Conformidade e Visualização Arquitetural em Linguagens Dinâmicas

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

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais, Brasil
{sergio.miranda,mtov}@dcc.ufmg.br

[2] Departamento de Ciência da Computação
Universidade Federal de Lavras, Brasil
terra@dcc.ufla.br

Abstract. As software systems evolve, the architectural erosion process nullifies the well-known benefits provided by an architectural design. Such erosion process is even more severe in software systems developed in dynamic typed languages. The reasons are twofold: (i) some resources provided by such languages make developers more propitious to break the planned architecture, and (ii) the developers' community lacks tool support for architectural purposes. To address these shortcomings, this paper presents a conformance and visualization approach based on static code analysis techniques and a type inference heuristic. The central idea is to provide the developers' community with means to control the architectural erosion process by reporting architectural violations and visualizing the high-level model of the concrete architecture. This paper also describes a tool—called ArchRuby—that implements our approach and reports results from applying our solution in two real-world systems.

1 Introdução

A arquitetura planejada de um sistema engloba um conjunto de padrões e boas práticas arquiteturais que permitem a evolução do sistema de software [4]. 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 [5,3].

Esse processo de erosão arquitetural é ainda mais severo em sistemas desenvolvidos em linguagens dinâmicas 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.

Este artigo é centrado na conjectura de que sistemas desenvolvidos em linguagens dinâmicas devem ter um projeto arquitetural bem definido. Assim, é proposta uma abordagem de conformidade e visualização arquitetural baseada

em técnicas de análise estática de código e uma heurística de inferência de tipo para sistemas desenvolvidos em linguagens dinâmicas. Este artigo demonstra que é possível monitorar a arquitetura de tais sistemas usando técnicas não-invasivas de análise estática, i.e., sem modificações no código fonte, sem prejuízos ao desempenho, etc.

No intuito de investigar a aplicabilidade da solução proposta, foi implementada uma ferramenta para a linguagem Ruby e conduzida uma avaliação em dois sistemas de médio porte. Como resultado, a ferramenta foi capaz de detectar 46 violações das quais os desenvolvedores não tinham conhecimento prévio. Mais importante, certas violações só puderam ser detectadas devido à heurística de inferência de tipo. A ideia central é prover à comunidade de desenvolvedores uma forma simples e objetiva de controlar o processo de erosão arquitetural mediante a detecção de violações arquiteturais e da visualização do modelo de alto nível da arquitetura implementada

O restante deste artigo está organizado como a seguir. A Seção 2 apresenta a abordagem proposta, descrevendo os processos de conformidade e visualização arquitetural. A Seção 3 apresenta `ArchRuby`, a ferramenta que implementa a solução proposta. A Seção 4 reporta uma avaliação em dois sistemas reais de médio porte. Por fim, a Seção 5 discute trabalhos relacionados e a Seção 6 conclui.

2 Abordagem Proposta

Este artigo propõe uma solução de conformidade e visualização arquitetural baseada em técnicas de análise estática de código e em uma heurística de inferência de tipos para sistemas desenvolvidos em linguagens dinamicamente tipadas. A ideia é prover à comunidade de desenvolvedores uma solução para controlar o processo de erosão arquitetural mediante a detecção de violações arquiteturais (conformidade) e da geração de um modelo de alto nível da arquitetura (visualização). A Figura 1 ilustra o funcionamento da abordagem proposta.

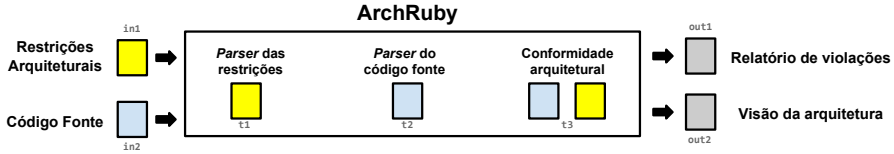

Figura 1. Abordagem proposta

Inicialmente, são recebidos como entrada apenas um arquivo de especificação das restrições arquiteturais (`in1`) e código fonte do sistema (`in2`). Após o *parser* das restrições arquiteturais (`t1`) e do código fonte (`t2`), é realizado o processo de conformidade arquitetural (`t3`), i.e., a detecção de decisões de implementação não coerentes com a arquitetura planejada do sistema alvo. Como resultado, é apresentado um relatório textual (`out1`) reportando e detalhando as violações arquiteturais detectadas (localização, restrição, etc.). Além disso, é apresentada uma visão de alto nível da arquitetura implementada (`out2`). Essa visão consiste em um grafo orientado cujos vértices representam os módulos e arestas repre-

sentam as dependências existentes entre os módulos, as quais são diferenciadas quando se tratam de violações.

