ArchCI: Uma Ferramenta de Verificação Arquitetural em Integração Contínua

Arthur F. Pinto[1], Nicolas Fontes[2], Eduardo Guerra[2], Ricardo Terra[1]

[1]Universidade Federal de Lavras (UFLA), Lavras, Brasil

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

fparthur@posgrad.ufla.br,
{nicolas.fontes,eduardo.guerra}@inpe.br,terra@dcc.ufla.br

Abstract. *As software evolves, developers usually introduce deviations from the planned architecture, due to unawareness, conflicting requirements, technical difficulties, deadlines, etc. Although architectural compliance processes identify architectural violations, (i) these tools are underused and (ii) detected violations are rarely corrected. Therefore, this article introduces ArchCI, a tool that provides architectural compliance check as part of the Continuous Integration (CI) process. The tool relies on DCL as underlying conformance technique and Jenkins as the CI server. It implies that the architectural compliance process is triggered by every code integration, performing preset actions when violations are detected. Such actions range from sending a warning e-mail to forbid the integration to the repository. Also, this article reports case studies in a controlled and a real-world scenarios to demonstrate the applicability of the tool.*

Resumo. *No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por novos requisitos, desconhecimento, dificuldades técnicas, prazos curtos, etc. Embora existam processos e ferramentas de conformidade arquitetural que identifiquem violações, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo implementa ArchCI, uma ferramenta que provê a verificação de conformidade arquitetural como parte do processo de Integração Contínua (IC). São utilizados DCL como técnica de conformidade e Jenkins como servidor de IC. Isso implica que o processo de conformidade arquitetural é ativado a cada integração de código, executando ações configuradas quando violações forem detectadas. Tais ações podem variar desde o envio de um e-mail de alerta ao bloqueio da integração do código ao repositório. Além disso, este artigo reporta uma avaliação controlada e uma avaliação em um cenário real que demonstram a aplicabilidade da ferramenta.*

Vídeo da ferramenta. *http://youtu.be/WhjK4M--jzc*

1. Introdução

No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por desconhecimento, requisitos conflitantes, dificuldades técnicas, prazos curtos, novos requisitos, etc. [7, 11]. Isso se agrava em projetos com vários desenvolvedores uma vez que o acúmulo dos possíveis desvios arquiteturais que podem ocorrer durante sua implementação, são potencializados pelo aumento do número de desenvolvedores em um projeto, levando ao fenômeno conhecido

como erosão arquitetural [5, 2]. Mais importante, esses desvios arquiteturais impactam negativamente o projeto, podendo anular características essenciais de um sistema, como manutenibilidade, reusabilidade, escalabilidade, etc. [6]. Ainda mais crítico, como parte de um fenômeno conhecido como dívida técnica [9], sabe-se que quanto mais esses problemas demorarem para serem eliminados, mais caro será para corrigí-los.

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo apresenta ArchCI, uma ferramenta que implementa uma solução de conformidade arquitetural em Integração Contínua (IC) proposta inicialmente em uma escola [8]. Nesse cenário, o processo de conformidade arquitetural é ativado a cada integração de código no servidor – o que auxilia na solução do problema (i), pois desenvolvedores não podem desativar a ferramenta quando lhes convier –, a qual executa uma ação configurada quando violações forem detectadas, que varia do envio de um e-mail de alerta ao arquiteto de software ao bloqueio da integração ao repositório – o que auxilia na solução do problema (ii). ArchCI utiliza DCL (*Dependency Constraint Language*) como linguagem de definição das restrições de conformidade [11] e o Jenkins como servidor de IC [10].

Neste artigo, estendeu-se um estudo prévio [8] nas seguintes direções: (a) uma descrição completa das funcionalidades da ferramenta (Seção 2.1); (b) um reporte detalhado da arquitetura da ferramenta com a inclusão de vários detalhes técnicos voltados à uma sessão de ferramentas (Seção 2.2); (c) uma avaliação em um cenário real de desenvolvimento (Seção 3.2); e (d) uma revisão do estado-da-prática em relação a ferramentas relacionadas (Seção 4).

O restante desse artigo está organizado como descrito a seguir. A Seção 2 provê uma visão geral da ferramenta ArchCI, descrevendo um exemplo de uso, suas principais funcionalidades, sua arquitetura e interface. A Seção 3 avalia a aplicabilidade da ferramenta em um cenário controlado e em um cenário real. A Seção 4 discute ferramentas relacionadas e a Seção 5 apresenta as considerações finais.

