
**ELENA AUGUSTA ARAUJO**

# UMA ABORDAGEM DE CONFORMIDADE ARQUITETURAL PARA ARQUITETURA DE MICROSSERVIÇOS

**LAVRAS – MG**

**2019**

**ELENA AUGUSTA ARAUJO**

# UMA ABORDAGEM DE CONFORMIDADE ARQUITETURAL PARA ARQUITETURA DE MICROSSERVIÇOS

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre.

Prof. Dr. Ricardo Terra Nunes Bueno Villela

Orientador

**LAVRAS – MG**

**2019**

**Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).**

Araujo, Elena Augusta.
Uma Abordagem de Conformidade Arquitetural para Arquitetura de Microsserviços / Elena Augusta Araujo. – 2019.
117 p.

Orientador(a): Ricardo Terra Nunes Bueno Villela.

Dissertação (mestrado acadêmico) – Universidade Federal de Lavras, 2019.
Bibliografia.

1. Arquitetura de Software. 2. Conformidade Arquitetural. 3. Arquitetura de Microsserviços. I. Villela, Ricardo Terra Nunes Bueno. II. Título.

**ELENA AUGUSTA ARAUJO**

# UMA ABORDAGEM DE CONFORMIDADE ARQUITETURAL PARA ARQUITETURA DE MICROSSERVIÇOS
# AN ARCHITECTURAL CONFORMANCE APPROACH TO MICROSERVICE ARCHITECTURE

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre.

APROVADA em 31 de Maio de 2019.

| | |
|---|---|
| Prof. Dr. Elder Cirilo | UFSJ |
| Prof. Dr. Rafael Serapilha Durelli | UFLA |

Prof. Dr. Ricardo Terra Nunes Bueno Villela
Orientador

**LAVRAS – MG**
**2019**

*Dedico esta dissertação à minha família, em especial à minha mãe Bernadete por ser minha inspiração de força e dedicação.*

## AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus por me proporcionar saúde, sabedoria e coragem para chegar até aqui.

Agradeço à minha famılia, em especial aos meus pais, Bernadete e Elson, aos meus irmãos Elder, Elaine, Elton, Euler e Everton e ao meu sobrinho Otávio, por todo amor, carinho e compreensão. Obrigada por estarem sempre presentes em minha vida e por compartilharem comigo mais essa vitória. Agradeço também aos meus avós, Maria e João, pelas fortes orações.

Agradeço ao Augusto, meu grande amor. Obrigada por seu meu porto seguro, meu melhor amigo, namorado e professor. Agradeço pela infinita paciência, dedicação e companheirismo.

Agradeço aos amigos que o mestrado me proporcionou, especialmente àqueles dos laboratórios de pesquisa LabMIDAS e LabRI. Obrigada pelo conhecimento, incentivo, conversas, alegrias e tristezas compartilhadas. Um agradecimento especial à Juliana, minha companheira de UFLA até altas madrugadas, e também à Vânia, exemplo de força e superação. Obrigada também às minhas amigas de república, Camila e Gabriela, a vocês, a minha eterna gratidão.

Agradeço aos meus colegas do PqES pelo apoio no desenvolvimento dessa dissertação, em especial ao Álvaro, Elder, Arthur e Carlos, a vocês, meu muito obrigada.

Agradeço ao professor Ricardo Terra pelo conhecimento, suporte e paciência durante o mestrado. Obrigada por além de ser um excelente orientador, ser exemplo de dedicação e comprometimento.

Agradeço à Universidade Federal de Lavras (UFLA), especialmente a todos do Departamento de Ciência da Computação, pela oportunidade de me tornar mestre nessa excelente instituição. Agradeço também à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa durante todo o período de realização deste mestrado.

Agradeço aos professores que compuseram a banca, Elder Cirilo e Rafael Serapilha Durelli, pelas sugestões construtivas durante todo o desenvolvimento desta dissertação de mestrado e também por participarem da banca de defesa.

Finalmente, agradeço a todos que contribuíram de alguma forma na elaboração desta dissertação e também em minha formação profissional.

*A persistência realiza o impossível*
*(Surama Jurdi).*

# RESUMO

Arquitetura de software é definida como um conjunto de decisões de projeto que possuem impacto na construção e evolução de sistemas de software. Essas decisões incluem como tais sistemas são estruturados em componentes e restrições de como esses componentes devem interagir. A arquitetura de microsserviços compreende a um estilo arquitetural que enfatiza a composição de um conjunto de microsserviços independentes que executam funcionalidades bem definidas, permitindo que cada microsserviço possa ser desenvolvido em diferentes linguagens de programação, possa utilizar diferentes *frameworks* e possa ser gerenciado 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, uma vez que, na prática, desvios em relação à arquitetura planejada podem ocorrer, tornando a base de código incompatível com a arquitetura planejada. Isso ocorre pois, durante o desenvolvimento e evolução de sistemas de software, anomalias são constantemente introduzidas no código-fonte. Essas anomalias consistem nas decisões que não são compatíveis com o modelo arquitetural especificado, tornando a arquitetura concreta inconsistente com a arquitetura planejada, ocasionando fenômenos conhecidos como violação e erosão arquitetural.

Diante desse cenário, esta dissertação de mestrado é centrada na proposta de uma abordagem de conformidade arquitetural específica para a arquitetura de microsserviços. Para isso, (i) definiu-se uma linguagem de restrição arquitetural, denominada DCL$^+$ – adaptada da linguagem DCL (*Dependency Constraint Language*) – a fim de restringir o espectro das comunicações e dependências aceitáveis entre os microsserviços e de cada um de seus projetos arquiteturais; (ii) propôs-se uma solução multiplataforma que permita restringir a comunicação entre os microsserviços e verificar os projetos arquiteturais de cada um deles; (iii) projetou-se DCL$^+$*check*, uma ferramenta que implementa a solução proposta; (iv) avaliou-se a solução proposta em uma aplicação real de médio porte composta por onze microsserviços, desenvolvidos em duas linguagens distintas (JavaScript e Java), em que foram detectadas 16 violações de comunicação e 171 violações do projeto estrutural.

As violações de comunicação, em geral, ocorreram devido à falta de conhecimento dos desenvolvedores sobre as restrições de comunicação entre os módulos do sistema orquestrador e demais microsserviços, bem como evolução de dois microsserviços. No que compete às violações do projeto estrutural, as violações foram ocasionadas devido à falta de conhecimento por parte dos desenvolvedores sobre os conceitos de arquitetura de software, especificamente na correta aplicação do *framework* Spring MVC e dependências do módulo `Util`.



**Palavras-chave:** Arquitetura de Microsserviços; Conformidade Arquitetural.

## ABSTRACT

Software architecture is defined as a set of design decisions that has impact on the construction and evolution of software systems. Such decisions include how those systems are structured in components and the constraints on how these components should interact on. The microservice architecture comprises an architectural style that emphasizes the composition of a set of microservices that execute well defined functionalities, thus microservices can be developed in different programming languages, running in different frameworks, and can be managed by different database technologies.

However, such heterogeneity implies in the difficulty of verifying the communication between the microservices and each of their architectural projects, since, in practice, deviations from the planned architecture may occur, making the code base incompatible with the planned architecture. During the development and evolution of software systems, anomalies are constantly introduced in the source code. These anomalies consist of decisions that are not compatible with the specified architectural model, making the concrete architecture inconsistent with the planned architecture, causing phenomena known as violation and architectural erosion.

Given this scenario, this dissertation is centered on the proposal of a specific architectural compliance approach for the microservice architecture. For this purpose, (i) an architectural constraint language, called DCL$^+$ – adapted from the DCL (Dependency Constraint Language) language – has been defined in order to restrict the spectrum of acceptable communications and dependencies among the microservices and each one of their architectural designs; (ii) a multiplatform solution was proposed that would allow to limit the communication between the microservices and verify the architectural projects of each one of them; (iii) we designed DCL$^+$*check*, a tool that implements the proposed solution; (iv) the proposed solution was evaluated in a real large application composed of eleven microservices, developed in two different languages (JavaScript and Java), in which 16 communication violations and 171 violations of the structural design were detected.

The communication violations occurred in general due to the lack of knowledge of the developers about the restrictions of communication among the modules of the orchestrator system and other microservices, as well as the evolution of two microservices. Regarding the violations of the structural project, they were caused due to the lack of knowledge of the developers about the concepts of software architecture, specifically in the correct application of the Spring MVC framework and dependencies of the `Util` module.



**Keywords:** Microservice Architecture; Architecture Conformance.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE SIGLAS

API – *Application Programming Interface*

AST – *Abstract Syntax Tree*

AWS – *Amazon Web Services*

CORBA – *Common Object Request Broker Architecture*

DAO – *Data Access Object*

DCL – *Dependency Constraint Language*

DSM – *Dependency Structure Matrixes*

HTTP – *HyperText Transmission Protocol*

JPA – *Java Persistence API*

LDM – *Latix Dependency Manager*

MDD – *Model Driven Development*

REST – *REpresentational State Transfer*

RMI – *Remote Method Invocation*

SOA – *Service-Oriented Architecture*

URL – *Uniform Resource Locator*

**SUMÁRIO**

# 1 INTRODUÇÃO

Este capítulo visa descrever os principais conceitos relacionados à pesquisa, apresentando os direcionadores do trabalho em termos de sua contextualização e motivação (Seção 1.1), justificativa (Seção 1.2), solução proposta (Seção 1.3) e principais contribuições (Seção 1.4). Tem-se ainda uma visão geral da estrutura da dissertação (Seção 1.5) e dos materiais relacionados (Seção 1.6).

## 1.1 Contextualização e Motivação

Arquitetura de software é comumente definida como um conjunto de decisões de projeto que possuem impacto na construção e evolução de sistemas de software (PERRY; WOLF, 1992). Essas decisões incluem como tais sistemas são estruturados em componentes e sobre quais restrições esses componentes devem interagir (FOWLER, 2002). O projeto da arquitetura de um sistema deve ser bem definido, uma vez que provê benefícios como a manutenibilidade, reusabilidade, escalabilidade e portabilidade (PRESSMAN; MAXIM, 2011).

Para auxiliar no projeto de sistemas que atenda aos princípios da arquitetura de software, paradigmas arquiteturais que utilizam serviços como elementos fundamentais de desenvolvimento têm sido propostos (PAPAZOGLOU et al., 2008; GRANCHELLI et al., 2017a; RODRÍGUEZ; DÍAZ-PACE; SORIA, 2018). Os serviços consistem em entidades computacionais autônomas e independentes de plataforma que podem ser descritos, publicados, descobertos e integrados (MACLENNAN; BELLE, 2014).

A arquitetura de microsserviços surgiu em 2015 como uma promessa de estilo arquitetural que visa promover a reutilização e facilitar a manutenção de aplicações por meio de serviços (FOWLER; LEWIS, 2014; ENGEL et al., 2018). Esse estilo arquitetural compreende à uma variação da Arquitetura Orientada a Serviços (*Service-Oriented Architecture* ou SOA) ao nível de desenvolvimento e implantação de aplicações distribuídas (ZIMMERMANN, 2016). Newman re-

força ainda que a arquitetura de microsserviços surgiu com o objetivo de melhor compreender e aplicar as práticas propostas por SOA, adotado por empresas como Netflix e Amazon (NEWMAN, 2015).

Fowler e Lewis definem a arquitetura de microsserviços como uma abordagem para desenvolver uma única aplicação como uma suíte de serviços independentes, cada um rodando em seu próprio processo e se comunicando por meio de mecanismos leves através de uma API que utiliza o protocolo HTTP (FOWLER; LEWIS, 2014). Nessa arquitetura, a aplicação deve ser dividida em serviços pequenos – microsserviços que implementam funcionalidades específicas – e se comunicam através de chamadas de rede por meio de interfaces bem definidas. Isto garante baixo acoplamento e alta coesão, permitindo agilidade, flexibilidade, escalabilidade e reusabilidade (FRANCESCO, 2017).

Os microsserviços são entidades autônomas e distribuídas, responsáveis por empregar paradigmas modernos de engenharia de software e tecnologias da *Web* (ZIMMERMANN, 2016). Eles são caracterizados por poderem estar hospedados em diferentes servidores, ser implementados em diferentes linguagens de programação, utilizar diferentes *frameworks* ou ser gerenciados por persistência de dados poliglota.

Neste contexto, é necessário estabelecer como tais serviços serão compostos a fim de fornecerem suas respectivas funcionalidades. Os conceitos mais abrangentes de fazer isso é por meio de orquestração e coreografia (PEDRAZA; ESTUBLIER, 2009; BUTZIN; GOLATOWSKI; TIMMERMANN, 2016). Na orquestração, uma única entidade conhecida como orquestrador é responsável por gerenciar e coordenar todo o fluxo de execução de solicitações à microsserviços (NANDA; CHANDRA; SARKAR, 2004). Sendo assim, toda a lógica da aplicação concentra-se no orquestrador e apenas ele conhece todos os microsserviços da aplicação. Por outro lado, em uma coreografia, cada serviço envolvido sabe exatamente quando executar suas operações e com quem interagir, sem que haja um controle centrali-

zado (BARKER; WALTON; ROBERTSON, 2009). Nesse sentido, as coreografias podem ser vistas como um acordo entre um conjunto de serviços sobre como uma determinada colaboração deve ocorrer (LEITE et al., 2013).

As características descritas pela arquitetura de microsserviços tornam esse padrão arquitetural muito heterogêneo, apresentando desafios para verificação da comunicação de tais microsserviços e de seus projetos arquiteturais, uma vez que, na prática, desvios em relação à arquitetura planejada são comuns (MAYER; WEINREICH, 2018). Isso se deve ao fato de durante o desenvolvimento e evolução de qualquer aplicação, anomalias arquiteturais são frequentemente introduzidas no código-fonte. Essas anomalias consistem nas decisões que não são compatíveis com o modelo arquitetural especificado, tornando a base de código atual inconsistente com a documentação existente, ocasionando fenômenos conhecidos como violação e erosão arquitetural (KAZMAN; CARRIÈRE, 1999; PERRY; WOLF, 1992; KNODEL; POPESCU, 2007).

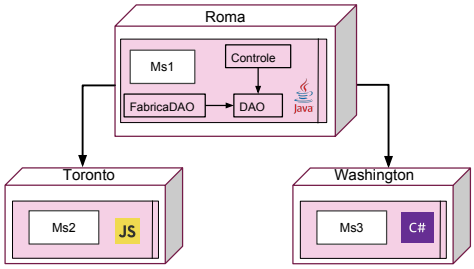
## 1.2 Justificativa

O processo de erosão arquitetural faz com que os benefícios, como manutenibilidade, reusabilidade, escalabilidade e portabilidade, proporcionados por um projeto arquitetural sejam anulados (TERRA; VALENTE, 2009). Assim, verificar a conformidade arquitetural é uma importante técnica para evitar o processo de erosão (KAZMAN; CARRIÈRE, 1999). Apesar de diversas pesquisas na área de arquitetura de software apresentarem propostas de soluções para conformidade arquitetural (MURPHY; NOTKIN; SULLIVAN, 1995; SANGAL et al., 2005; VERBAERE; GODFREY; GIRBA, 2008; TERRA; VALENTE, 2009), ainda não se têm conhecimento de estudos voltados à arquitetura de microsserviços.

Considerando uma aplicação desenvolvida na arquitetura de microsserviços e as características apresentadas para tal estilo arquitetural, assuma uma aplicação composta de três microsserviços (`Ms1`, `Ms2` e `Ms3`), conforme

apresenta a Figura 1.1. O microsserviço `Ms1` é implementado na linguagem Java e hospedado em Roma, `Ms2` é implementado na linguagem JavaScript e hospedados em Toronto. Já o microsserviço `Ms3` está implementado na linguagem C# e hospedado em um servidor na cidade de Washington. Como pode ser observado, o microsserviço `Ms1` comunica-se com os microsserviços `Ms2` e `Ms3`. Já em relação ao projeto estrutural, em `Ms1`, existe um módulo *FabricaDAO* (*Data Access Object*) e um módulo *Controle* criando objetos do tipo DAO. Nesse contexto, o *problema* é como restringir as comunicações válidas entre os microsserviços de uma aplicação?

Figura 1.1 – Exemplo de uma arquitetura de comunicação envolvendo três microsserviços.

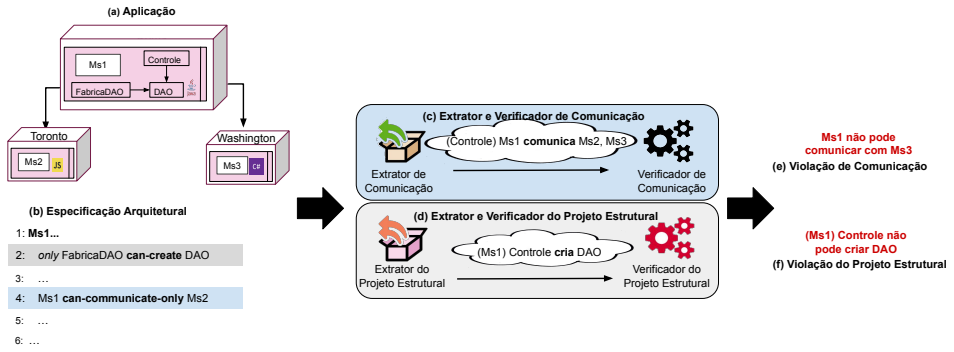



## 1.3 Solução Proposta

Diante do cenário apresentado anteriormente, esta dissertação de mestrado 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 `DCL`$^{+}$. Para isso, a Figura 1.2 ilustra o funcionamento da abordagem proposta para a verificação da conformidade arquitetural do exemplo apresentado na Figura 1.1.

Inicialmente, a abordagem requer o acesso à uma *Aplicação* desenvolvida com uma suíte de microsserviços (Figura 1.2a) juntamente com sua *Especificação*

Figura 1.2 – Abordagem proposta para verificação da conformidade arquitetural de sistemas sob uma arquitetura de microsserviços.




*Arquitetural* (Figura 1.2b). As informações sobre as comunicações devem ser extraídas e analisadas pelo *Extrator e Verificador de Comunicação* (Figura 1.2c) que reportará um conjunto de violações de *comunicação* (Figura 1.2e). Já em relação ao projeto estrutural de cada microsserviço, tais informações devem ser extraídas e verificadas pelo *Extrator e Verificador de Projeto Estrutural (Figura 1.2d)* que ao final reportará um conjunto de violações do *projeto estrutural* (Figura 1.2f).

Para o exemplo apresentado na Figura 1.1, na Especificação Arquitetural, uma restrição de projeto estrutural para o microsserviço `Ms1` define que **apenas** o módulo *FabricaDAO* pode criar objetos do tipo `DAO` (linha 2). Como uma classe da camada de Controle também cria objetos DAO, é reportada uma *violação do projeto estrutural*. Já para as comunicações válidas de `Ms1`, uma restrição define que os módulos de `Ms1` **não** podem comunicar-se com o microsserviço `Ms3` (linha 4). Como `Ms1` podem comunicar **apenas** com `Ms3`, é reportada uma *violação de comunicação*.

Embora a solução proposta possa ser facilmente adaptadas a outras arquiteturas que envolvam comunicação – tais como SOA, CORBA e RMI – esta dissertação de mestrado foca exclusivamente em microsserviços.

## 1.4 Contribuições

Esta dissertação contém as seguintes contribuições:

1. Uma linguagem de restrição arquitetural denominada DCL$^+$ proposta para representar dependências de comunicação entre microsserviços de uma arquitetura por meio da dependência `communicate`.

2. Uma solução de conformidade arquitetural denominada DCL$^+$ que provê a arquitetos de software uma solução multiplataforma que permite restringir a comunicação entre microsserviços e verificar os projetos arquiteturais de cada um deles.

3. Automatização da solução proposta por meio da ferramenta DCL$^+$*check*. Foram então implementados extratores de comunicação e extratores de dependência estrutural, além dos respectivos verificadores independentes de plataforma.

4. Uma demonstração da aplicabilidade da solução proposta em uma aplicação com cinco microsserviços – desenvolvidos nas linguagens Java e C# – e organizados por meio de coreografia.

5. Uma avaliação da solução proposta em um estudo de caso real sobre uma aplicação financeira de médio porte, focada na gestão de vendas e conciliação bancária. A aplicação avaliada é composta por onze microsserviços desenvolvidos em duas linguagens distintas e organizados por meio de orquestração. Nessa avaliação, foram extraídas 101 comunicações, sendo reportadas 16 violações de comunicação e 162 violações do projeto estrutural.

Vale ressaltar que, até o momento do desenvolvimento desta dissertação de mestrado, não foram encontrados na literatura estudos que proveem a arquitetos de

software soluções para analisar conformidade de comunicação entre microsserviços de uma aplicação, bem como verificar cada um de seus projetos arquiteturais. Dessa maneira, esta dissertação de mestrado torna-se pioneira nessa linha de pesquisa, tornando-se uma referência importante para futuros estudos na área.

## 1.5 Estrutura da Dissertação

O restante desta dissertação de mestrado está organizado como a seguir. O Capítulo 2 introduz os principais conceitos envolvidos, i.e., arquitetura de software, conformidade arquitetural e arquitetura de microsserviços. O Capítulo 3 descreve a abordagem de conformidade arquitetural proposta, uma solução multi-plataforma que permite restringir a comunicação entre microsserviços bem como verificar o projeto estrutural de cada um deles. O Capítulo 4 apresenta DCL$^{+}$*check*, a ferramenta desenvolvida para apoiar a solução proposta. O Capítulo 5 aponta os resultados de uma avaliação de uma aplicação real de médio porte composta por 11 microsserviços organizados por meio de orquestração. O Capítulo 6 discute os trabalhos relacionados. Por fim, o Capítulo 7 apresenta as considerações finais desta dissertação de mestrado, incluindo as contribuições e trabalhos futuros.

## 1.6 Artigos Relacionados

Esta dissertação gerou os seguintes artigos relacionados:

- **Elena A. Araujo**, Elder Rodrigues Jr., Arthur F. Pinto e Ricardo Terra. (2017). Em busca de uma abordagem de conformidade arquitetural para arquitetura de microsserviços. In: 5th *Workshop* de Visualização, Evolução e Manutenção de Software (VEM), páginas 68-75.

- **Elena A. Araujo**, Alvaro Espíndola Álvaro, Vinicius Garcia e Ricardo Terra. (2019). *Applying a Multi-plataform Architectural Conformance So-*

*lution in a Real-world Microservice-based System*. Service Oriented Computing and Applications, páginas 1-13. (a ser submetido).

- **Elena A. Araujo**, Juliana Botelho, Rafael S. Durelli e Ricardo Terra. (2020). Uma Abordagem Híbrida para Visualização da Comunicação entre Microsserviços. Relatório técnico.

# 2 REFERENCIAL TEÓRICO

Este capítulo apresenta os conceitos fundamentais necessários para o entendimento desta dissertação de mestrado. A Seção 2.1 introduz as definições adotadas sobre arquitetura de software, relacionadas à conformidade arquitetural e suas técnicas, objeto principal de estudo do presente trabalho. A Seção 2.2 descreve sobre a arquitetura de microsserviços, estilo arquitetural abordado. Por fim, a Seção 2.3 discute as considerações finais do capítulo.

## 2.1 Conformidade Arquitetural

Arquitetura de software é comumente definida como um conjunto de decisões de projeto que possuem impacto na construção e evolução de sistemas de software (PERRY; WOLF, 1992). Essas decisões incluem em como tais sistemas são estruturados em componentes e sobre quais restrições esses componentes devem interagir (FOWLER, 2002).

Assegurar que uma arquitetura de software esteja bem definida é de suma importância durante o desenvolvimento de um sistema, uma vez que provê características como manutenibilidade, reusabilidade, escalabilidade e portabilidade (PASSOS et al., 2010; TERRA; VALENTE, 2009). Portanto, a arquitetura de um software deve ser documentada a fim de obter um artefato para a comunicação entre as partes interessadas, facilitando a compreensão do sistema (BITTENCOURT, 2010).

Um desafio enfrentado por arquitetos de software é garantir que um sistema esteja implementado de acordo com a sua arquitetura planejada (MÜLLER; KLASHINSKY, 1988). Isso ocorre pois, durante a construção e evolução da implementação de um sistema, anomalias são frequentemente introduzidas no código-fonte. Essas anomalias consiste em decisões que não são compatíveis com o modelo arquitetural especificado, tornando a base de códigos atual inconsistente

com a documentação existente, ocasionando fenômenos conhecidos como violação e erosão arquitetural (PERRY; WOLF, 1992; KNODEL; POPESCU, 2007).

Violações arquiteturais, na prática, ocorrem devido ao desconhecimento por parte dos desenvolvedores, requisitos conflitantes, prazos de entrega reduzidos, dificuldades técnicas etc. (KNODEL; POPESCU, 2007). Como resultado, essas violações tornam a manutenção uma tarefa mais difícil e demorada, uma vez que o produto implementado não é aderente à arquitetura planejada e documentada (SARKAR; MASKERI; RAMACHANDRAN, 2009). A erosão arquitetural é causada por meio do acúmulo de violações arquiteturais, ocorrendo quando as regras que regem as dependências entre os componentes da arquitetura não são respeitadas (PERRY; WOLF, 1992). Com o objetivo de evitar erosões arquiteturais, técnicas de conformidade arquitetural têm sido propostas como solução (MURPHY; NOTKIN; SULLIVAN, 1995; SANGAL et al., 2005; VERBAERE; GODFREY; GIRBA, 2008; TERRA; VALENTE, 2009).