O restante desta seção é organizado como a seguir. A Seção 2.1 apresenta o sistema motivador. A Seção 2.2 aborda a especificação de restrições arquiteturais e a Seção 2.3 descreve o processo de conformidade arquitetural. Por fim, a Seção 2.4 detalha o grafo de visualização arquitetural.

2.1 Sistema Motivador

O sistema `FindMeOnTwitter`[3] foi desenvolvido pelo primeiro autor deste artigo para ilustrar os processos de conformidade e visualização arquitetural propostos neste artigo. `FindMeOnTwitter` é um sistema web simples cuja funcionalidade consiste em realizar uma busca no Twitter (nome do usuário), persistir um registro da busca em uma base de dados e gerar um PDF descrevendo as buscas realizadas. O sistema foi desenvolvido na linguagem Ruby utilizando o *framework* Rails. Embora simples, `FindMeOnTwitter` retrata – em pequena escala – as dependências mais comuns em sistemas Ruby, e.g., comunicação web, persistência e utilização de Gems.[4] A Figura 2 ilustra o diagrama de componentes do sistema. Basicamente, o funcionamento do sistema é guiado pelas três funcionalidades da classe `MainController` descritas a seguir:

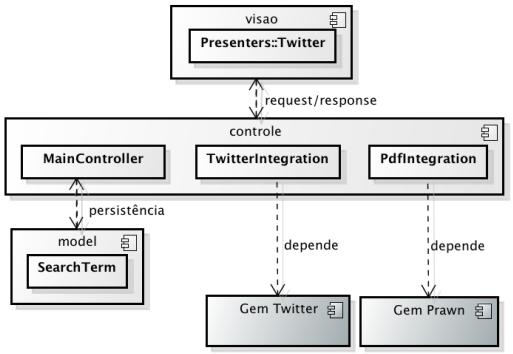

Figura 2. Arquitetura do `FindMeOnTwitter`

1. Ao receber o nome do usuário da camada de visão, instancia um objeto da classe `TwitterIntegration`, que representa a interface com o Gem `twitter`.
2. Ao obter os dados do Twitter, persiste as buscas realizadas. Para isso, deve instanciar um objeto da classe `SearchTerm`, que representa a interface com o Gem nativo `activerecord` de persistência no *framework* Rails.
3. Ao obter o pedido de geração de PDF da camada da visão, instancia um objeto da classe `PdfIntegration`, que representa a interface com o Gem `prawn` de geração de PDF.

[3] O sistema e seu código fonte estão publicamente disponíveis em: `http://aserg.labsoft.dcc.ufmg.br/archruby/cibse2015`

[4] Gem representa um pacote ou uma aplicação reusável escrita na linguagem Ruby.

2.2 Especificação de Regras Arquiteturais

As regras arquiteturais são especificadas em uma linguagem de domínio específico em YAML, formato largamente utilizado no ecossistema Ruby. Sumariamente, cada módulo do sistema alvo deve ter sua especificação formalizada como a seguir:[5]

```
1 <module_id>:
2   (files | gems): '<pattern_desc> {,<pattern_desc>}'
3   [(allowed | forbidden): '<module_id> {,<module_id>}']
4   [(required): '<module_id> {,<module_id>}']
```

