Modular Specification of Architectural Constraints

Sândalo Bessa and Marco Tulio Valente
Department of Computer Science
Federal University of Minas Gerais, Brazil
E-mail: {sandalo,mtov}@dcc.ufmg.br

Ricardo Terra
Department of Computer Science
Federal University of Lavras, Brazil
E-mail: terra@dcc.ufla.br

***Abstract*—Due to the abstract nature of software architecture concepts, ensuring the correct implementation of architectural decisions is not a trivial task. Divergences between the planned architecture and source code may occur in the early stages of the software development, which denotes a phenomenon known as software architectural erosion. Architectural Conformance Checking techniques have been proposed to tackle the problem of divergences between the planned architecture and source code. Among such techniques, we can note the DCL language (Dependency Constraint Language), which is a domain-specific language that has interesting results in architectural conformance contexts. However, the current version of DCL has some limitations, such as lack of modularity and low degree of reuse, which may prevent its adoption in real software development scenarios. In this master dissertation, we extend DCL with a reusable, modular, and hierarchical specification. We evaluate the extended DCL—named DCL 2.0 by us—in a real-world system used by public State Government of Minas Gerais, Brazil. As main result, we were able to detect 771 architectural violations where 74% of them could only be detected due to the new violations types proposed in DCL 2.0.**

***Index Terms*—architecture conformance, modularity, hierarchical specification.**

I. Introdução

Dada a natureza abstrata do conceito de arquitetura de software, garantir a correta implementação de decisões arquiteturais não é uma tarefa trivial. Alguns trabalhos apontam que divergências entre código e arquitetura planejada podem ocorrer já em fases iniciais de desenvolvimento [14], [7]. Outros trabalhos apresentam cenários onde o software, ao longo de sua evolução, perde gradualmente o alinhamento entre código e arquitetura dando origem a um fenômeno conhecido como erosão arquitetural [15], [11], [9]. As violações arquiteturais podem ser classificadas em dois grupos: (i) *Violações de Estrutura*, as quais referem-se a inconsistências relacionadas a criação do componente, ou seja, essas violações ocorrem quando questões como localização, convenção de nomes e caracterização (estereotipagem) do componente divergem do modelo arquitetural planejada; e (ii) *Violações de Relacionamento*, as quais consistem em inconsistências na forma com que os componentes relacionam se entre si.

Algumas técnicas de verificação de conformidade arquitetural vêm sendo propostas para atacar o problema de divergência entre arquitetura planejada e código fonte [13], [12], [17]. DCL *(Dependency Constraint Language)* é uma linguagem declarativa que tem por objetivo controlar dependências entre módulos de sistemas orientados por objeto [17]. Embora seja expressiva, possua um bom nível de abstração e seja facilmente aplicável, DCL possui algumas características que podem impedir a sua rápida adoção em cenários reais de desenvolvimento de software, tais como especificação monolítica das regras arquiteturais, dificuldade no reúso de regras, não estabelecimento de relações hierárquicas entre os elementos arquiteturais e não suporte à rastreabilidade conceitual.

Este artigo, portanto, propõe uma extensão de DCL, denominada DCL 2.0, na qual foram implementados os seguintes conceitos e funcionalidades com o intuito de preencher lacunas da linguagem, e aumentar seu poder de expressão e reutilização: (i) *Modelagem Hierárquica* para permitir uma captura mais fiel dos modelos lógicos das arquiteturas de sistemas; (ii) *Novos Tipos de Restrições* para permitir um maior controle sobre violações estruturais, incluindo restrições que garantem a rastreabilidade entre conceitos essenciais e código fonte; (iii) *Reúso* por meio de uma modelagem totalmente independente e desacoplada do projeto alvo; (iv) *Modularização* por meio de especificações modulares que favorecem o reúso de componentes entre módulos via referências. (v) *Apoio Ferramental* que facilita atividades de edição, validação e visualização de arquiteturas.

Com a intenção de avaliar a linguagem proposta neste trabalho, realizou-se um estudo, no qual um sistema real de grande porte do governo do Estado de Minas Gerais passou por um processo de conformidade arquitetural utilizando a linguagem DCL 2.0. Os resultados demostraram que a abordagem foi capaz de capturar com precisão o modelo arquitetural do sistema, além de inibir a geração de violações arquiteturais no período analisado. Mais importante, foram detectadas 771 violações arquiteturais, sendo que 74% delas somente puderam ser detectadas devido aos novos tipos de violações propostas em DCL 2.0. Após o período do estudo de caso, a ferramenta foi incorporada ao processo de desenvolvimento da empresa mantenedora do sistema avaliado.

Este artigo está organizado como a seguir: A Seção II introduz a linguagem DCL. A Seção III apresenta a extensão da linguagem, denominada DCL 2.0, com um modelo de especificação modular, reutilizável e hierárquico. É também demonstrada uma ferramenta que implementa a solução proposta. A Seção IV reporta uma avaliação do uso de DCL 2.0 em uma grande empresa desenvolvedora de sistemas de software do Estado de Minas Gerais. Por fim, a Seção V apresenta trabalhos relacionados e a Seção VI conclui e enumera trabalhos futuros.

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.23

II. DCL

DCL é uma linguagem declarativa que tem por objetivo controlar dependências entre módulos de sistemas orientados por objetos [17], [18], [16]. A linguagem restringe o espectro de dependências permitidas entre os módulos de um sistema utilizando-se de uma abordagem baseada em regras de dependência entre componentes arquiteturais. Basicamente, cada regra tem sempre dois elementos e um tipo de relacionamento entre eles, como a seguinte:

$$M_A \textit{ cannot-extend } M_B$$

Uma restrição declarada na forma **M_A cannot-extend M_B** indica que classes pertencentes ao módulo M_A não podem estender classes pertencentes ao módulo M_B. De forma geral, todas as declarações de restrição seguem essa sintaxe. Certamente, uma das principais vantagens de DCL é sua sintaxe simples e autoexplicativa. Sendo assim, uma vez que o arquiteto do sistema conheça as características e restrições arquiteturais, aplicar a técnica de DCL sobre a arquitetura do sistema é um processo relativamente simples.

Para se fazer uso de DCL é necessário a execução de algumas etapas. Primeiramente, definem-se os componentes de alto nível que, em DCL, são chamados de módulos. Em seguida, define-se o mapeamento entre os módulos e código fonte, ou seja, nessa etapa deve-se explicitar qual parte do código será representado por determinado módulo. Por fim, definem-se as restrições entre os módulos.

