UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

LUCAS FERREIRA DE LIMA

ORIENTADOR: RODRIGO BONIFÁCIO DE ALMEIDA

DISSERTAÇÃO DE MESTRADO EM
ENGENHARIA ELÉTRICA

PUBLICAÇÃO: MTARH.DM - 017 A/99

BRASÍLIA/DF: JULHO - 2016.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

LUCAS FERREIRA DE LIMA

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA.

APROVADA POR:

Prof. Rodrigo Bonifácio de Almeida, DSc. (ENE-UnB)
(Orientador)

Prof. XXX
(Examinador Interno)

YYY
(Examinador Externo)

BRASÍLIA/DF, 01 DE JULHO DE 2016.

DEDICATÓRIA

Este trabalho é dedicado a ...
continuação .....

AGRADECIMENTOS

(Página opcional) Agradeço .....
continuação .....

RESUMO

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

Autor: Lucas Ferreira de Lima
Orientador: Rodigo Bonifácio de Almeida
Programa de Pós-Graduação em Engenharia Elétrica
Brasília, julho de 2016

A adoção do paradigma arquiteturação baseado em serviços... O presente trabalho foi desenvolvido, ..., tendo como objetivo geral ..... Como objetivos específicos, o trabalho procurou .....

A proposta partiu...

Durante a realização...

Os resultados sugerem...

ABSTRACT

CONTRATOS REST ROBUSTOS E LEVES: UMA ABORDAGEM EM DESIGN-BY-CONTRACT COM NEOIDL

Autor: Lucas Ferreira de Lima
Orientador: Rodigo Bonifácio de Almeida
Programa de Pós-Graduação em Engenharia Elétrica
Brasília, julho de 2016

..

..

..

..

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

SOC: Software Oriented Computing, modelo arquitetural baseado em serviços.

DbC: Design by Contract, mecanismos de garantias com condições na chamadas a métodos, funções, serviços, etc.

1 INTRODUÇÃO

A computação orientada a serviços (*Service-oriented computing, SOC)* tem se mostrado uma solução de *design* de *software* que favorece o alinhamento às mudanças constantes e urgentes nas instituições [5]. Nessa abordagem, os recursos de software são empacotados como serviços, módulos bem definidos e auto-contidos, que provêem funcionalidades negociais e com independência de estado e contexto [22].

Os benefícios de SOC estão diretamente relacionados ao baixo acoplamento dos serviços que compõem a solução, de forma que as partes (nesse caso serviços) possam ser substituídas e evoluídas facilmente, ou ainda rearranjadas em novas composições. Contudo, para que isso seja possível, é necessário que os serviços possuam contratos bem escritos e independentes da implementação.

A relação entre quem provê e quem consome o serviço se dá por meio de um contrato. O contrato de serviço é o documento que descreve os propósitos e as funcionalidades do serviço, como ocorre a troca de mensagens, informações sobre as operações e condições para sua execução [8].

Nesse contexto, a qualidade da especificação do contrato é fundamental para o projeto de software baseado em SOC. Este trabalho de pesquisa aborda um aspecto importante para a melhoria da robustez de contratos de serviços: a construção de garantias mútuas por meio da especificação formal de contratos, agregando o conceito de Design by Contract.

1.1 PROBLEMA DE PESQUISA

As linguagens de especificação de contratos para SOC apresentam algumas limitações. Por exemplo, a linguagem WSDL (*Web-services description language*) [2] é considerada uma solução verbosa que desestimula a abordagem *Contract First*. Por essa razão, especificações WSDL são usualmente derivadas a partir de anotações em código fonte *Code First*. Além disso, os conceitos descritos em contratos na linguagem WSDL não são diretamente mapeados aos elementos que compõem as interfaces do estilo arquitetural REST (*Representational State Transfer*).