2. Ferramenta ArchCI

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante de tal cenário, ArchCI garante que o processo de verificação de conformidade arquitetural seja realizado em um servidor de IC sem a necessidade de instalações em máquinas de desenvolvedores. Assim, integrações de código com violações serão sempre detectadas e devidamente tratadas, conforme ilustrado na Figura 1, onde desenvolvedores, ao integrarem código, ativam uma tarefa de verificação de conformidade arquitetural no servidor de IC que realiza ações específicas ao detectar violações arquiteturais.

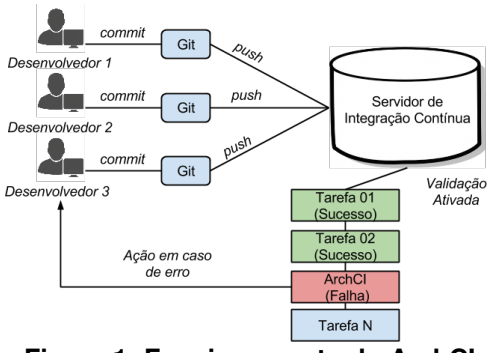


Figura 1. Funcionamento do ArchCI

ArchCI utiliza DCL como linguagem de definição das restrições de conformidade e o Jenkins como servidor de IC. ArchCI captura dois tipos de violações arquiteturais: *divergências* (quando uma dependência observada no código fonte não está de acordo com o modelo arquitetural do sistema) e *ausências* (dependência inexistente no código fonte, mas que é obrigatória de acordo com o modelo arquitetural). Essencialmente, esse modelo abrange qualquer forma de relação entre classes que podem ser verificadas estaticamente. Por restrições de espaço, uma descrição completa da linguagem DCL e do servidor de IC Jenkins podem ser encontrados em [11] e [10], respectivamente.

Como um exemplo de uso, suponha a especificação DCL (definida no arquivo `architecture.dcl`) ilustrada na Figura 2(a). A linha 3 restringe o módulo `Main` – composto pelas classes do pacote `project.main` (linha 1) – de acessar a classe `java.lang.Math`. O desenvolvedor ao tentar integrar ao repositório a classe `Main` (Figura 2(b)) que possui um acesso à classe `Math` (linha 5), ArchCI fornecerá uma mensagem de erro juntamente com as violações encontradas nas classes alteradas da integração, conforme ilustrado na Figura 2(c).

```
1 module Main: project.main.*
2
3 Main cannot-depend java.lang.Math
```

(a) Exemplo de Restrição de Dependência

```
1  package project.main;
2
3  public class Main {
4      public static void main(String[] args) {
5          System.out.println(Math.pow(2, 5));
6      }
7  }
```

(b) Exemplo de Violação

```
$ git add .
$ git commit -m "Integration Changes"
[dev a57ceb7] Integration Changes
 1 file changed, 4 insertions(+), 4 deletions(-)
$ git push

FAILURE
VIOLATION 1: 'project.main.Main' contains the method 'main' that statically invo
kes the method 'pow' of an object of 'java.lang.Math'
```

(c) Relatório após a tentativa de integração de código

Figura 2. Exemplo de Uso da ferramenta ArchCI

2.1. Funcionalidades

Verificação Arquitetural: As integrações são realizadas por intermédio do Gerrit[1] em um *branch* temporário. Quando ArchCI não detectar violações, o código é automaticamente unificado (*merge*) ao *branch* principal. No entanto, caso violações sejam detectadas, a integração ficará pendente de aprovação e ArchCI realizará, por meio de *hooks*, a ação configurada em caso de violação.

Ações: É possível (i) bloquear ou (ii) permitir a integração de código. No último caso, um *hook* do Gerrit ativará uma tarefa do Jenkins que enviará automaticamente alertas diários por e-mail ao desenvolvedor. Considerando que débitos técnicos são inevitáveis, por prazo e cobranças, cabe ao encarregado do projeto decidir qual ação corretiva é a mais adequada no projeto. Inclusive, a ferramenta pode ser desativada em atividades de reestruturação ou evolução da arquitetura.

Atomicidade: Na configuração de *bloqueio*, somente as integrações que estejam em total acordo com as regras arquiteturais do projeto serão aceitas pelo servidor.

[1] http://gerrithub.io

Verificação Incremental: ArchCI verifica somente as classes que sofreram alterações desde a última integração de forma a assegurar o bom desempenho da ferramenta.