Uma vez definida a especificação da arquitetura na linguagem DCL, ferramentas podem ler a especificação e confrontá-la com o código fonte a fim de detectar violações arquiteturais. Basicamente, uma especificação DCL é formada por módulos e restrições entre tais módulos. Um módulo é um conjunto de classes, como no seguinte exemplo:

```
1 module Math:       java.lang.Math
2 module Exception:  java.lang.RuntimeException,
3                    java.io.IOException
4 module JavaUtil:   java.util.*
5 module JavaSwing:  javax.swing.**
```

Listagem 1: Módulos DCL

Nesse exemplo, o módulo *Math* (linha 1) representa uma e somente uma classe, a classe *java.lang.Math*. Na linha 2, o módulo *Exception* representa duas classes, *java.lang.RuntimeException* e *java.io.IOException*. Na linha 3, o módulo *JavaUtil* representa todas as classes do pacote *java.util*. Na linha 4, o módulo *JavaSwing* refere-se a todos os tipos definidos no pacote *javax.swing* ou em quaisquer de seus sub-pacotes.

DCL classifica as violações em dois grupos: *divergência* e *ausência*. Uma *divergência* ocorre quando uma dependência existente no código-fonte viola uma restrição definida em DCL. Por outro lado, uma *ausência* ocorre quando o código-fonte não estabelece uma dependência que foi previamente definida na linguagem DCL. A fim de capturar divergências, DCL suporta a definição dos seguintes tipos de restrições entre os módulos:

- *only* M_A *can-**dep*** M_B: somente classes do módulo M_A podem depender de tipos definidos no modulo M_B.
- M_A *can-**dep**-only* M_B: classes do módulo M_A podem depender somente de tipos definidos no módulo M_B.
- M_A *cannot-**dep*** M_B: classes do módulo M_A não podem depender de tipos definidos em M_B.

Para capturar ausências, DCL suporta a definição da seguinte restrição:

- M_A *must-**dep*** M_B: classes do módulo M_A devem depender de tipos definidos no módulo M_B.

Nas declarações apresentadas, o literal ***dep*** refere-se ao tipo de dependência, a qual pode ser mais genérica como *depend* ou mais específica como *access*, *declare*, *create*, *extend*, *implement*, *throw* e *useannotation*. A Figura 1 resume a sintaxe da linguagem.

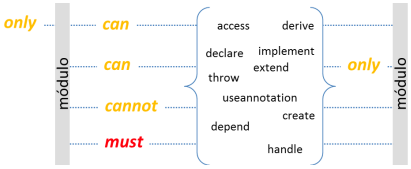

Figura 1: Restrições DCL (adaptada de [17])

DCLSuite[1] é um *plug-in* para a plataforma Eclipse que implementa a abordagem DCL. DCLSuite possui um editor para criar e editar as especificações DCL, as quais são armazenadas em um arquivo com a extensão *.dcl* que deve estar localizado na pasta raiz do projeto alvo. Na prática, DCLSuite lê este arquivo e verifica o código fonte com intuito de encontrar possíveis violações arquiteturais e, quando as encontra, elas são apresentadas como erros na aba de *Problemas* do Eclipse. Neste artigo, DCL será referenciada como DCL 1.0.

III. Solução Proposta: DCL 2.0

Este artigo propõe DCL 2.0, uma linguagem que estende DCL 1.0 com um modelo de especificação modular, reutilizável e hierárquico. Portanto, a principal motivação para a evolução de DCL 1.0 se deu no sentido de dotá-la de funcionalidades importantes para ser utilizada de forma efetiva em processos de verificação de conformidade arquitetural em cenários reais de desenvolvimento de software. A Tabela I mostra as funcionalidades impactadas pela proposta, a situação atual e a solução proposta em DCL 2.0, as quais são detalhadas nas subseções seguintes.

A. Especificação Hierárquica e Modular

DCL 2.0 provê o suporte à especificação hierárquica e modular, embora DCL 1.0 possua uma especificação *flat* em que cada módulo é definido de forma independente. A Figura 2 ilustra a estrutura lógica (a) e física (b) de um sistema fictício e, em seguida, é apresentada a sua especificação DCL 2.0. Na Listagem 2, é possível perceber um alinhamento entre os modelos lógico e físico (Figura 2-a e 2-b) e a especificação DCL 2.0, uma vez que os três modelos: diagrama, código e especificação DCL 2.0 têm natureza hierárquica. Por outro

[1] http://aserg.labsoft.dcc.ufmg.br/dclsuite/

	Funcionalidade	DCL 1.0	DCL 2.0
F1	Especificação	Monolítica e *flat*.	Tornar a especificação modular e hierárquica.
F2	Verificação	Limitada a artefatos Java. Verificações restritas ao controle de dependências.	Possibilitar a verificação de artefatos não Java e adicionar novos tipos de violações arquiteturais.
F3	Reusabilidade	Limitada.	Integrar a linguagem a ferramentas de gestão e configuração e desacoplar do sistema alvo.
F4	Visualização de arquitetura	Não há.	Implementar uma API (AST) para facilitar a visualização de arquitetura em qualquer modelo de visualização.
F5	Grau de Cobertura Arquitetural	Não há	Implementar funcionalidade que permita visualizar a porção do código fonte que está sendo coberto pela especificação DCL 2.0.

Tabela I: Comparação entre DCL 1.0 e DCL 2.0

lado, em DCL 1.0, cada módulo seria definido de forma isolada não havendo indicativo formal de que o mesmo possa fazer parte de outro módulo de mais alto nível e que também o mesmo módulo possa incluir módulos de níveis inferiores.

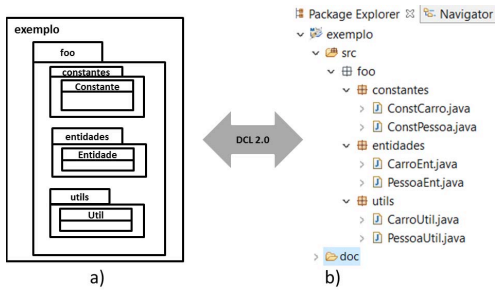

Figura 2: Modelagem de um sistema fictício - Modelo lógico e Modelo físico[2]

```
1  architecture exemplo {
2    foo {
3      entidades {
4        Entidade { matching: "{?}Ent";
5          restrictions {
6            can declare only platform.java.lang;
7          }
8        }
9      }
10     constantes {
11       Constante { matching: "{?}Const";
12         restrictions {
13           can declare only platform.java.lang;
14         }
15       }
16     }
17     utils {
18       Util { matching: "{?}Util";
19         restrictions {
20           can declare only platform.java.lang,
21           requires entidades.Entidade;
22         }
23       }
24     }
25   }
26   ignore "doc";
27 }
```

Listagem 2: Especificação DCL 2.0

É importante ressaltar que DCL 2.0 implementa uma especificação modular por meio de um mecanismo de referência cruzada, onde um elemento é definido em um arquivo de especificação e referenciado em outro. Na Listagem 2 pode-se observar que o componentes *plataform.java.lang* (linhas 6, 13 e 20) não foram definidos no arquivo corrente, mas apenas referenciados. Sendo assim, uma vantagem com DCL 2.0 é o reúso de componentes arquiteturais, i.e., *frameworks*, APIs, bibliotecas podem ter seus componentes especificados em DCL 2.0 apenas uma vez e reusados em outros projetos. Outra questão importante em relação a especificação de arquitetura é que DCL 2.0 exige mais formalidade ao se especificar arquiteturas. Como exemplo, pode-se observar a Linha 26 da Listagem 2, onde a palavra chave ***ignore*** é utilizada para informar que o componente *doc* não deve ser considerado como um artefato arquitetural; na prática artefatos relacionados a esse componente não serão verificados ou visualizados pelas ferramentas que leem a especificação DCL 2.0. Caso a palavra chave ***ignore*** não declarasse o componente *doc*, uma violação seria imediatamente gerada.