onde <*module_id*> é o identificador do módulo (linha 1). O módulo pode ser formado por arquivos (`files`) ou por Gems (`gems`) que devem ser definidos por um ou mais <*pattern_desc*> separados por vírgula (linha 2). O casamento de padrões é baseado em *shell glob* (padrão da biblioteca de arquivos do Ruby) para mapear vários arquivos de uma só vez. Para a detecção de divergências (dependências existentes no código, mas não autorizadas [5]), para cada módulo pode-se definir os módulos cujo acesso é autorizado (`allowed`) ou não autorizado (`forbidden`), os quais são definidos por um ou mais <*module_id*> separados por vírgula (linha 3). Similarmente, para a detecção de ausências (dependências planejadas, mas não encontradas no código [5]), para cada módulo pode-se definir os módulos cujo acesso é obrigatório (`required`), os quais são definidos conforme supracitado (linha 4). É importante mencionar que, para um mesmo módulo, a chave `required` pode ser utilizada em conjunto com as chaves `allowed` e `forbidden`. No entanto, não existe a possibilidade de se utilizar a chave `allowed` e `forbidden` na definição de um mesmo módulo.

Para ilustrar uma especificação YAML, a Figura 3 reporta a definição de regras arquiteturais para o sistema `FindMeOnTwitter`. Por exemplo, o módulo `controller` (linhas 1–4) contém os arquivos `application_controller.rb`, `main_controller.rb` e *pode* depender de classes do módulo `model`, `integration_twitter`, `integration_pdf`, `actioncontroller` e `view`. Por outro lado, o módulo `view` (linhas 6–8) contém o arquivo `presenters/twitter.rb` e *não* pode depender de classes do módulo `model`. É importante ressaltar que módulos formados por Gems não definem restrições arquiteturais (e.g., `pdf` e `twitter`), pois não são partes integrantes do sistema alvo.

2.3 Conformidade Arquitetural

A partir da especificação das regras e se dispondo do código fonte do sistema alvo, é realizado o processo de conformidade arquitetural. Esse processo (i) extrai os módulos alvo e suas restrições arquiteturais pelo *parser* da especificação YAML; (ii) extrai o grafo de dependências de todo o sistema; (iii) adiciona informações de tipo ao grafo de dependências (por meio de uma heurística de inferência de

[5] A formalização segue convenções da notação Extended Backus-Naur Form (EBNF).

```
1  controller:
2    files:      'app/controllers/**/*.rb'
3    allowed:    'model, integration_twitter, integration_pdf,
4                 actioncontroller, view'
5
6  view:
7    files:      'app/views/**/*.rb, app/presenters/**/*.rb'
8    forbidden:  'model'
9
10 model:
11   files:      'app/models/**/*.rb'
12   required:   'activerecord'
13
14 integration_twitter:
15   files:      'lib/twitter_integration.rb'
16   allowed:    'twitter, model'
17
18 integration_pdf:
19   files:      'lib/pdf_integration.rb'
20   allowed:    'pdf'
21
22 twitter:
23   gems:       'Twitter'
24
25 pdf:
26   gems:       'Prawn'
27
28 activerecord:
29   gems:       'ActiveRecord'
30
31 actioncontroller:
32   gems:       'ActionController'
```

Figura 3. Regras arquiteturais do sistema `FindMeOnTwitter`

tipos); e (iv) verifica se as dependências obtidas nos passos *ii* e *iii* 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 `FindMeOnTwitter` (Figura 3). Nessa especificação, o módulo `view` não pode depender do módulo `model` (linha 8). No entanto, assuma que uma classe de visão acessa um método da classe de modelo. Tal dependência representa uma divergência arquitetural que é reportada ao desenvolvedor.

Inferência de Tipo: Embora dinamicamente tipada, é possível inferir em Ruby parte dos tipos de variáveis e parâmetros formais. Para isso, foi proposta uma heurística – mais especificamente, uma simplificação da heurística formalizada por Furr et al. [1] – que visa construir um conjunto `TYPES` de triplas [`method`, `var_name`, `type`], onde `type` é um dos possíveis tipos inferidos para a variável ou parâmetro formal `var_name` do método `method`. Esse conjunto é construído de acordo 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 [`f`, `x`, `T`] ∈ `TYPES`.

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

iii) **Fechamento**: [`method`, `var_name`, `type`] ∈ `TYPES` somente se puder ser obtido a partir de (i) com um número finito de aplicações de (ii).