Conformidade arquitetural é definida como a medida do grau de aderência da arquitetura implementada em código-fonte de um sistema em relação à sua arquitetura planejada (KNODEL; POPESCU, 2007). Essa é uma atividade chave de controle de qualidade de software que objetiva revelar as decisões de implementação que denotam violações arquiteturais, ou seja, declarações concretas, expressões ou declarações no código-fonte que não correspondem às restrições impostas pela arquitetura pretendida (TERRA et al., 2015).

O processo de conformidade arquitetural em geral é apoiado por técnicas baseadas em análise estática e análise dinâmica (YAN et al., 2004; JERDING; RUGABER, 1997). Técnicas de análise estática abrangem as propriedades estruturais do sistema contidas no código-fonte, não necessitam de sua execução e não requerem acesso à base de dados. Já as técnicas baseadas em análise dinâmica requerem o sistema em execução, necessitam do acesso à base de dados e cobrem apenas os caminhos acessados na execução do sistema.

O presente trabalho foca em técnicas de análise estática, descritas por meio de Matrizes de Dependência Estrutural (BALDWIN; CLARK, 2000) e Linguagens de Restrição Arquitetural (TERRA; VALENTE, 2009).

### 2.1.1 Matrizes de Dependência Estrutural

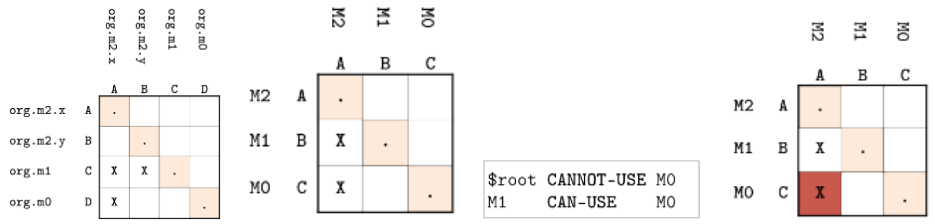
Matrizes de Dependência Estrutural (*Dependency Structure Matrixes* ou DSM) foram propostas por Baldwin e Clark com o intuito de demonstrar e avaliar a importância da organização modular em projetos de software (BALDWIN; CLARK, 2000; SULLIVAN et al., 2001). DSM consiste em matrizes de adjacência utilizadas para representar dependências estáticas entre módulos de um sistema, descritas por meio de classes em um sistema orientado a objetos. Um `X` na linha `A` e coluna `B` informa que a classe (ou módulo) `B` depende da classe (ou módulo) `A`, ou seja, existem referências explícitas (chamadas de métodos, parâmetros, exceções etc.) de elementos sintáticos de `B` para `A`.

Um exemplo de ferramenta comercial que gera DSM's é a ferramenta *Lattix Dependency Manager* (LDM) (SANGAL et al., 2005). LDM é uma ferramenta de conformidade e visualização arquitetural que utiliza DSM's para representar e gerenciar dependências interclasses em sistemas orientados a objetos (TERRA; VALENTE, 2009). Seu objetivo é prover aos arquitetos uma ferramenta gráfica que permita revelar padrões arquiteturais, detectar dependências que possam indicar violações arquiteturais ou mesmo uma reengenharia de código para melhorar a modularidade e reduzir a dívida técnica (LATTIX, 2017).

A Figura 2.1(a) apresenta a extração automática de uma DSM de um código-fonte. Como pode ser observado, o pacote `org.m2.x` depende do pacote `org.m1` e `org.m0`, e o pacote `org.m2.y` depende do pacote `org.m1`. Posteriormente, a ferramenta LDM agrupa as dependências em seus respectivos pacotes e os renomeia (pacotes `org.m2.x` e `org.m2.y` agrupados no módulo `M2`, e pacotes `org.m0` e `org.m1` renomeados para `M1` e `M2`, respectivamente), como apresenta

a Figura 2.1(b). Uma linguagem declarativa de domínio específico é utilizada para especificar regras de conformidade que devem ser seguidas pelo código-fonte avaliado (MAFFORT et al., 2013), como apresenta a Figura 2.1(c). Basicamente, as regras de conformidade representam que `root` – definido por todos os tipos de um sistema – não pode acessar serviços fornecidos por `M0` e que `M1` pode usar dependências de `M0`. Como o módulo `M2` utiliza dependências de `M0` (Figura 2.1(b)), a ferramenta LDM reporta uma violação, ilustrada na Figura 2.1(d).

Figura 2.1 – Matriz de dependência estrutural (MAFFORT et al., 2013).

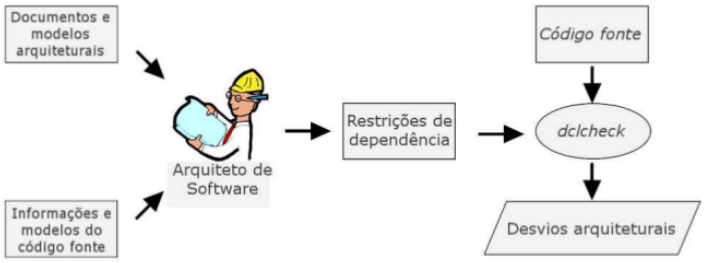



DSM extraída. (a) | DSM pós agrupamento (b) | Regras de projeto. (c) | DSM verificada (d)

### 2.1.2 Linguagem de Restrição Arquitetural

Esse tipo de técnica inclui linguagens de domínio específico que objetivam fornecer um método para especificar dependências estruturais de uma aplicação. Como exemplo, este trabalho descreve a linguagem `DCL` (*Dependency Constraint Language*) (TERRA; VALENTE, 2009), conforme ilustra a Figura 2.2.

Figura 2.2 – Solução de conformidade arquitetural (TERRA; VALENTE, 2009).



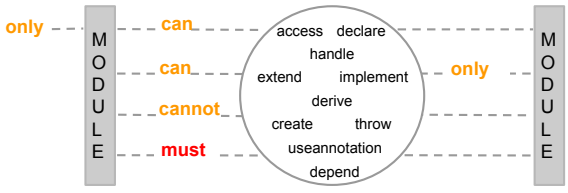
Inicialmente, o arquiteto de software deve definir as restrições de dependência do sistema, utilizando a linguagem DCL. Para definição dessas restrições, o arquiteto deve se basear em informações presentes em modelos do código-fonte (diagramas de classes) e em documentos e modelos arquiteturais (diagramas de pacotes e componentes). Para automação da solução proposta, os autores desenvolveram a ferramenta DCL$^{+}$*check* que detecta violações das restrições de dependências especificadas.

**Linguagem**: DCL é uma linguagem declarativa e estaticamente verificável que suporta as definições de dependências estruturais entre módulos de um sistema orientado a objetos (TERRA; VALENTE, 2009). Essa linguagem permite especificar os seguintes tipos de violações (SULLIVAN et al., 2001; KNODEL; POPESCU, 2007):

- Divergência: Ocorrem quando uma dependência existente no código-fonte viola uma restrição de dependência especificada; e
- Ausência: São detectadas quando o código-fonte não possui uma dependência que deveria existir de acordo com uma restrição de dependência especificada.

A linguagem DCL abrange módulos e restrições arquiteturais entre os módulos definidos, descritos conforme ilustra a Figura 2.3.

Figura 2.3 – Sintaxe da linguagem DCL (TERRA; VALENTE, 2009).




Para capturar as divergências, os arquitetos devem especificar que certas dependências podem (*only can* e *can- only*) ou não podem (*cannot*) ser estabe-

lecidas por módulos específicos. Para capturar as ausências, DCL permite aos arquitetos especificar que certas dependências devem estar presentes (*must*) em determinados módulos do sistema. DCL também permite a especificação da granularidade das dependências (*access*, *declare*, *handle*, *extend*, *implement*, *derive*, *create*, *throw*, *useannotation* e *depend*), presentes em sistemas orientado a objetos.

**Módulos:** Um módulo é um conjunto de classes. Suponha, por exemplo, as seguintes definições:

```
1 module View: org.foo.view.*
2 module DataStructure: org.foo.util.**
3 module Remote: java.rmi.UnicastRemoteObject+
4 module Frame: "org.foo.[a-zA-Z0-9/.]*Frame"
```

O módulo View inclui todas as classes do pacote **org.foo.view** (linha 1). O módulo DataStructure (linha 2) inclui todas as classes do módulo **org.foo.util** e todas as suas subclasses (operador **). O módulo Remote (linha 3) denota todas as subclasses de **java.rmi.UnicastRemoteObject** (operador +). Por fim, DCL permite também a definição de módulos por meio de uma expressão regular delimitada por aspas. Por exemplo, o módulo Frame (linha 4) é composto por todas as classes cujo nome qualificado inicia-se com **org.foo** e termina com Frame. Em complementação, DCL provê dois módulos pré-declarados: $java, que inclui todos os tipos da biblioteca de Java, e $system, que se refere aos tipos específicos do sistema.

**Restrições:** O exemplo abaixo ilustra a definição de restrições arquiteturais entre os módulos:

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

Essas restrições expressam o seguinte significado: (linha 1) apenas classes definidas no módulo `Fabrica` podem criar objetos da classe `DAO`; (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 classes de Visão não podem acessar classes do Modelo; e (linha 4) todas as classes definidas no módulo Entidade 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.

Como mencionado anteriormente, `DCL` especifica quatro tipos de restrições *only can, can only, cannot* e *must*. Assuma os módulos $M_A$ e $M_B$ – conjunto de classes de um sistema – e $\overline{M_A}$ e $\overline{M_B}$ o complemento dos respectivos conjuntos. Assuma também que `dep` corresponde a todas as dependências suportadas por `DCL`. Dessa maneira, uma violação na restrição $M_A$ `cannot-dep` $M_B$ é definido por:

$$\exists A \; \exists B \; [\; A \in M_A \;\; \wedge \;\; B \in M_B \;\; \wedge \;\; dep(A,B) \;]$$

Restrições *only-can* e *can-only* são definidas com base na restrição *cannot*.

$$only \;\; M_A \;\; can{-}dep \;\; M_B \implies \overline{M_A} \;\; cannot{-}dep \;\; M_B$$
$$M_A \;\; can{-}dep{-}only \;\; M_B \implies M_A \;\; cannot{-}dep \;\; \overline{M_B}$$

Finalmente, a semântica de uma violação atribuída à restrição $M_A \;\; must{-}dep \;\; M_B$ é definida como:

$$\exists A \; !\exists B \; [\; A \in M_A \;\; \wedge \;\; B \in M_B \;\; \wedge \;\; dep(A,B) \;]$$

Dessa maneira, uma violação ocorre quando um elemento A $\in M_A$ não estabelece uma dependência `dep` com um elemento B $\in M_B$.

Este trabalho adota a linguagem `DCL` para especificar as comunicações entre os microsserviços de uma aplicação. A escolha pela linguagem `DCL` se deu ao fato de que (i) é uma linguagem simples e com sintaxe autoexplicativa; (ii) é expressiva para tratar problemas de erosão arquitetural; e (iii) possui implementação de código-fonte aberta para linguagem Java.

## 2.2 Arquitetura de Microsserviços

A arquitetura de microsserviços é uma arquitetura nativa da nuvem que tornou-se popular nos últimos anos devido à sua característica particular de projetar aplicativos de software como suítes de serviços implementáveis independentemente, a fim de superar os principais problemas encontrados em aplicações monolíticas (FOWLER, 2014). Em essência, esse estilo arquitetural emergiu a partir da arquitetura SOA (*Service-Oriented Architecture*), a fim de buscar soluções em relação aos seus protocolos de comunicação, falta de orientação sobre a granularidade do serviço ou a orientação incorreta em lugares escolhidos para dividir o sistema (NEWMAN, 2015).

Segundo Fowler e Lewis, a arquitetura de microsserviço é 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 que utiliza o protocolo HTTP (FOWLER; LEWIS, 2014). Newman simplifica sua definição apresentando microsserviços como pequenos serviços autônomos que trabalham em conjunto (NEWMAN, 2015).

Os microsserviços são entidades autônomas, 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. Consequentemente, modificar a implementação de um serviço não impacta outros

serviços, uma vez que a comunicação ocorre por meio de interfaces bem definidas – REST (*REpresentational State Transfer*), por exemplo. Dessa maneira, é possível que os serviços sejam implantados, escalados e testados de maneira independente (THÖNES, 2015). Isso garante alta coesão e baixo acoplamento entre os serviços, enfatizando a agilidade, flexibilidade e escalabilidade (FOWLER, 2014).

Para melhor entendimento sobre à arquitetura de microsserviços, Newman descreve as principais características relacionadas a esse estilo arquitetural (NEWMAN, 2015):

**Heterogeneidade tecnológica**: Uma vez que as aplicações são compostas por um conjunto de serviços independentes e ligeiramente acoplados, tem-se a liberdade de escolher a melhor ferramenta que satisfaça as suas necessidades. Isso pode ser aplicado desde a linguagem de programação, tecnologias para o gerenciamento dos dados, até o servidor de aplicações.

**Escalabilidade:** Os serviços são definidos e organizados em torno das capacidades do negócio (CAMARGO et al., 2016). Esse tipo de granularidade permite escalar cada um dos serviços conforme necessário, reduzindo os custos e fornecendo apenas os recursos necessários (NEWMAN, 2015). Serviços oferecidos pela Amazon AWS[1] e provedores similares, permitem escalar as funcionalidades conforme as demandas atuais. Dessa forma, é possível gerenciar o custo de hospedar a aplicação de maneira mais eficiente (TOFFETTI et al., 2017).

**Resiliência:** O conceito-chave de resiliência consiste em isolar as falhas causadas por um serviço para que esse não afete toda a aplicação. A independência natural dos serviços em uma arquitetura microsserviços permite isolar os principais serviços em sua própria infraestrutura com o objetivo de mantê-los funcionando, mesmo que falhas sejam ocasionadas por outros serviços.

[1] *Amazon AWS*, disponível em: https://aws.amazon.com

**Alinhamento organizacional:** Os microsserviços permitem alinhar a arquitetura da aplicação à organização, objetivando organizar os times ao redor das áreas do negócio. Dessa maneira, não é necessário que os desenvolvedores possuam o domínio de toda a aplicação, mas apenas dos serviços individuais que são responsáveis, acarretando na maior produtividade da equipe (NEWMAN, 2015).

**Facilidade de implantação:** Em uma arquitetura de microsserviços, é possível modificar um único serviço e implantá-lo de forma independente (NEWMAN, 2015). Essas mudanças são desenvolvidas e testadas de maneira mais rápida e com menor risco quando comparadas a aplicações monolíticas em que mesmo pequenas mudanças exigem que todo o aplicativo seja implantado. Além disso, os problemas de implantação podem ser facilmente isolados em uma arquitetura de microsserviços que permite uma recuperação rápida e a restauração do serviço.

A arquitetura de microsserviços têm sido amplamente adotada por empresas como Netflix, Amazon e Linkedin a fim de fornecer aplicações de software em grande escala (VILLAMIZAR et al., 2015; NEWMAN, 2015). Contudo, a adoção desse estilo arquitetural apresenta problemas como (i) a descoberta dos serviços no ambiente de computação em nuvem, (ii) o roteamento e balanceamento de carga das chamadas entre os serviços para a realização das comunicações e (iii) a composição dos microsserviços que integram a aplicação.

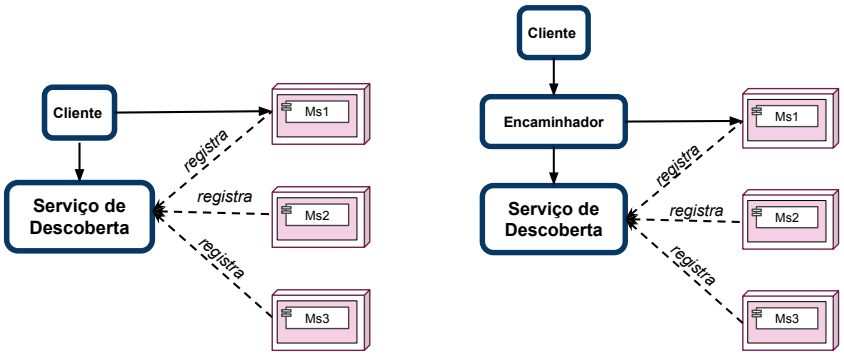
A fim de resolver tais problemas, ferramentas para o desenvolvimento de aplicações distribuídas implementadas na linguagem Java e que sigam as premissas abordadas pela arquitetura de microsserviços têm sido propostas pelo projeto *Spring Cloud* (PIVOTAL, 2018). Esse projeto fornece uma integração completa entre o *framework* Spring Boot e o projeto *Netflix Open Source* (Netflix OSS). Netflix OSS compreende um conjunto de *frameworks* e bibliotecas desenvolvidos pela Netflix para resolver problemas comuns em sistemas distribuídos escaláveis (NETFLIX, 2018b). Para o melhor entendimento dos conceitos apresentados, esses padrões e *frameworks* são discutidos nas próximas seções.

### 2.2.1 Descoberta de Serviço

A característica dinâmica presente em aplicações implantadas na nuvem faz com que a localização de um microsserviço possa não ser estaticamente conhecida em tempo de projeto. Isso é devido as possibilidades de replicação e realocação dos microsserviços em tempo de execução (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016), sendo necessário assim, mecanismos para a descoberta de serviços. Essa necessidade vem desde a arquitetura SOA com o serviço de registro (*Service Registry*), utilizado para obter informações entre serviços que se comunicam (PAPAZOGLOU et al., 2007).

A Figura 2.4 ilustra duas maneiras de utilização do padrão para a descoberta de serviços. No padrão de descoberta por parte do cliente (Figura 2.4a), cada microsserviço deve se registrar no `servio de descoberta`. O `cliente`, por sua vez, deve consultar o `servio de descoberta` para conhecer a localização do serviço ao qual deseja-se comunicar (como servidor e porta, por exemplo). Uma vez conhecida essa informação, o `cliente` realiza a comunicação com o serviço diretamente. Essa arquitetura é simples, porém há a necessidade de implementação da lógica de descoberta para cada linguagem de programação e/ou estrutura usada para a implementação de clientes (RICHARDSON, 2017b).

Figura 2.4 – Padrão de descoberta de serviço. Adaptado de (RICHARDSON, 2017b).




Serviço de descoberta do lado do cliente. (a)

Serviço de descoberta do lado do servidor. (b)

No padrão de descoberta de serviços por parte do servidor (Figura 2.4b), os serviços também devem se registrar no serviço de descoberta. Contudo, o `cliente` deve comunicar-se com um `encaminhador` que possui localização fixa. Ao receber uma solicitação, o `encaminhador` solicita ao `servio de descoberta` as informações do serviço desejado pelo `cliente` e encaminha a solicitação a ele. Nesse tipo de padrão, toda a lógica de descoberta de serviços concentra-se apenas no `encaminhador`. Dessa maneira, não é necessário que cada cliente implemente sua própria lógica. Em contrapartida, é necessário a inclusão e configuração do componente `encaminhador` no ambiente, podendo ser necessário também seu escalonamento dependendo da disponibilidade de solicitações.

Uma implementação prática do padrão de descoberta por parte do cliente é proposta pelo projeto Netflix OSS através do *framework* Eureka. O Eureka é um microsserviço baseado em REST que permite que os serviços encontrem e se comuniquem uns com os outros apenas com a informação sobre seus nomes. Um servidor Eureka é facilmente implementado através da criação de um projeto Maven com (i) inserção da dependência presente na Listagem 2.1, (ii) criação da classe principal com as anotações necessárias apresentadas pela Listagem 2.2 e (iii) criação de um arquivo de propriedades, conforme demonstra a Listagem 2.3.

```
1 <dependency
2    <groupId>org.springframework.cloud</groupId>
3    <artifactId>spring-cloud-starter-eureka-server</artifactId>
4 </dependency>
```

Listagem 2.1 – Dependência Maven para o servidor Eureka.

```
1 @SpringBootApplication
2 @EnableEurekaServer
3 public class EurekaServer {
4    public static void main(String[] args) {
5         SpringApplication.run(EurekaServer.class, args);
6    }
7 }
```

Listagem 2.2 – Classe principal do servidor Eureka.

```
server:
    port: 8761
eureka:
    client:
        registerWithEureka: false
    fetchRegistry: false
```

Listagem 2.3 – Arquivo de propriedade do servidor Eureka.

Uma vez implementado o serviço de descoberta, é necessário que cada microsserviço que compõe a aplicação defina suas configurações para que possa ser encontrado no ambiente distribuído. No contexto de aplicações Java, é necessário também a criação de um projeto Maven com (i) inserção das dependências (Listagem 2.4), (ii) classe principal com as anotações necessárias (Listagem 2.5) e (iii) arquivo de propriedades (Listagem 2.6).

```
...
<dependency>
   <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
...
```

Listagem 2.4 – Dependência Maven para o cliente Eureka.

```
@SpringBootApplication
@EnableEurekaClient
public class EurekaClient {
   public static void main(String[] args) {
        SpringApplication.run(EurekaClient.class, args);
   }
}
```

Listagem 2.5 – Classe principal do cliente Eureka.

```
spring:
    application:
        name: Ms1
    server:
        port: 8080
```

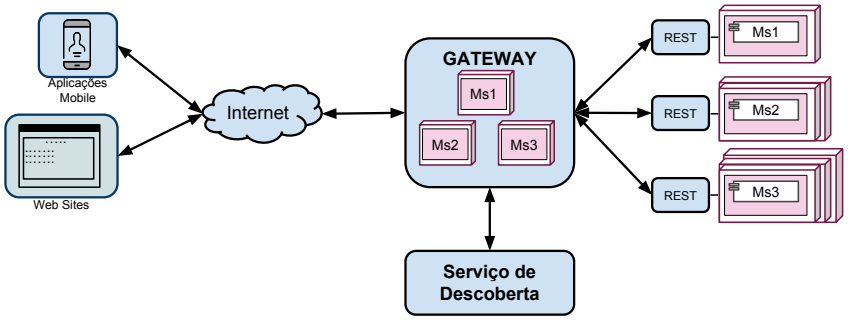
Listagem 2.6 – Arquivo de propriedade do cliente Eureka.

### 2.2.2 Padrão API Gateway

Com o objetivo de resolver o problema de roteamento das requisições e balanceamento de carga para diferentes tipos de cliente em uma arquitetura de microsserviços, o padrão *API Gateway* tem sido amplamente utilizado (RICHARDSON, 2017a; RICHARDSON, 2017b).

Como pode ser observado na Figura 2.5, o padrão permite que aplicações *web* tradicionais e aplicações *mobile*, por exemplo, utilizem funcionalidades fornecidas por serviços em uma arquitetura de microsserviços. Quando uma aplicação `cliente` solicita funcionalidades a um microsserviço, o serviço *gateway* descobre as informações do serviço requisitado através do `servio de descoberta` e realiza o roteamento da requisição. Os microsserviços que compõem a aplicação são geralmente expostos por meio de API's como REST no *API Gateway*.

Figura 2.5 – Padrão API Gateway.




No projeto *Spring Cloud*, o serviço responsável por implementar o padrão *API Gateway* é o microsserviço Zuul. Esse serviço é responsável por permitir roteamento dinâmico, balanceamento de carga, monitoramento, resiliência e segurança (NETFLIX, 2018a). Semelhante ao serviço Eureka, a configuração do serviço Zuul é simples, necessitando da (i) criação de um projeto Maven com as respectivas dependências (Listagem 2.7), (ii) configuração da classe principal (Listagem 2.8) e (iii) configuração no arquivo de propriedades (Listagem 2.9).

```
...
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
...
```

Listagem 2.7 – Dependência Maven para o serviço Zuul.

```
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZuulServer {
   public static void main(String[] args) {
        SpringApplication.run(ZuulServer.class, args);
   }
}
```

Listagem 2.8 – Classe principal do serviço Zuul.

```
server:
    port: 8765
zuul:
    routes:
        users: /ms1/api/ **
            serviceId: Ms1
        users: /ms2/api/ **
            serviceId: Ms2
        users: /ms3/api/ **
            serviceId: Ms3
        ...
    ...
...
```

Listagem 2.9 – Arquivo de propriedade do serviço Zuul.

O arquivo de propriedades do serviço Zuul deve conter informações sobre o servidor e a porta no qual o serviço será executado, e as rotas dos serviços que compõem a aplicação. No exemplo da Listagem 2.9, o Zuul é executado na porta 8765. Em relação as rotas a serem percorridas por esse serviço, por exemplo, todas as vezes que uma rota "/ms1/api/**" for requisitada, o microsserviço `Ms1` deve ser recuperado (linhas 5 e 6, respectivamente).

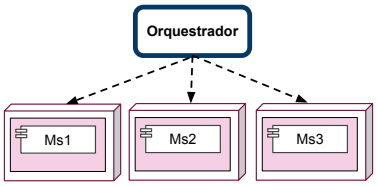
### 2.2.3 Composição entre Microsserviços

A preocupação em relação a maneira como os serviços devem se integrar em uma arquitetura de microsserviços, vem desde a arquitetura SOA, com a in-

trodução de conceitos como orquestração e coreografia de serviços (PEDRAZA; ESTUBLIER, 2009; LEITE et al., 2013).