[2]Para facilitar a visualização, as figuras deste artigo estão disponíveis em: https://github.com/aserg-ufmg/sbcars2016.

B. *Verificação de Novos Tipos de Violação*

O modelo hierárquico da DCL 2.0 possibilitou a verificação de novas violações arquiteturais, classificadas neste artigo como *Violações de Estrutura*, que ocorrem quando algum elemento de código diverge do modelo de estrutura especificado pela hierarquia de componentes. Os tipos de tais violações são detalhados a seguir:

1) Componente Desconhecido: Essa violação ocorre quando um artefato é encontrado no código fonte, mas não se enquadra em componentes previstos no modelo da arquitetura planejada. A Figura 3-b apresenta duas violações devido à existência da classe *FooHelper* e do pacote *bar*, para os quais não existem componentes correspondentes no modelo de arquitetura apresentado na Figura 3-a.

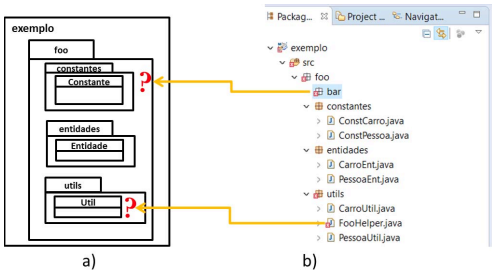

Figura 3: Violação do tipo *Componente Desconhecido*

2) Referência Desconhecida: Essa violação ocorre quando um elemento de código referenciar um segundo elemento que não está explicitamente especificado em DCL 2.0, seja como elemento interno ou externo ao sistema. A Figura 4 apresenta uma violação onde a classe *PessoaEnt* referencia a classe *java.rmi.Remote* que não possui componente correspondente.

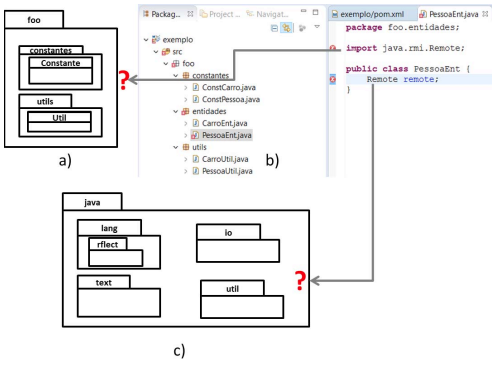

Figura 4: Violação do tipo *Referência Desconhecida*

3) Localização Incorreta: Essa violação ocorre quando um artefato de código é identificado como uma instância válida de um componente arquitetural, porém sua localização na arquitetura diverge da localização onde deveria ser implementado. A Figura 5 apresenta uma violação onde a classe de entidade *PessoaEnt* está localizada no componente *utils* (seta laranja), mas deveria estar no componente *entidades* (seta verde).

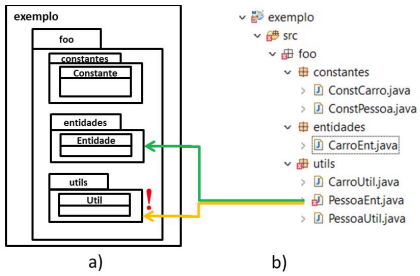

Figura 5: Violação do tipo *Localização Incorreta*

4) Ausência de Componente Dominante: DCL 1.0 não provê dependências entre elementos relacionados a um mesmo conceito. Por exemplo, para existir a classe `PessoaDAO` deve existir o conceito `Pessoa`. DCL 2.0 então propõe um novo tipo de restrição, representado pela palavra chave ***requires***, que tem como propósito controlar a rastreabilidade entre o modelo conceitual e elementos de código. Por exemplo, a linha 21 da Listagem 2 estabelece que "toda instância do componente *Util* deve possuir uma instância do componente *Entidade* relacionado ao mesmo conceito".

Uma violação do tipo *Ausência de Componente Dominante* ocorre quando não é possível detectar o componente dominante. A Figura 6-b apresenta uma violação em que *PessoaUtil* tem o conceito ausente, i.e., de acordo com as restrições, para que exista a classe *PessoaUtil* deve existir a classe *PessoaEnt*.

Na Figura 6-b pode-se observar que as partes sublinhadas dos nomes das classes representam os conceitos essenciais do sistema. Seguindo uma estratégia baseada em convenção de nomes, DCL 2.0 identifica essas partes essenciais (palavra chave ***matching***) e as utiliza como identificador para relacionar artefatos de um mesmo conceito.

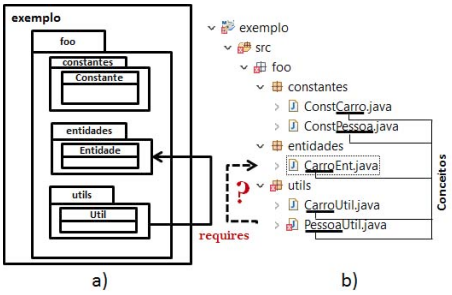

Figura 6: Violação do tipo *Ausência de Componente Dominante*

C. Reusabilidade Arquitetural

A especificação DCL 1.0 é parte integrante do sistema. Já em DCL 2.0, a especificação é desacoplada do código fonte do sistema alvo. Essa diferença possibilita que arquiteturas especificadas em DCL 2.0 possam ser reutilizadas. Além disso, essa mudança proporciona um maior controle da evolução das arquiteturas, uma vez que permite que as especificações possam ser geridas e distribuídas por ferramentas de gestão e configuração de software, de forma semelhante ao que é feito com código fonte.

D. Grau de Cobertura Arquitetural

DCL 2.0 provê uma métrica simples que indica o quanto do código fonte está sendo verificado. Devido à especificação hierárquica, todos os elementos de código deve ser verificados, salvo os declarados como *ignore*. Em linhas gerais, quanto menos elementos forem ignorados, maior será o Grau de Cobertura Arquitetural. Por exemplo, um código que possui 10 artefatos (classes, XML, JavaScript, etc.), mas somente 7 são considerados instância de componentes arquiteturais, diz-se que o *Grau de cobertura Arquitetural* é de 70%. Por padrão, arquivos binários são desconsiderados no cálculo.

E. Visualização Arquitetural

O entendimento da especificação arquitetural de um sistema é um importante fator para evitar a proliferação de violações arquiteturais [12]. Diante disso, o modelo hierárquico de DCL 2.0 permitiu três diferentes visualizações das arquiteturas especificadas: (i) forma textual, (ii) árvore de componentes, e (ii) artefatos do código seguido de seu respectivo nome do respectivo componente arquitetural entre « ». Um fragmento dessa última visualização pode ser observada na Figura 7.