Evolução da Arquitetura: A verificação arquitetural considera a especificação DCL armazenada no repositório (`architecture.dcl`). No entanto, em caso de uma integração em que houve modificações no arquivo `architecture.dcl` – para inclusão ou alteração de regras – ArchCI considera a especificação DCL a ser integrada ao repositório.

Uso local: ArchCI realiza a verificação de conformidade apenas no momento de integração de código. No entanto, para assegurar um menor número de violações em tentativas de integrações ao repositório, a ferramenta `dclcheck` [11] – um *plug-in* para a IDE Eclipse com a mesma finalidade – pode ser utilizada localmente.

Linguagem de Programação: Embora o conceito seja desacoplado a linguagens de programação, a atual implementação de ArchCI atua sobre projetos desenvolvidos em Java.

2.2. Estrutura Interna

Conforme ilustrado na Figura 3, a implementação de ArchCI segue uma arquitetura com cinco módulos principais:

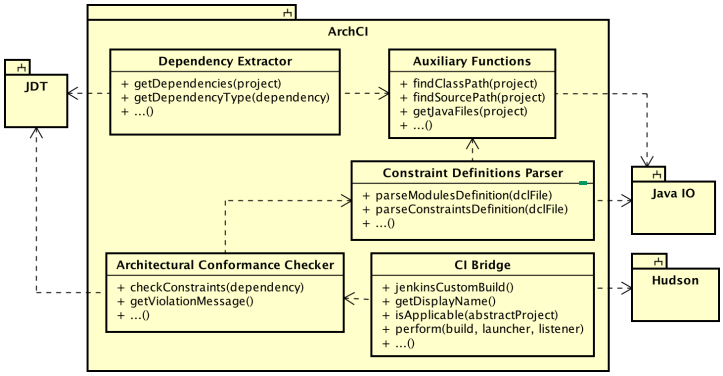


Figura 3. Arquitetura do ArchCI

Dependency Extractor: Módulo responsável pela obtenção das dependências do projeto, assim como a manipulação das mesmas. Apresenta funções que analisam cada elemento das classes a serem validadas, analisando o tipo de dependência ao qual o determinado elemento se refere. Para sua implementação, foi realizada uma adaptação da ferramenta `dclcheck` [11]. Desse modo, tornou-se necessário remover todas as dependências e partes de código que eram inteiramente exclusivos da IDE Eclipse, permanecendo apenas dependências às bibliotecas de manipulação de AST (*Abstract Syntax Tree*) fornecidas pelo Eclipse JDT (*Java Development Tools*). Todas as dependências ao Java Model, conjunto de classes que representam um projeto internamente na IDE Eclipse, tiveram de ser totalmente descartadas e, sendo assim, praticamente toda manipulação das classes a serem verificadas teve de ser reescrita. Por fim, foram utilizados métodos e funções da classe AST para que cada elemento pudesse ser compreendido e assim, determinadas as dependências do projeto.

Constraint Definitions Parser: Módulo encarregado do *parse* do arquivo contendo os módulos do projeto e as restrições de dependência estabelecidas para a arquitetura do sistema armazenadas no arquivo `architecture.dcl`.

Architectural Conformance Checker: Módulo envolvendo funções para garantir a conformidade arquitetural do projeto por meio da verificação e validação de desvios arquiteturais com base nas restrições de dependência definidas. Cada dependência extraída

pelo módulo *Dependency Extractor* é verificada frente às restrições obtidas pelo módulo *Constraint Definitions Parser* a fim de se encontrar violações arquiteturais.

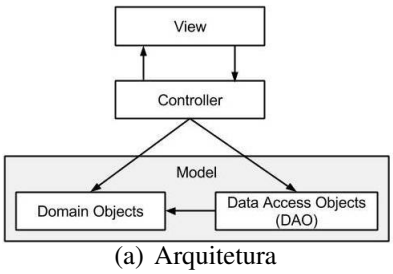
Auxiliary Functions: Módulo responsável por fornecer funções que auxiliem as tarefas do ArchCI de modo geral, por exemplo, de localização do caminho das bibliotecas e dos arquivos necessários para a resolução das dependências.