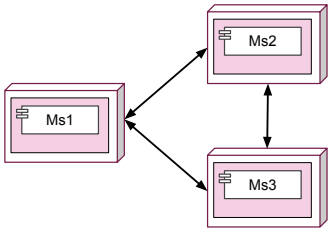
A orquestração refere-se a um paradigma de composição de serviços no qual os serviços são controlados de maneira centralizada (NANDA; CHANDRA; SARKAR, 2004), conforme ilustra a Figura 2.6. Como pode ser observado, uma única entidade conhecida como orquestrador é responsável por gerenciar e coordenar todo o fluxo de execução de solicitações a microsserviços. Nesse tipo de composição, toda a lógica da aplicação concentra-se no orquestrador e apenas ele conhece todos os microsserviços da aplicação.

Figura 2.6 – Orquestração de microsserviços.




Por outro lado, a Figura 2.7 ilustra como a composição de serviços por meio de coreografia deve ocorrer. Nesse tipo de composição, cada serviço envolvido sabe exatamente quando executar suas operações e com quem interagir, sem que haja um controle centralizado (BARKER; WALTON; ROBERTSON, 2009). Nesse sentido, as coreografias podem ser vistas como um acordo entre um conjunto de serviços sobre como uma determinada colaboração deve ocorrer (LEITE et al., 2013). Dessa maneira, os serviços comunicam diretamente entre si, eliminando assim a necessidade de uma entidade central.

Figura 2.7 – Coreografia de microsserviços.



Segundo Newman, a arquitetura de microsserviços tende a se adaptar melhor com a composição por meio de coreografia, uma vez que os serviços torna-se menos acoplados – devido a inexistência do orquestrador – melhor escalonados e passíveis de mudança (NEWMAN, 2015). Fowler reforça ainda que microsserviços buscam integrar-se através de *endpoints* inteligentes utilizando bibliotecas REST e comunicação simples por meio do protocolo HTTP (FOWLER, 2014).

Com o objetivo de fornecer maneiras de integração entre os microsserviços para o estabelecimento de comunicação, o projeto *Spring Cloud* define clientes descritos como *Feign* e *Spring RestTemplate*. Esses dois tipos de clientes são integrados junto ao serviço *Eureka* para o descobrimento de serviços locais. *Feign* é um cliente de serviço *web* declarativo que permite a criação de clientes HTTP. Por meio do *Feign*, é possível estabelecer requisições de comunicação entre microsserviços por meio de interfaces e anotações (NETFLIX, 2018b). *RestTemplate*, por sua vez, consiste em uma classe central do Spring Boot para acesso `http` do lado do cliente (SPRING, 2018), fornecendo vários métodos para o estabelecimento de requisições entre os microsserviços.

A Listagem 2.10 apresenta como é realizada a comunicação entre os microsserviços utilizando o cliente *Feign*. Suponha que há a necessidade de um microsserviço denominado `Ms1` estabelecer uma comunicação com um microsserviço denominado `Ms2`.

```
1  @FeignClient("Ms2")
2  public interface Ms2Cliente{
3      @RequestMapping("/clientMs2/")
4      public ClientMs2 bar();
5  }
6  public class Api{
7      ...
8      @Autowired
9      private Ms2Cliente clientMs2;
10     ...
11     public ClientMs2 foo(){
12         return client.bar();
13     }
14     ...
15 }
```

Listagem 2.10 – Comunicação por meio do cliente *Feign*.

Para isso, inicialmente é necessário declarar uma interface de comunicação *@FeignClient*, informando qual o nome do microsserviço (`Ms2`) que se deseja realizar a comunicação (linha 1). Posteriormente são definidos os métodos de acesso a essas informações, por exemplo, através do método *bar()* (linha 4). Para cada método, é definida uma anotação *@RequestMapping* (linha 3). Nessa anotação, deve-se informar qual é a interface acessada em tal microsserviço (*"/clientMs2/"*).

Para utilizar essa interface de comunicação, é necessário declarar o objeto *clientMs2* (linha 9), injetando uma dependência por meio da anotação *@Autowired* (linha 8), para solicitar informações do microsserviço `Ms2`. Essa solicitação é invocada por meio do método *foo()* (linha 11), através das informações retornadas pelo método *bar()* (linha 12).

Por outro lado, a Listagem 2.11 ilustra como é realizada a comunicação por meio de *RestTemplate*. Basicamente, deve-se definir uma injeção de dependência a um objeto *clientMs3* do tipo *RestTemplate* (linha 4) e invocar os métodos fornecidos pela classe. Nesse exemplo, o método *getForObject()* foi utilizado para a realização da requisição de informações fornecidas pelo microsserviço `Ms3`. Uma das maneiras de definir essa requisição é passando como parâmetro a esse método a *url* do microsserviço que se deseja obter informações, como pode ser observado na linha 8.

```
1  public class Api{
2      ...
3      @Autowired
4      private RestTemplate clientMs3;
5      ...
6      @RequestMapping("/clientMs3/")
7      public RestTemplate foo(){
8         clientMs3.getForObject("http://Ms3/clientMs3");
9      }
10     ...
11 }
```

Listagem 2.11 – Comunicação por meio do cliente *RestTemplate*.

## 2.3 Considerações Finais

Neste capítulo, foram apresentados os conceitos fundamentais para o entendimento desta dissertação de mestrado. Foram inicialmente introduzidas as definições sobre as técnicas de conformidade arquitetural, descritas por meio de matrizes de dependência estrutural (BALDWIN; CLARK, 2000) e linguagens de restrição arquitetural (TERRA; VALENTE, 2009). Posteriormente, foram apresentados os conceitos referentes à arquitetura de microsserviços.

Matrizes de dependência estrutural consiste em matrizes de adjacência utilizadas para representar dependências estáticas entre módulos de um sistema. Essa representação compreende à um instrumento simples e valioso para visualizar e avaliar arquiteturas de software. Outra técnica de conformidade apresentada foi `DCL`, uma linguagem de restrição arquitetural declarativa e estaticamente verificável que permite especificar dependências que podem (*only can*, *can-only*) ou não podem (*cannot*) ocorrer para capturar divergências e dependências que devem estar presentes (*must*) em determinados módulos do sistema para capturar ausências. Este trabalho adota a linguagem `DCL` pra especificar as comunicações entre microsserviços, uma vez que `DCL` é uma linguagem expressiva para tratar problemas de erosão arquitetural, além de possuir implementação de código-fonte aberta para a linguagem Java.

No que compete à arquitetura de microsserviços, esse estilo arquitetural tornou-se popular nos últimos anos devido à sua característica particular de projetar sistemas de software como suítes de serviços implementáveis independentemente, a fim de solucionar problemas encontrados em aplicações monolíticas. Os microsserviços são entidades autônomas que podem ser implementados em diferentes linguagens, estar hospedados em diferentes servidores e utilizar diferentes *frameworks*. Com o objetivo de prover mecanismos para o desenvolvimento de aplicações que utilizam microsserviços, diversas ferramentas têm sido propostas no projeto Spring Cloud (PIVOTAL, 2018). Esse projeto fornece uma integração

completa entre o *framework* Spring Boot e o projeto Netflix OSS para solução de problemas como descoberta de serviços, roteamento e balanceamento de carga e composição entre microsserviços.

# 3 SOLUÇÃO DE CONFORMIDADE ARQUITETURAL PROPOSTA

Este capítulo apresenta a solução de conformidade arquitetural proposta nesta dissertação de mestrado. Para isso, inicialmente foi definida uma linguagem de restrição arquitetural denominada DCL$^+$ a fim de especificar as comunicações desejáveis entre microsserviços, apresentada na Seção 3.1. Posteriormente, a Seção 3.2 apresenta a abordagem de conformidade arquitetural proposta. A Seção 3.3 ilustra a aplicabilidade da abordagem proposta por meio de um estudo piloto de uma aplicação composta por cinco microsserviços. Por fim, a Seção 3.4 apresenta as considerações finais deste capítulo. É válido ressaltar que detalhes do projeto técnico e implementação da solução proposta DCL$^+$ são apresentados no Capítulo 4.

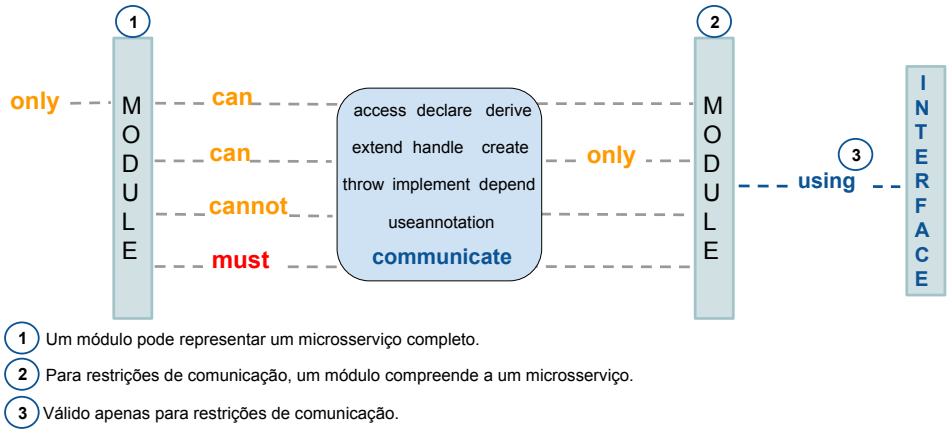
## 3.1 Linguagem de Restrição Arquitetural

Para prover uma solução multiplataforma que permita restringir a comunicação entre os microsserviços da arquitetura, foi necessária a utilização de alguma técnica de conformidade arquitetural para tal propósito. Para isso, optou-se por estender a linguagem DCL (Seção 2.1.2) para especificar as comunicações válidas entre os microsserviços. Isso se deu ao fato de que DCL é (i) uma linguagem simples e com sintaxe autoexplicativa; (ii) é expressiva para tratar erosões arquiteturais; e (iii) possui implementação aberta para sistemas desenvolvidos na linguagem Java.

DCL$^+$ é uma extensão da previamente proposta linguagem DCL – responsável por especificar dependências estruturais – para o contexto da arquitetura de microsserviços. Como pode ser observado na Figura 3.1, para representar esse novo tipo de arquitetura, a linguagem estabelece um novo tipo de dependência relacionado a comunicação entre microsserviços, denominado `communicate`, além de definir restrições *only can*, *can only* e *cannot* para detectar as divergências e restrições *must* para detectar as ausências de comunicação. DCL$^+$ permite também a especificação da nomenclatura `using` para restringir a interface de

comunicação entre os microsserviços. Mais além, DCL$^+$ considera as comunicações entre microsserviços da aplicação que se comunicam com outro(s) não pertencente(s) à aplicação, definidas por meio de `alertas`. É válido ressaltar que, na linguagem DCL$^+$, um módulo pode compreender a um conjunto de módulos de um microsserviço, ou seja, um módulo pode representar um microsserviço. A especificação completa da gramática que gera a linguagem DCL$^+$ é apresentada no Apêndice A.

Figura 3.1 – Sintaxe da linguagem DCL$^+$.




Seja $S$ o conjunto de todos os microsserviços pertencentes à aplicação. Seja $A$ um microsserviço, tal que $A \in S$, $M_A$ o conjunto de todos os módulos pertencentes à $A$, e $S_{M_A}$ um subconjunto de módulos de $M_A$. Analogamente, seja $S_B$ um conjunto de microsserviços diferentes de $A$ ($S_B \subseteq S - \{A\}$). Desta maneira, o novo tipo de dependência é definido a seguir.

**Divergências**: Para capturar as divergências de comunicações, DCL$^+$ suporta os seguintes tipos de restrições entre microsserviços:

- $S_{M_A}$ *cannot-communicate* $S_B$: os módulos pertencentes ao subconjunto $S_{M_A}$ *não* poderão comunicar com microsserviços em $S_B$. Formalmente, violações são definidas da seguinte forma:

$$\exists m\ \exists B\ [\ m\ \in S_{M_A}\ \ \wedge\ \ B\ \in\ S_B\ \ \wedge\ \ communicate(m,B)\ ]$$

- $S_{M_A}$ *cannot-communicate* $S_B$ **using** $I$: os módulos pertencentes ao subconjunto $S_{M_A}$ *não* poderão comunicar com microsserviços em $S_B$ utilizando a interface $I$. Formalmente, violações são definidas da seguinte forma:

$$\exists m\ \exists B\ [\ m\ \in S_{M_A}\ \ \wedge\ \ B\ \in\ S_B\ \ \wedge\ \ communicate\_using(m,B,I)\ ]$$

As restrições *only-can* e *can-only* são definidas com base na restrição *cannot*.

- *Only* $S_{M_A}$ *can-communicate* $S_B$: *apenas* os módulos em $S_{M_A}$ poderão comunicar com microsserviços em $S_B$. Essa restrição, formalmente, é definida por:

$$only\ S_{M_A}\ can{-}communicate\ S_B\ \Longrightarrow\ \overline{S_{M_A}}\ \ cannot{-}communicate\ S_B$$

- *Only* $S_{M_A}$ *can-communicate* $S_B$ **using** $I$: *apenas* os microsserviços em $S_{M_A}$ poderão comunicar com microsserviços em $S_B$ utilizando a interface $I$. Essa restrição, formalmente, é definida por:

$$only\ S_{M_A}\ can{-}communicate\ S_B\ using\ I\ \rightarrow\ \overline{S_{M_A}}\ cannot{-}communicate\ S_B\ using\ I$$

- $S_{M_A}$ *can-communicate-only* $S_B$: os módulos em $S_{M_A}$ poderão comunicar *apenas* com microsserviços em $S_B$. Formalmente, essa restrição define-se por:

$$S_{M_A}\ can{-}communicate{-}only\ S_B\ \Longrightarrow\ \ S_{M_A}\ cannot{-}communicate\ \overline{S_B}$$

- $S_{M_A}$ *can-communicate-only* $S_B$ **using** $I$: os módulos em $S_{M_A}$ poderão comunicar *apenas* com microsserviços em $S_B$. Formalmente, essa restrição define-se por:

$$S_{M_A}\ can{-}communicate{-}only\ S_B\ using\ I\ \Longrightarrow\ S_{M_A}\ cannot{-}communicate\ \overline{S_B}\ using\ I$$

**Ausências**: Para capturar as ausências, DCL$^{+}$ suporta apenas um tipo de restrição, definida como:

- $S_{M_A}$ *must-communicate* $S_B$: os módulos em $S_{M_A}$ *devem* (obrigatoriamente) comunicar com microsserviços em $S_B$. Violações são definidas da seguinte forma:

$$\forall m \; !\exists B \; [\; m \in S_{M_A} \;\wedge\; B \in S_B \;\wedge\; communicate(m,B) \;]$$

- $S_{M_A}$ *must-communicate* $S_B$ **using** $I$: os módulos em $S_{M_A}$ *devem* (obrigatoriamente) comunicar com microsserviços em $S_B$ utilizando a interface $I$. Violações são definidas da seguinte forma:

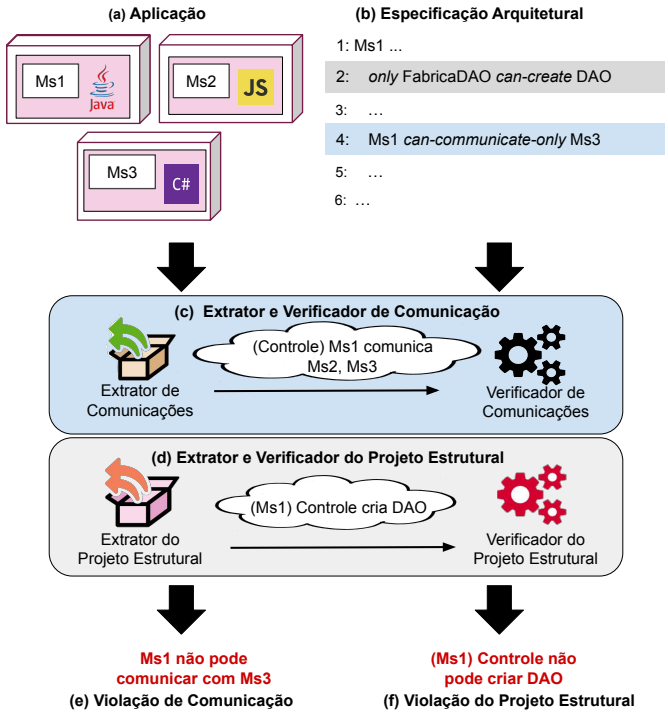
$$\forall m \; !\exists B \; [\; m \in S_{M_A} \;\wedge\; B \in S_B \;\wedge\; communicate\_using(m,B,I)]$$

**Alertas**: Ocorrem quando módulos de um microsserviço devidamente especificado comunica com um microsserviço desconhecido pela arquitetura de microsserviços. Logo, um alerta ocorre quando um elemento A estabelece uma comunicação `communicate` com um elemento B $\notin S$.

## 3.2 Abordagem de Conformidade Arquitetural

A Figura 3.2 ilustra a abordagem proposta para verificação da conformidade arquitetural para a 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 3.2a) juntamente com sua *Especificação Arquitetural* (Figura 3.2b). A execução da abordagem é apoiada por meio do *Extrator e Verificador de Comunicação* (Figura 3.2c) e *Extrator e Verificador do Projeto Estrutural* (Figura 3.2d). As saídas da execução da abordagem resultam em um conjunto de violações de *comunicação* (Figura 3.2e) e violações do *projeto estrutural* (Figura 3.2f).

Figura 3.2 – Abordagem de conformidade arquitetural para sistemas sob uma arquitetura de microsserviços.




### 3.2.1 Aplicação

Conforme ilustrado na Figura 3.2a, uma das entradas da abordagem consiste do conjunto de microsserviços que compõe a aplicação. Basicamente, a abordagem DCL$^+$ requer o acesso ao repositório do código-fonte de cada microsserviço.

### 3.2.2 Especificação Arquitetural

A Figura 3.2b apresenta a outra entrada da abordagem, consistindo na especificação arquitetural na linguagem proposta DCL$^+$. Como pode ser observado na Listagem 3.1, a especificação arquitetural é composta pela definição (a) cada microsserviço, (b) restrições do projeto estrutural de cada microsserviço e (c) comunicações válidas de um microsserviço em relação a outros.

```
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      Restricao DCL+ comunicacao #1 para ms1                         (c)
6      ...                                                            (c)
7      Restricao DCL+ comunicacao #N para msN                         (c)
8  ...
9  [id_msn]: [url_msn] ; [repositorio_msn] ; [linguagem_msn]          (a)
10     Restricao DCL #1 para msn                                      (b)
11     ...                                                            (b)
12     Restricao DCL #N para msn                                      (b)
13     Restricao DCL+ comunicacao #1 para msn                         (c)
14     ...                                                            (c)
15     Restricao DCL+ comunicacao #N para msN                         (c)
```

Listagem 3.1 – Arquivo de especificação arquitetural DCL$^+$.

**Definição de um microsserviço:** Como pode ser observado nas linhas 1 e 9, microsserviços são definidos por um `identificador`, `url` de acesso, `repositrio` de código-fonte e a `linguagem` em que foi implementado.

**Definição das restrições do projeto estrutural de cada microsserviço:** Como pode ser visto nas linhas 2 à 4 e nas linhas 10 à 12, as restrições estruturais de cada microsserviço iniciam-se logo após a sua respectiva definição. Essas restrições são especificadas na linguagem DCL.

**Definição das restrições de comunicação:** Como pode ser visto nas linhas 5 à 7 e nas linhas 13 à 15, as restrições de comunicação referente a cada microsserviços são apresentadas após as definições de restrições do projeto estrutural. Essas restrições são especificadas na linguagem DCL$^+$ (Seção 3.1).

No exemplo da Figura 3.2b, o microsserviço `Ms1` possui uma restrição de projeto estrutural no qual apenas as classes pertencentes ao módulo `FabricaDAO` podem criar objetos `DAO` (linha 2). Como restrição de comunicação, nenhhum módulo pertencente ao microsserviço `Ms1` pode comunicar-se com o microsserviço `Ms3` (linha 4).

### 3.2.3 Conformidade de Comunicação

A análise da conformidade de comunicação é apoiada pelo Extrator e Verificador de Comunicação (Figura 3.2c), que ao serem executados reportam um conjunto de Violação de Comunicação (Figura 3.2e).

**Extrator de comunicação:** O módulo extrator extrai do código-fonte de cada um dos microsserviços as declarações de chamadas à outros microsserviços. Para o exemplo ilustrado na Figura 3.2c, é extraído do módulo *Controle* do código-fonte do microsserviço `Ms1` o estabelecimento de 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 $\texttt{DCL}^{+}$ denominada `communicate`. No exemplo ilustrado na Figura 3.2b, uma restrição define que `Ms1` (ou todos os módulos daquele sistema) não pode se comunicar com `Ms3` e, como o módulo anterior detectou tal comunicação, esse módulo reportará uma *violação de comunicação* (Figura 3.2e).

### 3.2.4 Conformidade do Projeto Estrutural

Esse módulo é composto pelo Extrator e Verificador do Projeto Estrutural que, ao serem executados, reportam um conjunto de *violação de projeto estrutural* (Figuras 3.2d e 3.2f).

**Extrator do projeto 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. No exemplo ilustrado na Figura 3.2d, a partir de um extrator para a linguagem Java, foi extraída a se-

guinte tripla [`ClienteController`, `create`, `ClienteDAO`], o que indica que o módulo `Controle` do microsserviço `Ms1` está criando objetos DAO.

**Verificador do projeto 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 verifica se as restrições de dependência `DCL` (Seção 2.1.2) estão em conformidade com a especificação arquitetural. 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. Na Figura 3.2b, uma restrição define para `Ms1` que apenas o módulo `FabricaDAO` pode criar objetos DAO e, como o módulo anterior detectou uma instanciação na camada de Controle, esse módulo reportará uma *violação do projeto estrutural*.

## 3.3 Avaliação Controlada

Esta seção apresenta uma avaliação controlada da abordagem `DCL`$^+$ em uma aplicação com microsserviços compostos por meio de coreografia. Para isso, foram implementados um conjunto de cinco microsserviços no contexto de vendas de produtos, disponíveis no repositório GitHub[1]. As próximas subseções apresentam como cada um dos módulos da abordagem foram implementados para o contexto da aplicação proposta.

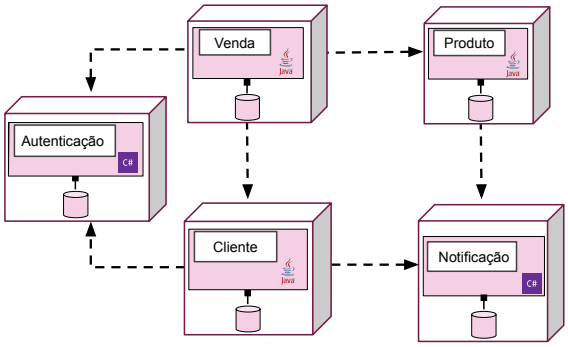
### 3.3.1 Aplicação

A Figura 3.3 ilustra a arquitetura planejada da aplicação. *Autenticação* autentica uma venda. *Produto* e *Cliente* gerenciam suas respectivas entidades. *No-*

[1] Aplicação exemplo, disponível em: https://github.com/PqES/ToyExampleMicrosservicos

*tificação* envia e-mails aos clientes no caso de novos produtos. *Venda* é o microsserviço que um vendedor autenticado efetua uma venda de produtos para clientes.

Figura 3.3 – Arquitetura de microsserviços para a aplicação de Controle de Vendas.




### 3.3.2 Especificação Arquitetural

A Listagem 3.2 ilustra a especificação DCL$^+$ pra a aplicação de Controle de Vendas.

```
1  Produto: http://singapura; /system/msProduto; Java
2      Main must-depend SpringBoot                                  "#Produto-RE1"
3      only Controller can-depend DAO                               "#Produto-RE2"
4      only DAO can-useannotation JPA                               "#Produto-RE3"[1]*
5      InterfaceNotificacao must-communicate Notificacao            "#Produto-RC1"+
6
7  Venda: http://nova_york; /system/msVenda; Java
8      Controller must-depend Service                               "#Venda-RE1"
9      Intercomm must-useannotation FeignClient                     "#Venda-RE2"[1]*
10     only InterfaceAutenticacao can-communicate Autenticacao      "#Venda-RC1"[1]+
11     only InterfaceProduto can-communicate Produto                "#Venda-RC2"[2]+
12     only InterfaceCliente can-communicate Cliente                "#Venda-RC3"
13     $system cannot-communicate Notificacao                       "#Venda-RC4"
14
15 Cliente: http://madrid; /system/msCliente; Java
16     Main must-depend SpringBoot                                  "#Cliente-RE1"
17     only Controller can-depend DAO                               "#Cliente-RE2"[2]*
18     Controller must-useannotation RestControllerAnnotation       "#Cliente-RE3"
19     $system can-communicate-only Notificacao                     "#Cliente-RC1"[2]+
20     InterfaceNotificacao must-communicate MsNotificacao using /api/notify
                                                                    "#Cliente-RC2"
21
22 Notificacao: http://londres; /system/MsNotificacao; C#
23     Main must-depend Controller                                  "#Notificacao-RE1"
24     only Controller can-depend DAO, Email                        "#Notificacao-RE2"
25     only Email can-depend MailKit.Net                            "#Notificacao-RE3"
26     DAO must-depend Entities                                     "#Notificacao-RE4"
27
28 Autenticacao: http://paris; /system/MsAutenticacao; C#
29     Main must-depend Controller                                  "#Autenticacao-RE1"
30     only Controller can-depend DAO                               "#Autenticacao-RE2"
31     Controller, DAO must-depend Entities                         "#Autenticacao-RE3"
```

Listagem 3.2 – Especificação arquitetural para a aplicação de Controle de Vendas.