Figura 7: Visualização da arquitetura e código

F. Ferramenta DCL 2.0

A ferramenta DCL 2.0 foi desenvolvida no intuito de verificar a aplicabilidade da solução proposta e apoiar a condução

da avaliação. A ferramenta é um *plug-in* para o IDE Eclipse que utiliza principalmente *Xtext*[3], um *framework* para desenvolvimento de linguagens de domínio específico. A ferramenta é integrada ao *framework* Maven, logo especificações ficam em repositórios remotos. Assim, o módulo de verificação da ferramenta identifica a especificação de arquitetura entre as dependências do projeto e a utiliza no processo de verificação de conformidade arquitetural.

Além de prover as funcionalidades essenciais – verificação de conformidade arquitetural e visualizações de alto nível da arquitetura –, a ferramenta também provê um conjunto de funcionalidades que possibilita a edição de especificações de arquiteturas na linguagem DCL 2.0. Por exemplo, a ferramenta disponibiliza um editor de texto adaptado à sintaxe da linguagem. Esse editor dispõe de funcionalidades como referência cruzada, recursos de auto completar e validação automática de erros de sintaxe.

IV. Avaliação da Solução Proposta

Nesta seção, é reportado um estudo de caso em um sistema de grande porte da PRODEMGE (Empresa de Tecnologia da Informação do Estado de Minas Gerais) em que foi aplicado um processo de conformidade arquitetural com DCL 2.0.

A. Questões de Pesquisa

Os objetivos principais se resumem em entender o fenômeno de erosão arquitetural no contexto da empresa (QP 1) e avaliar a linguagem DCL 2.0 (QP 2).

Questões de Pesquisa relacionadas à erosão arquitetural:

QP 1.1) Por que ocorrem violações arquiteturais?
QP 1.2) Como as violações são tratadas pela equipe?

Questões de Pesquisa relacionadas à DCL 2.0:

QP 2.1) DCL 2.0 pode ser usada para evitar violações arquiteturais?
QP 2.2) Os novos conceitos propostos por DCL 2.0 podem melhorar o processo de verificação arquitetural?
QP 2.3) A nova versão da ferramenta pode ser usada em processos reais de verificação de arquitetura?

B. Metodologia

Esta seção foi dividida em três partes: (i) escolha do sistema alvo, (ii) métricas e (iii) definição do processo de conformidade arquitetural.

1) Escolha do Sistema Alvo: Dentro da PRODEMGE, foi escolhido um sistema que atendesse a certos critérios: (i) sistema de médio a grande porte, uma vez em sistemas maiores tem mais propensão a proliferação de violações arquiteturais [19]; (ii) linguagem Java, embora os conceitos de DCL sejam aplicáveis a qualquer sistema orientado a objetos, a versão de DCL 1.0 possui implementação Java apenas; (iii) em fase de construção, uma vez que é pouco provável violações em sistemas estáveis ou sem desenvolvimento de novas funcionalidades; (iv) equipes médias, uma vez que equipes pequenas têm menor probabilidade de haver violação arquitetural, pois quem define a arquitetura é também quem implementa; e (v) equipes heterogêneas para evitar equipes compostas apenas de desenvolvedores pouco ou muito experientes.

Assim, escolheu-se o sistema *SSC-ADMIN*, que é o módulo de administração da solução de gestão de identidade digital do Estado de Minas Gerais. Ele é desenvolvido na linguagem Java e suas principais funcionalidades são Gestão de Unidades (órgãos), Gestão de Sistemas, Gestão de Usuários, Gestão de Recursos, Gestão de Perfil e Auditoria. A Tabela II sumariza as informações do sistema.

Métrica	Valor
Tempo de desenvolvimento	31 meses
Linhas de código	30.000 (aprox.)
Classes	187
Pacotes	39
Camadas	6
Mudanças - *Commits*	9.368
Conjunto de mudanças	1.454
Tipos de artefatos	18
Número de artefatos	602
Desenvolvedores	21

Tabela II: Métricas do sistema SSC-ADMIN

2) Definição do Processo: O processo de remoção de violações arquiteturais foi inspirado em [20] e adaptado de [14] para torná-lo mais iterativo e incremental, conforme ilustrado na Figura 8. Após as adaptações, o processo proposto para o estudo de caso incluiu as seguintes etapas que serão descritas a seguir: *1. Preparação da equipe, 2. Preparação do código fonte, 3. Formalização da arquitetura, 4. Verificação arquitetural e análise de violações, 5. Remoção das violações, 6. Geração de nova versão e 7. Acompanhamento e evolução.*

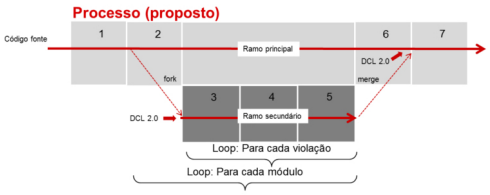

Figura 8: Processo proposto para o estudo de caso

1. Preparação da equipe: Essa etapa contemplou a explicação sobre DCL 2.0. Mais importante, definiu-se que a equipe do projeto deveria ser separada em dois grupos. O Grupo 1, composto por arquitetos, os quais seriam responsáveis por formalizar a arquitetura no padrão DCL 2.0 e refatorar o código para remover as violações na arquitetura e os demais desenvolvedores seriam o Grupo 2, responsável por continuar com o desenvolvimento normal do sistema.

2. Preparação do código fonte: Nessa fase foi extraída a versão atual e anteriores do código fonte do sistema para serem inseridas na base de dados previamente mencionada.

3. Formalização da Arquitetura: O Grupo 1 (arquitetos) deveriam especificar em DCL 2.0 a arquitetura dos seus sistemas e mapear os componentes arquiteturais para componentes de código. Para o mapeamento, os arquitetos deveriam se basear no modelo de estereótipos usados na arquitetura da empresa. Por exemplo, uma classe *PessoaCtr* (sufixo=Ctr) deveria ser classificada como um componente de «controle».

[3]https://eclipse.org/Xtext

4. Verificação arquitetural e análise de violações: Após a *formalização da arquitetura*, o módulo de verificação da ferramenta DCL deveria ser habilitado nas máquinas dos arquitetos do Grupo 1 para confrontar a arquitetura planejada e o código implementado para que as violações arquiteturais fossem reveladas. Como é um sistema de grande porte, a estratégia para validar a arquitetura deveria seguir o seguinte critério: habilitar módulo a módulo – em ordem do módulo menos dependente para o módulo mais dependente –, avaliar cada violação, e planejar mudanças no código, mudanças na arquitetura ou em ambos.

5. Remoção das violações: As violações deveriam ser corrigidas, documentadas e entregues no repositório remoto. O Grupo 1 foi instruído a repetir os Passos 2, 3, 4 e 5 para cada módulo individualmente até que todas as violações fossem corrigidas em todos os módulos.