Outras alternativas para REST, como Swagger e RAML[1], usam linguagens de propósito geral (em particular JSON e YAML) adaptadas para especificação de contratos. Ainda que façam uso de contratos mais sucintos que WSDL, essas linguagens não se beneficiam da clareza típica das linguagens específicas para esse fim (como IDLs CORBA) e não oferecem mecanismos semânticos de extensibilidade e modularidade.

Com o objetivo de mitigar esses problemas, a linguagem NeoIDL foi proposta para simplificar a especificação de serviços REST com mecanismos de modularização, suporte a anotações, herança em tipos de dados definidos pelo desenvolvedor, e uma sintaxe simples e concisa semelhante às *Interface Description Languages* – IDLs – presentes em *Apache Thrift*™e CORBA™.

Por outro lado, a NeoIDL, da mesma forma que WSDL, Swagger e RAML não oferece construções para especificação de contratos formais com aspecto comportamental como os presentes em linguagens que suportam DBC (*Design by Contract*) [18], como Eiffel, JML e Spec#. Em outras palavras, a NeoIDL admite apenas contratos fracos (*weak contracts*), sem suporte a construções como pré e pós-condições.

1.2 OBJETIVO GERAL

O objetivo geral de trabalho é investigar o uso de construções de Design by Contract no contexto de computação orientada a serviços, verificando a viabilidade e utilidade de sua adoção na especificação de contratos e implentação de serviços REST.

1.2.1 Objetivos específicos

1. Realizar análise empírica de expressividade e reuso da especificação de contratos em NeoIDL em comparação com *Swagger*, a partir de contratos reais do Exército Brasileiro;

2. Extender a sintaxe da NeoIDL para admitir construções de Design by Contract, com pré e pós condições para operações de serviços REST;

3. Implementar um estudo de caso de geração de código em *Python Twisted* com suporte a Design by Contract a partir de contratos especificados em NeoIDL;

[1] http://raml.org/spec.html

4. Coletar a percepção de desenvolvedores sobre a aceitação da especificação de contratos REST com Design by Contract na NeoIDL.

1.3 JUSTIFICATIVA E RELEVÂNCIA

A demanda por integração entre sistemas de várias origens e tecnologias diversas fez aumentar a adoção de soluções baseada em computação orienta a serviços. Isso se deve justamente à necessidade de tornar a interoperabilidade de soluções heterogênias o menos acopladas possível, de modo que mudanças nos requisitos de negócio ou na inclusão de novos serviços sejam atendidas com simplicidade, eficiência e rapidez.

O uso de *Web Service* é a forma mais comum de se implementar os serviços. O desenvolvimento de *Web Service*, que eram inicialmente projetados sobre a abordagem SOAP, com o tráfego de mensagens codificadas em XML, tem gradativamente se intensificado no sentido da utilização de REST.

Uma dos principais benefícios do uso de SOC está na possibilidade de reuso de seus componentes. Porém, reuso requer serviços bem construídos e precisos em relação a sua especificação [12]. A qualidade e precisão do contrato de serviço torna-se claramente um elemento fudamental para que auferir os benefícios da abordagem SOC.

Nesse contexto, REST não dispõe de um meio padrão para especificação de contratos. Linguangens como Swagger, YAML e WADL cumprem com o propósito de especificar contratos REST, porém padecem do mesmo problema: são voltados para computadores e de escrita e leitura complexa para humanos, o que prejudica a prática de *Contract-first*. A linguagem NeoIDL foi concebida com o objetivo de ser mais expressiva para humanos, além de outros propósitos.

Todas essas linguagens tem, entretanto, um outra limitação em comum: não dão suporte a contratos robustos, com garantias. A estratégia para superar essa limitação foi de buscar no paradigma de orientação a objetos, que é uma das principais influências de orientação a serviços [8], o conceito de Design by Contract. Ambas as abordagens, orientação a serviços e a objetos, tem em comum a ênfase no reuso e comunicação entre componentes (serviços e classes).

