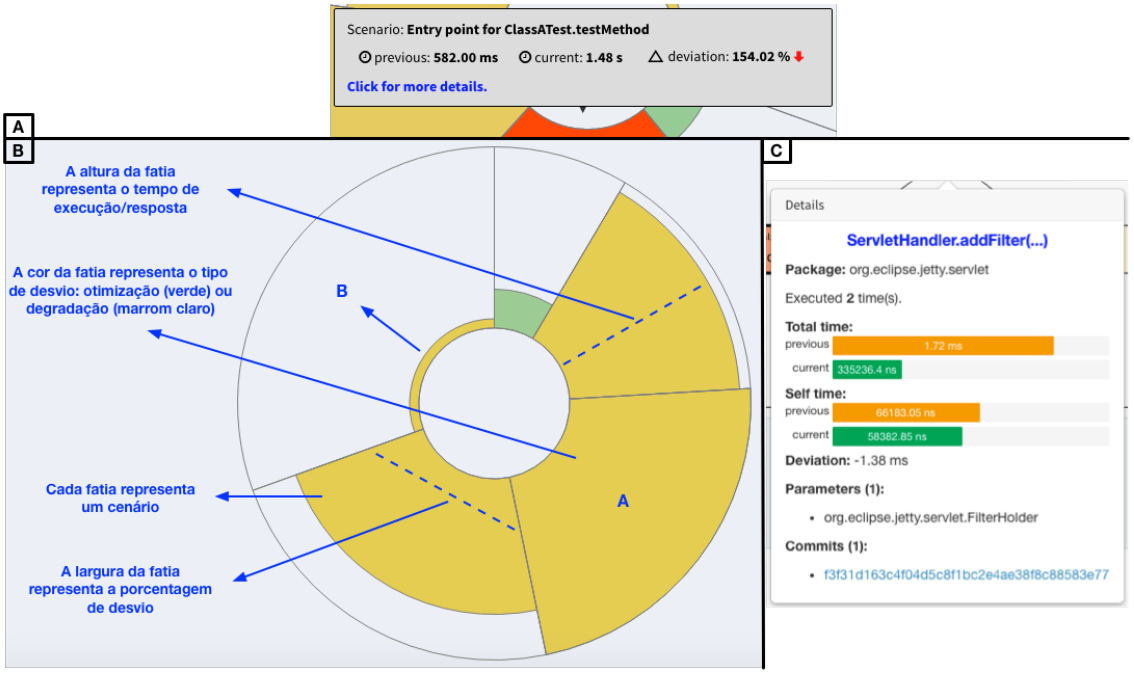

Figura 2. (A) Exemplo do *tooltip* da visualização de sumarização de cenários. (B) Exemplo da visualização Sumarização de Cenários. (C) Informações sobre o nó `ServletHandler.addFilter(...)` no grafo de chamadas.

A fatia `A` da Figura 2-B mostra o cenário com o maior tempo de execução dentre os exibidos. Essa conclusão é possível pois a fatia tem a maior altura preenchida. Por outro lado, a fatia `B` mostra o cenário com a maior porcentagem de desvio de desempenho dentre os apresentados. Tal conclusão se dá pois a fatia tem a maior largura preenchida. Em ambos os exemplos os cenários são de degradação. Cada fatia é clicável, redirecionando o usuário para a visualização do Grafo de Chamadas.

O usuário pode posicionar o *mouse* em cima de uma das fatias e obter mais detalhes, como o tempo de execução na versão anterior e atual, e a porcentagem de desvio. A Figura 2-A acima exemplifica esses detalhes.

2.4. Visualização do Grafo de Chamadas

Esta visualização objetiva mostrar, para dadas duas versões de um sistema, os métodos que potencialmente causaram o desvio de desempenho para um cenário. Os métodos são

exibidos em um grafo direcionado de chamadas com propriedades visuais que destacam quais desses métodos tiveram desvios de desempenho. A Figura 3 exemplifica esse grafo.