6. Geração de nova versão: Após todas as remoções de violação terem sido feitas no ramo secundário, todas as alterações deveriam ser levadas para o ramo principal e uma nova versão do sistema gerada. Após a junção dos dois ramos, a ferramenta DCL deveria ser instalada nas máquinas do Grupo 2, a fim de coibir violações.

7. Acompanhamento e evolução: Após a geração da nova versão do sistema, seriam apresentados a arquitetura resultante e os resultados obtidos no estudo de caso. Os arquitetos deveriam acompanhar a arquitetura para motivos de adaptação, evolução ou correção.

C. Execução do Estudo de Caso

1) Preparação da equipe: DCL 2.0 foi explicado e a equipe foi dividida em dois grupos: o Grupo 1 com dois arquitetos e o Grupo 2 com oito desenvolvedores.

2) Preparação do código fonte: As ferramentas *IBM-RTC*[4] *(Rational Team Concert)* e *Maven*[5], adotadas na PRODEMGE, foram adotadas para controle de versão e gestão de dependência, respectivamente.

3) Formalização da arquitetura: Os arquitetos se basearam na documentação existente para realizar o mapeamento entre arquitetura e código. Essa tarefa necessitou de duas semanas (i.e., 80 homens/hora), pois em várias situações a documentação estava incompleta e as convenções estavam definidas informalmente entre os arquitetos. O *SSC-ADMIN* é organizado em seis camadas: *ssc-admin-comum, ssc-admin-dominio, ssc-admin-interfaces, ssc-admin-infraestrutura, ssc-admin-negocio e ssc-admin-web*. Cada camada lógica (visão lógica) do sistema corresponde a um projeto Eclipse específico (visão de implementação). A Figura 9 mostra um diagrama contendo todas as camadas, bem como o respectivo projeto no IDE Eclipse. De acordo com a metodologia, a camada *ssc-admin-comum* foi a primeira a ser formalizada, por ser a menos dependente de todas as camadas, e a *ssc-admin-web* foi a última, por ser a camada mais dependente.

Após a definição da hierarquia de componentes arquiteturais, os arquitetos realizaram o mapeamento entre arquitetura

[4]https://jazz.net/products/rational-team-concert
[5]https://maven.apache.org

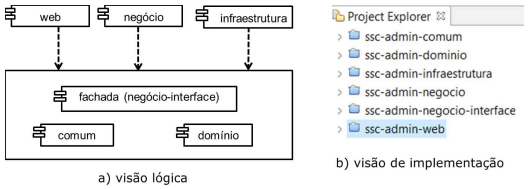

Figura 9: Componentes por camada.

e os artefatos de código utilizando *convenção de nomes* ao invés de anotações que é mais intrusiva e não auxilia na compreensão do sistema. Além de classes Java, o mapeamento também considerou arquivos JavaScript e XML, e.g., componentes *Page* (XML) requerem um componente *Controle* (Java). A Listagem 3 mostra a especificação final em linguagem DCL 2.0 para a camada de *domínio*, onde é possível observar o mapeamento de cada componente definido por meio da palavra reservada *matching*. Pode-se observar nas *Linhas 6 e 9*, os mapeamentos definidos para os componentes *Entity* e *AudityEntity*, respectivamente. De fato, essas declarações significam que qualquer artefato cujo nome termine com *VO* é uma instância do componente *Entity* e qualquer artefato cujo nome termine com *AudVO* é uma instância do componente *AudityEntity*. Por exemplo, a classe *UsuarioVO* é uma instância do componente *Entity* e a classe *AuditoriaAudVO* é um instância do componente *AudityEntity*. As especificações relativas às demais camadas estão publicamente disponíveis.[6]

```
1  architecture dominio {
2    ecosistema { matching: "xx.yyy.zzzzzzzz.{?}";
3      sistema { matching: "{?}";
4        entidades { matching: "entidades";
5          modulo { matching: "{?}";
6            Entity { matching: "{?}VO"; }
7            AudityEntity { matching: "{?}AudVO"; }
8          }
9          BaseEntity { matching: "{?}BaseVO"; }
10       }
11       enums { matching: "enums";
12         modulo { matching: "{?}";
13           Enum { matching: "{?}Enum"; }
14         }
15       }
16     }
17   }
18   ignore "xx", "yyy", "zzzzzzzz"...;
19 }
```

Listagem 3: Especificação da camada de domínio usando DCL 2.0

De maneira geral, quase todos os componentes de todas as camadas foram mapeados utilizando *convenção de nome* como estratégia. No entanto, em alguns poucos casos, foi utilizada a estratégia de mapeamento por *extensão de arquivo*. A Tabela III reporta o número de componentes e o grau de cobertura por camada. No estudo foi observado que o *Grau de cobertura* pode variar de módulo para módulo e a decisão de tornar uma arquitetura mais ou menos formal depende, principalmente, das características do módulo.

4) Verificação arquitetural e análise de violações: Nessa etapa, arquitetos adicionaram restrições do tipo relacionamento para cada módulo (camada). A Tabela IV reporta as camadas, o número de restrições e o número de referências formalizadas na linguagem DCL 2.0. Por exemplo, a Listagem 4 apresenta

[6]https://github.com/aserg-ufmg/sbcars2016

Projeto (camada)	Nº de componentes	Grau de cobertura	
		Por artefatos(%)	Por linhas(%)
web	25	82	97
negocio	21	63	94
infraestrutura	11	50	77
negocio-interface	14	53	77
dominio	10	67	93
comum	7	33	47
Total do sistema	**88**		

Tabela III: Métricas - Componentes e Grau de cobertura arquitetural por camada

o componente *Entity* (previamente mencionado na Listagem 3) após a inserção de restrições indicando que entidades devem ser serializáveis (linha 4) e que devem estender de *BaseVO* (linha 5). Ademais, como existiam referências a 147 componentes externos, como é o caso do *java.io.Serializable* na linha 4, os componentes relativos a *frameworks*, APIs e a própria plataforma Java foram especificados em um projeto específico no intuito de favorecer o reúso.

Componentes internos		
Projeto (camada)	Nº de restrições	Nº de referências
web	8	42
negocio	11	72
infraestrutura	11	31
negocio-interface	8	20
dominio	8	22
comum	2	6
TOTAL	**45**	**193**

Tabela IV: Restrições - Referências

```
1 ...
2   Entity { matching: "{?}VO";
3     restrictions{
4       must implement platform.java.io.Serializable;
5       must extend BaseVO message "";
6     }
7 }
8 ...
```

Listagem 4: Especificação da camada de domínio usando DCL 2.0

Na etapa anterior – Formalização da arquitetura – restrições de hierarquia já haviam sido definidas, uma vez que a própria especificação da hierarquia impõe, implicitamente, restrições à inserção de componentes à sua estrutura. Logo, artefatos de código que não se enquadrem no modelo hierárquico consistirá violação. Em princípio, observou-se um número alto de *violações estruturais* (91%), se comparado com o número de violações do tipo relacionamento (9%). Em uma análise, arquitetos detectaram duas deficiências: (i) elementos que não tinham correspondência com esteriótipos, e.g., a classe *EspecieFunctionalTest* foi considerada como *Componente Desconhecido*; e (ii) elementos que não se enquadravam pela não observância à convenção de nomes, e.g., a classe *MailService*, possuía caraterísticas de uma classe de persistência, portanto, foi renomeada para *NotificacaoDAO*. Diante do volume de violações na hierarquia de componentes, resolveu-se dividir o processo de conformidade arquitetural em duas etapas, antes e depois da remoção dessas violações.