A Figura 4 ilustra o funcionamento da heurística proposta. Ao executar o passo base do algoritmo, `TYPES` é inicializado com [`A::f`, `x`, `Foo`], [`A::f`, `b`, `B`], [`A::f`, `self`, `A`], [`B::g`, `c`, `C`] e [`C::h`, `d`, `D`] já que são as inferências diretas. Na primeira aplicação do passo recursivo, as triplas [`B::g`, `x`, `Foo`] e [`B::g`, `z`, `A`] são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `self` na chamada de `g`. Na segunda aplicação do passo recursivo, as triplas [`C::h`, `y`, `Foo`] e [`C::h`, `y`, `A`] são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `z` na chamada de `h`. Na terceira aplicação do passo recursivo, as triplas [`D::m`, `k`, `Foo`] e [`D::m`, `k`, `A`] são incluídas em `TYPES` (onde `k` é o nome do parâmetro em `D`), uma vez que se conhece o tipo da variável `y` na chamada de `m`. A aplicação do passo recursivo se repete até que nenhuma nova tripla seja adicionada ao conjunto.

```
1  class A                    class B                    class C
2    def f                      def g x z                  def h y
3      x = Foo.new                c = C.new                  d = D.new
4      b = B.new                  c.h(x)                     d.m(y)
5      b.g(x, self)               c.h(z)                   end
6    end                        end                      end
7  end                        end
```

Figura 4. Código para ilustração da heurística de inferência de tipo

2.4 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. [3], 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 motivador `FindMeOnTwitter`. Os vértices em retângulos na cor cinza claro representam módulos internos do sistema (e.g., `integration_pdf`) e vértices em trapézios na cor cinza escuro representam módulos externos do sistema (e.g., `actioncontroller`). As arestas são apresentadas conforme definições a seguir (assuma uma aresta de A para B):

(→) Aresta na cor preta: indica uma dependência permitida (*allowed*) entre os módulos A e B. Por exemplo, o módulo `controller` estabelece duas (#2) dependências com o módulo `integration_twitter` (veja linhas 3–4, Figura 3).

(→) Aresta na cor vermelha: indica uma violação do tipo ausência, i.e., existe alguma classe do módulo A que não depende do módulo B, mesmo tal dependência sendo obrigatória (*required*). Por exemplo, existe uma classe do módulo `model` que não estabelece dependência com o módulo `activerecord` (veja linha 12, Figura 3).

(→) Aresta na cor laranja: indica uma violação do tipo divergência, i.e., existe alguma classe do módulo A que depende do módulo B, porém tal dependência (i) não é permitida (*forbidden*) ou (ii) não foi explicitamente permitida (*allowed*). Por exemplo, o módulo `view` depende do módulo `model`, mas tal

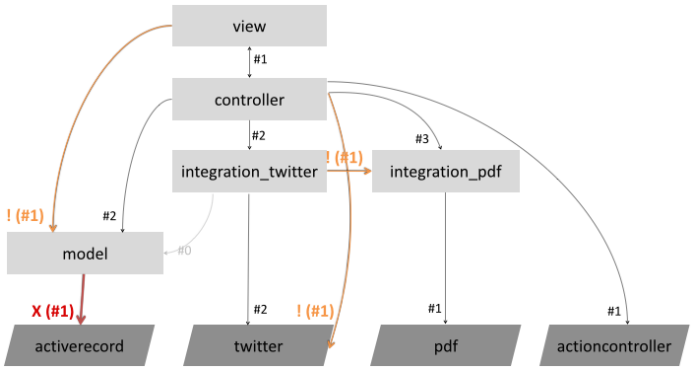

Figura 5. Modelo arquitetural do `FindMeOnTwitter` provida pelo `ArchRuby`[6]

dependência não é permitida (caso *i*; veja linha 8, Figura 3). Como um outro exemplo, o módulo `controller` depende do módulo `twitter`, mas tal dependência não foi explicitamente permitida (caso *ii*; veja linhas 3–4, Figura 3).

(→) Aresta na cor cinza: não indica uma violação, mas sim, um alerta de que o módulo A não depende do módulo B, mesmo tal dependência sendo prescrita como permitida (*allowed*). Por exemplo, foi explicitamente permitido que o módulo `integration_twitter` dependa do módulo `model` (veja linha 16, Figura 3), porém tal dependência não é estabelecida.