CI Bridge: Módulo contendo as funções necessárias para integrar o código ao servidor Jenkins, o qual engloba funções para a customização do *build*, obtenção do *workspace* com o código a ser integrado, identificação das classes a serem validadas, etc. O projeto da ferramenta ArchCI foi estruturado como um projeto Maven para funcionar como *plug-in* no Jenkins. Em seguida, foram designadas dependências à classes da biblioteca Hudson, para que assim fosse possível manipular os elementos e componentes envolvidos na execução das tarefas no Jenkins. Por fim, foram criadas uma classe de descrição (onde as propriedades do *build* são definidas) e uma classe contendo métodos para a obtenção de informações fornecidas pela tarefa do servidor, bem como as ações realizadas pelo *build*. Assim, com o acesso ao *workspace*, o processo de verificação e validação das dependências torna-se possível pelo *plug-in*. Já o processo de *bloqueio* é executado pelo servidor de IC em conjunto com o Gerrit.

3. Avaliação

3.1. Cenário Controlado

Sistema Alvo: `myAppointments` [6], um sistema de gerenciamento de informação pessoal simples. Apesar de ser um sistema de pequeno porte, suas restrições arquiteturais são provavelmente utilizadas em diversos projetos reais. Conforme ilustrado na Figura 4(a), o sistema segue o padrão arquitetural *Model-View-Controller* (MVC). Internamente ao componente *Model*, estão contidos *Domain Objects*, que representam entidades de domínio, e *Data Access Objects* (DAOs), que encapsulam o *framework* de persistência.



(a) Arquitetura

```
%Modulos
module Controller:    myappointments.controller.*
module View:          myappointments.view.*
module Model:         myappointments.model.**
module Domain:        myappointments.model.domain.*
module Util:          myappointments.util.*
module DB:            myappointments.model.DB
module DAO:           "myappointments.model.[a-zA-Z0-9/.]*DAO"
module JavaAwtSwing:  java.awt.**, javax.swing.**
module JavaSql:       java.sql.**

%Restricoes
only View can-depend JavaAwtSwing                          %RA1
only DAO, DB can-depend JavaSql                            %RA2
View cannot-depend Model                                   %RA3
Domain can-depend-only $java                               %RA4
DAO can-depend-only Domain, Util, javaSql                  %RA5
Util cannot-depend $system                                 %RA6
```

(b) Especificação DCL

Figura 4. Avaliação Controlada com myAppointments

Restrições: `myAppointments` deve respeitar as seguintes restrições arquiteturais: (RA1) Somente *View* pode depender de *AWT/Swing*; (RA2) Somente DAOs e `model.DB` podem depender dos serviços de persistência; (RA3) A camada *View* não pode acessar componentes do *Model* diretamente; (RA4) *Domain Objects* devem depender apenas da API de Java; (RA5) DAOs podem depender somente de *Domain Objects*, do pacote *Util* e dos serviços de persistência; (RA6) O pacote *Util* não pode depender de classes específicas do projeto. Para a utilização de ArchCI, tais definições foram traduzidas para restrições de dependência na linguagem DCL, conforme ilustrado na Figura 4(b).

Violações: Como `myAppointments` foi projetado para ilustrar técnicas de conformidade, sua arquitetura original não possui violações. No entanto, para ilustrar o funcionamento

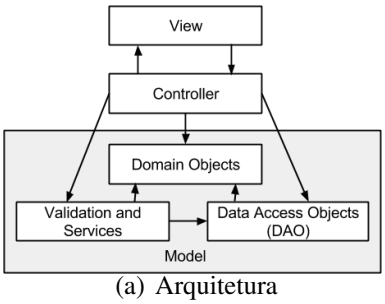
da ferramenta proposta, foram intencionalmente incorporadas seis violações arquiteturais, uma para cada restrição arquitetural. Por exemplo, a classe `Appointment`, que pertence a camada *Domain*, teve incorporadas dependências com `javax.swing.JOptionPane` e `java.sql.Date` que violam, respectivamente, as RA1 e RA2.

Resultado: Ao realizar a integração de código, ArchCI foi capaz de encontrar as seis violações com sucesso. Como a ferramenta, nesta avaliação, foi configurada de forma a não ser possível integrar código com violação arquitetural, ArchCI cancelou a integração (*push*) e informou as violações ao desenvolvedor.

Limitações: A avaliação foi realizada em um ambiente controlado – um sistema de pequeno porte, um único desenvolvedor, poucas integrações de código e um pequeno conjunto de violações. Entretanto, o objetivo da avaliação de se verificar a aplicabilidade de ArchCI foi atingido ao se demonstrar que é sim possível integrar um processo de conformidade arquitetural em IC.