5) Remoção das violações: Nessa etapa, devido ao alto número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*, as ações seguiram em dois sentidos: (i) ajustar a especificação da arquitetura, alterando a especificação em DCL 2.0; e (ii) ajustar os artefatos de código fonte, refatorando o código para que os artefatos se enquadrassem no modelo de componentes da arquitetura. Em seguida, iniciou-se o processo de remoção das demais violações, até que cerca de 80% das violações fossem removidas. Uma parte das violações não pôde ser removida devido ao risco que essas violações apresentavam para o projeto. Para evitar que essas violações impactassem o processo de desenvolvimento (sendo recorrentemente reportadas), a funcionalidade de filtrar violações da ferramenta DCL 2.0 foi utilizada. A maioria dessas violações eram *violações estruturais* e ocorreram em artefatos não Java, o que atesta a hipótese de que não basta controlar apenas uma linguagem.

6) Geração de nova versão: Após a remoção das violações, foi realizada a junção dos dois ramos de código. Todos os módulos então passaram por uma nova validação para capturar possíveis violações ocorridas no período. Foram removidas 11 violações. Após a refatoração, tanto arquitetura quanto código ganharam novas versões e a ferramenta DCL 2.0 foi então instalada nas máquinas da equipe do Grupo 2 a fim de garantir a conformidade arquitetural desse ponto em diante.

7) Acompanhamento e Evolução: Logo nas primeiras semanas, os arquitetos identificaram violações no código, o que não deveria estar ocorrendo já que DCL 2.0 tinha sido instalado nas máquinas dos desenvolvedores. Ao verificar o ocorrido, os arquitetos perceberam que esses desenvolvedores estavam desabilitando a ferramenta de validação no IDE Eclipse, por problemas de desempenho. Em média, a ferramenta levava entre um (menor módulo, *comum*) a oito segundos (maior módulo, *web*). Para evitar tal cenário, a ferramenta foi ajustada para possibilitar validação durante a compilação ou por demanda, além da incremental. Além disso, para garantir a integridade da arquitetura, a tarefa de verificação arquitetural com a ferramenta DCL 2.0 passou a ser obrigatória para o *builder* (membro da equipe responsável por gerar a versão), já que não havia garantia de que os desenvolvedores aplicariam a ferramenta no código entregue no repositório. Mais importante, a gestão definiu que o *builder* do projeto não aceitaria violações em novas versões. Essa fase durou 8 meses e 39 violações foram evitadas. Inclusive, durante esse período, houve mudanças de versão de alguns componentes arquiteturais, mas com relevância nula para estrutura geral da arquitetura.

D. Resultados

Os resultados apresentados nessa seção tiveram como base o histórico de desenvolvimento do sistema, período compreendido entre junho de 2013 e abril de 2016, 35 meses. O processo de especificação da arquitetura e remoção das violações do sistema teve início em 2 de julho de 2015 e durou até 27 de agosto de 2015, 58 dias. Nesse período, DCL 2.0 foi utilizada para revelar as violações ocorridas desde o início do sistema. Após a remoção das violações, a ferramenta de verificação de DCL 2.0 foi instalada nas máquinas da equipe. A etapa de acompanhamento teve inicio no mês de setembro de 2015 e se estendeu até abril de 2016, 8 meses.

Como discutido nas seções anteriores, o processo de conformidade arquitetural do sistema foi dividido em duas etapas

em função do número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*. A Figura 10 reporta as violações encontradas classificadas em três grupos: Grupo 1, composto por *Componente Desconhecido* e *Referência Desconhecida*; Grupo 2, contendo as demais violações de DCL 2.0; e Grupo 3, contendo as violações DCL 1.0. É possível observar um aumento nos demais tipos de violações depois da remoção das violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*, o que reforça a hipótese de que essas violações encobrem violações dos demais tipos.

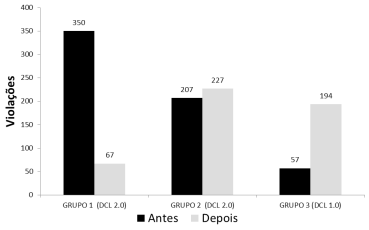

Figura 10: Antes e depois da remoção de violações

A Figura 11 reporta a quantidade de mudanças (*commits*) feitas no período, bem como a quantidade de mudanças com violações no período. Não há relação direta entre quantidade de mudanças (*commits*) e a quantidade de violações adicionadas, já que é possível observar períodos com muitas mudanças e poucas violações e vice-versa. Observa-se um padrão tendencioso de adição de violação ao longo do desenvolvimento. Particularmente em dois períodos, *06* e *07/2013*, o maior número de violações se deve ao fato de que nesse período foi adicionada a maior parte das classes Java no projeto.

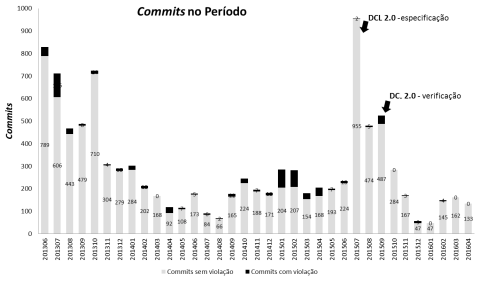

Figura 11: Quantidade de violações/mudanças no período

Nos meses de janeiro a abril de 2015, dois fatores contribuíram para o aumento das violações: (i) adição de novas funcionalidades para permitir a funcionalidades de solicitação e aprovação de alteração de dados no sistema; e (ii) adição de código para extensão de um *framework* JSON. Observando-se a Figura 12, em que é reportada a quantidade de funcionalidades adicionadas ao projeto, nota-se certa relação entre adição de novas funcionalidades e adição de violações. No entanto, fica evidente que o crescimento das duas métricas seguem proporções diferentes. Note que as violações foram acumuladas até a incorporação de DCL 2.0 no processo. Após esse período, as violações foram reiteradamente removidas. Logo, pode-se concluir que períodos com adição de novas funcionalidades geraram mais violações do que em períodos onde houve apenas manutenções de funcionalidades já existentes. Mais importante, pode-se também observar o impacto do processo de conformidade arquitetural a partir do mês 08/2015, uma vez que nesse período o número de violações cai a níveis de início do projeto.