A proposta deste trabalho de pesquisa de mestrado está justamente em incluir garantias

na especificação de contratos REST, extendendo a linguagem NeoIDL para suportar construções de Design by Contract.

1.4 ESTRUTURA

Este trabalho está organizado em quatro capítulos...

2 REFERENCIAL TEÓRICO

2.1 COMPUTAÇÃO ORIENTADA A SERVIÇO

As empresas precisam estar preparadas para responder rápida e eficientemente a mudanças impostas por novas regulações, por aumento de competição ou ainda para usufruir de novas oportunidades. No contexto atual, em que as informações fluem de modo extremamente veloz, o tempo disperdiçado pelas organizações para se adaptar a um novo cenário tem um preço elevado, gerando expressiva perda de receita e, em um determinados casos, podendo causar a falência.

No campo das instituições governamentais, a eficiência na condução das ações do Estado impõem que a estrutura de troca de informações entre os mais variados entes seja continuamente adaptável, mutuamente integrada. Pode-se tomar como exemplo a edição de nova lei que implique alteração no cálculo do tempo de serviço para aposentadoria. A nova fórmula deve se propagar para ser aplicada em várias instituições que compõem a máquina pública.

Nessas situações, os sistemas de informação das organizações devem possibilitar que a dinâmica de adaptação ocorra sem demora, sob pena de, em vez de serem ferramental para apoiar continuamente os processos de negócio, se tornem entrave para a ágil incorporação dos novos processos. Por outro lado, a nova configuração deve se manter integra e funcional com o já complexo cenário de TI.

A eficiência na integração entre as soluções de TI é determinante para que se consiga alterar uma parte sem comprometer todo o ecossistema. A integração possibilita a combinação de eficiência e flexibilidade de recursos para otimizar a operação através e além dos limites de uma organização, proporcionando maior interoperabilidade [23].

A computação orientada a serviços – SOC – endereça essas necessidades em uma plataforma que aumenta a flexibilidade e melhora o alinhamento com o negócio, a fim de reagir rapidamente a mudanças nos requisitos de negócio. Para obter esses benefícios, os serviços devem cumprir com determinados quesitos, que incluem alta autonomia ou baixo acoplamento [7]. Assim, o paradigma de SOC está voltado para o projeto de

soluções preparadas para constantes mudanças, substituindo-se continuamente pequenas peças – os serviços – por outras atualizadas.

Portando, o objetivo da SOC é conceber um estilo de projeto, tecnologia e processos que permitam às empresas desenvolver, interconectar e manter suas aplicações e serviços corporativos com eficiência e baixo custo. Embora esses objetivos não sejam novos, SOC procura superar os esforços prévios como programação modular, reuso de código e técnicas de desenvolvimento orientadas a objetos [24].

As vertentes mais visionárias da computação orientada a serviços prevêem, em seu estado da arte, uma coordenação de serviços cooperantes por todo o mundo, onde os componentes possam ser conectados facilmente em uma rede de serviços pouquíssimo acoplados e, assim, criar processos de negócio dinâmicos e aplicações ágeis entre organizações e plataformas de computação [15].

2.1.1 Terminologia

Computação orientada a serviço é um termo *guarda-chuva* para descrever uma nova geração de computação distribuída. Desse modo, é um conceito que engloba vários pontos, como paradigmas e princípios de projeto, catálogo de padrões de projeto, padronização de linguagem, modelo arquitetural específico, e conceitos correlacionados, tecnologias e plataformas. A computação orientada a serviços é baseada em modelos anteriores de computação distribuída e os extendem com novas camadas de projeto, aspectos de governança, e uma grande gama de tecnologias de implementações especializadas, em grande parte baseadas em *Web Service* [8].

Orientação a serviço é um paradígma de projeto cuja intenção é a criação de unidades lógicas moldadas individualmente para podem serem utilizadas conjutamente e repetidamente, atendendo assim a objetivos e funções específicos associados com SOA e computação orientada a serviço.