3.2. Cenário Real

Sistema Alvo: LEONA - Rede Colaborativa para Estudos de Eventos Luminosos Transientes (ELT). O projeto se concentra em desenvolver uma plataforma web capaz de monitorar estações remotas localizadas em toda a América Latina e obter imagens para análise e estudos sobre os ELTs. É um sistema com uma razoável quantidade de usuários e com uma arquitetura que envolve diversos nós. O sistema é baseado no padrão arquitetural MVC, porém também possui camadas adicionais no componente Model. Existem camadas para classes de domínio, serviços com regras de negócio e de persistência, que implementam o padrão DAO. O software utiliza bibliotecas externas, sendo elas Esfinge, JavaxSwing e VRaptor, que podem ser utilizados apenas por certos componentes. A Figura 5(a) apresenta o modelo arquitetural do LEONA. É importante ressaltar que a construção do sistema já era feita dentro de um processo de IC.



(a) Arquitetura

```
%Modules
module Controller:    br.leona.server.controller.*
module Model:         br.leona.server.model.*
module Service:       br.leona.server.service*
module DAO:           br.leona.server.dao.*
module Persistence:   javax.persistence.*
module Esfinge:       org.esfinge.*
module Serializable:  java.io.Serializable.*
module VRaptor:       br.com.caelum.vraptor.*
module JavaxSwing;    javax.swing.*

%Constraints
Service can-depend-only Controller                    %RA1
only Model can-declare Persistence                    %RA2
only Controller can-depend Controller                 %RA3
only DAO can-declare Esfinge                          %RA4
View cannot-access Model                              %RA5
Model must-declare Serializable                       %RA6
only Controller can-declare VRaptor                   %RA7
only Controller can-declare JavaxSwing                %RA8
```

(b) Especificação DCL

Figura 5. Avaliação em Cenário Real com LEONA

Restrições: LEONA deve respeitar as seguintes restrições arquiteturais: (RA1) *Service* só pode depender de *Controller*; (RA2) Somente *Model* pode declarar *Persistence*; (RA3) Somente *Controller* pode depender de *Controller*; (RA4) Somente *DAO* pode declarar *Esfinge*; (RA5) *View* não pode acessar *Model*; (RA6) *Model* deve declarar *Serializable*; (RA7) Somente *Controller* pode declarar *VRaptor*; (RA8) Somente *Controller* pode declarar *JavaxSwing*. Para utilizar a ferramenta ArchCI, as definições das restrições de dependências do LEONA foram traduzidas na linguagem DCL conforme na Figura 5(b).

Violações: Como ArchCI foi implementada depois de boa parte do projeto LEONA estar desenvolvida, na sua primeira compilação foram identificadas três violações. Duas delas foram no módulo *DAO* onde estava utilizando classes do módulo *Persistence* também e a

outra no módulo *Service* que estava utilizando umas classe do *JavaxSwing*. As violações encontradas foram recusadas com base nas restrições RA4 e RA8, respectivamente.

Resultado: A ferramenta identificou as violações na integração do código à produção. Sendo assim, por intermédio da ferramenta, o ato de integrar código foi bloqueado e as violações informadas ao desenvolvedor. Deve-se destacar que a equipe encontrou soluções diferentes para conseguir atender a estrutura. Nas violações do módulo DAO foi identificado que a classe `Consultas` estava localizada incorretamente, e para corrigir foi movida para o módulo `Model`. Já na violação do módulo `Service` um diálogo de erro era instanciado, indo contra o que estava definido na arquitetura. A correção do problema foi o envio de um erro que era tratado usando o `JavaxSwing` no módulo `Controller` .

Limitações: A avaliação foi realizada em um ambiente real de desenvolvimento, onde se encontra uma equipe com três integrantes desenvolvendo e integrando os mesmos módulos de código. A ferramenta foi utilizada em uma versão do sistema com funcionalidades já concluídas, e o uso de ferramenta foi considerado apenas na última versão. Porém, o objetivo de conseguir identificar falhas na arquitetura como parte do processo de IC foi concluído com sucesso. A ferramenta continuará integrada no ambiente de desenvolvimento do projeto LEONA para que possa ser avaliado seu uso de forma contínua.