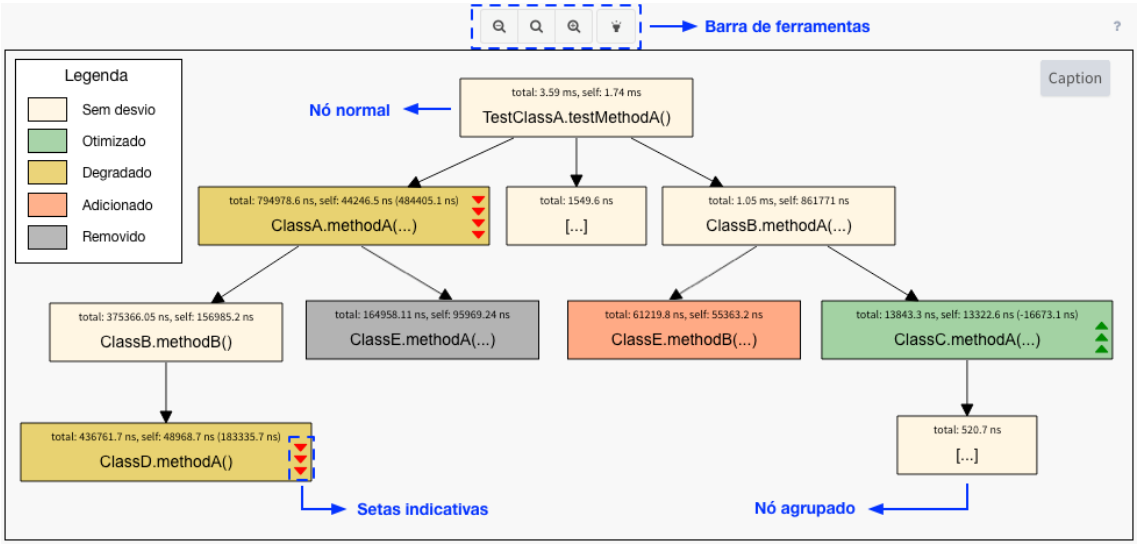

Figura 3. Exemplo da visualização Grafo de Chamadas.

Os nós possuem um nome (ao centro) e os tempos total, do próprio nó e o desvio, caso haja (no topo). Seis tipos de nós estão destacados na figura: (i) *sem desvio*: indica um nó que não teve desvio de tempo de execução entre as versões; (ii) *degradado*: representa um nó que teve o seu tempo de execução degradado da versão anterior para a versão atual; (iii) *otimizado*: indica um nó que teve o seu tempo de execução otimizado; (iv) *adicionado*: mostra um nó adicionado na versão atual; (v) *removido*: representa um nó que foi removido na versão atual; e (iv) *agrupado*: representa um conjunto de nós que não estão diretamente relacionados com os nós com desvio apresentados no grafo.

A borda dos nós indicam a quantidade de vezes que ele foi executado. Uma borda espessa (▣) significa que o nó executou mais de uma vez (em um *loop*, por exemplo).

No canto direito de um nó com desvio de desempenho podem ser vistas setas indicativas (▼ ou ▲) que representam a porcentagem de desvio. Na Figura 3, o nó `ClassC.methodA(...)` possui 3 setas verdes para cima, indicando que a sua otimização foi de 50% a 75%. Já o nó `ClassA.methodA(...)` possui 4 setas vermelhas para baixo, o que representa uma degradação de mais de 75%.

A interação com o grafo se dá através da barra de ferramentas da Figura 3. Através dos botões, o usuário pode realizar o *zoom out*, *zoom to fit* e *zoom in* ( ) e destacar os nós com desvios, adicionados e removidos ( ). Há, também, uma legenda explicativa no canto superior direito do grafo. Essa visualização ainda possui duas seções: sumário e histórico. No sumário são mostradas informações sobre o cenário do grafo, como a quantidade de nós exibidos. No histórico é apresentado um gráfico de linha mostrando, ao longo de todas as versões analisadas, os tempos de execução para o cenário em questão.

2.5. Aspectos de Implementação

A ferramenta foi implementada utilizando o *framework* Grails, além das linguagens de programação Groovy e Java. Os bancos de dados DAM e QAV foram implementados no

PostgreSQL. O arquivo usado para montar as visualizações é enviado pelo servidor em formato JSON. No lado cliente, a partir desse arquivo, as visualizações são montadas e renderizadas usando uma biblioteca JavaScript chamada JointJS [JointJS 2017].

3. Avaliação

Para avaliar as visualizações, a ferrramenta PMV foi aplicada na análise da evolução de versões de dois sistemas reais: o Jetty e o VRaptor. O primeiro trata-se de um *framework open-source* que provê um servidor *web* além de um *servlet container* Java. Já o segundo é um *framework* para desenvolvimento *web* MVC. Os critérios de seleção foram: (i) ser escrito em Java, (ii) ter um mínimo de 10 *releases*, (iii) possuir testes automatizados em JUnit, (iv) possuir repositório no GitHub e (v) estar listado no site http://java-source.net.

As Tabelas 1 e 2 exibem um resumo das principais informações encontradas para cada conjunto de versões dos sistemas analisados. Para ambos, os códigos-fonte das versões listadas nas tabelas foram baixados e, para cada versão, foi realizado o processamento necessário até que os dados das visualizações fossem gerados.

Tabela 1. Resumo para o Jetty.