3 Ferramenta `ArchRuby`

`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:

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. O *parser* do arquivo YAML é realizado pela Gem `YAML` padrão da linguagem Ruby.

[6] Para facilitar a visualização, a Figura 5 está disponível em: `http://aserg.labsoft.dcc.ufmg.br/archruby/cibse2015`

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`.

3. *Verificação arquitetural*: Responsável por verificar se a arquitetura implementada (código fonte) segue a arquitetura planejada (restrições arquiteturais), conforme previamente descrito na Seção 2.3. O objetivo é identificar dependências que *não* respeitam as restrições arquiteturais e, caso detectadas, armazenar as informações detalhadas de tais violações.

4. *Geração do relatório de violações*: Responsável por estruturar os dados das violações arquiteturais detectadas em um arquivo no formato YAML (`archruby_report.yml`).

5. *Geração da visão de alto nível da arquitetura*: Responsável pela geração da visão arquitetural do sistema alvo conforme previamente descrito na Seção 2.4. O grafo é gerado por meio do Gem `Ruby-Graphviz`.

4 Avaliação

4.1 Sistemas Alvo

A solução foi avaliada em dois sistemas reais[7]: `Dito Social`, plataforma social da Dito para atender aos seus clientes; e `Tim Beta`, canal de comunicação da TIM com o público jovem. Tabela 1 reporta principais informações dos sistemas.

Tabela 1. Sistemas avaliados

	`Dito Social`	`Tim Beta`
LOC	13.304	17.817
# classes / # gems	142 / 34	141 / 50
Cyclomatic complexity	2.73 ± 2.94 (avg ± sd)	2.57 ± 2.79 (avg ± sd)
Principais Tecnologias	Ruby on Rails, Resque, Rspec, RSA, Twitter, Google Plus, Koala, Suspot Rails, Mysql2	Ruby On Rails, Resque, Twitter, YoutubeIt, Google Plus, Instagram, Devise, Foursquare2

4.2 Metodologia

A metodologia de avaliação dos dois sistemas obedeceu as três etapas a seguir:

(i) *Especificação de regras arquiteturais*: Os arquitetos especificaram as regras arquiteturais, logo após um tutorial sobre a linguagem de especificação de módulo e restrições (Seção 2.2).

(ii) *Conformidade arquitetural*: Os arquitetos executaram a ferramenta, logo após um breve tutorial no qual foram explicados principalmente suas entradas e saídas.

(iii) *Visualização arquitetural*: Os arquitetos avaliaram o modelo de alto nível produzido pela ferramenta.

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

4.3 Dito Social

Especificação de regras arquiteturais: O arquiteto desse sistema especificou 62 módulos e 43 restrições arquiteturais. Um subconjunto das regras definidas é reportado na Figura 6.[8] O módulo `dashboard_controller` é responsável por apresentar informações aos clientes e, portanto, pode acessar diversos módulos provedores de dados (linhas 3–7). O módulo `facebook_info_retriever` é responsável por buscar dados do Facebook e, portanto, pode acessar somente os módulos `facebook` e `airbrake` (linha 11). Já o módulo `post_model` é responsável pela persistência de dados e, portanto, deve implementar classes do módulo `activerecord` (linha 15) e pode acessar módulos que realizam trabalho em segundo plano (linhas 16–18), e.g., `post_workers`.

```
1  dashboard_controller:
2    files:    'app/controllers/dashboard/**/*.rb'
3    allowed:  'dashboard_finder, stats_model, network_model,
4               action_model, app_model, interaction_model, post_model,
5               social_helper, user_network_model, stats_model,
6               controller_base, referral_model, origin_model, http_party,
7               user_agent_model, user_model, airbrake'
8
9  facebook_info_retriever:
10   files:    'lib/facebook_info_retriever.rb'
11   allowed:  'facebook, airbrake'
12
13 post_model:
14   files:    'app/models/post.rb'
15   required: 'activerecord'
16   allowed:  'resque, post_workers, post_logger, facebook_info_retriever,
17              social_helper, interaction_model, question_option_model,
18              logger, activerecord, rails'
```

Figura 6. Subconjunto de regras arquiteturais para `Dito Social`

Conformidade arquitetural: `ArchRuby` foi capaz de detectar 24 violações no sistema `Dito Social`, conforme reportado na Tabela 2. Duas dessas violações são discutidas a seguir.