A lógica concebida de acordo com orientação a serviço pode ser designada de **orientada a serviço**, e as unidades da lógica orientada a serviço são referenciadas como **serviços**. Como um paradigma de computação distribuída, a orientação a serviço pode ser comparada a orientação a objetos, de onde advém várias de suas raízes, além da influência de EAI, BMP e *Web Service*[8].

A orientação a serviços é composta principalmente de oito princípios de projeto (descritos na subseção 2.1.3).

Arquitetura orientada a serviço - SOA representa um modelo arquitetural cujo objetivo é elevar a agilidade e a redução de custos e ao mesmo tempo reduzir o peso da TI para a organização. Isso é feito colocando o serviço como elemento central da representação lógica da solução [8].

Como uma arquitetura tecnológica, uma implementação SOA consiste da combinação de tecnologias, produtos, APIs, extensões da infraestrutura, etc. A implantação concreta de uma arquitetura orientada a serviço é única para cada organização, entretanto é caracterizada pela introdução de tecnologias e plataformas que suportam a criação, execução e evolução de soluções orientadas a serviços. O resultado é a formação de um ambiente projetado para produzir soluções alinhadas aos princípios de projeto de orientação a serviço.

Segundo Thomas Erl [8], o termo arquitetura orientada a serviço – SOA – vem sendo amplamente utilizado na mídia e nos produtos de divugação de fabricantes e se tornado quase que sinônimo de computação orientada a serviço – SOC.

Serviço é a unidade da solução no qual foi aplicada a orientação a serviço. É a aplicação dos princípios de projeto de orientação a serviço que distigue uma unidade de lógica como um serviço comparada a outras unidades de serviços que podem existir isoladamente como um objeto ou componente [8].

Após a modelagem conceitual do serviço, os estágios de projeto e desenvolvimento produzem um serviço que é um programa de *software* independente com características específicas para suportar a realização dos objetivos associados a computação orientada a serviço.

Cada serviço possui um contexto funcional distinto e é composto de uma lista de capacidades relacionadas a esse contexto. Então um serviço pode ser considerado um conjunto de capacidades descritas em seu contrato.

Contrato de serviço é o conjunto de documentos que expressam as meta-informações do serviço, sendo a parte que descreve a sua interface técnica a mais fundamental. Esses documentos compõem o contrato técnico do serviço, cuja essência é estabelecer uma API com as funcionalidades providas pelo serviço por meio de suas capacidades [8].

Os serviços implementados como *Web Service* SOAP normalmente são descritos em seu WSDL [1], *XML schemas* and políticas (*WS-policy*). Já os serviços im-

[1] *Web Service Description Language*

plementados como *Web Service* REST não possuem uma linguagem padrão para especificação de contratos. Já foram propostas algumas alternativas como WADL [10], Swagger [1], e NeoIDL [16].

O contrato de serviço também pode ser composto de documentos de leitura humana, como os que descrevem níveis de serviços (*SLA*), comportamentos e limitações. Muitas dessas características também podem ser descritas em linguagens formais (para processamento computacional).

No contexto de orientação a serviço, o projeto do contrato do serviço é de suma importância de tal forma que o princípio de projeto contrato de serviço padronizado é dedicado exclusivamente para cuidar da contratos de serviços uniformes e com qualidade [8].

2.1.2 Objetivos, benefícios e características

De modo diferente de arquiteturas convencionais, ditas monolíticas, em que os sistemas são concebidos agregando continuamente funcionalidades a um mesmo pacote de *software*, a arquitetura orientada a serviço prega o projeto de pequenas aplicações distribuídas que podem ser consumidas tanto por usuários finais como por outros serviços [24].

A unidade lógica da arquitetura orientada a serviços é exatamente o serviço. Serviços são pequenos *softwares* que provêem funcionalidades específicas para serem reutilizadas em várias aplicações. Cada serviço é uma entidade isolada com dependências limitadas de outros recursos compartilhados [27]. Assim, é formada uma abstração entre os fornecedores e consumidores dos serviços, por meio de baixo acoplamento, e promovendo a flexibilidade de mudanças de implementação sem impacto aos consumidores.

