Em Busca de uma Abordagem de Conformidade Arquitetural para Arquitetura de Microsserviços

Elena A. Araujo, Elder Rodrigues Jr,
Arthur F. Pinto, Ricardo Terra

Universidade Federal de Lavras, Lavras, Brasil

{elena.araujo,fparthur}@posgrad.ufla.br,
elderjr@computacao.ufla.br, terra@dcc.ufla.br

Abstract. *Microservice architectures are composed of a set of independent microservices that execute well-defined functionalities, allowing each one to be developed in different programming languages and data management technologies. However, such heterogeneity implies in a harder verification process of communication between microservices and the architectural designs of each microservice. Thereupon, this paper proposes an architectural conformance approach to the microservice architecture. It consists of a multi-platform solution that allows the restriction of communication between microservices together with the verification of the architectural designs of each microservice.*

Resumo. *A arquitetura de microsserviços é composta por um conjunto de microsserviços independentes que executam funcionalidades bem definidas, permitindo que cada microsserviço seja desenvolvido em diferentes linguagens de programação e gerenciados por diferentes tecnologias de banco de dados. No entanto, tal heterogeneidade implica na dificuldade de verificação da comunicação entre os microsserviços e de cada um dos seus projetos arquiteturais. Diante desse cenário, o presente artigo propõe uma abordagem de conformidade arquitetural para arquitetura de microsserviços. Essa abordagem consiste em uma solução multiplataforma que permite a restrição da comunicação entre os microsserviços juntamente com a verificação dos seus projetos arquiteturais.*

1. Introdução

A arquitetura de microsserviços pode ser definida como uma abordagem para desenvolver uma única aplicação como uma suíte de serviços, cada um rodando em seu próprio processo e se comunicando através de mecanismos leves por meio de uma API (*Application Programming Interface*) que utiliza o protocolo HTTP (*Hypertext Transfer Protocol*) [Fowler and Lewis 2014, Alpers et al. 2015].

Os microsserviços são autônomos, o que indica que podem estar em diferentes servidores, ser implementados em diferentes linguagens de programação, utilizar diferentes *frameworks* e gerenciar sua própria base de dados. Tais características tornam esse padrão arquitetural muito heterogêneo, apresentando desafios para verificação da comunicação de tais microsserviços e de seus projetos arquiteturais. Assuma, por exemplo, uma aplicação composta de três microsserviços (`Ms1`, `Ms2` e `Ms3`). Os microsserviços `Ms1` e `Ms2` estão implementados na linguagem Java e hospedados nas cidades de Roma e

Toronto, respectivamente. Já o microsserviço `Ms3` está implementado na linguagem C# e hospedado em um servidor na cidade de Washington. O *problema* é como restringir as comunicações válidas entre tais microsserviços, bem como verificar cada um de seus projetos arquiteturais?

Diante desse cenário, este artigo propõe uma *solução* de conformidade arquitetural para a arquitetura de microsserviços. O objetivo é prover a arquitetos de software uma solução multiplataforma que permita restringir a comunicação entre os microsserviços e verificar os projetos arquiteturais de cada um deles, através de uma linguagem de restrição arquitetural denominada MsDCL (*Microservice* DCL).

A Figura 1 ilustra o resultado da verificação de conformidade sobre uma arquitetura envolvendo os três microsserviços mencionados anteriormente. Como pode ser observado, o microsserviço `Ms1` pode se comunicar apenas com o microsserviço `Ms2`. Como `Ms1` também comunica com `Ms3`, é reportada uma *violação de comunicação*. Já em relação ao projeto estrutural, na arquitetura planejada do `Ms1`, existe uma fábrica de DAO (*Data Access Object*), em que apenas esse módulo pode fazer a criação de objetos DAOs. Como uma classe da camada de Controle também cria objetos DAO, é reportada uma *violação do projeto estrutural*.