4. Ferramentas Relacionadas

Pelo menos as seguintes ferramentas podem ser utilizadas para se tratar de problemas arquiteturais juntamente com o processo de IC:

SonarQube[2] [1, 4] é uma plataforma de código aberto que provê integração com diferentes IDE's e com o servidor Jenkins para se gerenciar a qualidade do código de um projeto. Com a definição de regras arquiteturais no SonarQube, torna-se possível obter informações a respeito do software, por exemplo, cobertura de testes, métricas, matriz de dependências, aderência a boas práticas de código, etc. A partir dessas informações também é feita a quantificaçao da dívida técnica.

Checkstyle[3] utiliza métodos de manipulação de AST para inspecionar cada elemento do código e verificar sua conformidade com regras pré-definidas. Utilizando Java, é possível definir regras customizadas sobre diferentes aspectos do desenvolvimento e, posteriormente, exportá-las para aplicação em ferramentas de análise de qualidade, revisão de código, ou mesmo em servidores de IC. Um trabalho recente reportou o uso dessas regras para conformidade arquitetural [3], onde cada regra precisava ser criada a partir da criação de uma classe. O conjunto de regras desenvolvido no Checkstyle pode ser integrado ao SonarQube para análise de qualidade.

Code Climate[4] é uma plataforma para avaliação e revisão de projetos Ruby, Javascript, PHP e Python. É possível importar repositórios remotos do Github[5] e, utilizando a métrica GPA, fornecer uma nota geral sobre qualidade do projeto. A ferramenta provê dados e relatórios relacionados a complexidade, segurança, más práticas, duplicação de código, etc. Ademais, como a plataforma provê métricas a respeito dos testes, caso o projeto contenha testes arquiteturais, o Code Climate auxilia no processo de se manter um boa arquitetura de software durante o desenvolvimento e integração de código ao repositório.

[2]http://www.sonarqube.org

[3]http://checkstyle.sourceforge.net/

[4]https://codeclimate.com/

[5]https://github.com

As ferramentas encontradas apresentam funcionalidades relacionadas a avaliação da qualidade do software durante o processo de IC, porém nenhuma delas foca diretamente em um mecanismo para verificação de conformidade arquitetural. Como visto em [3], é possível criar regras personalizadas para essa finalidade, porém é possível ver que isso ainda é muito trabalhoso, necessitando a criação de uma classe para cada regra. Por fim, em relação com à `dclcheck` [11], a qual também define regras usando a linguagem DCL, o principal ponto de originalidade de ArchCI é estender DCL de forma a aliar a verificação de conformidade arquitetural como parte do processo de IC.

5. Conclusão

Este artigo descreve ArchCI, uma ferramenta de conformidade arquitetural (DCL) incorporada em um servidor de IC (Jenkins). Como principal contribuição, a ferramenta minimiza os problemas decorrentes de um processo de erosão arquitetural através de um processo de conformidade arquitetural mais rígido, e.g., integrações de código só ocorrem quando não foram detectadas violações arquiteturais.

Como trabalho futuro, pretende-se: (i) aplicar a solução proposta em projetos reais em andamento com alta taxa de integrações, a fim de avaliar sua expressividade, aplicabilidade e desempenho; inclusive a avaliação do ArchCI pelo próprio ArchCI; (ii) avaliar as características mais importantes para aceitação dos desenvolvedores, e.g., exibição de *warnings* ou bloqueio de integração de código; (iii) definir grau de severidade para cada restrição de forma a configurar ações pelo grau de severidade, e.g., bloquear a integração caso viole restrição arquitetural que afete segurança; e (iv) conduzir estudos relacionados à detecção e tratamento de violações nos diferentes momentos de integração do código.

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

Referências

[1] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning, 2013.

[2] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[3] Paulo Merson. Ultimate architecture enforcement: custom checks enforced at code-commit time. In *Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 153–160, 2013.

[4] Paulo Merson, Joseph Yoder, Eduardo Guerra, and Ademar Aguiar. Continuous inspection. In *2nd Asian Conference on Pattern Languages of Programs (AsianPLoP)*, pages 1–13, 2014.

[5] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *20th International Conference on Program Comprehension (ICPC)*, pages 3–10, 2012.

[6] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. Static architecture conformance checking: An illustrative overview. *IEEE Software*, 27(5):132–151, 2010.

[7] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[8] Arthur F. Pinto and Ricardo Terra. Processo de conformidade arquitetural em integração contínua. In *2nd Latin-American School on Software Engineering (ELA-ES)*, pages 1–12, 2015.

[9] Carolyn B. Seaman and Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[10] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, 2011.

[11] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, 2009.