A arquitetura SOC busca atingir um conjunto de objetivos e benefícios [6]:

(a) Ampliar a iterperabilidade intrínseca, de modo a se ter uma rápida resposta a mudanças de requisitos de negócio por meio da efetiva reconfiguração das composições de serviços;

(b) Ampliar a federação da solução, permitindo que os serviços possam ser evoluídos e governados individualmente, a partir da uniformização de contratos;

(c) Ampliar a diversificação de fornecedores, fazendo com que se possa evoluir a arquitetura em conjunto com o negócio, sem ficar restrito a características de derminados fornecedores;

(d) Ampliar o alinhamento entre a tecnologia e o negócio, especializando-se alguns serviços ao contexto do negócio e possibilitando sua evolução;

(e) Ampliar o retorno sobre investimento, pois muitos serviços podem ser rearranjados em novas composições sem que se tenha que se construir grandes soluções de custo elevado;

(f) Ampliar a agilidade, remontando as composições por reduzido esforço, beneficiando-se do reuso e interoperabilidade nativas dos serviços;

(g) Reduzir o custo de TI, como resultado de todos os benefícios acima citados.

Para possibilitar que esses benefícios sejam atingidos, quatro características são observadas em qualquer plataforma SOA. A primeira é o direcionamento efetivo ao negócio, levando-se em conta dos objetivos estratégicos de negócio na concepção do projeto arquitetural. Se isso não ocorrer, é inevitável que o desalinhamento com os requisitos de negócio cheguem a níveis muito elevados bem rapidamente [6].

A segunda característica é a independência de fabricante. O projeto arquitetural que considere apenas um fabricante específico levará inadvertidamente a implantação dependente de características proprietárias. Essa dependência também reduzirá a agilidade na reação às mudanças e tornará a arquitetura inefetiva. A arquitetura orientada a serviço deve fazer uso de tecnologias providas pelos fornecedores, sem, no entanto, se tornar dependente dela, por meio de APIs e protocolos padrões de mercado.

Outra característica da aplicação da plataforma SOA é os serviços são considerados recursos corporativos, ou seja, da empresa como um todo. Serviços desenvolvidos para atender um único objetivo perdem esta característica e se assemelham a soluções de propósito específico, tal como soluções monolíticas. O modelo arquitetural deve se guiar pela premissa de que os serviços serão compartilhados por várias áreas da empresa ou farão parte de soluções maiores, como serviços compatilhados.

A capacidade de composição é a quarta característica. Os serviços devem ser projetados não somente para serem reusados, mas também possuir flexibilidade para serem

compostos em diferentes estruturas de variadas soluções. Confiabilidade, escalabilidade, troca de dados em tempo de execução com integridade são pontos chave para essa característica.

2.1.3 Princípios SOA

O paradigma de orientação a serviço é estruturado em oito princípios fundamentais [8]. São eles que caraterizam a abordagem SOA e a sua aplicação fazem com que um serviço se diferencie de um componente ou de um módulo. Os contratos de serviços permeiam a maior parte destes princípios:

Contrato padronizado - Serviços dentro de um mesmo inventário estão em conformidade com os mesmos padrões de contrato de serviço. Os contratos de serviços são elementos fundamentais na arquitetura orientada a serviço, pois é por meio deles que os serviços interagem uns com os outros e com potenciais consumidores. Este princípio tem como foco principal o contrato de serviço e seus requisitos. O padrão de projeto *Contract-first* é uma consequência direta deste princípio [8].