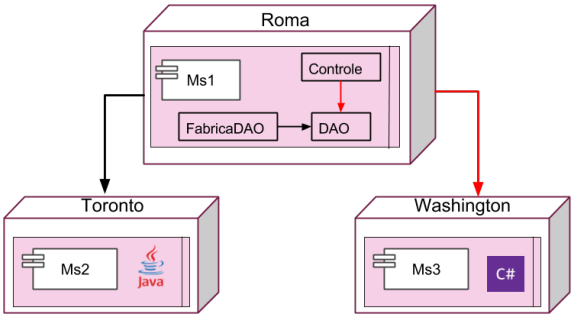

Figura 1. Verificação de conformidade sobre uma arquitetura envolvendo três microsserviços.

O restante deste trabalho está organizado da seguinte maneira. A Seção 2 apresenta conceitos fundamentais, mais precisamente a linguagem de restrição arquitetural DCL. A Seção 3 descreve a linguagem MsDCL, a qual estende a linguagem DCL para o contexto de microsserviços. A Seção 4 descreve a abordagem proposta. A Seção 5 apresenta os trabalhos relacionados e a Seção 6 conclui o trabalho.

2. Background

DCL é uma linguagem declarativa, de domínio específico e de verificação estática, que restringe o espectro de dependências aceitáveis em um sistema de software [Terra and Valente 2009]. DCL permite especificar restrições arquiteturais dos tipos *only can*, *can only*, *cannot* ou *must* com o intuito de detectar violações de divergência e ausência [Murphy et al. 1995]. Divergências ocorrem quando uma dependência presente no código fonte viola a arquitetura planejada, detectadas por meio das restrições *only can*, *can only* e *cannot*. Ausências são detectadas quando o código fonte não estabelece uma dependência prescrita pela arquitetura planejada, detectadas por meio da restrição *must*. DCL também permite a especificação da granularidade das dependências (*access*, *declare*, *create*, *extend*, *implement*, *throw* e *useannotation*). A Listagem 1 ilustra restrições DCL.

```
1 only Fabrica can-create DAO
2 Util can-only-depend $API, Util
3 Visao cannot-access Modelo
4 Entidade must-implement Serializable
```

Listagem 1. Exemplo de restrições DCL

Essas restrições expressam o seguinte significado: (linha 1) apenas classes definidas no módulo `Fabrica` podem criar objetos da classe `Produto`; (linha 2) classes definidas no módulo `Util` podem estabelecer dependências apenas com a API da linguagem de programação e com classes do seu próprio módulo; (linha 3) as devidas classes de Visão não podem acessar classes do Modelo; e (linha 4) todas as classes definidas no módulo Produto devem implementar a interface mencionada.

No exemplo acima é possível perceber a ocorrência de três tipos de violações de divergência (linhas 1, 2 e 3) e uma violação de ausência (linha 4). Na linha 1, foi especificado que apenas a classe `Fabrica` pode criar instâncias de `DAO`. Caso alguma outra classe crie instâncias de `DAO`, ocasionará uma *divergência*. A única violação de ausência está presente na linha 4, caso `Entidade` não implemente a interface mencionada.

A abordagem proposta neste artigo adota a linguagem DCL devido a sua (i) sintaxe simples e auto-explicativa, (ii) expressividade em lidar com o problema de erosão arquitetural e (iii) implementação de código aberto para sistemas Java.

3. MsDCL

Este artigo propõe MsDCL – uma extensão da previamente proposta linguagem DCL – para o contexto da Arquitetura de Microsserviços. Para representar esse novo tipo de arquitetura, a linguagem estabelece um novo tipo de dependência denominado *communicate*, além de definir restrições *only can*, *can only* e *cannot* para representar as divergências e restrições *must* para representar as ausências de comunicação. Considerando S_A e S_B como sendo um conjunto de microsserviços, o novo tipo de dependência é definido a seguir.