**Microsserviço:** A Listagem 3.2 ilustrada a especificação DCL$^{+}$ de todos os microsserviços. O primeiro microsserviço especificado é identificado como `Produto`, possui URL `http://singapura`, com repositório `/system/msProduto` e desenvolvido na linguagem Java (linha 1). Os microsserviços `Venda` (linha 7), `Cliente` (linha 15), `Notificacao` (linha 22) e `Autenticacao` (linha 28) são especificados de maneira semelhante.

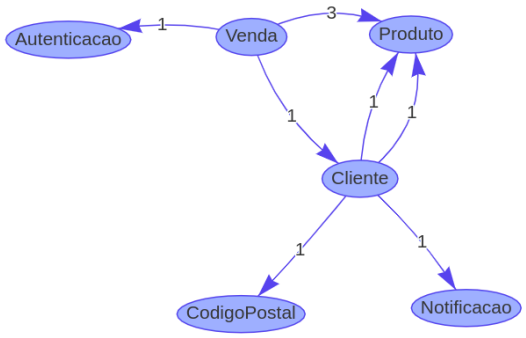
**Restrições de comunicação:** As restrições de comunicação (RC's) são especificadas apenas para os microsserviços `Produto`, `Venda` e `Cliente`. Para o microsserviço `Produto`, a restrição `Produto−RC1` exige que o módulo *InterfaceNotificacao* comunique-se com `Notificao` (linha 5). Já as restrições do microsserviço `Venda`, `Venda−RC1` permite apenas o módulo *InterfaceAutenticacao* comunicar-se com `Autenticao` (linha 10); `Venda−RC2` permite apenas o módulo *InterfaceProduto* comunicar-se com `Produto` (linha 11); `Venda−RC3` permite apenas o módulo *InterfaceCliente* comunicar-se com `Cliente` (linha 12); e `Venda−RC4` não permite que nenhum módulo do microsserviço `Venda` comunique-se com o microsserviço `Notificacao` (linha 13); Por fim, para o microsserviço `Cliente`, a restrição `Cliente−RC1` não permite que nenhum módulo do microsserviço `Cliente` comunique-se com `Notificao` (linha 19); e `Cliente−RC2` exige que o módulo *InterfaceNotificacao* comunique-se com o microsserviço `Notificao` por meio da interface `/api/notify` (linha 20).

**Restrições do projeto estrutural**: As restrições do projeto estrutural (RE's) na linguagem `DCL` são apresentadas para cada microsserviço. Por exemplo, para o microsserviço `Produto`, `Produto−RE1` define que classes `Main` devem depender do *framework* Spring Boot (linha 2); `Produto−RE2` permite apenas a camada de `Controller` de depender de classes `DAO` (linha 3); e `Produto−RE3` restringe o uso da anotação JPA a apenas classes `DAO` (linha 4). As especificações dos demais microsserviços são realizadas de maneira semelhante.

### 3.3.3 Conformidade de Comunicação

**Extrator de comunicação:** A Figura 3.4 apresenta as comunicações extraídas por DCL$^{+}$*check* nos módulos dos microsserviços da aplicação de Controle de Vendas. Em resumo, esses microsserviços realizam nove comunicações, variando entre uma e três comunicações. Como é possível perceber, uma comunicação entre os microsserviços `Cliente` e `CodigoPostal` não estava prevista na arquitetura planejada, conforme apresenta a Figura 3.3 e a Listagem 3.2.

Figura 3.4 – Comunicações extraídas por DCL$^{+}$*check* para a aplicação de Controle de Vendas.




**Verificador de comunicação:** Quando DCL$^{+}$*check* verifica se as comunicações extraídas estão em conformidade com a especificação arquitetural, são detectadas as seguintes *violações de comunicação*, identificadas por + e por [n], onde n compreende ao número de violações, como pode ser observado na Listagem 3.2.

- *Divergências*:
  - Um módulo nomeado `ControlerVenda` presente no microsserviço `Venda` estabelece comunicações não permitidas com os microsserviços `Autenticacao` (uma comunicação) e `Produto` (duas comunicações), violando respectivamente as restrições `Venda`−`RC1` e `Venda`−`RC2`. Essas violações ocorreram pois apenas os módulos

`InterfaceProduto` e `InterfaceAutenticacao` poderiam estabelecer comunicações com os microsserviços `Autenticacao` e `Produto`, respectivamente.

  - Um módulo nomeado `InterfaceProduto` no microsserviço `Cliente` estabelece duas comunicações não permitidas com o microsserviço `Produto`, violando a restrição `Cliente`−`RC1`.

- *Ausência*:

  - Na funcionalidade relacionada ao lançamento de novos produtos, nenhum módulo de `Produto` notifica `Notificao` para enviar e-mails aos clientes, violando a restrição `Produto`−`RC1`.

- *Alerta*:

  - Quando um novo cliente está sendo criado, `Cliente` pede informações ao `CodigoPostal`, através de duas comunicações, para verificar o endereço informado. Essas comunicações não foram especificadas na arquitetura.

### 3.3.4 Conformidade do Projeto Estrutural

**Extrator do projeto estrutural:** `DCL`$^{+}$*check* executou as ferramentas de dependência do projeto estrutural `JavaDepExtractor` e `CsDepExtractor` sob os microsserviços da aplicação Controle de Vendas. A Tabela 3.1 apresenta a quantidade de dependências extraídas por `DCL`$^{+}$*check* em todos os microsserviços pertencentes a essa aplicação.

**Verificador do Projeto Estrutural:** Quando `DCL`$^{+}$*check* executa piDCLcheck para verificar se as dependências internas de cada microsserviço estão em conformidade com a arquitetura planejada, apresentada na Listagem 3.2, as seguintes violações do *projeto estrutural* foram detectadas:

Tabela 3.1 – Dependências do projeto estrutural extraídas por DCL$^{+}$*check*.

| Microsserviço | Dependências Extraídas |
|---|---|
| Venda | 56 |
| Autenticacao | 99 |
| Cliente | 69 |
| Produto | 72 |
| Notificacao | 85 |
| **Total** | **381** |

- *Divergências*:
  - `Produto` tem uma classe `ProdutoController` fora do módulo `DAO` usando JPA, o que viola a restrição `Produto`−`RE3` que estabelece que somente `DAO` pode usar anotações `JPA` (linha 4 da Listagem 3.2);
  - `Cliente` tem uma classe `Cliente` e uma classe `ClienteApp` fora do módulo `Controller` declarando DAO, o que viola `Cliente`−`RE2` que estabelece que somente `Controller` pode depender de `DAO` (linha 17 da Listagem 3.2);
- *Ausência*:
  - `Venda` tem uma interface `InterfaceCliente` no módulo `Intercomm` que não usa a anotação `FeignClient`, o que viola `Venda`−`RE2` que estabelece que todas as interfaces do módulo `Intercomm` devem utilizar a anotação `FeignClient` (linha 9 da Listagem 3.2).

## 3.4 Considerações Finais

Este capítulo apresentou a solução de conformidade arquitetural proposta nesta dissertação. Inicialmente foi definida a linguagem de restrição arquitetural DCL$^{+}$ – estendida da previamente proposta linguagem DCL – para especificar as comunicações desejáveis entre microsserviços de uma aplicação. Para isso, a linguagem estabelece um novo tipo de dependência denominado `communicate`, além de definir restrições *only-can*, *can-only* e *cannot* para detectar divergências e

restrições *must* para detectar ausências de comunicação. A linguagem DCL$^+$ permite também a especificação da nomenclatura `using` para restringir a interface de comunicação entre microsserviços, além de considerar comunicações de um microsserviço com outro(s) que não faz(em) parte da aplicação, definida por meio de `alertas`.

Posteriormente, foi apresentada a abordagem de conformidade arquitetural DCL$^+$, a qual consiste na principal contribuição deste estudo. Essa abordagem possui como entrada uma Aplicação desenvolvida com uma suíte de microsserviços juntamente com sua Especificação Arquitetural. A execução da abordagem é apoiada por meio dos Extratores e Verificadores de Comunicação e do Projeto Estrutural. A execução da abordagem resulta em um conjunto de violações de comunicação e do projeto estrutural.

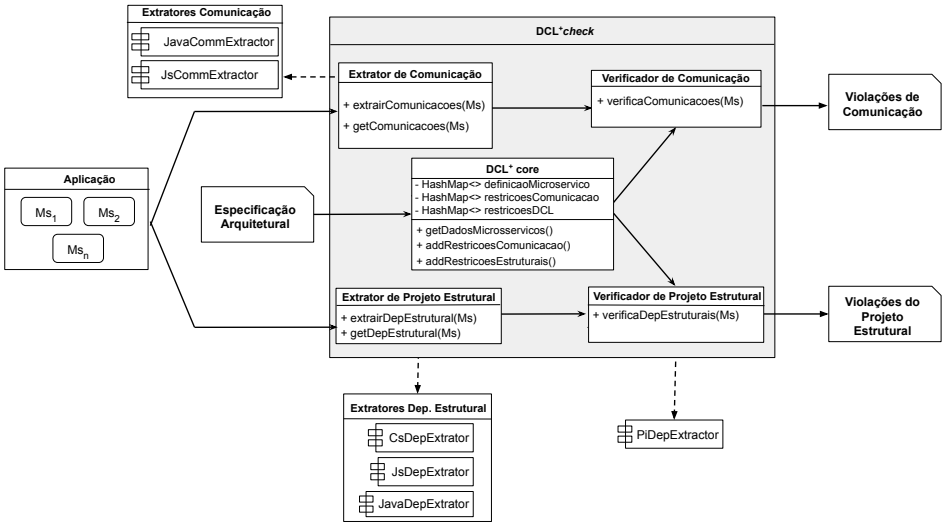
Por fim, para ilustrar a aplicabilidade da abordagem proposta, foi realizada uma avaliação controlada em uma aplicação com microsserviços compostos por meio de coreografia. Para isso, foram implementados um conjunto de cinco microsserviços (desenvolvidos em Java e C#) para o contexto de vendas de produtos. Nessa avaliação, foram especificadas sete restrições de comunicação e 14 restrições do projeto estrutural. A execução da abordagem resultou na extração de nove comunicações e 381 dependências do projeto estrutural. Como resultados, no que compete à conformidade de comunicação, DCL$^+$ reportou sete violações de comunicação (cinco divergências, uma ausência e um alerta). Já para a conformidade do projeto estrutural, DCL$^+$ reportou quatro violações (três divergências e uma ausência).

# 4 FERRAMENTA DE CONFORMIDADE ARQUITETURAL

Este capítulo apresenta a implementação da abordagem de conformidade arquitetural proposta denominada DCL$^{+}$*check*. A Seção 4.1 apresenta as entradas necessárias para a ferramenta. A Seção 4.2 descreve os módulos internos de DCL$^{+}$*check*. A Seção 4.3 ilustra as saídas da ferramenta. A Seção 4.4 destaca as limitações atuais da ferramenta DCL$^{+}$*check* e, por fim, a Seção 4.5 apresenta as considerações finais deste capítulo.

A Figura 4.1 ilustra a arquitetura da ferramenta DCL$^{+}$*check*. Basicamente, a ferramenta possui como entrada a *Aplicação* composta por uma suíte de microsserviços, juntamente com sua *Especificação Arquitetural*. A execução da ferramenta é apoiada pelos módulos *DCL$^{+}$core*, *Extratores* e *Verificadores de Comunicação* e *do Projeto Estrutural*. As saídas retratam as *Violações de Comunicação* e *Violações do Projeto Estrutural*.

Figura 4.1 – Arquitetura DCL$^{+}$*check*.



## 4.1 Entradas

Conforme apresenta a Figura 4.1, as entradas para DCL$^+$*check* são a *Aplicação* e sua respectiva *Especificação Arquitetural* em DCL$^+$. No que compete à aplicação, DCL$^+$*check* requer o acesso ao repositório do código-fonte de cada microsserviço. Isso é necessário para que sejam extraídas as comunicações estaticamente realizadas entre os microsserviços, bem como a extração das dependências estruturais de cada microsserviço. Em contrapartida, a especificação arquitetural consiste em um arquivo de texto onde são definidas as restrições em DCL$^+$ (Seção 3.2.2).

## 4.2 DCL$^+$*check*

A arquitetura interna da ferramenta DCL$^+$*check* possui, como componentes principais, o módulo DCL$^+$*core*, Extratores e Verificadores de Comunicação e do Projeto Estrutural (Figura 4.1). As próximas subseções detalham cada um desses componentes.

### 4.2.1 DCL$^+$*core*

Nesse módulo, as informações presentes na especificação arquitetural são capturadas por meio de expressões regulares (Regex[1]) a fim de detectar padrões de definição de um microsserviço, restrições de projeto estrutural e restrições de comunicação. As informações capturadas são armazenadas em estruturas de dados *HashMap* com o intuito de posteriormente verificar a conformidade de comunicação e do projeto estrutural.

### 4.2.2 Extrator de Comunicação

Esse módulo extrai, do código-fonte de cada um dos microsserviços, as declarações de chamadas à outros microsserviços e armazena tais informações

[1] *Regex*, disponível em: https://docs.oracle.com/javase/7/docs/api/java/util/regex/

em uma estrutura de dados *HashMap*. A atual implementação da ferramenta DCL$^{+}$*check* utiliza técnica de análise estática para detectar as chamadas entre os microsserviços, conforme ilustra a Figura 4.2. A análise estática consiste em extrair informações do código-fonte de uma aplicação, sem que seja necessária sua execução (ERNST, 2003). Isso garante uma compreensão precisa da estrutura do programa, independentemente de quais entradas ou em que ambiente o programa é executado.

Figura 4.2 – Extrator de comunicação


As extrações estáticas das comunicações realizadas entre os microsserviços deve ser implementada para cada uma das diferentes linguagens que compõe a aplicação de microsserviços (MAYER; WEINREICH, 2018). Esta dissertação de mestrado possui implementação da extração estática das comunicações por meio de AST (*Abstract Syntax Tree*) para os microsserviços desenvolvidos em Java e JavaScript. Espera-se a implementação do extrator de comunicação para a linguagem C# como trabalho futuro desta dissertação de mestrado, assim como para as demais linguagens de programação.

**Extração estática de comunicação em Java:** Para o contexto dos serviços implementados em Java, a extração é realizada por meio da identificação de clientes *Feign* (Seção 2.2.3). Com a utilização desse tipo de cliente, é possível identificar a composição dos serviços realizadas por meio de coreografia e orquestração quando todos os serviços, inclusive o orquestrador, são implementados na linguagem Java.

Nessa extração, todos os nós representados na árvore são subclasses de um nó principal denominado *ASTNode*. Por exemplo, há nós para representar anotações, declarações de métodos, declarações de variáveis etc. O Algoritmo 1 apresenta o pseudocódigo de como é realizada a descoberta das comunicações por meio da AST.

**Algoritmo 1:** Descoberta de comunicação estática na AST Java

```
1  Entrada: Microsserviço msOrigem
2  Saída: Conjunto de comunicações de msOrigem
3  Comunicacoes ← ∅;
4  forall class C_i ∈ msOrigem do
5      dependenciasClass ← extrair_dependencias(C_i);
6      forall dependencia D_i ∈ dependenciasClass do
7          if D_i is a FieldAnnotationDependency and D_i.Nome ==
             "Autowired" then
8              declaracao ← D_i.Declaracao
9              if declaracao is a ClassSingleAnnotationDependency
                 and declaracao.Nome == "FeignClient" then
10                 msDestino = declaracao.Valor
11             end
12             else if declaracao is a MethodAnnotation then
13                 interface ← declaracao.Valor
14             end
15             Comunicacoes ←
                 Comunicacoes ∪ {comunicacao(msDestino, interface)}
16         end
17     end
18 end
```

O algoritmo recebe como entrada o microsserviço a ser analisado, denominado *msOrigem* e retorna o conjunto de comunicações *Comunicacoes* do microsserviço. Para cada classe $C_i$ (linha 5) de *msOrigem*, são extraídas as dependências a partir do nó principal *ASTNode* da AST Java para lista *dependenciasClass*. Para cada dependência $D_i$ da classe analisada, caso a dependência seja uma anotação

do tipo *@Autowired* (linha 7), são extraídas pela AST, as classes que possuem dependências com alguma interface *@FeignClient* (linha 9) e é obtido então, o valor presente na expressão da anotação referenciando *msDestino* (linha 10). Por consequência, são identificadas as declarações de anotações de métodos (linha 13) e são obtidas todas as interfaces de comunicação entre *msOrigem* e *msDestino*. Por fim, as comunicações mapeadas são adicionadas à lista *Comunicacoes* do microsserviço analisado (linha 15.

A Listagem 4.1 apresenta um fragmento de código do microsserviço `Venda`, da aplicação Controle de Vendas (ver Seção 3.3) para comunicações com o microsserviço `Produto`.

```
1 package com.intercomm;
2
3 @FeignClient(value="Produto")
4 public interface ProductInterface {
5     @RequestMapping( method = RequestMethod.GET,
6     value="/getProducts")
7     public String getAllProduct();
8     ...
9 }
```

Listagem 4.1 – Fragmento da interface de comunicação do microsserviço `Venda` com o microsserviço `Produto`.

Nesse tipo de comunicação, um módulo denominado `JavaCommExtractor` é utilizado e as comunicações são mapeadas para uma estrutura HashMap. Tais informações apresentam-se no formato `dir−classe communicate microservice using interface`, onde `dir−classe` é o diretório e classe onde a comunicação foi identificada. No exemplo apresentado `ProductInterface` comunica com o microsserviço `Produto` usando a interface `getProducts` (linha 4). Dessa maneira, a saída é gerada: `com.intercomm.ProductInterface communicate Produto using getProducts`.

É importante ressaltar que, o módulo `JavaCommExtractor` pode ser utilizado externamente à ferramenta `DCL`$^{+}$*check*[2]. Para isso, basta executar o comando `java−jar JavaCommExtractor` [`folder−microservice`], onde a en-

[2] *JavaCommExtractor*, disponível em: https://github.com/PqES/JavaCommExtractor

trada [`folder`-microservice] compreende a pasta do microsserviço a ser analisado. Como saída, `JavaCommExtractor` reporta um arquivo texto denominado *communications.txt*.

O motivo pelo qual esta dissertação de mestrado extrai somente as anotações *@FeignClient*, é devido ao fato que o projeto *Spring Cloud* (Seção 2.2.3) define esse tipo de anotação para estabelecer a comunicação entre microsserviços implementados na linguagem Java.

**Extração estática das comunicações em JavaScript:** Para o contexto de microsserviços desenvolvidos na linguagem *JavaScript*, foi desenvolvido um módulo extrator voltado ao *framework* Node.js. Essa necessidade surgiu devido a grande aplicabilidade de tal linguagem para o contexto de aplicações *web* e também a sua utilização em aplicações acadêmicas (ADERALDO et al., 2017; ZHOU et al., 2018), tais como (i) *AcmeAir*, (ii) Spring Cloud Demo Apps e (iii) TrainTicket.

Para auxiliar na identificação das comunicações na AST JavaScript, foram utilizadas as bibliotecas `Espree`[3] e `Estraverse`[4]. A biblioteca `Espree` compreende a um *parser* para realizar análise léxica e sintática de códigos JavaScript. Já a biblioteca `Estraverse` provê funcionalidades para navegar na AST, permitindo alterações ao longo da mesma. Para o contexto do *framework* Node.js, as bibliotecas mais utilizadas para requisições HTTP são as bibliotecas `Express`[5] e `Request`[6]. `Express` é uma biblioteca para o contexto do *framework* Node.js, utilizada no desenvolvimento de aplicações *web*. Já a biblioteca `Request` é uma biblioteca projetada para ser a maneira mais simples de fazer chamadas HTTP.

A Listagem 4.2 apresenta um exemplo de utilização da API Express para a realização de comunicações HTTP. Inicialmente, é necessário importar a biblioteca `express` (linha 1), onde todas as definições para criação de rotas são

[3] *Espree*, disponível em: https://www.npmjs.com/package/espree
[4] *Estraverse*, disponível em: https://github.com/estools/estraverse
[5] *Express*, disponível em: https://expressjs.com/pt-br/
[6] *Request*, disponível em: https://www.npmjs.com/package/request

importadas junto a classe `Router()`[7] (linha 3). A fim de manter uma maneira padronizada, permitir a reutilização e facilitar a manutenção, uma constante (`const ms1`) é definida com a rota para um microsserviço (linha 5) e um exemplo de requisição `get` (linha 7) é apresentado.

```
1  import { Router } from 'express'
2  ...
3  const routes = Router()
4  // Rota para comunicacao do microsservico Ms1
5  const ms1 = '/ms1/getFoo'
6  routes.get('/', (req, res) => {
7      const route = getRoute(ms1)
8      ...
9  }
10 const getRoute = (route) => {
11   return 'http://localhost:8080' + route
12 }
```

Listagem 4.2 – Comunicação utilizando a API Express no arquivo `rotaMicrosservicoMs1`.

Para as requisições definidas através da biblioteca `Express`, DCL$^{+}$*check* inicialmente mapeia todas as constantes `const`, as quais possuem um `path` (como apresentado na linha 5) ou uma atribuição de alguma biblioteca, definidas no código-fonte e seu valor. Na Listagem 4.2, DCL$^{+}$*check* mapeia a constante `ms1` = '`/ms1/getFoo`'. Posteriormente, DCL$^{+}$*check* mapeia todas as utilizações de funções da biblioteca `Router` (por meio das funções `get`, `post`, `update`, `delete` etc. da API REST) e analisa se naquele bloco de código há a utilização de alguma constante mapeada. Por exemplo, DCL$^{+}$*check* analisa o bloco de código das linhas 6 à 9 e verifica a utilização da constante `route` mapeada na linha 7. Dessa maneira, DCL$^{+}$*check* identifica a comunicação do arquivo `rotaMicrosservicoMs1` com o microsserviço `ms1` através da interface `getFoo`.

A Listagem 4.3 apresenta um exemplo de utilização da biblioteca `Request` para a realização de comunicações HTTP. Com a utilização dessa biblioteca, a maneira de realizar a importação das bibliotecas a serem utilizadas no código é por meio da função `require` (linha 1) e não através do `import` como na Lista-

[7] *Router*, disponível em: https://expressjs.com/pt-br/guide/routing.html

gem 4.2. Nesse caso, uma constante `const` deve representar uma biblioteca a ser utilizada, como apresentado também na linha 1 por meio da importação da biblioteca `Request`, com atribuição à constante `request`. No exemplo apresentado, a função `foo` (linha 3), realiza uma requisição `get` para a rota do microsserviço `Ms1` (linha 5).

```
1 const request = require('request');
2 ...
3 module.exports.foo = function(req, res) {
4     let routeMs1 = getMsURL('http://localhost:8080/ms1/getFoo/');
5     request.get(routeMs1,...);
6     );
7     ...
8 }
```

Listagem 4.3 – Comunicação utilizando a API Request.

Semelhante à estratégia apresentada para a biblioteca `Express`, `DCL`$^{+}$*check* mapeia a constante que representa a atribuição da biblioteca `Request` (linha 1) e analisa todas as chamadas das funções relacionadas a ela (`get`, `post`, `delete` e `update` da API REST) (linha 5) e mapeia na AST o caminho referente ao primeiro parâmetro, a qual representa a rota para o microsserviço, até encontrar um `path` para esse serviço. No exemplo apresentado, a rota capturada compreende a `ms1/getFoo`, ou seja, uma comunicação é realizada com o microsserviço `Ms1` através da interface `getFoo`.

Para o contexto de aplicações onde os microsserviços sejam desenvolvidos na linguagem JavaScript, um módulo denominado `JsCommExtractor` é utilizado e as comunicações são mapeadas para uma estrutura HashMap. Essas informações apresentam-se no formato `dir−arquivo communicate microservice using interface`, onde `dir−arquivo` é o diretório e arquivo onde a comunicação foi identificada. Para os exemplos apresentados nas Listagens 4.2 e 4.3, no arquivo `rotaMicrosservicoMs1`, uma comunicação é realizada com o microsserviço `ms1` através da interface `getFoo`, ou seja, a seguinte saída é gerada: `.../rotaMicrosservicoMs1.js communicate ms1 using getFoo`.

É válido ressaltar que o módulo `JsCommExtractor`[8] pode ser utilizado a partir do comando *bash* ./`JsCommExtractor` [`dir−microservice`], onde a tag [`dir−microservice`] é o diretório completo do código-fonte do microsserviço. Como saída, `JsCommExtractor` reporta um arquivo texto denominado *communications.txt*.