Versão Inicial	Versão Final	Cenários Degradados	Cenários Otimizados	Total de Cenários
9.3.10	9.3.11	5 (3,18 %)	1 (0,63 %)	157
9.3.11	9.3.12	0 (0 %)	36 (21,55 %)	167
9.3.12	9.3.13	0 (0 %)	7 (4,51 %)	155
9.3.13	9.3.14	0 (0 %)	0 (0 %)	157
9.3.14	9.3.15	3 (1,84 %)	2 (1,22 %)	163
9.3.15	9.3.16	2 (1,21 %)	1 (0,60 %)	165
9.3.16	9.4.0	28 (14,97 %)	6 (3,20 %)	187
9.4.0	9.4.1	1 (0,53 %)	0 (0 %)	186
9.4.1	9.4.2	4 (2,15 %)	0 (0 %)	186
Total		43	53	1523

Tabela 2. Resumo para o VRaptor.

Versão Inicial	Versão Final	Cenários Degradados	Cenários Otimizados	Cenários com Desvio
4.0.0.Final	4.1.0.Final	36 (4,86 %)	41 (5,54 %)	740
4.1.0.Final	4.1.1	0 (0 %)	0 (0 %)	745
4.1.1	4.1.2	20 (2,66 %)	10 (1,33 %)	750
4.1.2	4.1.3	9 (1,19 %)	2 (0,26 %)	752
4.1.3	4.1.4	0 (0 %)	0 (0 %)	739
4.1.4	4.2.0.RC1	0 (0 %)	1 (0,12 %)	774
4.2.0.RC1	4.2.0.RC2	4 (0,51 %)	1 (0,12 %)	776
4.2.0.RC2	4.2.0.RC3	6 (0,77 %)	0 (0 %)	771
4.2.0.RC3	4.2.0.RC4	15 (1,94 %)	3 (0,38 %)	773
Total		90	58	6820

Para exemplificação da avaliação quantitativa, serão usadas informações do Jetty. Como pode ser visto na Tabela 1, para os *patches* 13 e 14 da versão 9.3 não houve desvios de desempenho em nenhum dos cenários. Várias otimizações importantes (total de 36) aconteceram entre os *patches* 11 e 12 da versão 9.3 e a mudança de versão minoritária de 9.3.16 para 9.4 causou degradações (total de 28) na maioria dos cenários. A Figura 4-A representa a sumarização de cenários entre os *patches* 14 e 15 da versão 9.3.

Os 5 cenários com desvios de desempenho são vistos na Figura 4-A: 3 de degradação e 2 de otimização. O cenário indicado pela letra `A` foi o que possuiu maior desvio: uma otimização de 92,99% em relação a versão anterior. A letra `B` indica o cenário com maior tempo de execução: 221,42 ms. No entanto, o seu desvio em com relação a versão anterior foi uma degradação de apenas 2,37%.

O fato de apenas 5 cenários terem sido indicados com desvios na Figura 4-A não significa que existe apenas essa quantidade de cenários no sistema, mas que a avaliação feita pelo PM os apontou como cenários com desvios significativamente relevantes.

Na Figura 4-B, o nó `A` foi otimizado em mais de 75%, já o nó `B` foi adicionado na versão 9.3.15. Além disso, ambos executaram mais de uma vez. A adição de um nó no grafo é um fator potencialmente responsável por causar degradações, entretanto, o nó `B`

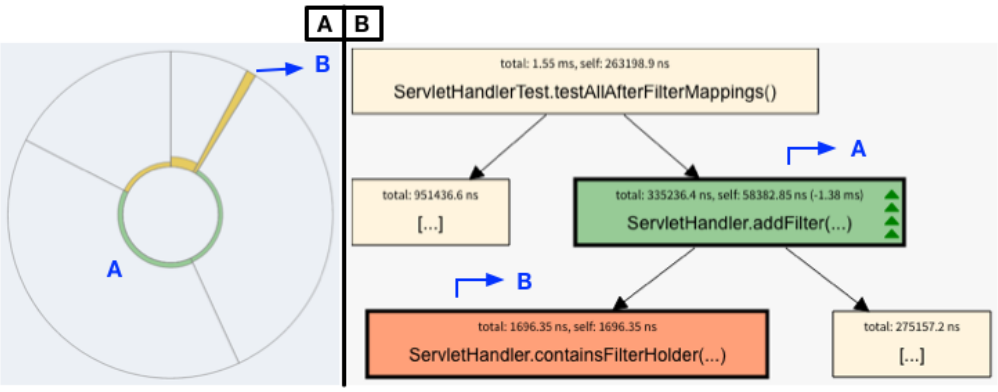

Figura 4. Sumarização de Cenários dos *patches* 14 e 15 da versão 9.3 do Jetty (A) e Grafo de Chamadas para o cenário `Entry point for ServletHandlerTest.testAllAfterFilterMappings` (B).

possui um tempo baixo em relação ao nó `A`. Dessa forma, a otimização do nó `A` foi maior do que a degradação inserida pelo nó `B`, resultando em um cenário otimizado.

