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).

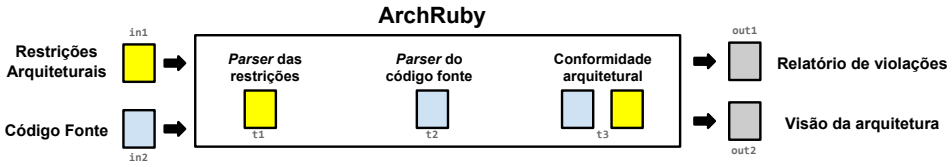

Figura 1. Funcionamento da ferramenta `ArchRuby`

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.

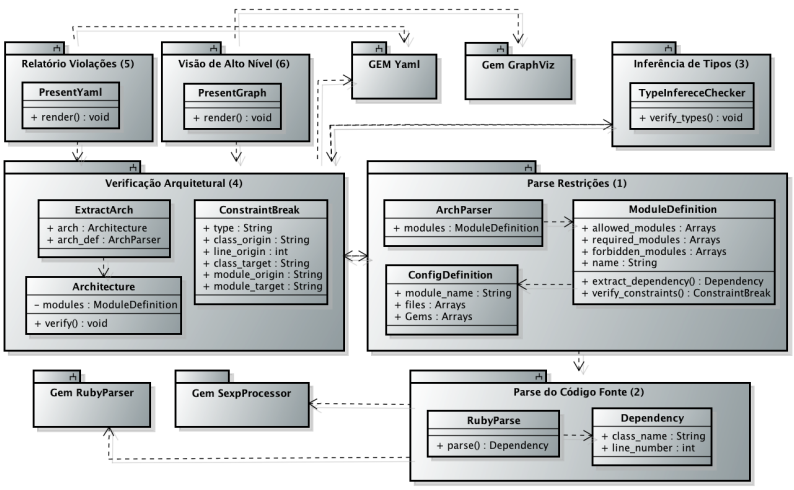

Figura 2. Arquitetura do `ArchRuby`

[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.

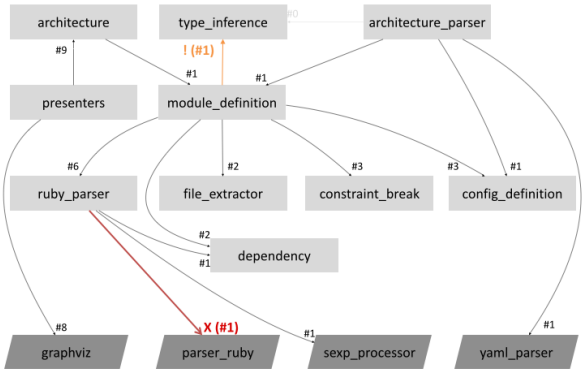

Figura 5. Visualização automática do `ArchRuby` provida pelo `ArchRuby`

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.