É importante salientar que a abordagem proposta implementa a descoberta das comunicações estáticas para a linguagem Java e JavaScript. Contudo, a descoberta dessas informações é extensível para outras linguagens, uma vez que é necessária a utilização da AST da linguagem especificada.

### 4.2.3 Verificador de Comunicação

Esse módulo verifica se as informações contidas na estrutura de dados *HashMap*, reportada pelo extrator de comunicação, estão em conformidade com as informações contidas na estrutura *HashMap* extraídas da especificação arquitetural. Essas informações são analisadas a fim de identificar violações arquiteturais de comunicação.

### 4.2.4 Extrator do Projeto Estrutural

Esse módulo é apoiado por extratores externos desenvolvidos para as linguagens Java por meio do *JavaDepExtractor*, JavaScript por meio do *JsDepExtractor* e C# por meio do *CsDepExtractor*. Essas ferramentas extraem as dependências de projeto estrutural conforme a linguagem implementada em cada microsserviço que compõe a aplicação. Em outras palavras, essas ferramentas extraem as dependências de um microsserviço em um conjunto de triplas no formato [`dir−file`, `dependency−type`, `target−file`] e definem como saída um arquivo texto com as dependências especificadas (*dependencies.txt*). As próximas subseções apresentam o funcionamento das ferramentas abordadas. Para isso, fo-

[8] *JsCommExtractor*, disponível em: https://github.com/PqES/jsCommExtractor

ram utilizados fragmentos dos códigos desenvolvidos para a aplicação de Controle de Vendas, previamente apresentada na Seção 3.3.

#### 4.2.4.1 JavaDepExtractor

Essa ferramenta extrai as dependências do projeto estrutural dos microsserviços implementados na linguagem Java. A Listagem 4.4 apresenta o fragmento da implementação da interface `ProductInterface` do microsserviço `Venda`, a fim de ilustrar o funcionamento de *JavaDepExtractor*.

```
1  package com.intercomm;
2  @FeignClient(value="Produto")
3  public interface ProductInterface {
4      @RequestMapping(method=RequestMethod.GET,
           value="/product/getProduct/{id}")
5      public String getProduct(@PathVariable("id") String id);
6
7      @RequestMapping(method = RequestMethod.GET,value="/getProducts")
8      public String getAllProduct();
9
10     @RequestMapping(method =
           RequestMethod.POST,value="/venda/{id}/{qnt}")
11     public String updateStock(@PathVariable("id") String[] id,
           @PathVariable("qnt") String[] qnt);
12 }
```

Listagem 4.4 – Interface de comunicação do microsserviço `Venda` com `Produto`.

Como pode ser observado, foi implementada a interface `ProductInterface` a qual implementa abstrações para requisições ao microsserviço `Produto`, definida pela anotação `FeignClient` (linha 3) (ver Seção 2.2.3). As anotações para mapeamento de requisições `RequestMapping` definidas nas linhas 4, 7 e 10 são disponibilizadas pelo *framework* Spring Boot[9] para o mapeamento de requisições REST.

A Listagem 4.5 apresenta o arquivo de dependências (dependencies.txt) da interface `ProductInterface` extraídas por *JavaDepExtractor*. Para isso, é necessário executar o comando `java −jar javadepextractor` [`dir−projeto`], onde [`dir−projeto`] compreende ao diretório do projeto da aplicação. Como é

[9] *Spring Boot*, disponível em: https://projects.spring.io/spring-boot/

possível perceber, o extrator retorna, além das dependências dessa interface (linhas 2, 4, 5, e 6), dependências da mesma com classes da `API` Java (linha 1).

```
1 com.intercomm.ProductInterface,declare,java.lang.String
2 com.intercomm.ProductInterface,useannotation,org.springframework.
      web.annotation.RequestMapping
3 com.controller.SaleController,declare,com.controller.
      ProductInterface
4 com.controller.Services,access,com.controller.ProductInterface
5 com.controller.Services,declare,com.controller.ProductInterface
6 com.intercomm.ProductInterface,useannotation,org.springframework.
      netflix.feign.FeignClient
```

Listagem 4.5 – Dependências da interface `ProductInterface`.

#### 4.2.4.2 JsDepExtractor

O extrator de dependências estruturais para a linguagem JavaScript foi desenvolvido na linguagem Java com a utilização da biblioteca *Closure Compiler*[10]. Essa biblioteca compreende a um compilador desenvolvido na linguagem Java para manipulação da AST em JavaScript. A Listagem 4.6 apresenta um fragmento de código disponível no arquivo `authenticationRoutes` do microsserviço `Venda`. Basicamente, esse fragmento compreende à uma requisição `get` para as interfaces disponibilizadas pelo microsserviço `Autenticacao`.

```
1  // Folder: Venda/routes/authenticationRoutes
2  var authenticationController =
       require('../controllers/authenticationController');
3  ...
4  module.exports = function(app) {
5      app.get('/', authenticationController.login);
6      app.get('/login', authenticationController.login);
7      app.get('/autenticacao/:username/:password',
           authenticationController.authenticate);
8      app.get('/')
9  }
10 ...
```

Listagem 4.6 – Requisições as interfaces disponibilizadas pelo microsserviço `Autenticacao`.

A Listagem 4.7 apresenta o arquivo de dependência (*dependencies.txt*) extraído pelo *JsDepExtractor* do arquivo `authenticationRoutes`. Para isso, é

[10] *Closure Compiler*, disponível em: https://github.com/google/closure-compiler

necessário executar o comando `java −jar JsDepextractor [dir−projeto]`, onde `[dir−projeto]` compreende ao diretório do projeto da aplicação. Um exemplo é a dependência extraída do arquivo `authenticationRoutes` com a função `AuthenticationController` presente no pacote `controllers` (linha 1).

```
1 Venda.routes.authenticationRoutes,access,Venda.controllers.
      authenticationController.AuthenticationController
2 Venda.routes.authenticationRoutes,access,Venda.controllers.
      authenticationController.login
3 Venda.routes.authenticationRoutes,access,Venda.controllers.
      authenticationController.authenticate
4 Venda.routes.authenticationRoutes,access,Venda.routes.
      authenticationRoutes.exports
```

Listagem 4.7 – Dependências do arquivo `authenticationRoutes`.

#### 4.2.4.3 CsDepExtractor

Essa ferramenta extrai as dependências estruturais dos microsserviços implementados na linguagem C#. É válido ressaltar que a atual implementação dessa ferramenta é considerado um protótipo, uma vez que a extração das dependências são realizadas por meio de análise textual.

A Listagem 4.8 apresenta o fragmento da classe – presente no microsserviço `Autenticacao` – `AuthenticateController` . Basicamente, a classe realiza a autenticação de um funcionário por meio de um `username` e uma `senha` (linha 9), através da interface `api/[authenticate]` (linha 4) por meio do método POST (linha 8) de requisições REST.

```
1  namespace MsAuthentication.Controllers{
2      [Route("api/[authenticate]")]
3      public class AuthenticateController : Controller{
4          public bool Authenticate(string username, string password){
5              string userLine = UserDAO.GetUserByUsername(username);
6              if(userLine != null){
7                  string[] userSplited = userLine.Split(';');
8                  User user = new User(userSplited[0], userSplited[1]);
9                  return user.Password == password;
10             }
11             return false;
12         }
13     }
14 }
```

Listagem 4.8 – Fragmento da classe `AuthenticationController`.

A Listagem 4.9 apresenta o arquivo de dependências (*dependencies.txt*) extraídas por *CsDepExtractor* para a classe `AuthenticationController`. Para obter tais dependências do projeto estrutural, é necessário executar o comando `CsDepextractor` [`dir−projeto`]. Um exemplo de dependência extraída é a extensão da classe `Controller` realizada pela classe `AuthenticationController` (linha 5 da Listagem 4.8), apresentada na linha 1.

```
1  Auth.Controller.AuthenticateController,extend,Controller
2  Auth.Controller.AuthenticateController,useannotation,Route
3  Auth.Controller.AuthenticateController,declare,bool
4  Auth.Controller.AuthenticateController,useannotation,HttpPost
5  Auth.Controller.AuthenticateController,declare,string
6  Auth.Controller.AuthenticateController,access,Auth.DAO.UserDAO
7  Auth.Controller.AuthenticateController,declare,char
8  Auth.Controller.AuthenticateController,access,string
9  Auth.Controller.AuthenticateController,declare,Auth.Entities.User
10 Auth.Controller.AuthenticateController,create,Auth.Entities.User
11 Auth.Controller.AuthenticateController,access,Auth.Entities.User
```

Listagem 4.9 – Dependências da classe `AuthenticationController` do microsserviço `Autenticacao`.

### 4.2.5 Verificador do Projeto Estrutural

Esse módulo analisa se dependências estabelecidas internamente à algum microsserviço – independente de sua linguagem – estão de acordo com o seu projeto arquitetural. Para tal propósito, foi utilizada a ferramenta `piDCL`*check*[11], o qual constitui outra contribuição prática deste estudo. A ferramenta define um conjunto de triplas no formato [`dcl−file`, `folder−dir`, `dependencies−file`]. A entrada [`dcl−file`] compreende às restrições de projeto arquitetural de cada microsserviço, [`folder−dir`] especifica o diretório onde encontra-se o arquivo de dependências e, por fim, [`dependencies−file`] compreende ao arquivo de dependências gerados pelos extratores estruturais. Como resultado, a ferramenta reporta um conjunto de violações do projeto estrutural. Na ferramenta `DCL`$^{+}$*check*, essas violações são agrupadas e reportadas em um arquivo texto juntamente com as violações de comunicação.

[11] *PiDCLcheck*, disponível em: https://github.com/rterrabh/pi-dclcheck

## 4.3 Saídas

As saídas da ferramenta DCL$^{+}$*check* compreende a dois arquivos texto, o arquivo de *Violações de Comunicação* e *de Projeto Estrutural*.

**Violações de comunicação:** A Listagem 4.10 apresenta as violações de comunicação reportadas por DCL$^{+}$*check* para a aplicação Controle de Vendas (ver Seção 3.3). Nessa aplicação, DCL$^{+}$*check* reportou sete violações de comunicação. O microsserviço `Produto` (linha 1) foi responsável pela ausência de comunicação. Já o microsserviço `Venda` (linha 4) foi responsável por três divergências e o microsserviço `Cliente` (linha 14) por outras duas divergências e um alerta.

Como pode ser observado, DCL$^{+}$*check* reportou sete violações de comunicação (cinco divergências, uma ausência e um alerta). Por exemplo, uma ausência foi detectada no módulo `InterfaceNotificacao` do microsserviço `Produto` onde esse módulo não comunica com o microsserviço `Notificacao` (linha 2). Um exemplo de divergência reportada é apresentada no módulo `InterfaceProduto` do microsserviço `Venda` (linhas 5 e 6). Nesse caso, uma restrição define que apenas o módulo `InterfaceProduto` do microsserviço `Venda` pode comunicar com o microsserviço `Produto` (linha 5), porém o módulo `ControllerVenda` também estabelece comunicações com o microsserviço `Produto` usando a interface `getAllProducts` (linha 6). É válido ressaltar que as demais divergências são apresentadas de maneira semelhante.

```
1  Produto
2     AUSENCIA: [.../InterfaceNotificacao] MUST_COMUNICATE Notificacao
3
4  Venda:
5     DIVERGENCIA: [.../InterfaceProduto] ONLY_CAN_COMMUNICATE
          Produto
6     [.../ControlerVenda] COMMUNICATE Produto using /getAllProducts/
7
8     DIVERGENCIA: [.../InterfaceProduto] ONLY_CAN_COMMUNICATE
          Produto
9     [.../ControlerVenda] COMMUNICATE Produto using
          /product/getProduct/{id}
10
```

```
11    DIVERGENCIA: [.../InterfaceAutenticacao] ONLY_CAN_COMMUNICATE
          Autenticacao
12    [.../ControlerVenda] COMMUNICATE Autenticacao using /autheticate
13
14  Cliente:
15    ALERTA: [.../Cliente/*] CAN_COMMUNICATE_ONLY Notificacao
16    [.../ClienteController] COMMUNICATE viacep using ws/{cep}/json/
17
18    DIVERGENCIA: [.../Cliente/*] CAN_COMMUNICATE_ONLY Notificacao
19    [.../InterfaceProduto] COMMUNICATE Produto using
          /product/getProduct/{id}
20
21    DIVERGENCIA: [.../Cliente/*] CAN_COMMUNICATE_ONLY Notificacao
22    [.../InterfaceProduto] COMMUNICATE Produto using
          /product/getAllProducts
```

Listagem 4.10 – Violações de Comunicação reportadas por DCL$^{+}$*check* da aplicação Controle de Vendas.

**Violações do projeto estrutural:** No que compete as saídas referentes as violações do projeto estrutural, a Listagem 4.11 apresenta como são reportadas essas violações pela ferramenta DCL$^{+}$*check*. Como pode ser observado, para esse tipo de restrição, DCL$^{+}$*check* reportou quatro violações (três divergências e uma ausência). O microsserviço `Produto` é responsável por uma divergência (linha 2). O microsserviço `Venda` é responsável pela ausência detectada (linha 6). Por fim, o microsserviço `Cliente` é responsável pelas outras duas divergências reportadas (linhas 9 e 12).

```
1   Produto:
2     DIVERGENCIA: [.../ProdutoDAO] ONLY_CAN_USEANNOTATION [JPA]
3     [.../ProdutoController] USEANNOTATION
          [javax.transaction.Transactional]
4
5   Venda:
6     AUSENCIA: [...InterfaceCliente] MUST_USEANNOTATION
          [.*.netflix.feign.FeignClient.**]
7
8   Cliente:
9     DIVERGENCIA: [.../controller.**] ONLY_CAN_DEPEND [javax.**]
10    [.../Cliente] USEANNOTATION [javax.transaction.Transactional]
11
12    DIVERGENCIA: [.../controller.**] ONLY_CAN_DEPEND [javax.**]
13    [.../ClienteApp] USEANNOTATION [javax.transaction.Transactional]
```

Listagem 4.11 – Violações de projeto estrutural reportadas por DCL$^{+}$*check* da aplicação Controle de Vendas.

Como exemplo, uma restrição define que apenas o módulo `ProdutoDAO` pode usar anotações `JPA` (linha 2), mas a classe `ProdutoController` faz uso dessa anotação (linha 3). As demais violações são interpretadas similarmente.

## 4.4 Limitações

As principais limitações da ferramenta DCL$^{+}$*check* estão concentradas nos módulos *Extratores de Comunicação e do Projeto Estrutural*. Para o contexto do módulo *Extrator de Comunicação*, as limitações estão ligadas à capacidade da ferramenta em extrair comunicações entre microsserviços. Em sua implementação atual, a ferramenta é capaz de extrair comunicações por meio da análise estática de código-fonte para o contexto das bibliotecas `FeignClient` (Java) e `Express` e `Request` (JavaScript). Assim, comunicações estabelecidas através de métodos detectáveis apenas durante a execução ou através do uso de outras bibliotecas não serão detectadas, gerando falsos negativos. Por outro lado, não ocorrem falsos positivos, isto é, todas as comunicações detectadas pela ferramenta representam de fato comunicações estabelecidas entre os microsserviços.

No que se refere as limitações do módulo *Extrator do Projeto Estrutural*, DCL$^{+}$*check* também pode gerar falsos negativos, uma vez que a ferramenta não é capaz de capturar e tratar informações dinâmicas, tais como execuções de métodos por reflexão. Novamente, em contrapartida, DCL$^{+}$*check* não reporta falsos positivos, i.e., a ferramenta não reporta violações do projeto estrutural de forma equivocada.

Para superar as limitações apresentadas, planeja-se como trabalhos futuros, estender a descoberta de comunicações por meio da aplicação de técnicas de análise dinâmica de código-fonte, a qual permite examinar o comportamento real e exato durante sua execução, gerando resultados precisos para um determinado tipo de entrada. Ademais, planeja-se também aprimorar a análise estática através da utilização de recursos capazes de detectar em baixo nível, requisições

e respostas entre serviços *web* (*request* e *response*), tornando o mapeamento das comunicações independente de *frameworks* e bibliotecas utilizadas na aplicação.

## 4.5 Considerações Finais

Neste capítulo foi apresentada a arquitetura da ferramenta DCL$^+$*check*. Para isso, foram apresentados os principais módulos que compõem a ferramenta, descrevendo suas principais estruturas de dados e bibliotecas utilizadas. Inicialmente, foram ilustradas as duas entradas da ferramenta: *Aplicação* e sua respectiva *Especificação Arquitetural*. A aplicação refere-se ao repositório do código-fonte de cada microsserviço. Já a especificação arquitetural consiste em um arquivo de texto onde são definidas as restrições em DCL$^+$ (Seção 3.2.2).

A execução da ferramenta é realizada por meio dos componentes DCL$^+$*core*, *Extratores e Verificadores de Comunicação e Projeto Estrutural*. Basicamente, o módulo DCL$^+$*core* é responsável por mapear as informações presentes na especificação arquitetural para estruturas de dados *HashMap* no intuito de posteriormente verificar a conformidade de comunicação e de projeto estrutural.

O módulo *Extrator de Comunicação* extrai as comunicações realizadas entre os microsserviços da aplicação. Para isso, foram desenvolvidos extratores de comunicação para as linguagens Java (para o contexto de comunicações *FeignClient*) e JavaScript (para o contexto de requisições HTTP definidas pelas bibliotecas `Express` e `Request`). Na ferramenta DCL$^+$*check*, as informações extraídas nos extratores de comunicação são também mapeadas para estruturas de dados *HashMap*. Porém, é possível ter acesso a tais ferramentas a partir de arquivos executáveis disponibilizados em repositórios do GitHub. Uma vez mapeadas as informações das comunicações extraídas, essas são comparadas às restrições definidas na especificação arquitetural, através do módulo *Verificador de Comunicação*.

Já o módulo *Extrator de Projeto Estrutural* é apoiado por ferramentas externas para extrair dependências estruturais para microsserviços implementados

nas linguagens Java por meio do *JavaDepExtractor*, JavaScript por meio do *JsDepExtractor* e C# por meio do *CsDepExtractor*. As dependências extraídas são também mapeadas para estruturas de dados *HashMap* e posteriormente comparadas às restrições do projeto estrutural definidas na especificação arquitetural, através do módulo *Verificador do Projeto Estrutural*. É válido ressaltar que os extratores de dependências estruturais também são contribuições práticas desta dissertação de mestrado.

Como saída, a ferramenta DCL$^+$*check* reporta aos arquiteto dois arquivos texto, sendo um referente às *Violações de Comunicação* e outro referente às *Violações do Projeto Estrutural*. Nesses arquivos são apresentadas as respectivas divergências e ausências de comunicação entre os microsserviços que compõem a aplicação, bem como as divergências e ausências presentes nos projetos estruturais de cada microsserviço.

Por fim, este capítulo apontou as atuais limitações da ferramenta DCL$^+$*check* e como tais limitações podem ser contornadas em trabalhos futuros no intuito de melhorar o espectro da descoberta de comunicações entre microsserviços de uma aplicação.

# 5 AVALIAÇÃO

Este capítulo apresenta uma avaliação de um estudo de caso real a fim de demonstrar a aplicabilidade da abordagem DCL$^+$. A aplicação avaliada compreende à uma aplicação financeira de médio porte e complexa, focada na gestão de vendas e conciliação com cartões de crédito, cartões de débito, *tickets* e outros meios de pagamentos online. Dessa maneira, devido a um acordo de confidencialidade, será omitido o nome da empresa neste documento e nos referiremos a essa aplicação como Conciliação Bancária.

A Seção 5.1 apresenta a metodologia de avaliação conduzida no estudo de caso da aplicação *Conciliação Bancária*. A Seção 5.2 detalha a aplicação *Conciliação Bancária* e a Seção 5.3 descreve sua especificação arquitetural em DCL$^+$. As Seções 5.4 e 5.5 apresentam, respectivamente, os resultados obtidos na extração e verificação da comunicação e do projeto estrutural da aplicação *Conciliação Bancária*. Por fim, a Seção 5.6 destaca as considerações finais deste capítulo.

## 5.1 Metodologia

Para demonstrar a aplicabilidade e a utilidade da abordagem DCL$^+$ foi proposto um estudo de caso por meio da avaliação de um sistema de médio porte real composto por microsserviços, implementados em diferentes linguagens de programação utilizando diferentes *frameworks*. A avaliação realizada representa diretamente os ambientes reais nos quais a ferramenta poderá ser utilizada.

A avaliação foi conduzida pela autora desta dissertação de mestrado com o apoio de um aluno de iniciação científica e o arquiteto de software responsável pelo sistema, entre os dias 30 de julho à 3 de agosto de 2018. Durante esse período, foram realizadas as tarefas de compreensão do sistema, extração das dependências, avaliação das divergências e apresentação do resultado final aos envolvidos.

Os itens a seguir detalham as tarefas desempenhadas durante os cinco dias do processo de avaliação:

- Primeiro dia (30/07): O arquiteto apresentou aos avaliadores a aplicação de forma ampla – com ênfase nas funções que o mesmo realiza – oferecendo uma visão geral das comunicação entre os microsserviços, sem apresentar detalhes sobre como tais comunicações ocorrem. Com base nessas informações, os avaliadores realizaram inspeções do código-fonte de todos os serviços, a fim de melhor compreendê-lo, o que permitiu discussões mais detalhadas com o arquiteto.

- Segundo dia (31/07): O arquiteto apresentou cada microsserviço, seus contextos e possíveis dependências estruturais e de comunicação. Com base nessas informações, os avaliadores especificaram as restrições DCL$^+$ conforme detalhado pelo arquiteto.

- Terceiro dia (01/08): As restrições DCL$^+$ especificadas foram validadas pelo arquiteto, sendo realizados alguns ajustes nas dependências de comunicação e de projeto estrutural definidas. Assim, foi possível obter um conjunto de restrições DCL$^+$ condizentes com a arquitetura existente.

- Quarto dia (02/08): A ferramenta DCL$^+$*check* foi executada na aplicação *Conciliação Bancária*. Inicialmente, foram executados os módulos *Extrator de Comunicação e de Extrator de Projeto Estrutural*, sendo gerado como saída as dependências de comunicação entre microsserviços, bem como as dependências de projeto estrutural de cada um deles. As dependências extraídas foram analisadas pelos avaliadores, a fim de validar o correto funcionamento de tais módulos. Posteriormente, foram executados os módulos *Verificador de Comunicação e Verificador de Projeto Estrutural*, resultando nas divergências e ausências de comunicação e de projeto estrutural da aplicação.

- Quinto dia (03/08): No último dia, foram apresentados aos responsáveis as divergências de comunicação e de projeto estrutural reportados pela ferra-

menta DCL$^{+}$*check*. Após a apresentação, obtivemos o *feedback* dos envolvidos no desenvolvimento e manutenção da aplicação, visando justificar e fornecer possíveis soluções para os desvios arquiteturais encontrados. Nessa etapa, ficou claro para os avaliadores como a participação do arquiteto responsável foi crucial para a avaliação, uma vez que ele possui conhecimento extenso sobre a aplicação, o que é essencial tanto nas etapas iniciais do processo (de descrição do funcionamento do sistema), quanto na discussão dos resultados.

## 5.2 Aplicação

A aplicação Conciliação Bancária é composta por um sistema orquestrador nomeado `Node-Middleware` e por outros demais dez microsserviços (`Audit`, `Authentication`, `Authorization`, `Conciliation`, `Dashboard`, `Entries`, `FileLoad`, `FileProcess`, `Reports` e `Summary`).

O sistema `Node-Middleware` é implementado na linguagem JavaScript através do Node.js e é composto por 27 módulos. Os demais microsserviços são desenvolvidos em Java e utilizam o *framework* Spring Boot.