Tabela 2. Violações arquiteturais detectadas no `Dito Social`

Módulo	Restrições	# Violações
dashboard_controller	allowed: 'dashboard_finder, ...'	16
dashboard_finder	allowed: 'stats_model, ...'	3
report_model	allowed: 'post_model, ...'	2
event_model	allowed: 'action_model'	1
user_model	allowed: 'user_infos, ...'	1
facebook_info_retriever	allowed: 'facebook, airbrake'	1

Exemplo de violação #1: O envio de notificações ao usuário (e.g., e-mail) foi movido para um outro sistema e, portanto, não faz mais parte do sistema `Dito Social`. No entanto, conforme pode ser observado na Figura 7, `ArchRuby` detectou quatro divergências arquiteturais (linhas 2, 3, 5 e 7) na classe `EmailsController` – pertencente ao módulo `dashboard_controller` –

[8] Os dados completos da avaliação dos dois sistemas estão disponíveis em: `http://aserg.labsoft.dcc.ufmg.br/archruby/cibse2015`

com a classe `Email`, o que não é explicitamente permitido de acordo com a especificação de regras arquiteturais (linhas 3–7 da Figura 6).

```
1  def create                                  #from Module dashboard_controller
2    email = Email.new params['email']
3    email.save!
4    send_template_to_mandrill
5    if email.action
6      redis_action_id = SocialHelper::RedisData.get_action_id_by_name
7                        email.action.name, email.app_id
8    end
9  end
```

Figura 7. Exemplo #1 – Violação arquitetural no `Dito Social`

Exemplo de violação #2: O módulo `post_model` pode acessar o módulo `facebook_info_retriever`, mas não o contrário. No entanto, conforme pode ser observado na Figura 8, `ArchRuby` detectou duas divergências arquiteturais (linhas 16 e 19) na classe `FacebookInfoRetriever` – pertencente ao módulo `facebook_info_retriever` – com a classe `Post` do módulo `post_model`, o que não é permitido de acordo com a especificação de regras arquiteturais (linha 17 da Figura 6). É importante mencionar que a detecção dessa violação só foi possível devido a heurística de inferência de tipos, uma vez que seu tipo é conhecido na classe `Post` (linha 5) e propagado na chamada do método `get_first_likes_comments_and_people` (linha 8).

```
1  class Post                                          # from Module post_model
2    def first_update_complete_info_from_facebook(post_info, update_freq,
3        limit = 50, is_customer = false)
4      ...
5      vpost = self.class.select('id, fb_id, likes_count, comments_count,
6              updated_info, premium, international').find_by_fb_id(fb_id)
7      facebook = FacebookInfoRetriever.new
8      facebook.get_first_likes_comments_and_people(vpost, limit,
9                special_token.present?) do |info|
10     ...
11   end
12 end
13
14
15 class FacebookInfoRetriever       # from Module facebook_info_retriever
16   def get_first_likes_comments_and_people post, limit = 25,
17       special_token = false, &block
18     ...
19     likes_count = post['likes']
20     ...
21   end
22 end
```

Figura 8. Exemplo #2 – Violação arquitetural no `Dito Social`

Visualização arquitetural: `ArchRuby` produziu o modelo de alto nível da arquitetura implementada, conforme ilustrado na Figura 9 (que, por restrições de espaço, apresenta apenas um fragmento de tal modelo). É possível notar divergências (setas na cor laranja) dos módulos `dashboard_controller` (conforme visto no Exemplo #2), `report_model` e `dashboard_finder` com classes não pertencentes a módulos definidos. Também é possível notar a comunicação entre o módulo `post_model` e o módulo `facebook_info_retriever` (seta na cor

preta), porém o contrário, conforme visto no Exemplo #2, é destacado como uma divergência.