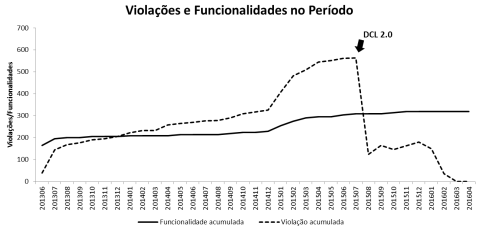

Figura 12: Quantidade de violações/funcionalidades no período

A Figura 13-a reporta uma relação de violações e mudanças com perfil de desenvolvedores. Observa-se que todos os perfis geraram violações no sistema. Observando-se a Figura 13-b em relação à Figura 13-a pode-se notar uma correlação entre conjunto de mudanças e violações, ou seja, quem muda mais o sistema, incorpora mais violações. No entanto, observando-se apenas as violações de *alta* complexidade, percebe-se que, o perfil *sênior* inseriu aproximadamente dez vezes mais violações que os demais perfis. Isso ocorre pois a esse perfil são delegadas as tarefas mais complexas e consequentemente com maior probabilidade de gerar impacto na arquitetura.

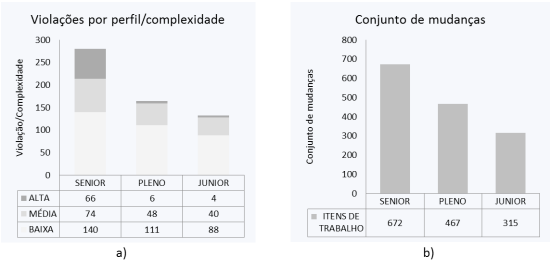

Figura 13: Violações por perfil e mudanças por perfil

E. Discussão

Esta seção responde às questões de pesquisa estabelecidas no início do estudo.

QP 1.1) Por que ocorrem violações arquiteturais? Os resultados indicam uma relação entre geração de violação e criação de novas funcionalidades, ou seja, mudanças que adicionam funcionalidades no sistema são mais propensas a incorporar violações no sistema do que mudanças que tenham por objetivo a manutenção de funcionalidade. Já segundo os membros da equipe, as causas são: (i) desconhecimento das convenções arquiteturais, devido a uma documentação extensa; e (ii) não evolução da arquitetura, pois os desenvolvedores, ao se depararem com uma limitação da arquitetura, construíam suas próprias soluções. Pressões de projeto foram citadas como fatores secundários, uma vez que os desenvolvedores

disseram que nunca fizeram uma escolha contrária às convenções arquiteturais quando a convenção era conhecida. No entanto, os desenvolvedores ressaltaram que a pressão de projeto pode ter dificultado a busca por mais informações sobre a arquitetura quando determinada convenção não era conhecida. Outro resultado interessante indica que, embora todos os perfis adicionem violação, violações mais complexas são adicionadas por desenvolvedores mais experientes.

QP 1.2) Como as violações são tratadas pela equipe? O histórico indica que violações eram removidas, portanto conclui-se que a equipe corrigia violações sem planejamento ou processo. No entanto, a maioria das violações removidas antes de DCL 2.0 era de baixa complexidade. Os desenvolvedores relataram que o risco de introduzir *bugs* no sistema foi o fator principal para a não remoção das violações de alta complexidade. Os desenvolvedores também alegaram que, não terem tido uma ferramenta que apontasse as violações arquiteturais desde o início do projeto, pode ser considerado um fator que dificultou a remoção das mesmas, uma vez que a inspeção manual é bastante difícil e custosa.

QP 2.1) DCL 2.0 pode ser usada para evitar violações arquiteturais? Após dez meses do início do processo de conformidade arquitetural, os resultados indicam que o número de violações se manteve controlado. Mesmo que tenha havido a adição de funcionalidades e manutenção nos meses avaliados, o sistema se manteve íntegro em relação aos conceitos arquiteturais.

QP 2.2) Os novos conceitos propostos por DCL 2.0 podem melhorar o processo de verificação arquitetural? (i) modelagem hierárquica e modular se mostrou bastante efetiva, permitindo que a arquitetura dos sistemas pudesse ser especificada de maneira fiel à estrutura dos mesmos, tanto do ponto de vista de decomposição, permitindo especificar vários módulos independentes, quanto do ponto de vista de níveis de abstração, onde foi possível especificar aspectos desde o nível de sistema até o nível de arquivo e classes; (ii) novos tipos de restrições para *Componente Desconhecido* e *Referência Desconhecida* foram importantes para explicitar o nível de informalidade no qual o sistema se encontrava; (iii) referência cruzada entre módulos foi importante para a prevenção de erros e para a celeridade no processo de especificação da arquitetura do sistema; (iv) reusabilidade apresenta um grande potencial, uma vez que as definições de arquitetura são completamente independentes dos sistemas alvo; (v) desacoplamento entre arquitetura e sistemas alvo trazida por DCL 2.0 possibilitou que conceitos e práticas de *Gestão e Configuração* fossem aplicadas à especificação de arquitetura, o que permitiu a fácil evolução da arquitetura.

QP 2.3) A nova versão da ferramenta pode ser usada em processos reais de verificação de arquitetura? Ao longo do estudo, a ferramenta foi adaptada para um cenário real de desenvolvimento, e.g., novos tipos de verificação devido ao desempenho. As visualizações e as funcionalidades de referência cruzada, verificação de erros e auto-completar fornecidas pelo editor da linguagem DCL 2.0 facilitaram bastante os trabalhos. Ademais, a integração com Maven foi importante para garantir a distribuição das versões das especificações arquiteturais. Enfim, a ferramenta foi de fato incorporada em um processo real de verificação arquitetural em uma empresa de grande porte.

V. Trabalhos Relacionados

Pesquisadores vêm propondo diferentes técnicas para, de alguma forma, combater erosões arquiteturais em sistemas de software. Esta seção é dividida em (i) abordagens de conformidade arquitetural e (ii) estudos empíricos relacionados.

Conformidade arquitetural: Modelos de reflexão comparam dois modelos (código fonte e modelo de alto nível do sistema) [12]. Uma etapa de mapeamento é necessária para que seja possível a comparação entre os dois modelos. Do processo de comparação entre os dois modelos, surge o modelo de reflexão que revela três tipos de relações entre os modelos: *convergência*, *divergência* e *ausência*. Vespucci implementa um modelo modular e hierárquico de especificação e controle de dependência baseado em blocos de especificação de arquitetura chamados *slices*, os quais são constituídos de pequenos blocos conceituais nomeados *ensembles* [11]. A técnica combina diagramas *box-and-line*, que são usados para definir a estrutura da arquitetura, com linguagem textual para fazer o mapeamento entre arquitetura e código. DCL 2.0, entretanto, além do controle de dependência entre módulos, provê restrições em relacionamentos estruturais.