Baixo acomplamento - Os contratos de serviços impõem aos consumidores do serviço requisitos de baixo acoplamento e são, os próprios contratos, desacoplados do seu ambiente. Este princípio também possui forte relação com o contratos de serviço, pois a forma como o contrato é projetado e posicionado na arquitetura é que gerará o benefício do baixo acoplamento. O projeto deve garantir que o contrato possua tão somente as informações necessárias para possibilitar a compreensão e o consumo do serviço, bem como não possuir outras características que gerem acoplamento.

São considerados negativos, e que devem ser evitados, os acoplamentos

(a) do contrato com as funcionalidades que ele suporta, agregando ao contrato características específicas dos processos que o serviço atende;

(b) do contrato com a sua implementação, invertendo a estratégia de conceber primeiramente o contrato;

(c) do contrato com a sua lógica interna, expondo aos consumidores características que levem os consumidores a inadivertidamente aumentarem o acoplamento;

(d) do contrato com a tecnologia do serviço, causando impactos indesejáveis em caso de substituição de tecnologia.

Por outro lado, há um tipo de acoplamento positivo que é o que gera dependência da lógica em relação ao contrato [8]. Ou seja, idealmente a implementação do serviço deve ser derivada do contrato, pondendo se ter inclusive a geração de código a partir do contrato.

Abstração - Os constratos de serviços devem conter apenas informações essenciais e as informações sobre os serviços são limitadas àquelas publicadas em seus contratos. O contrato é a forma oficial a partir da qual o consumidor do serviço faz seu projeto e tudo o que está além do contrato deve ser desconhecido por ele. Por um lado este princípio busca a ocultação controlada de informações. Por outro, visa a simplificação de informações do contrato de modo a assegurar que apenas informações essenciais estão disponíveis.

Reusabilidade - Serviços contém e expressam lógica agnóstica e podem ser disponibilizados como recursos reutilizáveis. Este princípio contribui para se entender o serviço como um produto e seu contrato com uma API genérica para potenciais consumidores. Essa abordagem aplicada ao projeto dos serviços leva a desenhá-lo com lógicas não dependentes de processos de negócio específicos, de modo a torná-los reutilizáveis em vários processos.

Autonomia - Serviços exercem um elevado nível de controle sobre o seu ambiente em tempo de execução. O controle do ambiente não está ligado a dependência do serviço à sua plataforma em termos de projeto, mas sim ao aumento da confiabilidade sobre a execução e redução da dependência dos recursos sobre os quais não se tem controle. O que se busca é a previsibilidade sobre o comportamento do serviço.

Ausência de estado - Serviços reduzem o consumo de recursos restringindo a gestão de estado das informações apenas a quando for necessário. Este princípio visa reduzir ou mesmo remover a sobrecarga gerada pelo gerenciamento do estado de cada operação, aumentando a escalabilidade da plataforma de arquitetura orientação a serviço como um todo. Na composição do serviço, o serviço deve armazenar apenas os dados necessários para completar o processamento, enquanto se aguarda o processamento do serviço acionado.

Descoberta de serviço - Serviços devem conter metadados por meio dos quais os serviços possam ser descobertos e interpretados. Tornar cada

serviço de fácil descoberta e interpreção pelas equipes de projeto é o foco deste princípio. Os próprios contratos de serviço devem ser projetados para incorporar informações que auxiliem na sua descoberta.

Composição - Serviços são participantes efetivos de composição, independentemente do tamanho ou complexidade da composição. O princípio da composição faz com que os projetos de serviços sejam projetados para possibilitar que eles se tornem participantes de composições. Deve-se levar em conta, entretanto, os outros princípios no planejamento de uma nova composição, considerando a complexidade das composições a serem formadas.

2.1.4 Contract First

O princípio do baixo acoplamento tem por objetivo principal reduzir o acoplamento entre o cliente e o fornecedor do serviço. Há vários tipos de acopamentos negativos, como citado acima. Porém, um acoplamento é considerado positivo e desejável: da implementação a partir do contrato. Ou seja, a lógica do serviço deve corresponder ao que está especificado no contrato.