Divergências: Para capturar as divergências de comunicações, MsDCL suporta os seguintes tipos de restrições entre microsserviços:

- *Only* S_A *can-communicate* S_B: *apenas* os microsserviços em S_A poderão comunicar com microsserviços em S_B.
- S_A *can-communicate-only* S_B: os microsserviços em S_A poderão comunicar *apenas* com microsserviços em S_B.
- S_A *cannot-communicate* S_B: os microsserviços em S_A *não* poderão comunicar com microsserviços em S_B.

Ausências: Para capturar as ausências, MsDCL suporta apenas um tipo de restrição, definida como:

- S_A *must-communicate* S_B: os microsserviços em S_A *devem* (obrigatoriamente) comunicar com microsserviços em S_B.

Alertas: MsDCL considera também as comunicações entre microsserviços que não fazem parte do sistema. Esses alertas ocorrem quando um microsserviço devidamente especificado comunica com um microsserviço desconhecido pela arquitetura de microsserviços.

4. Abordagem de Conformidade Arquitetural para Arquitetura de Microsserviços

A Figura 2 ilustra a abordagem proposta para verificação da conformidade arquitetural para arquitetura de microsserviços, a qual consiste na principal contribuição deste estudo.

Essa abordagem possui como entrada uma *Aplicação* desenvolvida com uma suíte de microsserviços (Figura 2a) juntamente com sua *Especificação Arquitetural* (Figura 2b). A execução da abordagem é apoiada por meio do *Extrator e Verificador de Comunicação* (Figura 2c) e *Extrator e Verificador de Projeto Estrutural (Figura 2d)*. As saídas da execução da abordagem resultam em um conjunto de violações de *comunicação* (Figura 2e) e violações de *projeto estrutural* (Figura 2f).

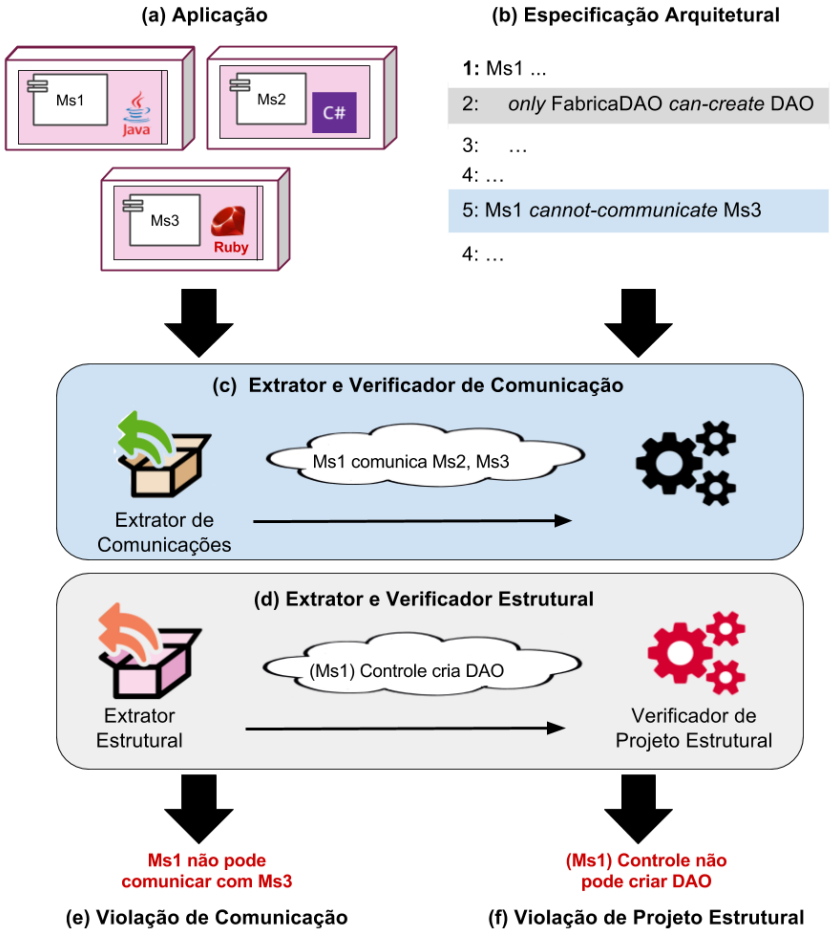

Figura 2. Abordagem de conformidade arquitetural para arquitetura de microsserviços

Todas essas etapas são automatizadas por uma ferramenta denominada MsDCLcheck[1], a qual constitui uma contribuição prática deste estudo. As próximas seções detalham cada uma das etapas.