Matriz de Dependências Estruturais (DSM) consiste em uma matriz quadrada, na qual a interseção entre linhas e colunas denota a relação entre classes de um sistema orientado por objetos [2], [15]. A estratégia de agrupamento permite aos arquitetos trabalhar com DSM de forma hierárquica. Mais importante, DSM podem ser combinadas com a definição de regras arquiteturais. Diferentemente de DCL2.0, em DSM não há especificação formal da arquitetura, o que dificulta o reúso.

.QL é uma linguagem baseada em SQL para auxiliar na localização de determinados elementos de código [5]. No contexto de conformidade arquitetural, a abordagem pode ser usada para gerar consultas que detectam padrões de codificação que não estão de acordo com a arquitetura desejada. Ao contrário de DCL 2.0, essa técnica possui baixo nível de abstração, o que dificulta a análise e compreensão da arquitetura. Teste de *Design* é uma técnica que verifica a conformidade arquitetural usando testes automatizados de maneira similar aos testes de unidade [3]. As regras são especificadas em baixo nível, o que dificulta a compreensão da arquitetura do sistema, o que não ocorre em DCL 2.0.

Framework-Specific Model Language (FSML), em contraste com outras abordagens, não se baseia apenas no controle de dependência, mas sim, em um rígido modelo de instanciação de *frameworks* que garante a conformidade arquitetural. Cada *FSML* contém informações e regras necessárias para se utilizar corretamente um determinado *framework*. A abordagem propõe o mapeamento de cada ponto de extensão do *framework* e de suas regras de instanciação de forma que essa definição possa ser usada para instanciar, entender e verificar a arquitetura de sistemas baseados em *frameworks* [1], [8].

ArchLint é uma abordagem de verificação de conformidade arquitetural que não requer a especificação manual das

regras [9], [10]. A técnica requer (a) histórico de versões do sistema e (b) um modelo documental de alto nível do sistema. Em uma estratégia combinada de análise estática código, análise do histórico de alterações e um conjunto de heurísticas, ArchLint classifica as dependências (ou a falta delas) como suspeitas baseando-se em hipóteses de frequência de uso e manutenções realizadas sobre essas dependências, ao contrário de DCL 2.0 em que restrições são explicitamente especificadas.

Estudos Empíricos: Rosik et al. [14] realizaram uma avaliação durante o desenvolvimento de um sistema de software comercial real, em que *Modelos de Reflexão* foram utilizados para detectar violações arquiteturais. Com principal resultado, os autores concluem que mesmo quando uma violação é detectada, ela não é removida do código-fonte. Brunet et al. [4] analisaram cinco anos de evolução da plataforma Eclipse IDE e concluem que a maioria das restrições estão relacionadas ao controle de extensão de classes ou interfaces, contrastando com a literatura onde a maioria dos trabalhos relacionados à validação de conformidade arquitetural baseiam-se em regras do tipo "A não pode depender de B".

Knodel et al. [7] introduzem o conceito de *verificação de conformidade construtiva* em Modelos de Reflexão, i.e., respostas instantâneas ao desenvolvedor em relação às violações, o que melhorou em 60% o número de violações inseridas. Em um trabalho similar, Knodel et al. [6] argumentam que ferramentas em estágio de protótipos não são adequadas para se fazer uma avaliação em cenários reais e geralmente exigem esforço adicional de adaptação. É ressaltada a necessidade de se ter cenários reais para se testar novas abordagens antes de torná-las disponíveis às fábricas de software.

VI. Conclusão

Este artigo descreve a linguagem DCL 2.0, a qual estende DCL 1.0 com o suporte à especificação arquitetural de forma modular, reutilizável e hierárquica. Além disso, o artigo apresenta a ferramenta DCL 2.0 que implementa a solução proposta. Os resultados de uma avaliação em um sistema real de grande porte comprovaram a aplicabilidade da extensão proposta, uma vez que 74% das violações somente puderam ser detectados devido às novas funcionalidades propostas em DCL 2.0. Além de bons resultados na detecção de violações existentes, os resultados também mostraram que DCL 2.0 foi importante no controle permanente de violações arquiteturais. Passados dez meses de implantação de DCL 2.0, o número de violações se manteve estável por um período, sendo zerado no fim do processo. Diante dos resultados apresentados, conclui-se que DCL 2.0 pode desempenhar um papel positivo no contexto de conformidade arquitetural.

Como trabalhos futuros, pode-se citar (i) novos tipos de visualização; (ii) especificações que suportam elementos de granularidade mais fina, como métodos e atributos; (iii) definição de regras baseadas em métricas para componentes, e.g., um certo componente que deve possuir baixo acoplamento; e (iv) criação de catálogo de arquiteturas de referência.

Agradecimentos

Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] Michał Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *Model Driven Engineering Languages and Systems*, pages 692–706. Springer, 2006.

[2] Carliss Y Baldwin and Kim B Clark. *Design rules: The power of modularity*. MIT Press, 1999.

[3] Joao Brunet, Dalton Guerrero, and Jorge Figueiredo. Design tests: An approach to programmatically check your code against design rules. In *31st International Conference on Software Engineering-Companion (ICSE)*, pages 255–258, 2009.

[4] Joao Brunet, Gail C Murphy, Dalton Serey, and Jose Figueiredo. Five years of software architecture checking: A case study of Eclipse. *IEEE Software*, 32(5):30–36, 2015.

[5] Oege De Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *7th IEEE International Working Conference on Source code Analysis and Manipulation (SCAM)*, pages 3–16, 2007.

[6] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Architecture compliance checking-experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52, 2008.

[7] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking-an experiment on support by live feedback. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 287–296, 2008.

[8] Herman Lee, Michal Antkiewicz, and Krzysztof Czarnecki. Towards a generic infrastructure for framework-specific integrated development environment extensions. In *2nd International Workshop on Domain-Specific Program Development*, pages 1–6, 2008.

[9] Cristiano Maffort, Marco Tulio Valente, Nicolas Anquetil, Andre Hora, and Mariza Bigonha. Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231, 2013.

[10] Cristiano Maffort, Marco Tulio Valente, Ricardo Terra, Mariza Bigonha, Nicolas Anquetil, and André Hora. Mining architectural violations from version history. *Empirical Software Engineering*, 21(3):854–895, 2016.

[11] Ralf Mitschke, Michael Eichberg, Mira Mezini, Alessandro Garcia, and Isela Macia. Modular specification and checking of structural dependencies. In *12th Annual International Conference on Aspect-oriented Software Development (AOSD)*, pages 85–96, 2013.

[12] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM Software Engineering Notes*, pages 18–28, 1995.

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

[14] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1):63–86, 2011.

[15] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, pages 167–176, 2005.

[16] Ricardo Terra and Marco Tulio Valente. Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 256–263, 2008.

[17] 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.

[18] Ricardo Terra and Marco Tulio Valente. Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. In *7th Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–8, 2010.

[19] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 335–340, 2012.

[20] R.T. Tvedt, M. Lindvall, and P. Costa. A process for software architecture evaluation using metrics. In *27th Annual NASA Goddard/IEEE - Software Engineering Workshop*, pages 191–196, 2002.