Duas abordagens podem ser seguidas para se produzir esse efeito. A primeira é a geração do contrato a partir da lógica implementada, conhecida como *Code-first*. A outra propõem um sentido inverso, partindo-se do contrato para a geração do código, chamada *Contract-first*. A abordagem *Contract-first* é recomendada para a arquitetura orientada a serviço [8].

Embora muitas vezes preferível pelo desenvolvedor, a desvantagem do uso *Code-first* está no elevado impacto que alterações na implementação causam ao contrato, fazendo com que os clientes dos serviços sejam também afetados. Reduz-se a flexibilidade e extensibilidade, de modo que o reuso é prejudicado. Ainda, eleva-se o risco de os serviços serem projetados para aplicações específicas e não voltados para reuso e composição [13].

A abordagem *Contract-first* preocupa-se principalmente com a clareza, completude e estabilidade do contrato para os clientes dos serviços. Toda a estrutura da informação é definida sem a preocupação sobre restrições ou características das implementações subjacentes. Do mesmo modo, as capacidades são definidas para atenderem a funcionalidades a que se destinam, porém com a preocupação em se promover estabilidade e

reuso.

As principais vantagens do *Contract-first* estão no baixo acoplamento do contrato em relação a sua implementação, na possibilidade de reuso de esquemas de dados (XML ou JSON Schema), na simplificação do versionamento e na facilidade de manutenção [13]. A desvantagem está justamente na complexidade de escrita to contrato. Porém várias ferramentas já foram e vem sendo desenvolvidas para facilitar essa tarefa.

2.2 WEB SERVICES

Web Service são aplicações modulares e autocontidas que podem ser publicadas, localizadas a acessadas pela *Web* [3]. A diferença entre o *Web Service* e a aplicação *Web* propriamente dita é que o primeiro se preocupa apenas com o dado gravado ou fornecido, deixando para o cliente a atribuição de apresentar a informação [27].

A necessidade das organizações de integrar suas soluções, seja entre os sistemas internos ou entre esses e sistemas de outras empresas [26], não é recente. Essa é uma das principais motivações do uso de *Web Service*, por possibilitar que soluções contruídas com tecnologias distintas possam trocar informações por meio da *Web*. Nesse contexto, as arquiteturas orientadas a serviço fazem amplo uso de *Web Service* como meio para disponibilização de serviços.

Há dois tipos de *Web Service*: baseados em SOAP e baseados em REST. Os mais diversos tipo de aplicações podem ser concebidas utilizando *Web Services* SOAP ou REST, situação também aplicável a serviços. Originalmente os serviços utilizaram *Web Service* SOAP, trafegando as informações em uma mensagem codificada em um formato de troca de dados (XML), por meio do protocolo SOAP (seção 2.2.1). Entretanto, a adoção de *Web Service* REST (seção 2.2.2) tem ganhado popularidade [21].

2.2.1 SOAP (W3C)

SOAP – *Simple Object Access Protocol* – é um protocolo padrão W3C que provê uma definição de como trocar informações estruturadas, por meio de XML, entre partes em um ambientes descentralizado ou distribuído [2]. SOAP é um protocolo mais antigo que REST, e foi desenvolvido para troca de informações pela Internet se utilizando de protocolos como HTTP, SMTP, FTP, sendo o primeiro o mais comumente utilizado.

Por ser anterior, SOAP é o padrão de *Web Service* mais comumente utilizado pela indústria. Algumas pessoas chegam a tratar *Web Service* apenas como SOAP e WSDL [27]. SOAP atua como um envelope que transporta a mensagem XML, e possui vastos padrões para transformar e proteger a mensagem e a trasmissão.

2.2.1.1 Especificação de contratos

Os contratos em SOAP são especificados no padrão WSDL – *Web Services Description Language* – que define uma gramática XML para descrever os serviços como uma coleção de *endpoints* capazes de atuar na troca de mensagens. As mensagens e operações são descritas abstratamente na primeira seção do documento. Uma segunda seção, dita concreta, estabelece o protocolo de rede e o formato das mensagens.