[1]https://github.com/rterrabh/msdclcheck

4.1. Aplicação

Conforme ilustrado na Figura 2a, uma das entradas da abordagem consiste do conjunto de microsserviços que compõe a aplicação. Basicamente, a abordagem proposta requer o acesso ao repositório do código fonte de cada microsserviço.

4.2. Especificação Arquitetural

Conforme ilustrado na Figura 2b, a outra entrada da abordagem consiste na especificação arquitetural na linguagem proposta MsDCL. Como pode ser observado na Listagem 2, trata-se de um arquivo texto que define (a) cada microsserviço, (b) restrições estruturais internas a cada microsserviço e (c) comunicações válidas entre os microsserviços.

```
1  [id_ms1]: [url_ms1] ; [repositorio_ms1] ; [linguagem_ms1]                    (a)
2      Restricao DCL #1 para ms1                                                (b)
3      ...                                                                      (b)
4      Restricao DCL #N para ms1                                                (b)
5
6  ...
7
8  [id_msn]: [url_msn] ; [repositorio_msn] ; [linguagem_msn]                    (a)
9      Restricao DCL #1 para msn                                                (b)
10     ...                                                                      (b)
11     Restricao DCL #N para msn                                                (b)
12
13 Restricao MsDCL #1 entre os microsservicos previamente definidos             (c)
14 ...                                                                          (c)
15 Restricao MsDCL #N entre os microsservicos previamente definidos             (c)
```

Listagem 2. Arquivo de especificação arquitetural

Definição de um microsserviço: Como pode ser observado nas linhas 1 e 8, microsserviços são definidos por um *identificador*, *url* de acesso, *repositório* de código fonte e a *linguagem* em que foi implementado.

Definição das restrições estruturais de cada microsserviço: Como pode ser visto nas linhas 2 a 4 e nas linhas 9 a 11, as restrições estruturais de cada microsserviço iniciam-se indentadas logo após a definição de cada microsserviço. Essas restrições são especificadas na linguagem DCL (ver Seção 2).

Definição das restrições de comunicação: Como pode ser visto nas linhas 13 a 15, as restrições de comunicação entre os microsserviços vêm (não indentadas) logo após as definições completas dos microsserviços. Essas restrições são especificadas na linguagem MsDCL (ver Seção 3).

4.3. Conformidade de Comunicação

A análise da conformidade de comunicação é apoiada pelo Extrator e Verificador de Comunicação, que ao serem executados reportam um conjunto de *violações de comunicação*, conforme apresenta as Figuras 2c e 2e.

Extrator de Comunicação: O módulo extrator extrai do código fonte de cada um dos microsserviços as declarações de chamadas à outros microsserviços. A atual implementação de MsDCLcheck detecta as chamadas entre microsserviços a partir de busca textual. Basicamente, uma expressão regular obtém todas as URLs presentes no código fonte do microsserviço, as quais podem ou não representar microsserviços existentes. Porém, essa implementação pode levar a *falsos positivos* (e.g., não se pode garantir que há de fato a comunicação com microsserviço) e *falsos negativos* (e.g., URLs podem estar armazenadas em artefatos externos). Como trabalho futuro, implementar-se-á a detecção de tais chamadas a partir da AST (*Abstract Syntax Tree*) e a análise de artefatos externos. No exemplo ilustrado na Figura 2c, o microsserviço `Ms1` estabelece comunicação com os microsserviços `Ms2` e `Ms3`.