## 5.3 Especificação Arquitetural

A Listagem 5.1 ilustra um subconjunto da especificação arquitetural da aplicação Conciliação Bancária. Para melhor visualização, essa especificação mostra apenas as restrições que foram violadas. A especificação arquitetural completa de todos os microsserviços e suas respectivas violações podem ser observadas nos Apêndices B e C.

```
1  Node-Middleware: http://localhost:8099; /home/microservices/node-middleware/; JavaScript
2     #Restricoes de Projeto Estrutural RE's
3    AuditLog can-access-only AuditLog, Node_module, Config, TransfData "#NM-RE8"[2]
4    BankStatement can-access-only BankStatement, Node_module, Config,
          TransfData                                                    "#NM-RE10"[2]
5    ConciliationReport can-access-only ConciliationReport, Node_module, Config,
          TransfData                                                    "#NM-RE15"[4]
6    ConciliationSummary can-access-only ConciliationSummary, Node_module, Config,
          TransfData                                                    "#NM-RE16"[2]
7    Dashboard can-access-only Dashboard, Node_module, Config,
          TransfData                                                    "#NM-RE17"[4]
8    FinancialMovements can-access-only FinancialMovements, Node_module, Config,
          TransfData                                                    "#NM-RE21"[4]
9    ManualConcil can-access-only ManualConcil, Node_module, Config,
          TransfData                                                    "#NM-RE23"[8]
10   OccurrenceReport can-access-only OccurrenceReport, Node_module, Config,
          TransfData                                                    "#NM-RE25"[6]
11   Operator can-access-only Operator, Node_module, Config,
          TransfData                                                    "#NM-RE26"[2]
12 #Restricoes de Comunicacao NM-RC's
13   Assignment, BankStatement, OccurrenceReport, ConciliationReport,
          FinancialMovements can-communicate-only Reports               "#NM-RC3"[6]
14   AuditLog can-communicate-only Audit                                "#NM-RC5"[1]
15   ConciliationSummary can-communicate-only Summary                   "#NM-RC8"[3]
16   only ConciliationSummary can-communicate Summary                   "#NM-RC9"[2]
17   ConciliationSummary must-communicate Summary                       "#NM-RC10"[2]
18   ManualConciliation can-communicate-only Conciliation               "#NM-RC18"[2]
19   ProcessedFiles cannot-communicate Conciliation, Authentication, Authorization,
          Entries, Audit, Dashboard, Summary, Reports, FileProcess, FileLoad
                                                                        "#NM-RC22"[2]
20 Audit: http://localhost:8085; /home/microservices/audit/; Java
21   only Service can-depend Jndi                                       "#Audit-RE3"[3]
22   Main can-access-only Controller, MainAnnotations, Logger, Apache, $API
                                                                        "#Audit-RE5"[4]
23
24 Authentication: http://localhost:8082; /home/microservices/authentication/; Java
25   only Service can-depend MyBatis                                  "#Authentication-RE5"[2]
26   Service must-useannotation ServiceAnnotation                     "#Authentication-RE9"[2]
27
28 Authorization: http://localhost:8083; .../microservices/authorization/; Java
29   only Service can-depend Jndi                                     "#Authorization-RE3"[6]
30   Main can-depend-only $API, MainAnnotations, Apache               "#Authorization-RE5"[5]
31
32 Conciliation: http://localhost:8093; .../microservices/conciliation/; Java
33   Util can-depend-only Util, $API, Logger                          "#Conciliation-RE7"[24]
34
35 Dashboard: http://localhost:8095; .../microservices/dashboard/; Java
36
37 Entries: http://localhost:8092; /home/microservices/microservices/entries/; Java
38   only Main can-depend BCMultitenancy                                "#Entries-RE4"[3]
39   Util can-depend-only Util, $API                                    "#Entries-RE5"[2]
40   Controller cannot-access DAO                                       "#Entries-RE7"[6]
41   Controller must-useannotation CtlrAnnotations                      "#Entries-RE8"[2]
42   Service must-useannotation ServiceAnnotation                       "#Entries-RE9"[7]
43   Model, Serializer must-implement Serializable                      "#Entries-RE11"[11]
44
45 FileLoad: http://localhost:8075; /home/microservices/fileLoad/; Java
46    Main cannot-depend DAO, Model, BCDAO, JPA                       "#FileLoad-RE6"[4]
47    Model must-implement Serializable                               "#FileLoad-RE10"[6]
48
49 FileProcess: http://localhost:8076; /home/microservices/fileProcess/; Java
50    only DAO, Model can-depend JPA                                "#FileProcess-RE4"[8]
51    Util can-depend-only Util, BCUtil, API, Logger                "#FileProcess-RE6"[22]
52    Model must-implement Serializable                             "#FileProcess-RE11"[1]
53
54 Reports: http://localhost:8094; /home/microservices/reports/; Java
55    Controller must-useannotation CtlrAnnotations                    "#Reports-RE7"[1]
56    Controller must-extend BCController                              "#Reports-RE8"[1]
57    Serializer must-depend JSONSerializer                            "#Reports-RE10"[4]
58
59 Summary: http://localhost:8096; /home/microservices/summary/; Java
60    Service must-useannotation ServiceAnnotation                    "#Summary-RE7"[1]
61    Controller must-useannotation CtlrAnnotations                   "#Summary-RE8"[1]
```

Listagem 5.1 – Subconjunto da especificação arquitetural DCL$^+$ da aplicação.

**Microsserviços:** A Listagem 5.1 ilustra a especificação DCL$^{+}$ para todos os microsserviços. O primeiro microsserviço é identificado por Node-Middleware, tem sua localização definida pela URL http://localhost:8089, repositório de código-fonte /home/microservices/node−Middleware/, e é implementado na linguagem JavaScript (linha 1). O segundo é identificado por Audit, URL http://localhost:8085, repositório /home/microservices/audit e é implementado em Java (linha 20). Os outros microsserviços são especificados de maneira semelhante.

**Restrições de comunicação**: A arquitetura de comunicação da aplicação é baseada em orquestração. Nesse sentido, apenas o Node-Middleware comunica-se com os demais microsserviços. As restrições de comunicação violadas (NM-RC's) são ilustradas nas linhas 13 até 19. Por exemplo, **NM-RC3** restringe os módulos Assignment, BankStatement, OccurrenceReport, ConciliationReport e FinancialMovements a comunicarem-se apenas com o microsserviço Reports (linha 13); **NM-RC9** estabelece o módulo ConciliationSummary como o único autorizado a comunicar-se com o microsserviço Summary (linha 16); **NM-RC10** exige ConciliationSummary a comunicar-se com o microsserviço Summary (linha 17); e **NM-RC22** proíbe o módulo ProcessedFiles a comunicar-se com qualquer microsserviço da aplicaçao (linha 19).

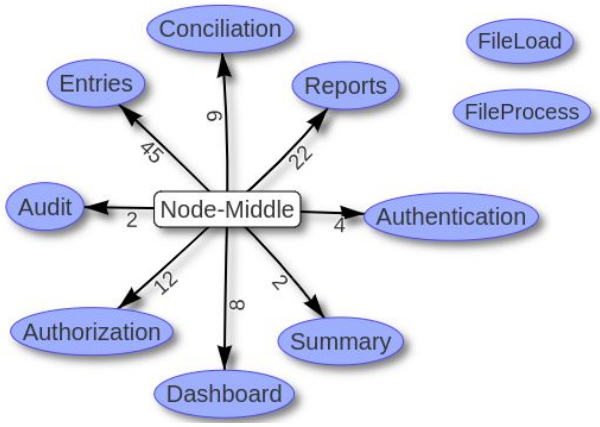
**Restrições do projeto estrutural**: A especificação arquitetural da Listagem 5.1 ilustra apenas as Restrições do Projeto Estrutural (RE's) com violações na linguagem DCL para todos os microsserviços. Por exemplo, para o microsserviço Node-Middleware, uma decisão do projeto estrutural através de **NM-RE8** estabelece que arquivos do módulo AuditLog dependa apenas de arquivos de seu próprio módulo e de arquivos dos módulos Node_module, Config e TransfData (linha 3); Para o microsserviço Audit, por exemplo, a restrição **Audit-RE5** res-

tringe classes do módulo `Main` a acessar apenas classes dos módulos `Controller`, `MainAnnotations`, `Logger`, `Apache` e classes da API Java (linha 22).

## 5.4 Conformidade de comunicação

**Extrator de Comunicação:** A Figura 5.1 apresenta as comunicações extraídas por DCL$^{+}$*check* a partir dos 27 módulos do sistema `Node-Middleware` com os demais microsserviços da aplicação. Em resumo, esses módulos realizam 101 comunicações com os microsserviços, variando entre duas comunicações com o microsserviço `Summary` e 45 comunicações com o microsserviço `Entries`.

Figura 5.1 – Comunicações extraídas da aplicação Conciliação Bancária.




É válido ressaltar que, apesar da ferramenta DCL$^{+}$*check* apresentar limitações no módulo *Extrator de Comunicação* conforme apresentado na Seção 4.4, este estudo de caso não é impactado pois a ferramenta possui suporte para as linguagens de programação e *frameworks* utilizados pela aplicação *Conciliação Bancária*.

**Verificador de comunicação:** Depois de extrair as comunicações, DCL$^{+}$*check* analisa cada comunicação definida na especificação arquitetural (linhas 13 até 19 da Listagem 5.1) e verifica se as comunicações estabelecidas são permitidas ou

não. A Figura 5.2 apresenta o grafo das comunicações com destaque das violações detectadas por DCL$^{+}$*check*. Os microsserviços são representados por círculos, os módulos do microsserviço `Node-Middleware` são representados por retângulo, uma aresta de um módulo `A` para um microsserviço `B` indica que `A` comunica-se com `B` e um rótulo na aresta representa o número de violações detectadas.

A Tabela 5.1 apresenta a relação dos identificadores e nomes dos 27 módulos definidos na especificação arquitetural de comunicação, para melhor reporte das violações detectadas.

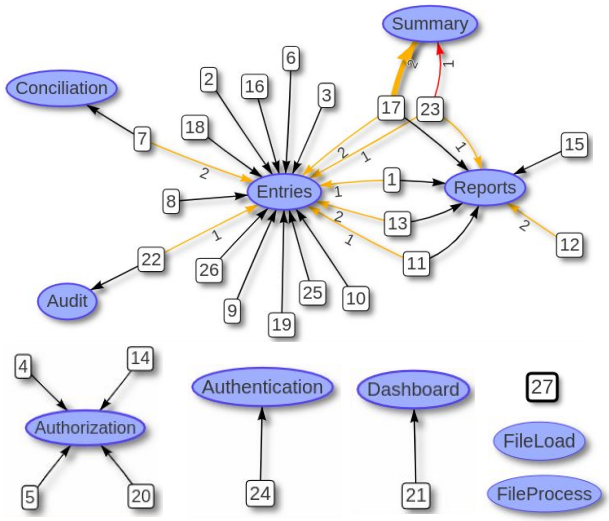
Tabela 5.1 – Relação dos identificadores e nomes dos módulos do sistema `Node-Middleware`.

| ID | Nome | ID | Name | ID | Nome |
|---|---|---|---|---|---|
| 1 | OccurrenceReport | 10 | FileUpload | 19 | BankAccounts |
| 2 | Categories | 11 | FinancialMovements | 20 | Client |
| 3 | EstablishmentGroups | 12 | ProcessedFiles | 21 | Dashboard |
| 4 | Operator | 13 | Assignment | 22 | AuditLog |
| 5 | AccessProfile | 14 | Home | 23 | ConciliationSummary |
| 6 | Task | 15 | BankStatement | 24 | User |
| 7 | ManualConciliation | 16 | Establishment | 25 | AttributesSet |
| 8 | ConciliationPlan | 17 | ConciliationReport | 26 | Cities |
| 9 | States | 18 | OccurrenceReason | 27 | Tregrid |

Em toda a aplicação, DCL$^{+}$*check* detectou 16 violações de comunicação, sendo 15 causadas por divergências e uma causada por ausência de comunicação, conforme a seguir:

- *Divergências*:
    1. Os módulos *OccurrenceReport* (1), *FinancialMovements* (11), *Assignment* (13), *ConciliationReport* (17) e *AuditLog* (22) estabeleceram comunicações proibidas com o microsserviço `Entries`, violando as restrições **NM-RC3** e **NM-RC5**;
    2. *ConciationSummary* comunica com o microsserviço `Reports`, violando a restrição **NM-RC8**;

Figura 5.2 – Violações de comunicação da aplicação Conciliação Bancária.




3. *ConciliationReport* (17) também comunica com o microsserviço `Summary`, violando a restrição **NM-RC9**;
4. *ManualConciliation* (7) realiza comunicações não previstas na especificação arquitetural com o serviço `Entries`, o que viola a restrição **NM-RC18**; e
5. *ProcessedFiles* (12) comunica com o microsserviço `Reports`, violando a restrição **NM-RC22**.

- *Ausência*:
  - O módulo *ConciliationSummary* (23) não estabelece comunicações com o microsserviço `Summary`, violando a restrição **NM-RC10**.

De acordo com os arquitetos, as violações reportadas por DCL$^{+}$*check* ocorreram devido ao desconhecimento por parte dos desenvolvedores e também devido a evolução da aplicação. As divergências 1 e 4 ocorreram devido ao desconhecimento dos desenvolvedores sobre as restrições de que cada módulo no sistema `Node-Middleware` poderia comunicar-se apenas com um microsserviço.

Em outras palavras, os módulos *OccurrenceReport, FinancialMovements, Assignment, ConciliationReport* e *ManualConciliation* estabelecem comunicações com os microsserviços `Reports` e `Conciliation`, mas não com o microsserviços `Entries`. Similarmente, a divergência 5 refere-se novamente ao desconhecimento dos desenvolvedores, uma vez que o módulo *ProcessedFile* – que deveria manter um histórico dos arquivos processados e não pode comunicar com nenhum microsserviço – comunica-se com o microsserviço `Reports`.

As divergências 2 e 3 e a ausência detectada ocorreram devido a evolução da aplicação do microsserviço `Reports` para o microsserviço `Summary`. Consequentemente, o módulo *ConciliationReport* vai evoluir para o módulo *ConcilationSummary*. Essa evolução justifica as violações detectadas, uma vez que a completa evolução ocorrerá nas próximas versões.

É válido ressaltar que o módulo *ConciliationReport* possui o maior número de violações detectadas, sendo responsável por 26,6% das divergências (quatro de quinze). Em seguida, os módulos *ConciliationSummary*, *ProcessedFiles*, *Assignment* e *ManualConciliation* representam cada 13,3% das divergências. Os módulos *AuditLog*, *FinancialMovements* e *OccurrenceReport* representam cada 6,6% das divergências (apenas uma cada). No que compete a ausência de comunicação, o módulo *ConciliationSummary* é totalmente responsável por esse tipo de violação. No total, dentre os 27 módulos especificados, em apenas oito ocorreram violações, correspondendo a 29,6% dos módulos violados. Esse resultado indica que, apesar que não haver uma especificação prévia da aplicação, os membros da equipe se mostraram empenhados em conduzir o desenvolvimento seguindo os padrões definidos pelo arquiteto.

## 5.5 Conformidade do projeto estrutural

**Extrator do projeto estrutural:** `DCL`$^{+}$*check* executou as ferramentas extratoras do projeto estrutural `JsDepExtractor` e `JavaDepExtractor` para extrair as de-

pendências do sistema `Node-Middleware` e dos dez microsserviços. A Tabela 5.2 apresenta a quantidade de dependências extraídas por DCL$^{+}$*check* em todos os microsserviços pertencentes a aplicação Conciliação Bancária.

Embora o módulo *Extrator do projeto estrutural* não reporte dependências causadas por falsos positivos, falsos negativos podem ocorrer devido às atuais limitações da ferramenta DCL$^{+}$*check*, conforme mencionadas na Seção 4.4.

**Verificador do projeto estrutural:** DCL$^{+}$*check* aplicou `piDCLcheck` para verificar se as dependências internas de cada microsserviços estão em conformidade com a arquitetura planejada (em restrições `DCL`) ou não. As violações do projeto estrutural detectadas podem ser vistas na Listagem 5.1 por [`n`], onde `n` compreende ao número de violações. Como pode ser observado, DCL$^{+}$*check* detectou 162 violações do projeto estrutural nos microsserviços da aplicação, sendo 125 divergências e 37 ausências. A matriz de violações detectadas de cada microsserviço é apresentada no Apêndice C. Em geral, as violações detectadas foram causadas devido ao desconhecimento por parte dos desenvolvedores no que diz respeito aos conceitos de arquitetura de software, especificamente na correta aplicação do framework *Spring MVC* e dependências do módulo *Util*.

Tabela 5.2 – Dependências do projeto estrutural extraídas por DCL$^{+}$*check* da aplicação Conciliação Bancária.

| **Microsserviço** | **Dependências Extraídas** |
|---|---|
| Node–Middleware | 2.141 |
| Audit | 99 |
| Authentication | 206 |
| Authorization | 257 |
| Conciliation | 329 |
| Dashboard | 297 |
| Entries | 498 |
| FileLoad | 347 |
| FileProcess | 353 |
| Reports | 324 |
| Summary | 439 |
| **Total** | **5.290** |

No que compete as dependências desconhecidas para o framework *Spring MVC*, DCL$^+$*check* detectou 45 divergências e cinco ausências, representando, respectivamente 36% e 10,8% das violações detectadas. Para as violações detectadas através de implementações de dependências para o módulo *Util*, DCL$^+$*check* reportou 48 divergências, correspondendo a 38,4% das violações. A Listagem 5.1 mostra que os microsserviços `Entries` e `FileProcess` são responsáveis pelo maior número de violações (31), respectivamente, representando 38,2% do número total de violações. Segundo o arquiteto, isso possivelmente se deve ao fato de esse serviço ser o maior – em linhas de código – dentre os demais. Em contrapartida, para o microsserviço `Dashboard` não houve nenhuma violação de projeto estrutural reportada.

## 5.6 Considerações Finais

Este capítulo apresentou uma avaliação de um estudo de caso real a fim de demonstrar a aplicabilidade da abordagem DCL$^+$, executada pela ferramenta DCL$^+$*check*. Esta avaliação foi realizada sobre uma aplicação financeira de médio porte, focada na gestão de vendas e conciliação com cartões de crédito, débito, *tickets* e outros meios de pagamentos online. A aplicação avaliada foi nomeada Conciliação Bancária e é composta por um sistema orquestrador nomeado `Node-Middleware` – implementado na linguagem JavaScript – e por dez microsserviços – desenvolvidos em linguagem Java.

Na aplicação Conciliação Bancária, foram especificadas 22 restrições de comunicação no sistema `Node-Middleware` envolvendo os dez microsserviços e um total de 148 restrições de projeto estrutural. DCL$^+$*check*, por sua vez, extraiu um conjunto de 101 comunicações e 5.290 dependências do projeto estrutural. Como resultado, DCL$^+$*check* reportou 16 violações de comunicação e 162 violações do projeto estrutural. No que compete às violações de comunicação, segundo os arquitetos, elas ocorreram devido ao desconhecimento por parte dos desenvol-

vedores sobre as restrições de que cada módulo do sistema `Node-Middleware` poderia comunicar-se apenas com um microsserviço e também devido a evolução da aplicação. Já as violações do projeto estrutural ocorreram pelo desconhecimento por parte dos desenvolvedores do projeto arquitetural, especificamente na correta aplicação do framework *Spring MVC* e dependências do módulo *Util*.

# 6 TRABALHOS RELACIONADOS

A arquitetura de microsserviços é uma área de pesquisa nova e pouco explorada, sendo a maioria dos aplicativos proprietários e, portanto, não são facilmente acessíveis para a comunidade acadêmica (ADERALDO et al., 2017; ZHOU et al., 2018). Embora exista uma quantidade reduzida de trabalhos que buscam soluções para verificação de conformidade na arquitetura de microsserviços, esta dissertação de mestrado discute estudos empíricos a respeito dos conceitos relacionados a microsserviços, uso de *benchmarks* para avaliação de aplicações, abordagens para descoberta arquitetural e propostas de verificação de conformidade para decomposição de microsserviços.

**Estudos empíricos em microsserviços**: Trabalhos de mapeamento sistemático apresentam quais os interesses de pesquisa relacionados a arquitetura de microsserviços, seus principais problemas e aplicabilidades (ALSHUQAYRAN; ALI; EVANS, 2016; FRANCESCO; MALAVOLTA; LAGO, 2017). Alshuqayran et al. confirmam que os principais interesses de pesquisa em microsserviços são guiados por estudos referentes ao estabelecimento da comunicação, integração e composição de microsserviços, implantação, descoberta arquitetural, desempenho, segurança, APIs e tecnologias de contêineres (ALSHUQAYRAN; ALI; EVANS, 2016). Francesco et al. destacam que os problemas mais recorrentes nos estudos primários referem-se a complexidade, flexibilidade, gerenciamento e composição nesse estilo arquitetural (FRANCESCO; MALAVOLTA; LAGO, 2017). Isso ocorre pois, por um lado, os microsserviços ajudam a alcançar um bom nível de flexibilidade (com baixo acoplamento de serviços e maior manutenibilidade), mas, por outro lado, adotar esse estilo arquitetural pode trazer maior complexidade, principalmente porque implica em um elevado número de serviços distribuídos para operar. A relação dos trabalhos categorizados por esses artigos de mapeamento sistemático, com esta dissertação de mestrado, é motivado

pelo fato de suas soluções serem potenciais alvos para aplicação da abordagem proposta, com extensão para trabalhos futuros.

**Comunicação, integração e composição de microsserviços:** 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 de serviços de granularidade fina (microsserviços). Os autores consideram agentes como entidades prestadores e fornecedoras de serviços, comparados a objetos em uma linguagem orientada a objetos. O agrupamento de agentes nessa linguagem é denominado *caste*, semelhantes às classes, fornecendo também mecanismos de herança entre tais *caste*, a fim de permitir o polimorfismo. Para automação da linguagem proposta, Liu et al. apresentam o ambiente de desenvolvimento denominado CIDE (*CAOPLE Integrated Development Environment*) (LIU et al., 2016). A proposta de uma linguagem de programação específica para o projeto de microsserviços e de um ambiente para sua execução se mostra relevante para a comunidade. Caso a linguagem proposta se mostre realmente efetiva para tal contexto, um potencial trabalho futuro desta dissertação de mestrado seria o projeto de um extrator para essa linguagem.

**Descoberta arquitetural:** Trabalhos apresentados por Granchelli et al. (GRANCHELLI et al., 2017b), Mayer e Weinreich (MAYER; WEINREICH, 2018) e Engel et al. (ENGEL et al., 2018), propõem abordagens para a descoberta arquitetural de aplicações implementadas na arquitetura de microsserviços.

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

extrai automaticamente a arquitetura de implantação da aplicação em um repositório GitHub por meio da ferramenta MicroART (GRANCHELLI et al., 2017a), gerando um modelo da arquitetura física. Posteriormente, esse modelo é analisado pelos arquitetos a fim de realizar modificações que julguem necessário. Ao final, a abordagem gera o modelo da arquitetura lógica com as informações fornecidas. Apesar de os autores extraírem a arquitetura física da aplicação, eles não proveem formas de definir a arquitetura planejada da aplicação tampouco verificar se as comunicações estabelecidas são realizadas corretamente. Nesse sentido, a abordagem proposta nesta dissertação de mestrado pode ser aplicada para verificar a conformidade de tais comunicações, a partir de uma arquitetura desejada pelo arquiteto.

Mayer e Weinreich propõem uma abordagem para extração da arquitetura de sistemas baseados em microsserviços (MAYER; WEINREICH, 2018) que permite extrair de maneira contínua a arquitetura de sistemas de software de microsserviços baseados em REST, através da combinação de informações estáticas, dinâmicas e de infraestrutura. Os autores definiram um modelo de dados guiado pelos princípios do projeto orientado a domínio (do inglês *Domain-Driven Design* (DDD)) (EVANS, 2004), a fim de representar as informações arquiteturais descritas por meio de serviços, suas respectivas infraestruturas e interações. Para realizar a extração das informações de cada serviço e suas interações, o trabalho aplica técnicas de análise estática e dinâmica de código apenas para os microsserviços implementados em Java para o contexto do *Spring Framework*. A avaliação da abordagem proposta considera um sistema composto por três microsserviços. A proposta dos autores possui finalidades parecidas ao módulo Extrator de Comunicações apresentado nesta dissertação de mestrado, uma vez que ambos fornecem informações sobre as comunicações realizadas em aplicações desenvolvidas em microsserviços REST. Apesar de a abordagem proposta realizar extrações estáticas e dinâmicas para mapear todas as comunicações, elas são realizadas apenas

para a linguagem Java e validada em um ambiente restrito (sistema com apenas três microsserviços). Esta dissertação de mestrado, por sua vez, realiza a extração de comunicações de maneira estática para microsserviços implementados em Java e JavaScript, além de fornecer aos arquitetos de software uma solução que permite verificar a conformidade das comunicações extraídas de acordo com a arquitetura planejada.

Engel et al., por sua vez, propõem uma abordagem para avaliar a arquitetura de sistemas implementados com microsserviços (ENGEL et al., 2018). Para isso, foi conduzido um estudo exploratório em cinco projetos, compostos por no mínimo dois e no máximo 50 microsserviços. Foram realizadas entrevistas com os *stakeholders* responsáveis a fim descobrir as principais dificuldades ao desenvolver tais microsserviços. Como resultado do estudo, dificuldades em relação a definição do contexto de um microsserviço, dependências cíclicas, monitoramento e testes foram reportadas. Assim, os autores propuseram princípios para o projeto de microsserviços por meio de métricas utilizando a abordagem GQM (*Goal Question Metric*). Para aplicar as métricas propostas, foi apresentada uma ferramenta que realiza a extração dinâmica das comunicações realizadas entre microsserviços, denominada MAAT (*Microservice Architecture Analysis Tool*). A aplicabilidade da abordagem proposta foi avaliada em um dos projetos utilizados no estudo inicial. Como resultado, constatou-se a necessidade de aprofundar em técnicas para extração das comunicações, uma vez que a ferramenta não foi capaz de reportar, no momento da avaliação, um microsserviço presente na arquitetura. Portanto, os autores poderiam se beneficiar dos extratores de comunicação propostos nesta dissertação de mestrado para os microsserviços implementados nas linguagens Java e JavaScript. Além disso, a dificuldade de reportar microsserviços motivou um dos nossos trabalhos futuros de combinar análise estática e dinâmica de código.