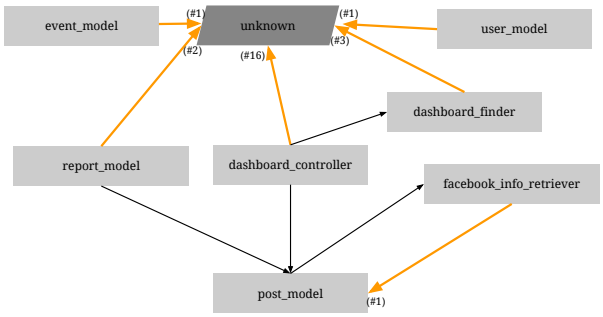

Figura 9. Fragmento do modelo arquitetural de alto nível no `Dito Social`

4.4 Tim Beta

Especificação de regras arquiteturais: O arquiteto desse segundo sistema especificou 43 módulos e 7 restrições arquiteturais. Um subconjunto das regras definidas pode ser visualizado na Figura 10. O módulo `models` implementa a camada de Modelo do padrão arquitetural MVC e, portanto, pode acessar módulos que auxiliam na persistência dos dados (linhas 3–7). O módulo `core` implementa as funcionalidades básicas do sistema e, portanto, deve acessar módulos que proveem os dados necessários para realizar suas tarefas (linhas 11–14).

```
1  models:
2    files:    'app/models/**/*.rb'
3    allowed: 'core, helpers, resque, logistica, dito_social_p,
4              postage_app, workers, facebook, devise, csv, olap,
5              twitter_oauth, datapoints, can_can, tim_points, linker,
6              twitter, rails, active_record, image_magick,
7              action_controller'
8
9  core:
10   files:    'app/core/**/*.rb'
11   allowed: 'models, helpers, facebook, twitter, foursquare, gmail,
12             mailers, instagram, dito_social_p, twitter_oauth,
13             contact_us, resque, sanitize, active_record, workers,
14             hoptoad'
```

Figura 10. Subconjunto de regras arquiteturais para `Tim Beta`

Conformidade arquitetural: `ArchRuby` foi capaz de detectar 22 violações, conforme reportado na Tabela 3. Um exemplo de violação detectada por `ArchRuby` no `Tim Beta` é discutido a seguir.

Tabela 3. Violações arquiteturais detectadas no `Tim Beta`

Módulo	Restrições	# Violações
`core`	`allowed: 'models, ...'`	6
`models`	`allowed: 'core, ...'`	16

Exemplo de violação #3: As funcionalidades vinculadas à rede social Orkut foram removidas do `Tim Beta` e, consequentemente, todo o respectivo código. No entanto, conforme ilustrado na Figura 11, a classe `User` – pertencente ao módulo `models` – acessa a classe `Core::Datapoints::Orkut` (linha 2), o que não é explicitamente permitido pela especificação de regras arquiteturais (linhas 3–7 da Figura 10).

```
1 def update_orkut_stats user_net = nil, app = nil     #from Module model
2   orkut_collector = Core::Datapoints::Orkut.new(
3                       user_net.access_token,
4                       user_net.access_secret,
5                       user_net.social_id
6                     )
7   orkut_datapoints = orkut_collector.collect
8 end
```

Figura 11. Exemplo #3 – Violação arquitetural no `Tim Beta`

Visualização arquitetural: `ArchRuby` produziu o modelo de alto nível da arquitetura implementada. Um fragmento desse modelo é mostrado na Figura 12. É possível notar divergências (setas na cor laranja) dos módulos `core` e `models` com classes não pertencentes a módulos definidos (conforme visto no Exemplo #3).