Verificador de Comunicação: Esse módulo analisa se as comunicações extraídas no passo anterior estão em conformidade com as definições da especificação arquitetural. Basicamente, esse módulo é responsável pela verificação da nova restrição de dependência incorporada na linguagem MsDCL denominada *communicate*, a qual é descrita na Seção 3. No exemplo ilustrado na Figura 1b, uma restrição define que `Ms1` não pode se comunicar com `Ms3` e, como o módulo anterior detectou tal comunicação, esse módulo reportará uma *violação de comunicação*.

4.4. Conformidade Estrutural:

A conformidade estrutural é composta pelo Extrator e Verificador Estrutural, que ao serem executados reportam um conjunto de *violações do projeto estrutural*, conforme apresenta as Figuras 2d e 2f.

Extrator Estrutural: A extração das dependências estruturais deve ser apoiada por ferramentas externas, implementadas para cada uma das diferentes linguagens que compõem a arquitetura de microsserviços. Em outras palavras, uma ferramenta deve extrair as dependências de um sistema em um conjunto de triplas no formato $[\texttt{source-class}, \texttt{dependency-type}, \texttt{target-class}]$. No exemplo ilustrado na Figura 2d, a partir de um extrator para a linguagem Java[2], foi extraída a seguinte tripla $[\texttt{ClienteController}, \texttt{create}, \texttt{ClienteDAO}]$, o que indica que o módulo `Controle` do microsserviço `Ms1` está criando objetos DAO. É importante ressaltar que extratores de dependências para outras linguagens já estão sendo desenvolvidos, por exemplo, C#[3].

Verificador Estrutural: Esse módulo analisa se dependências estabelecidas internamente à algum microsserviço – independente de sua linguagem – estão de acordo com o seu projeto arquitetural. Basicamente, esse módulo é responsável pela verificação das restrições de dependência de DCL, as quais são descritas na Seção 2. Embora a linguagem DCL seja independente de plataforma, a implementação existente (*dclcheck*) é acoplada à linguagem Java. Portanto, as dependências extraídas no módulo anterior são analisadas por um verificador DCL independente de plataforma[4], o qual constitui outra contribuição prática deste estudo.

[2] *JavaDepExtractor*, disponível em https://github.com/rterrabh/javadepextractor
[3] *CsDepExtractor*, disponível em https://github.com/rterrabh/csdepextractor
[4] *piDCLcheck*, disponível em https://github.com/rterrabh/pi-dclcheck

No exemplo ilustrado na Figura 1b, uma restrição define para o microsserviço `Ms1` que somente o módulo `FabricaDAO` pode criar objetos DAO e, como o módulo anterior detectou uma instanciação a partir da camada de Controle, esse módulo reportará uma *violação de projeto estrutural*.

5. Trabalhos Relacionados

Diversos trabalhos têm sido propostos para resolver os desafios emergentes no contexto de arquitetura de microsserviços. No entanto, conforme apresentado por Alshuqayaran et al. [Alshuqayran et al. 2016], as pesquisas destinam-se geralmente em abordagens que apresentam propostas para o estabelecimento da comunicação de microsserviços [Xu et al. 2016, Liu et al. 2016] e descoberta arquitetural [Granchelli et al. 2017b].

Xu et al. apresentam uma linguagem para programação de microsserviços baseada em modelos conceituais orientadas a agentes, denominada CAOPLE (*Caste-centrric Agent-Oriented Programming Language*) [Xu et al. 2016]. Essa linguagem foi proposta com o intuito de fornecer um mecanismo para o desenvolvimento paralelo, comunicação e implantação microsserviços. Para automatização da linguagem proposta, Liu et al. apresentam o ambiente de desenvolvimento denominado CIDE (*CAOPLE Integrated Development Environment*) [Liu et al. 2016]. Apesar de os autores fornecerem uma linguagem para estabelecer a comunicação entre os microsserviços, o trabalho não apresenta propostas para verificar se tais comunicações são realizadas conforme a arquitetura planejada.