**Benchmarks:** No ambiente acadêmico, as pesquisas em arquitetura de microsserviços ainda são limitadas, em parte devido à falta de aplicações de referência

que reflitam as características das aplicações de microsserviços desenvolvidos na Indústria (ZHOU et al., 2018). Com o objetivo de solucionar tal lacuna, trabalhos apresentam (ADERALDO et al., 2017) e propõem (ZHOU et al., 2018) arquiteturas de referências para o contexto da arquitetura de microsserviços.

Aderaldo et al. propõem e discutem um conjunto de doze requisitos relevantes para a comunidade acadêmica em Engenharia de Software, a fim de auxiliar na seleção de *benchmarks* para avaliação de suas pesquisas sobre abordagens propostas em microsserviços (ADERALDO et al., 2017). Segundo os autores, os requisitos propostos refletem a forma como as aplicações típicas de microsserviços estão sendo desenvolvidas e entregues em produção, conforme relatam profissionais da Indústria e especialistas em microsserviços. Esses requisitos foram agrupados em três contextos diferentes: referentes à arquitetura (dois requisitos), técnicas *DevOps* (seis requisitos) e contexto geral (três requisitos). Requisitos referentes à arquitetura descrevem *benchmarks* relacionados arquiteturas baseada em padrões, por exemplo. Requisitos referentes à práticas *DevOps* abrangem conceitos referentes a integração contínua e implantação automática, por exemplo. Já os requisitos classificados em contexto geral referem-se aos *benchmarks* que fornecem implementações alternativas em termos de linguagens de programação e/ou decisões arquiteturais, por exemplo. Para avaliação dos requisitos propostos, os autores selecionaram quatro *benchmarks* candidatos, denominados *Acme Air*[1]*, Spring Cloud Demo Apps*[2]*, Socks Shop*[3] e *Music Store*[4]. Esses *benchmarks* possuem implementação de código-fonte aberto desenvolvidos na arquitetura de microsserviços e são geralmente utilizados pela comunidade acadêmica. Após avaliação de tais *benchmarks* sobre os requisitos propostos, os autores concluem que o *benchmark Socks*

[1] *Acme Air*, disponível em: https://github.com/acmeair/acmeair

[2] *Spring Cloud Demo Apps*, disponível em: https://github.com/kbastani/spring-cloud-microservice-example

[3] *Socks Shop*, disponível em: https://github.com/microservices-demo/microservices-demo

[4] *Music Store*, disponível em: https://github.com/aspnet/MusicStore

*Shop* melhor atende aos requisitos apresentados, satisfazendo amplamente dez dos doze requisitos.

Zhou et al. propõem uma arquitetura de referência para auxiliar a comunidade acadêmica nas diversas lacunas presentes na arquitetura de microsserviços (ZHOU et al., 2018). Para isso, os autores inicialmente realizaram uma revisão de literatura em sistemas de código aberto para identificar a lacuna entre os sistemas de referência existentes e os sistemas de microsserviços presentes na Indústria. Nesse estudo, foram categorizados sete *benchmarks* comumente utilizados pela comunidade, onde quatro compreende àqueles apresentados por Aderaldo et al. (ADERALDO et al., 2017), além dos *benchmarks Bifrost*[5], Staffjoy[6] e NServiceBus[7]. Com base no estudo, os autores identificaram problemas nos *benchmarks* analisados em relação (i) ao mau uso de diferentes modos de interação como mecanismos de comunicação síncrona e assíncrona, (ii) ao pequeno número de microsserviços presente nas aplicações variando entre cinco e nove microsserviços, (iii) a não adequação dos *benchmarks* aos princípios desse estilo arquitetural, não sendo estruturados em torno de suas respectivas capacidade de negócio, e (iv) a insuficiência na geração de casos de teste de unidade ou casos de teste de integração. Com o intuito de solucionar tais problemas, foi proposto um *benchmark* para o contexto de vendas de passagens ferroviárias, denominado *TrainTicket*. Essa arquitetura de referência é composta por 24 microsserviços implementados nas linguagens Java com a utilização do Spring Boot e para a linguagem JavaScript para o contexto do *framework* Node.js.

Para o contexto deste projeto de dissertação, os trabalhos propostos por Aderaldo et al. e Zhou et al. são de suma importância, uma vez que o uso de tais *benchmarks* podem ser considerados para a validação da abordagem DCL$^{+}$ na

[5] *Bifrost*, disponível em: https://github.com/sealuzh/bifrost-microservices-sample-application

[6] *Staffjoy*, disponível em: https://github.com/Staffjoy/v2

[7] *NServiceBus*, disponível em: https://github.com/dotnet-architecture/eShopOnContainers

análise de suas respectivas conformidades. A atual implementação de DCL$^{+}$*check* permite a verificação da conformidade de comunicação e do projeto estrutural dos microsserviços presentes nos *benchmarks Acme Air*, *Spring Cloud Demo Apps* e *TrainTicket*, uma vez que eles são implementados nas linguagens Java (através do framework Spring Boot) e JavaScript (por meio do Node.js). Já para o contexto dos *benchmarks Socks Shop* e *Music Store*, a atual implementação de DCL$^{+}$*check* ainda não contempla extratores de comunicação para as linguagens Go e C# (para o *framework* ASP.NET) e do projeto estrutural também para a linguagem Go, o que foi considerado como trabalho futuro.

**Conformidade arquitetural para microsserviços**: Foi encontrado até o presente momento, apenas um trabalho que fornece uma verificação de conformidade em arquiteturas de microsserviços. Zdun, Navarro e Leymann sugerem um conjunto de restrições centradas na decomposição de arquiteturas baseadas em microsserviços para avaliar e medir a conformidade de tais padrões (ZDUN; NAVARRO; LEYMANN, 2017). Para atingir tal objetivo, três questões de pesquisa foram propostas: (Q1) quais medidas podem ser definidas para verificar ou avaliar automaticamente o quanto conforme a decomposição de uma arquitetura de microsserviços está em relação a um padrão?, (Q2) o quanto bem tais medidas são funcionais sobre o julgamento de especialistas? e (Q3) as restrições propostas expressam um conjunto mínimo necessário em uma arquitetura de decomposição para calcular medidas significativas?

Para verificação da conformidade, foi utilizada uma abordagem que aplica uma linguagem de domínio específico por meio de técnicas baseadas no desenvolvimento orientado a modelo (MDD), para especificar abstrações arquiteturais que podem ou não serem estabelecidas, propostas por Haitzer e Zdun (HAITZER; ZDUN, 2014). As restrições e métricas propostas estão centradas nas características de microsserviços como independência de implantação e dependências compartilhadas entre tais microsserviços. Para a característica de independência de

implantação, foi especificada uma restrição que refere-se a todos os componentes independentemente implantáveis e derivada duas métricas para tal restrição. Para a característica referente a dependências compartilhadas, foi especificada uma restrição referente a ausência de dependências de componentes não externos, derivando também duas métricas para essa restrição.

Para responder às questões de pesquisa propostas, os autores modelaram 13 modelos arquiteturais presentes na literatura e compararam sua abordagem em relação a uma análise manual feita por arquitetos experientes. Como resultado, para responder à primeira questão de pesquisa, os autores concluem que é possível gerar automaticamente um conjunto de métricas para avaliar a conformidade de padrões em relação a decomposição arquitetural em microsserviços. Para a segunda questão de pesquisa, os autores concluem que as métricas propostas estão próximas das avaliações realizadas por especialistas. Por fim, para responder a terceira questão, os autores consideram necessário um refinamento das restrições e métricas propostas.

Embora apresente a aplicação de técnicas de conformidade arquitetural para a arquitetura de microsserviços, o escopo de verificação desse artigo difere do escopo desta dissertação de mestrado, cujo o objetivo é verificar a conformidade de comunicação entre os microsserviços e não a conformidade de decomposição arquitetural dos microsserviços. Até o presente momento, não se tem conhecimento de pesquisas que objetivam restringir a conformidade da comunicação entre os microsserviços bem como verificar cada um de seus projetos arquiteturais. Dessa maneira, a presente dissertação de mestrado torna-se pioneira nessa linha de pesquisa.

# 7 CONCLUSÃO

Este capıtulo apresenta as considerações finais dessa dissertação de mestrado. A Seção 7.1 ressalta os pontos principais da solução proposta, como seus objetivos, o propósito da abordagem DCL$^+$ etc. A Seção 7.2 destaca as principais contribuições. Por fim, a Seção 7.3 apresenta possíveis trabalhos futuros.

## 7.1 Visão Geral da Solução Proposta

A solução de verificação da conformidade arquitetural em uma arquitetura de microsserviços, proposta nesta dissertação de mestrado, se baseia na ideia de que as dependências de comunicação entre microsserviços e dependências intramodulares de cada microsserviço constituem fontes importantes de violações arquiteturais. Isso deve-se à heterogeneidade desse estilo arquitetural, uma vez que os serviços podem estar hospedados em diferentes servidores, implementados em diferentes linguagens de programação e utilizar diferentes *frameworks*.

Diante desse cenário e da questão de pesquisa desta dissertação de mestrado, para restringir as comunicações entre microsserviços foi especificada a linguagem de restrição arquitetural DCL$^+$ a fim de restringir o espectro de comunicações desejáveis entre microsserviços de uma aplicação. Posteriormente, foi proposta uma abordagem de conformidade arquitetural para a arquitetura de microsserviços. Essa abordagem consiste em uma solução multiplataforma que permite restringir as comunicações entre microsserviços bem como verificar o projeto arquitetural de cada um deles. Para isso, a abordagem requer uma aplicação composta por uma suíte de microsserviços juntamente com sua especificação arquitetural definida na linguagem DCL$^+$. Essas informações são extraídas por meio de extratores de comunicação e do projeto estrutural, sendo posteriormente conduzidas verificações de conformidade. Como saída, a abordagem reporta o conjunto de violações de comunicação e do projeto estrutural.

A abordagem proposta também inclui DCL$^{+}$*check*, uma ferramenta que extrai e verifica se as dependências de comunicação e do projeto estrutural estão em conformidade com dependências definidas na especificação arquitetural da aplicação. A ferramenta DCL$^{+}$*check* utiliza técnicas de análise estática de código para extrair as dependências de comunicação – para microsserviços implementados nas linguagens Java e JavaScript – e dependências do projeto estrutural – para microsserviços implementados nas linguagens Java, JavaScript e C#. Ademais, DCL$^{+}$*check* utiliza verificadores de conformidade independentes de plataforma.

A solução proposta nesta dissertação de mestrado responde à questão de pesquisa de como restringir comunicações em uma arquitetura de microsserviços. Além do mais, conduziu-se, por critérios de validação da abordagem proposta, uma demonstração de aplicabilidade de DCL$^{+}$ sob uma aplicação composta com cinco microsserviços. Essa aplicação foi desenvolvida para o contexto de vendas de produtos, sendo composta por cinco microsserviços – três desenvolvidos na linguagem Java e dois na linguagem C#.

Outra, e mais importante validação da abordagem DCL$^{+}$, foi conduzida em um estudo de caso real sobre uma aplicação desenvolvida por uma empresa de médio porte. A aplicação avaliada é composta por 11 microsserviços compostos por meio de orquestração, implementados em duas linguagens distintas (Java e JavaScript). DCL$^{+}$*check* pôde extrair um conjunto de 101 comunicações e reportou 16 violações de comunicação. Em geral, tais violações ocorreram devido ao desconhecimento por parte dos desenvolvedores em relação as restrições de comunicação entre os módulos do sistema orquestrador e os demais microsserviços, bem como a evolução de dois microsserviços na versão analisada da aplicação. Essas violações ocorreram em apenas oito dos 27 módulos especificados.

No que diz respeito às especificações do projeto arquitetural de cada microsserviço, DCL$^{+}$*check* detectou 162 violações do projeto estrutural, sendo 125 divergências e 37 ausências. No geral, essas violações também foram causadas

devido ao desconhecimento por parte dos desenvolvedores em relação aos conceitos sobre arquitetura de software, especificamente na correta aplicação do padrão `Spring MVC` e dependências inesperadas no módulo `Util`. Esse resultado indica que, embora não houvesse especificação prévia da aplicação, os membros da equipe se mostraram comprometidos em conduzir o desenvolvimento seguindo os padrões definidos pelo arquiteto.

## 7.2 Contribuições

Esta dissertação contém as seguintes contribuições:

1. Uma linguagem de restrição arquitetural denominada `DCL`$^+$ proposta para representar dependências de comunicação entre microsserviços de uma arquitetura por meio da dependência `communicate`. Além disso, DCL$^+$ define também as restrições *only-can*, *can-only* e *cannot* para detectar divergências e restrições *must* para detectar ausências de comunicação.

2. Uma solução de conformidade arquitetural denominada `DCL`$^+$ que provê a arquitetos de software uma solução multiplataforma que permite restringir a comunicação entre microsserviços e verificar os projetos arquiteturais de cada um deles.

3. Automatização da solução proposta por meio da ferramenta `DCL`$^+$*check*. Para isso, foram implementados extratores de comunicação para as linguagens Java e JavaScript e extratores do projeto estrutural para as linguagens C#, Java e JavaScript. Além disso, foi realizada a implementação de verificadores de comunicação e do projeto estrutural independentes de plataforma.

4. Uma demonstração da aplicabilidade da solução proposta em uma aplicação com cinco microsserviços – desenvolvidos nas linguagens Java e C# – e organizados por meio de coreografia. Nessa demonstração, foram

simuladas violações de comunicação e do projeto estrutural a quais foram corretamente detectadas pela ferramenta DCL$^{+}$*check*.

5. Uma avaliação da solução proposta em um estudo de caso real sobre uma aplicação financeira de médio porte, focada na gestão de vendas e conciliação bancária. A aplicação avaliada é composta por onze microsserviços – desenvolvidos nas linguagens Java e JavaScript – e organizados por meio de orquestração. Nessa avaliação, foram extraídas 101 comunicações, sendo reportadas 16 violações de comunicação e 162 violações do projeto estrutural.

## 7.3 Trabalhos Futuros

Como trabalho futuro, pretende-se prosseguir com pesquisas na área de extração de comunicações realizadas e dependências do projeto estrutural em aplicações desenvolvidas em linguagens diferentes de Java e JavaScript. Assim, a seguir são descritas algumas possíveis linhas de pesquisa:

1. Implementar extratores de comunicação e do projeto estrutural para linguagens como C#, PHP, Python, Go e .NET, de forma que possa ser possível aplicar a solução proposta em uma maior gama de linguagens de programação.

2. Propor metodologias para a extração de comunicações realizadas dinamicamente, de forma que possa ser possível reduzir as limitações atuais na captura estática de comunicações realizadas.

3. Avaliar a abordagem DCL$^{+}$ em outras aplicações reais, a fim de validar sua aplicabilidade em aplicações onde os microsserviços sejam compostos por meio de coregrafia e desenvolvidos em uma maior heterogeneidade tecnológica de linguagens e *frameworks*.

4. Aprimorar a análise estática através da utilização de recursos capazes de detectar em baixo nível, requisições e respostas entre serviços *web* (*request* e *response*), tornando o mapeamento das comunicações independente de *frameworks* e bibliotecas utilizadas na aplicação.

# REFERÊNCIAS

ADERALDO, C. M. et al. Benchmark requirements for microservices architecture research. In: **1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)**. [S.l.: s.n.], 2017. p. 8–13.

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: **9th International Conference on Service-Oriented Computing and Applications (SOCA)**. [S.l.: s.n.], 2016. p. 44–51.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables DevOps: migration to a cloud-native architecture. **IEEE Software**, v. 33, n. 3, p. 42–52, 2016.

BALDWIN, C. Y.; CLARK, K. B. **Design rules: The power of modularity**. 4. ed. [S.l.]: MIT Press, 2000.

BARKER, A.; WALTON, C. D.; ROBERTSON, D. Choreographing web services. **Transactions on Services Computing**, v. 2, n. 2, p. 152–166, 2009.

BITTENCOURT, R. A. Conformance checking during software evolution. In: **17th Working Conference on Reverse Engineering (WCRE)**. [S.l.: s.n.], 2010. p. 289–292.

BUTZIN, B.; GOLATOWSKI, F.; TIMMERMANN, D. Microservices approach for the internet of things. In: **21st International Conference on Emerging Technologies and Factory Automation (ETFA)**. [S.l.: s.n.], 2016. p. 1–6.

CAMARGO, A. et al. An architecture to automate performance tests on microservices. In: **18th International Conference on Information Integration and Web-based Applications and Services (iiWAS)**. [S.l.: s.n.], 2016. p. 422–429.

ENGEL, T. et al. Evaluation of microservice architectures: A metric and tool-based approach. In: **30th International Conference on Advanced Information Systems Engineering (CAiSE)**. [S.l.: s.n.], 2018. p. 74–89.

ERNST, M. D. Static and dynamic analysis: Synergy and duality. In: **3rd Workshop on Dynamic Analysis (WODA)**. [S.l.: s.n.], 2003. p. 24–27.

EVANS, E. **Domain-driven design: tackling complexity in the heart of software**. 1. ed. [S.l.]: Addison-Wesley Professional, 2004.

FOWLER, M. **Patterns of enterprise application architecture**. 1. ed. [S.l.]: Addison-Wesley Longman Publishing., 2002.

FOWLER, M. **Microservices Resource Guide**. 2014. Acesso em: 28 maio 2017. Disponível em: <https://martinfowler.com/microservices/.>

FOWLER, M.; LEWIS, J. **Microservices: a definition of this new architectural term**. 2014. Acesso em: 28 maio 2017. Disponível em: <https://martinfowler.com/articles/microservices.html.> Acesso em: 28/05/2017.

FRANCESCO, P. D. Architecting microservices. In: **1st International Conference on Software Architecture Workshops (ICSAW)**. [S.l.: s.n.], 2017. p. 224–229.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: trends, focus, and potential for industrial adoption. In: **4rd International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2017. p. 21–30.

GRANCHELLI, G. et al. MicroART: A software architecture recovery tool for maintaining microservice-based systems. In: **1st International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2017. p. 298–302.

GRANCHELLI, G. et al. Towards recovering the software architecture of microservice-based systems. In: **1st International Conference on Software Architecture Workshops (ICSAW)**. [S.l.: s.n.], 2017. p. 46–53.

HAITZER, T.; ZDUN, U. Semi-automated architectural abstraction specifications for supporting software evolution. **Science of Computer Programming**, v. 90, n. 24, p. 135–160, 2014.

JERDING, D.; RUGABER, S. Using visualization for architectural localization and extraction. In: **4th Working Conference on Reverse Engineering (WCRE)**. [S.l.: s.n.], 1997. p. 56–65.

KAZMAN, R.; CARRIÈRE, S. J. Playing detective: Reconstructing software architecture from available evidence. **Automated Software Engineering**, v. 6, n. 2, p. 107–138, 1999.

KNODEL, J.; POPESCU, D. A comparison of static architecture compliance checking approaches. In: **7th International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2007. p. 12–21.

LATTIX, I. **Lattix.** 2017. Acesso em: 06 set. 2017. Disponível em: <http://lattix.com/solutions/enterprise.>

LEITE, L. A. et al. A systematic literature review of service choreography adaptation. **Service Oriented Computing and Applications**, v. 7, n. 3, p. 199–216, 2013.

LIU, D. et al. CIDE: An integrated development environment for microservices. In: **13th International Conference on Services Computing (SCC)**. [S.l.: s.n.], 2016. p. 808–812.

MACLENNAN, E.; BELLE, J.-P. V. Factors affecting the organizational adoption of service-oriented architecture (SOA). **Information Systems and e-Business Management**, v. 12, n. 1, p. 71–100, 2014.

MAFFORT, C. et al. Heuristics for discovering architectural violations. In: **20th Working Conference on Reverse Engineering (WCRE)**. [S.l.: s.n.], 2013. p. 222–231.

MAYER, B.; WEINREICH, R. An approach to extract the architecture of microservice-based software systems. In: **12th Symposium on Service-Oriented System Engineering (SOSE)**. [S.l.: s.n.], 2018. p. 21–30.

MÜLLER, H. A.; KLASHINSKY, K. Rigi-a system for programming-in-the-large. In: **10th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 1988. p. 80–86.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: Bridging the gap between source and high-level models. In: **3rd International Symposium on Foundations of Software Engineering (FSE)**. [S.l.: s.n.], 1995. p. 18–28.

NANDA, M. G.; CHANDRA, S.; SARKAR, V. Decentralizing execution of composite web services. In: **19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)**. [S.l.: s.n.], 2004. p. 170–187.

NETFLIX, O. **Zuul**. 2018. Acesso em: 25 mar. 2018. Disponível em: <https://github.com/Netflix/zuul/wiki.>

NETFLIX, S. C. **Spring Cloud Netflix**. 2018. Acesso em: 19 jan. 2018, Disponível em: <https://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html.>

NEWMAN, S. **Building microservices: designing fine-grained systems**. 1. ed. [S.l.]: O Reilly Media, 2015.

PAPAZOGLOU, M. P. et al. Service-oriented computing: State of the art and research challenges. **Computer**, v. 40, n. 11, p. 38–45, 2007.

PAPAZOGLOU, M. P. et al. Service-oriented computing: a research roadmap. **International Journal of Cooperative Information Systems**, v. 17, n. 2, p. 223–255, 2008.

PASSOS, L. et al. Static architecture-conformance checking: An illustrative overview. **IEEE Software**, v. 27, n. 5, p. 82–89, 2010.

PEDRAZA, G.; ESTUBLIER, J. Distributed orchestration versus choreography: The focas approach. In: **3rd International Conference on Software Process (ICSP)**. [S.l.: s.n.], 2009. p. 75–86.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Software Engineering**, v. 17, n. 4, p. 40–52, 1992.

PIVOTAL. **Spring Cloud Netflix**. 2018. Acesso em: 03 mar. 2018, Disponível em: <https://cloud.spring.io/spring-cloud-netflix.>

PRESSMAN, R.; MAXIM, B. **Engenharia de Software**. 8. ed. [S.l.]: McGraw Hill Brasil, 2011.

RICHARDSON, C. **Pattern: API Gateway / Backend for Front-End**. 2017. Acesso em: 17 out. 2017. Disponível em: <http://microservices.io/patterns/apigateway.html.>

RICHARDSON, C. **Pattern: Client-side service discovery**. 2017. Acesso em: 20 mar. 2018. Disponível em: <http://microservices.io/patterns/client-side-discovery.html.>

RODRÍGUEZ, G.; DÍAZ-PACE, J. A.; SORIA, Á. A case-based reasoning approach to reuse quality-driven designs in service-oriented architectures. **Information Systems**, Elsevier, v. 77, p. 167–189, 2018.

SANGAL, N. et al. Using dependency models to manage complex software architecture. In: **20th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)**. [S.l.: s.n.], 2005. p. 167–176.

SARKAR, S.; MASKERI, G.; RAMACHANDRAN, S. Discovery of architectural layers and measurement of layering violations in source code. **Journal of Systems and Software**, v. 82, n. 11, p. 1891–1905, 2009.

SPRING, P. **REST in Spring 3: RESTTemplate**. 2018. Acesso em: 27 mar. 2018. Disponível em: <https://spring.io/blog/2009/03/27/rest-in-spring-3-resttemplate.>

SUDKAMP, T. A.; COTTERMAN, A. **Languages and machines: an introduction to the theory of computer science**. 3. ed. [S.l.]: Pearson, 1988.

SULLIVAN, K. J. et al. The structure and value of modularity in software design. In: **9th International Symposium on Foundations of Software Engineering (FSE)**. [S.l.: s.n.], 2001. p. 99–108.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. **Software: Practice and Experience**, v. 39, n. 12, p. 1073–1094, 2009.

TERRA, R. et al. A recommendation system for repairing violations detected by static architecture conformance checking. **Software: Practice and Experience**, v. 45, n. 3, p. 315–342, 2015.

THÖNES, J. Microservices. **IEEE Software**, v. 32, n. 1, p. 116–126, 2015.

TOFFETTI, G. et al. Self-managing cloud-native applications: Design, implementation, and experience. **Future Generation Computer Systems**, v. 72, n. 10, p. 165–179, 2017.

VERBAERE, M.; GODFREY, M. W.; GIRBA, T. Query technologies and applications for program comprehension (QTAPC 2008). In: **16th International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2008. p. 285–288.

VILLAMIZAR, M. et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: **10th Computing Colombian Conference (CCC)**. [S.l.: s.n.], 2015. p. 583–590.

XU, C. et al. CAOPLE: A programming language for microservices SaaS. In: **10th Symposium on Service-Oriented System Engineering (SOSE)**. [S.l.: s.n.], 2016. p. 34–43.

YAN, H. et al. Discotect: A system for discovering architectures from running systems. In: **26th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2004.

ZDUN, U.; NAVARRO, E.; LEYMANN, F. Ensuring and assessing architecture conformance to microservice decomposition patterns. In: **15th International Conference on Service-Oriented Computing (ICSOC)**. [S.l.: s.n.], 2017. p. 411–429.

ZHOU, X. et al. Poster: Benchmarking microservice systems for software engineering research. In: **40th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2018. p. 323–324.

ZIMMERMANN, O. Microservices tenets: agile approach to service development and deployment. **Computer Science-Research and Development**, v. 32, n. 3, p. 301–310, 2016.