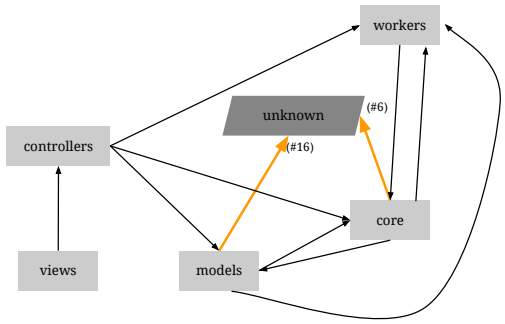

Figura 12. Fragmento do modelo arquitetural de alto nível no `Tim Beta`

4.5 Discussão

É importante destacar alguns pontos: (i) algumas vezes, 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` produziu uma 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 de os desenvolvedores estabelecerem comunicação com módulos não permitidos pela arquitetura; (iv) os arquitetos não tinham prévio conhecimento das violações arquiteturais detectadas e alegaram que tais violações impactam negativamente a manutenibilidade dos sistemas; e (v) os arquitetos alegaram dificuldade em analisar o modelo de alto nível da arquitetura à medida que o número de módulos definidos aumenta, indicando um problema de escalabilidade nesse modelo, que deve ser investigado como trabalho futuro.

4.6 Ameaças à Validade

Existem duas principais ameaças à validade do estudo. Em primeiro lugar, como é comum em avaliações empíricas em engenharia de software, não se pode generalizar os resultados para outros sistemas (validade externa). No entanto, foram utilizados dois sistemas reais de médio porte desenvolvidos com equipes diferentes. Em segundo lugar, foram entrevistados dois arquitetos (um por sistema) para especificar as restrições arquiteturais e verificar os resultados da conformidade e visualização arquitetural. Devido à classificação por pessoas, os resultados podem ter sido afetados por certa subjetividade (validade de construção). No entanto, é importante destacar que foram entrevistados os arquitetos que projetaram a arquitetura e que são responsáveis por sua manutenção e evolução.

5 Trabalhos Relacionados

No ecossistema Ruby, não se conhece uma solução de conformidade e visualização arquitetural como a proposta neste artigo. `DCLsuite` implementa uma abordagem de conformidade e reparação arquitetural para sistemas Java, na qual nossa solução 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, é voltada a linguagens dinamicamente tipadas (o que exigiu a definição e implementação de uma heurística de inferência de tipos), permite a especificação de restrições arquiteturais em arquivos YAML e provê uma visão de alto nível da arquitetura implementada, embora não contemple reparação arquitetural.

O modelo de alto nível gerado pela nossa solução é inspirado em `SAVE` [2], uma abordagem baseada em modelos de reflexão. Essa técnica compara (i) o modelo da arquitetura planejada, criado pelo arquiteto, com (ii) o modelo da arquitetura implementada, extraído do código fonte. Como resultado, é computado o modelo de reflexão que destaca divergências e ausências entre os dois modelos. O nosso modelo resultante é similar, embora seja computado por meio de regras arquiteturais textuais. Portanto, `ArchRuby` e `SAVE` compartilham o mesmo problema de escalabilidade. Diante disso, está sendo avaliado o uso de matrizes de dependência estrutural (DSMs), que podem representar uma solução mais escalável [6].

No estado da prática, existem ferramentas que têm o intuito de incrementar a qualidade de sistemas de software na linguagem Ruby através de técnicas de análise estática de código. Por exemplo, para auxiliar em revisões de código (`Code Climate`), verificar erros e regras de estilo (`Rubocop`, `LASER` e `ruby-lint`), detectar vulnerabilidades (`Brakeman`) e indicar padrões e boas práticas de orientação a objetos (`Pelusa`). `ArchRuby`, por sua vez, complementa tais ferramentas uma vez que auxilia o desenvolvedor na garantia de sua arquitetura planejada.

6 Conclusão

Erosão arquitetural é um problema recorrente no desenvolvimento de software. Desvios 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 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 ferramentas voltadas a propósitos arquiteturais. Para mitigar esse problema, este artigo apresentou uma solução que provê formas de controlar o processo de erosão arquitetural através da detecção de violações e da visualização do modelo de alto nível da arquitetura implementada. Como trabalho futuro, propõem-se a integração da solução proposta em IDEs para aumentar a usabilidade e a extensão para outras linguagens dinâmicas. Como resultado prático, `ArchRuby` foi integrada ao processo de desenvolvimento de software adotado pela Dito, 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. 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.
2. Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294, 2006.
3. 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.
4. David Lorge Parnas. Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.
5. 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.
6. Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005.
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*, 45(3):315–342, 2015.