Granchelli et al. propõem uma abordagem para descoberta arquitetural de aplicações orientadas a microsserviços a fim de resolver problemas como a complexidade de tal arquitetura [Granchelli et al. 2017b]. A abordagem proposta é capaz de extrair automaticamente a arquitetura de implantação da aplicação em um repositório GitHub por meio da ferramenta proposta denominada MicroART [Granchelli et al. 2017a]. Essa ferramenta é capaz de gerar modelos a partir da arquitetura física de aplicações orientadas a microsserviços. Apesar de os autores extraírem a arquitetura física da aplicação, eles não proveem formas de definir a arquitetura planejada da aplicação tampouco de verificar se as comunicações estabelecidas são realizadas corretamente.

É possível perceber que nenhum dos trabalhos apresentados acima foca em restringir a comunicação entre os microsserviços de uma arquitetura, tampouco verificar os projetos arquiteturais de cada um deles. Dessa maneira, o presente trabalho torna-se pioneiro nessa linha de pesquisa.

6. Conclusão

Este artigo descreve uma abordagem de conformidade arquitetural para arquitetura de microsserviços. Essa abordagem é definida pela MsDCL, uma linguagem de restrição arquitetural, estendida da linguagem DCL, voltada à arquitetura de microsserviços. Como maior contribuição, abordagem consiste em uma solução multiplataforma que permite restringir as comunicações desejadas entre os microsserviços bem como verificar os projetos arquiteturais de cada um deles.

A abordagem recebe como entrada um sistema desenvolvido com base nos princípios da arquitetura orientada a microsserviços e a especificação arquitetural de

cada microsserviço. Tal ferramenta extrai e verifica as comunicações realizadas entre os microsserviços e os projetos estruturais de cada microsserviço. Como saída, a ferramenta provê um conjunto de violações de comunicações entre os microsserviços e um conjunto de violações do projeto estrutural inerentes às arquiteturas internas de cada microsserviço.

Como trabalhos futuros, planeja-se implementar extratores de dependências estruturais para as linguagens JavaScript através do Node.js, PHP e Ruby. Planeja-se também acrescentar a construção `using` na linguagem MsDCL de forma a permitir especificar como um microsserviço deve se comunicar com outro microsserviço. Mais importante, planeja-se avaliar a abordagem proposta em sistemas reais baseados na arquitetura de microsserviços.

Agradecimentos: Este trabalho foi apoiado pela CNPq (projeto n^o 460401/2014-9), CAPES e FAPEMIG.

Referências

[Alpers et al. 2015] Alpers, S., Becker, C., Oberweis, A., and Schuster, T. (2015). Microservice based tool support for business process modelling. In *19th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 71–78.

[Alshuqayran et al. 2016] Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic mapping study in microservice architecture. In *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51.

[Fowler and Lewis 2014] Fowler, M. and Lewis, J. (2014). Microservices: a definition of this new architectural term. Acesso em: 29 jun. 2017. Disponível em: https://martinfowler.com/articles/microservices.html.

[Granchelli et al. 2017a] Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., and Di Salle, A. (2017a). MicroART: A software architecture recovery tool for maintaining microservice-based systems. In *2017 International Conference on Software Architecture (ICSA)*.

[Granchelli et al. 2017b] Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., and Di Salle, A. (2017b). Towards recovering the software architecture of microservice-based systems. In *2017 International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53.

[Liu et al. 2016] Liu, D., Zhu, H., Xu, C., Bayley, I., Lightfoot, D., Green, M., and Marshall, P. (2016). CIDE: An integrated development environment for microservices. In *13rd International Conference on Services Computing (SCC)*, pages 808–812.

[Murphy et al. 1995] Murphy, G. C., Notkin, D., and Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. *ACM SIGSOFT Software Engineering Notes*, 20(4):18–28.

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

[Xu et al. 2016] Xu, C., Zhu, H., Bayley, I., Lightfoot, D., Green, M., and Marshall, P. (2016). CAOPLE: A programming language for microservices SAAS. In *10th Symposium on Service-Oriented System Engineering (SOSE)*, pages 34–43.