Quando o *mouse* é passado sobre um nó, mais informações sobre ele são exibidas em um *tooltip*. A Figura 2-C mostra mais informações do nó `A` da Figura 4-B. Na figura, o *commit* apresentado foi o responsável pela modificação que potencialmente resultou na otimização do seu desempenho. Ao clicar no *commit*, o usuário tem acesso a ele diretamente no GitHub. Além dessa informação, o usuário pode verificar o pacote, quantidade de execuções, os tempos totais e do próprio nó para as versões, o tempo de desvio e os parâmetros.

4. Trabalhos Relacionados

Na literatura foram encontrados alguns trabalhos que se relacionam a este. Sandoval [Sandoval Alcocer et al. 2013] propõem uma abordagem visual para entender a causa de degradações de desempenho, comparando duas versões de um sistema. A ferramenta é compatível com a linguagem SmallTalk. O presente trabalho se diferencia por (i) ser compatível com a linguagem Java, aumentando a gama de sistemas alvo; (ii) exibir todos os cenários comparativamente entre as versões do software; e (iii) utilizar elementos visuais diferentes para destacar os nós e suas propriedades.

Bergel, Robbes e Binder [Bergel et al. 2010] propuseram uma abordagem cujo objetivo é comparar duas versões de um software, também resultando em um grafo de chamadas. Entretanto, essa abordagem considera apenas se um método gastou ou não mais tempo de execução do que na versão anterior. O PMV se diferencia por adicionar mais métricas, como a porcentagem de desvio dos métodos, se houve métodos adicionados/removidos, além de facilitar a identificação da causa do desvio apresentando a listagem de *commits* potencialmente responsáveis pela modificação no código-fonte.

5. Conclusão

Motivados pela necessidade de acompanhamento da evolução dos atributos de qualidade, este trabalho apresentou uma ferramenta *web*, intitulada *PerfMiner Visualizer*, que provê

visualizações de software para ajudar desenvolvedores e arquitetos dos sistemas de software a analisar a evolução do atributo de qualidade de desempenho, em termos de tempo de execução, ao longo das versões de um sistema.

Através da aplicação da ferramenta a dois sistemas reais, pode-se concluir que seus usuários podem se beneficiar das seguintes maneiras: (i) ter uma visão geral dos cenários com desvios de desempenho entre uma determinada versão do software e a anterior; (ii) descobrir qual desses cenários teve o maior tempo de execução; (iii) saber qual desses cenários teve o maior desvio; (iv) acompanhar a evolução de cada cenário ao longo das versões analisadas; (v) saber, para cada cenário, os nós que foram detectados com algum tipo desvio, além de ter conhecimento sobre os nós que foram adicionados e removidos; e (vi) obter uma listagem dos *commits* que possivelmente causaram o desvio.

Como trabalhos futuros, (i) estão sendo conduzidos estudos qualitativos com os desenvolvedores e arquitetos dos sistemas analisados. Objetiva-se, com esse estudo, obter retorno sobre a usabilidade das visualizações, a facilidade de se encontrar informações e a possível aplicabilidade da ferramenta como parte integrante dos seus processos de desenvolvimento. Além disso, (ii) pretende-se integrar a ferramenta ao Github de modo que seja utilizada através de *webhooks*. Dessa forma, a cada *release* lançada, a ferramenta realizaria a análise e publicação dos resultados de maneira automática.

Agradecimentos. Este trabalho é parcialmente financiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software, processo CNPq/465614/2014-0.

Referências

Bergel, A., Robbes, R., and Binder, W. (2010). Visualizing dynamic metrics with profiling blueprints. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6141 LNCS:291–309.

Caserta, P. and Zendra, O. (2011). Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933.

D'Ambros, M. and Lanza, M. (2009). Visual software evolution reconstruction. *Journal of Software Maintenance and Evolution*, 21(3):217–232.

Diehl, S. (2007). *Software visualization: Visualizing the structure, behaviour, and evolution of software*.

Ghanam, Y. and Carpendale, S. (2008). A survey paper on software architecture visualization. *University of Calgary, Tech. Rep*.

JointJS (2017). JointJS.

Pinto, F., Kulesza, U., and Treude, C. (2015). Automating the performance deviation analysis for multiple system releases: An evolutionary study. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015 - Proceedings*, pages 201–210.

Sandoval Alcocer, J. P., Bergel, A., Ducasse, S., and Denker, M. (2013). Performance evolution blueprint: Understanding the impact of software evolution on performance. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*.