# APÊNDICE A – Gramática da Linguagem DCL$^+$

Este apêndice apresenta a gramática completa da linguagem DCL$^+$ na notação BNF (*Backus-Nahur Form*) (SUDKAMP; COTTERMAN, 1988). É válido ressaltar que as definições descritas na cor azul são específicas para as restrições de comunicação e as demais já são apresentadas pela linguagem DCL (TERRA; VALENTE, 2009).

```
S: {MsDef}
MsDef: MsHeader {ModDecl | DCDecl |CCDecl}
MsHeader: MsId ; MsUrl ; MsRepo ; MsPL
ModDecl: module ModId: ModDef {, ModDef}
ModDef: ClassName | ClassName+ | Pkg* | Pkg** | RegExpr
DCDecl:
  only RefMod can-Type {, can-Type} RefMod |
  RefMod can-Type-only {, can-Type-only} RefMod |
  RefMod cannot-Type {, cannot-Type} RefMod |
  RefMod must-Type {, must-Type} RefMod
CCDecl:
  only RefMod can-communicate RefMs |
  RefMod can-communicate-only RefMs |
  RefMod cannot-communicate RefMs |
  RefMod must-communicate RefMs
RefMod: (ModDef | ModId) {, RefMod}
RefMs: MsId using EndPoint {, RefMs}
Type: access | declare | handle | create | depend | extend
    | implement | derive | throw | useannotation
```

Com o intuito de simplificar o entendimento da gramática, foram ocultadas as definições de nove variáveis que geram terminais de uso comum: `ModId` e `MsId` que se referem a nomes de identificadores (de módulo e de microsserviço, respectivamente), `MsUrl` que se refere a URL de acesso ao microsserviço, `MsRepo` que se refere ao caminho do código fonte do microsserviço, `MsPL` que se refere ao nome da linguagem de programação na qual o microsserviço foi implementado, `ClassName` ao nome qualificado de uma classe, `Pkg` ao nome de um pacote, `RegExpr` a qualquer expressão regular e `EndPoint` ao nome de um *end-point* de um microsserviço.

# APÊNDICE B – DCL$^+$ do Projeto Estrutural do Sistema Avaliado

Este apêndice apresenta a descrição completa das especificações em DCL$^+$ (Listagens 1 a 11) do projeto estrutural dos microsserviços da aplicação Conciliação Bancária abordada no Capítulo 5.

```
1  Node-Middleware: http://localhost:8099; .../node-middleware/; JavaScript
2  #Only-can
3  only App can-access Routes                                                    "NM-RE1"
4  #Can-only
5  Index can-access-only App, Node_module, Config, TransfData                    "NM-RE2"
6  App can-access-only Routes, Node_module, Config, TransfData                   "NM-RE3"
7  Routes can-access-only ConciliacaoBancaria, Node_module, Config,
       TransfData                                                                "NM-RE4"
8  AccessProfile can-access-only AccessProfile, Node_module, Config,
       TransfData                                                                "NM-RE5"
9  Assignment can-access-only Assignment, Node_module, Config,
       TransfData                                                                "NM-RE6"
10 AttributesSet can-access-only AttributesSet, Node_module, Config,
       TransfData                                                                "NM-RE7"
11 AuditLog can-access-only AuditLog, Node_module, Config, TransfData            "NM-RE8"[2]*
12 BankAccounts can-access-only BankAccounts, Node_module, Config,
       TransfData                                                                "NM-RE9"
13 BankStatement can-access-only BankStatement, Node_module, Config,
       TransfData                                                                "NM-RE10"[2]*
14 Categories can-access-only Categories, Node_module, Config,
       TransfData                                                                "NM-RE11"
15 Cities can-access-only Cities, Node_module, Config, TransfData                "NM-RE12"
16 Client can-access-only Client, Node_module, Config, TransfData                "NM-RE13"
17 ConcilPlan can-access-only ConcilPlan, Node_module, Config,
       TransfData                                                                "NM-RE14"
18 ConcilReport can-access-only ConcilReport, Node_module, Config,
       TransfData                                                                "NM-RE15"[2]*
19 ConcilSummary can-access-only ConcilSummary, Node_module, Config,
       TransfData                                                                "NM-RE16"[2]*
20 Dashboard can-access-only Dashboard, Node_module, Config,
       TransfData                                                                "NM-RE17"[2]*
21 Establishment can-access-only Establishment, Node_module, Config,
       TransfData                                                                "NM-RE18"
22 EstablishGroups can-access-only EstablishGroups, Node_module, Config,
       TransfData                                                                "NM-RE19"
23 FileUpload can-access-only FileUpload, Node_module, Config,
       TransfData                                                                "NM-RE20"
24 FinancMovements can-access-only FinancMovements, Node_module, Config,
       TransfData                                                                "NM-RE21"[2]*
25 Home can-access-only Home, Node_module, Config, TransfData                    "NM-RE22"
26 ManualConcil can-access-only ManualConcil, Node_module, Config,
       TransfData                                                                "NM-RE23"[2]*
27 OccurrenceReason can-access-only OccurrenceReason, Node_module, Config,
       TransfData                                                                "NM-RE24"
28 OccurrenceReport can-access-only OccurrenceReport, Node_module, Config,
       TransfData                                                                "NM-RE25"[2]*
29 Operator can-access-only Operator, Node_module, Config, TransfData            "NM-RE26"[2]*
30 ProcessedFiles can-access-only ProcessedFiles, Node_module, Config,
       TransfData                                                                "NM-RE27"
31 States can-access-only States, Node_module, Config, TransfData                "NM-RE28"
32 Task can-access-only Task, Node_module, Config, TransfData                    "NM-RE29"
33 Treegrid can-access-only Treegrid, Node_module, Config, TransfData            "NM-RE30"
34 User can-access-only User, Node_module, Config, TransfData                    "NM-RE31"
35 TransfData can-access-only TransfData, Node_module, Config                    "NM-RE32"
36 #Cannot
37 App cannot-depend Index                                                       "NM-RE35"
38 Routes cannot-depend Index, App                                               "NM-RE36"
39 #Must
40 Index must-access App                                                        "NM-RE37"
41 App must-access Routes                                                       "NM-RE38"
```

Listagem 1 – Especificação do projeto estrutural do sistema orquestrador `Node-Middleware`.

```
1  Audit: http://localhost:8085; .../microservices/audit/; Java
2  #Only-can
3  only Controller, Service can-depend Service                          "#Audit-RE1"
4  only Controller can-useannotation CtlrAnnotations                    "#Audit-RE2"
5  only Service can-depend Jndi                                         "#Audit-RE3"[3]*
6  only Service can-throw ExceptionCommons                              "#Audit-RE4"
7
8  #Can-only
9  Main can-access-only Controller, MainAnnotations, Logger, Apache, $API
                                                                        "#Audit-RE5"[4]*
10
11 #Cannot
12 Domain cannot-depend JPA                                             "#Audit-RE6"
13 Main cannot-depend CBMultitenancy                                    "#Audit-RE7"
14
15 #Must
16 Service must-useannotation ServiceAnnotation                         "#Audit-RE8"
```

Listagem 2 – Especificação do projeto estrutural do microsserviço Audit.

```
1  Authentication: http://localhost:8082; .../microservices/authentication/; Java
2  #Only-can
3  only Controller can-useannotation CtlrAnnotations          "#Authentication-RE1"
4  only Main can-useannotation MainAnnotations                "#Authentication-RE2"
5  only Service can-useannotation ServiceAnnotation           "#Authentication-RE3"
6  only Domain, Controller, Service can-depend PltDomain      "#Authentication-RE4"
7  only Service can-depend MyBatis                            "#Authentication-RE5"[2]*
8
9  #Cannot
10 system cannot-depend BaseCommons                           "#Authentication-RE6"
11 Service cannot-access Controller                           "#Authentication-RE7"
12
13 #Must
14 Controller must-access Service, Domain                     "#Authentication-RE8"
15 Service must-useannotation ServiceAnnotation               "#Authentication-RE9"[2]*
16 Main must-useannotation MainAnnotations                    "#Authentication-RE10"
```

Listagem 3 – Especificação do projeto estrutural do microsserviço Authentication.

```
1  Authorization: http://localhost:8083; .../microservices/authorization/; Java
2  #Only-can
3  only Controller, Service can-depend Service                "#Authorization-RE1"
4  only Controller can-useannotation CtlrAnnotations          "#Authorization-RE2"
5  only Service can-depend Jndi                               "#Authorization-RE3"[5]*
6  only Main can-depend MainAnnotations                       "#Authorization-RE4"
7
8  #Can-only
9  Main can-depend-only $API, MainAnnotations, Apache         "#Authorization-RE5"[6]*
10
11 #Cannot
12 Controller cannot-access Main                              "#Authorization-RE6"
13 system cannot-depend BCMultitenancy                        "#Authorization-RE7"
14 Service cannot-access Controller                           "#Authorization-RE8"
15
16 #Must
17 ServiceImpl must-useannotation ServiceAnnotation           "#Authorization-RE9"
18 ServiceImpl must-implement ServiceInterface                "#Authorization-RE10"
19 Controller must-useannotation CtlrAnnotations              "#Authorization-RE11"
```

Listagem 4 – Especificação do projeto estrutural do microsserviço Authorization.

```
Conciliation: http://localhost:8093; .../microservices/conciliation/; Java
#Only-can
only Controller can-useannotation CtlrAnnotations                    "#Conciliation-RE1"
only REheduled can-depend AmazonSQS                                  "#Conciliation-RE2"
only Service can-access DAO                                          "#Conciliation-RE3"
only Controller, Service can-access Service                          "#Conciliation-RE4"
only Main can-depend BCMultitenancy                                  "#Conciliation-RE5"
only DAO, Service can-depend BCDAO                                   "#Conciliation-RE6"

#Can-only
Util can-depend-only Util, $API, Logger                              "#Conciliation-RE7"[24]*
only REheduled can-depend AmazonSQS                                  "#Conciliation-RE8"
Domain can-access-only BCModel, BCDomain, java                       "#Conciliation-RE9"
ImplementDAO can-depend-only JPA, Hibernate, java, SpringRepository,
     BCModel                                                         "#Conciliation-RE10"
Main can-access-only Controller, MainAnnotations, Logger, java
                                                                     "#Conciliation-RE11"

#Cannot
Domain cannot-depend JPA                                             "#Conciliation-RE12"
Controller cannot-access DAO                                         "#Conciliation-RE13"

#Must
Main must-useannotation MainAnnotation                               "#Conciliation-RE14"
ImplementDAO must-depend SpringRepository                            "#Conciliation-RE15"
Service must-useannotation ServiceAnnotation                         "#Conciliation-RE16"
```

Listagem 5 – Especificação do projeto estrutural do microsserviço Conciliation.

```
Dashboard: http://localhost:8095; .../microservices/dashboard/; Java
#Only-can
only Controller can-useannotation CtlrAnnotations                    "#Dashboard-RE1"
only Controller can-depend BCController                              "#Dashboard-RE2"
only Main can-depend BCMultitenancy                                  "#Dashboard-RE3"
only Service, DAO can-access DAO                                     "#Dashboard-RE4"

#Can-only
ImplementDAO can-depend-only JPA, Hibernate, java, SpringRepository,
     BCModel                                                         "#Dashboard-RE5"

#Cannot
Controller cannot-access DAO                                         "#Dashboard-RE6"
Service cannot-depend Controller                                     "#Dashboard-RE7"

#Must
Main must-useannotation MainAnnotations                              "#Dashboard-RE8"

Controller must-access Service                                       "#Dashboard-RE9"
ImplementDAO must-implement DAOInterface                             "#Dashboard-RE10"
ImplementDAO must-depend SpringRepository                            "#Dashboard-RE11"
Service must-useannotation ServiceAnnotation                         "#Dashboard-RE12"
```

Listagem 6 – Especificação do projeto estrutural do microsserviço Dashboard.

```
Entries: http://localhost:8092; .../microservices/entries/; Java
#Only-can
only Controller can-useannotation CtlrAnnotations                    "#Entries-RE1"
only Controller, Service can-access Service                          "#Entries-RE2"
only Service can-access DAO                                          "#Entries-RE3"
only Main can-depend BCMultitenancy                                  "#Entries-RE4"[3]*

#Can-only
Util can-depend-only Util, $API                                     "#Entries-RE5"[2]*
ImplementDAO can-depend-only JPA, Hibernate, SpringRepository, java
                                                                     "#Entries-RE6"

#Cannot
Controller cannot-access DAO                                         "#Entries-RE7"[6]*

#Must
Controller must-useannotation CtlrAnnotations                        "#Entries-RE8"[2]*
Service must-useannotation ServiceAnnotation                         "#Entries-RE9"[7]*
Main must-useannotation MainAnnotations                              "#Entries-RE10"
Model, Serializer must-implement Serializable                        "#Entries-RE11"[11]*
ImplementDAO must-implement SpringRepository                         "#Entries-RE12"
```

Listagem 7 – Especificação do projeto estrutural do microsserviço Entries.

```
FileLoad: http://localhost:8075; .../microservices/fileLoad/; Java
#Only-can
only Model, DAO can-depend JPA, Beans                                "#FileLoad-RE1"
only Main can-useannotation MainAnnotations, DataSource              "#FileLoad-RE2"
only Main can-depend BCMultitenancy                                  "#FileLoad-RE3"
only Service can-useannotation ServiceAnnotation                     "#FileLoad-RE4"

#Can-only
ImplementDAO can-depend-only JPA, Hibernate, BCModel, SpringRepository
                                                                     "#FileLoad-RE5"

#Cannot
Main cannot-depend DAO, Model, BCDAO, JPA                            "#FileLoad-RE6"[4]*
Service cannot-access Main                                           "#FileLoad-RE7"

#Must
ImplementDAO must-implement DAOInterface                             "#FileLoad-RE8"
ImplementDAO must-depend SpringRepository                            "#FileLoad-RE9"
Model must-implement Serializable                                    "#FileLoad-RE10"[6]*
Service must-useannotation ServiceAnnotation                         "#FileLoad-RE11"
```

Listagem 8 – Especificação do projeto estrutural do microsserviço FileLoad.

```
FileProcess: http://localhost:8076; .../microservices/fileProcess/; Java
#Only-can
only Main can-useannotation MainAnnotations                          "#FileProcess-RE1"
only REheduling can-access REdFramework                              "#FileProcess-RE2"
only Main can-depend BCMultitenancy                                  "#FileProcess-RE3"
only DAO, Model can-depend JPA                                       "#FileProcess-RE4"[8]*
only Service, REheduling can-depend AmazonService                    "#FileProcess-RE5"

#Can-only
Util can-depend-only Util, BCUtil, java, Logger                     "#FileProcess-RE6"[22]*

#Cannot
Service cannot-depend Controller                                     "#FileProcess-RE7"

#Must
Main must-useannotation MainAnnotations                              "#FileProcess-RE8"
Service must-useannotation ServiceAnnotation                         "#FileProcess-RE9"
ImplementDAO must-implement DAOInterface, SpringRepository           "#FileProcess-RE10"
Model must-implement Serializable                                    "#FileProcess-RE11"[1]*
REheduling must-depend REdFramework                                  "#FileProcess-RE12"
REheduling must-useannotation REdAnnotation                          "#FileProcess-RE13"
```

Listagem 9 – Especificação do projeto estrutural do microsserviço FileProcess.

```
1  Reports: http://localhost:8094; .../microservices/reports/; Java
2  #Only-can
3  only Controller can-useannotation CtlrAnnotations                    "#Reports-RE1"
4  only Main can-depend BCMultitenancy                                  "#Reports-RE2"
5  only Main can-useannotation MainAnnotations                          "#Reports-RE3"
6
7  #Cannot
8  Service cannot-access Controller                                     "#Reports-RE4"
9
10 #Must
11 Main must-useannotation MainAnnotation                               "#Reports-RE5"
12 Controller must-useannotation CtlrAnnotations                        "#Reports-RE6"[1]*
13 Controller must-extend BCController                                  "#Reports-RE7"[1]*
14 Service must-useannotation ServiceAnnotation                         "#Reports-RE8"
15 Serializer must-depend JSONSerializer                                "#Reports-RE9"[4]*
```

Listagem 10 – Especificação do projeto estrutural do microsserviço Reports.

```
1  Summary: http://localhost:8096; .../microservices/summary/; Java
2  #Only-can
3  only Service can-useannotation ServiceAnnotation                     "#Summary-RE1"
4  only Controller can-useannotation CtlrAnnotations                    "#Summary-RE2"
5  only Main can-useannotation MainAnnotations                          "#Summary-RE3"
6  only Main can-depend BCMultitenancy                                  "#Summary-RE4"
7
8  #Can-only
9  Util can-access-only Util, java                                      "#Summary-RE5"
10
11 #Must
12 Main must-useannotation MainAnnotations                              "#Summary-RE6"
13 Service must-useannotation ServiceAnnotation                         "#Summary-RE7"[1]*
14 Controller must-useannotation CtlrAnnotations                        "#Summary-RE8"[1]*
15 ServiceSubClass must-extend BaseService                              "#Summary-RE9"
16 ImplementDAO must-implement DAOInterface                             "#Summary-RE10"
17 ImplementDAO must-useannotation Repository                           "#Summary-RE11"
18 Controller must-access Stream                                        "#Summary-RE12"
```

Listagem 11 – Especificação do projeto estrutural do microsserviço Summary.

# APÊNDICE C – Violações do Projeto Estrutural do Sistema Avaliado

Este apêndice apresenta as DSMs (Figuras 1 e 2a-j) com ênfase nas violações do projeto estrutural dos microsserviços da aplicação Conciliação Bancária abordada no Capítulo 5.

Figura 1 – Violações do projeto estrutural do microsserviço `Node-Middleware`.

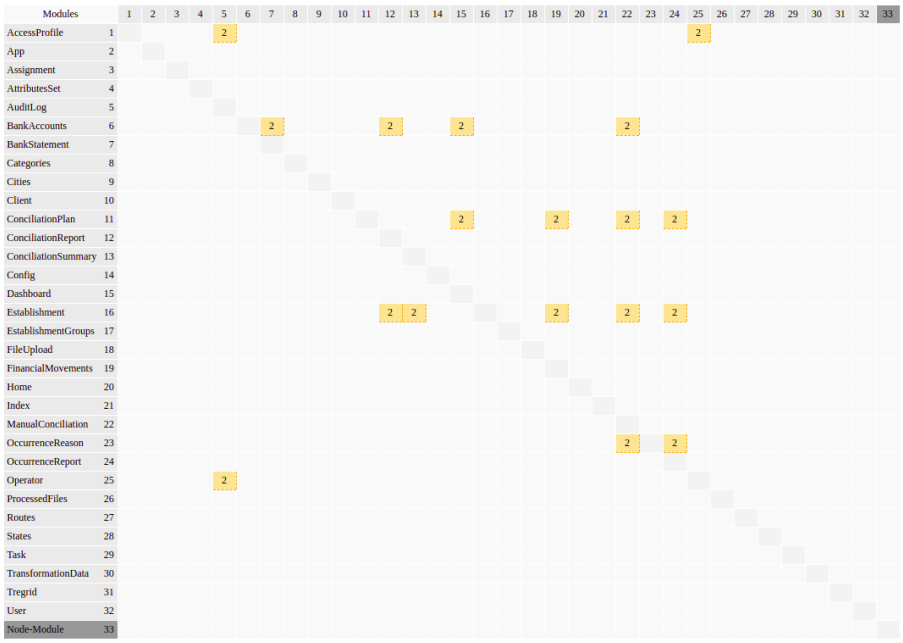
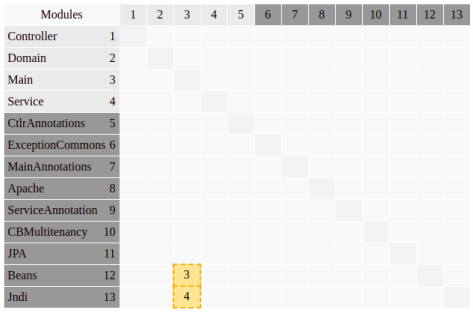
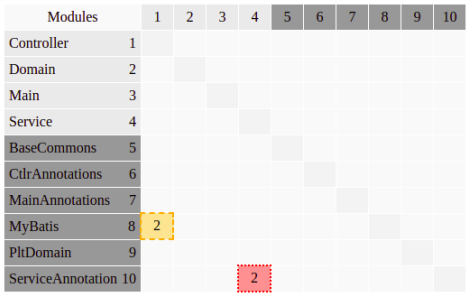


Figura 2 – Violações do projeto estrutural dos microsserviços da aplicação Conciliação Bancária.



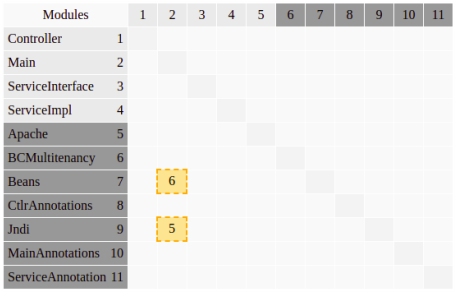

Violações do microsserviço Audit.

(a)



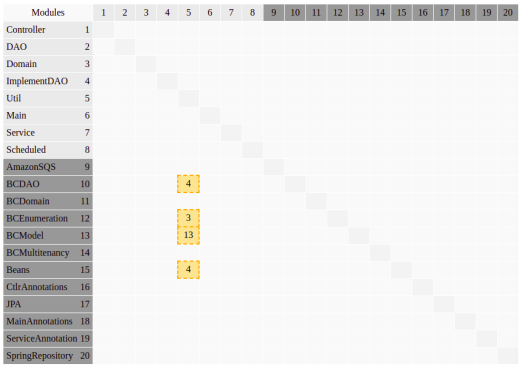

Violações do microsserviço Authentication.

(b)



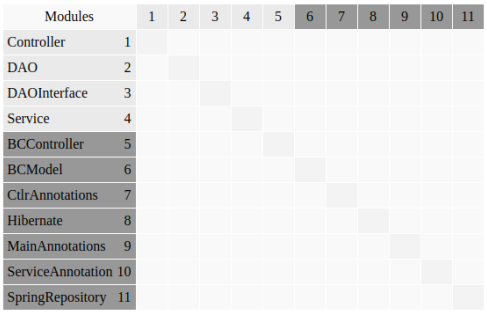

Violações do microsserviço Authorization.

(c)



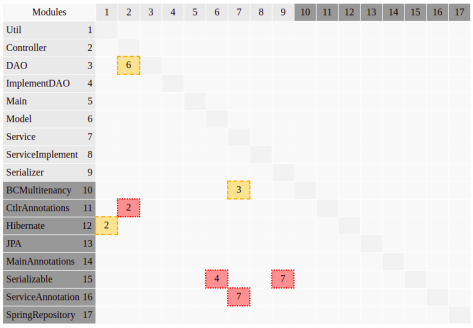

Violações do microsserviço Conciliation.

(d)



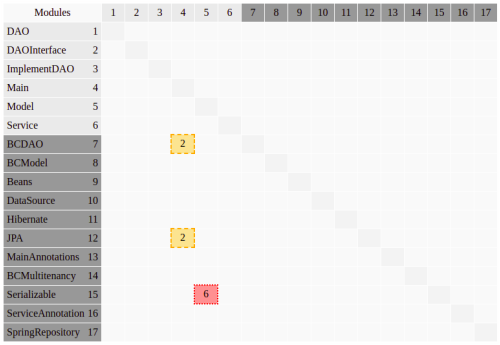

Violações do microsserviço Dashboard.

(e)



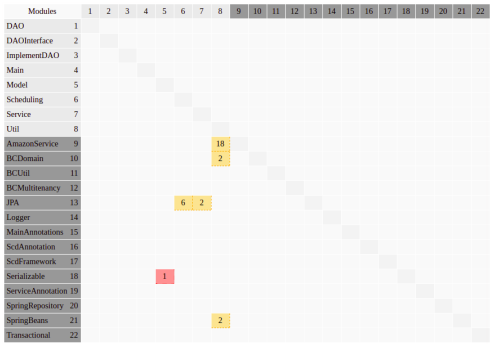

Violações do microsserviço Entries.

(f)



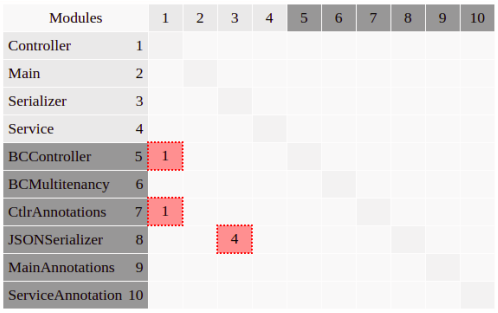

Violações do microsserviço FileLoad.

(g)



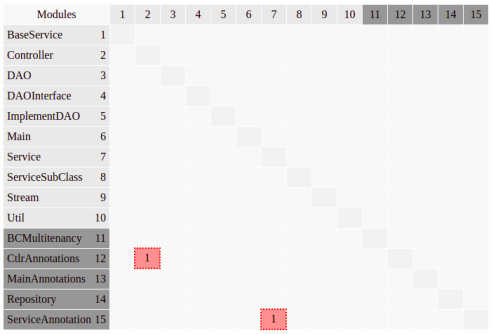

Violações do microsserviço FileProcess.

(h)




Violações do microsserviço Reports.

(i)




Violações do microsserviço Summary.

(j)