Muitas organizações preferem utilizar SOAP por ele dispor de mais mecanismos de segurança e tratamento de erros [27]. Além disso a tipagem de dados é mais forte em SOAP que em REST [21].

2.2.2 REST (Fielding)

O termo REST foi criado por Roy Fielding, em sua tese de doutorado [9], para descrever um modelo arquitetural distribuído de sistemas hipermedia. Um *Web Service* REST é baseado no conceito de recurso (que é qualquer coisa que possua uma URI) que pode ter zero ou mais representações [11].

O estilo arquitetural REST é cliente-servidor, em que o cliente enviar uma requisição por um determinado recurso ao servidor e este retorna uma resposta. Tanto a requisição como a resposta ocorrem por meio da transferência de representações de recursos [21], que podem ser de vários formatos, como XML e JSON [27]. Toda troca de informações ocorre por meio do protocolo HTTP, com uma semântica específica para cara operação:

1. HTTP GET é usado para obter a representação de um recurso.
2. HTTP DELETE é usado para remover a representação de um recurso.
3. HTTP POST é usado para atualizar ou criar a representação de um recurso.
4. HTTP PUT é usado para criar a representação de um recurso.

As transações são independentes entre si e com as transações anteriores, pois o servidor não guarda qualquer informação de sessão do cliente. Todas as informações de estado são trafegadas nas proprias requisições, de modo que as respostas também são independentes. Essas características tornam os *Web Services* REST simples e leves [21].

O uso de REST tem se tornado popular por conta de sua flexibilidade e performance em comparação com SOAP, que precisa envelopar suas informações em um pacote XML [21], de armazenamento, transmissão e processamento onerosos.

2.2.2.1 Especificação de contratos

Ao contrário de SOAP, REST não dispõe de um padrão para especificação de contratos. Essa carência, que no início não era considerada um problema, foi se tornando uma necessidade cada vez mais evidente a medida em que se amplia o conjunto de *Web Services* implantados. Atualmente existem algumas linguagens com o propósito de documentar o contrato REST.

A linguagem mais popular atualmente é *Swagger* cujo projeto se iniciou por volta de 2010 para atender a necessidade de um projeto específico, sendo posteriormente vedida para uma grande empresa. Em janeiro de 2016, *Swagger* foi doada para o *Open API Iniciative (OAI)* e denominada de *Open API Specification*. O propósito da iniciativa é tornar *Swagger* padrão para especificação de APIs com independencia de fornecedor. Apoiam o projeto grandes empresas como Google®, Microsoft® e IBM®.

WADL (*Web Application Description Language*), uma especição baseada em XML semelhante ao WSDL, foi projetada e proposta pela *Sun Microsystems*® e sua última versão submetida ao W3C em 2009. Outra linguagem proposta é a RAML – abreviação de *RESTful API Modeling Language* – baseada em YAML e projetada pela MuleSoft®. Muitos projetos *open source* adotam RAML.

Todas estas linguagens possuem suporte tanto para *Code-first* como para *Contract-first* [28].

2.3 DESIGN BY CONTRACT

Design by Contract [18] - DbC - é um conceito oriundo da orientação a objetos, no qual consumidor e fornecedor firmam entre si garantias para o uso de métodos ou classes. De um lado o consumidor deve garantir que, antes da chamada a um método, algumas condições sejam por ele satisfeitas. Do outro lado o fornecedor deve garantir, se respeitadas suas exigências, o sucesso da execução.

O mecanismo que expressa essas condições são chamados de asserções (*assertions*, em inglês). As asserções que o consumidor deve respeitar para fazer uso da rotina são chamadas de **precondições**. As asserções que fornecem, de parte do fornecedor, as garatias ao consumidor, são denominadas **pós-condições**.

DbC tem o objetivo de aumentar a robustez do sistema e tem na linguagem Eiffel [17] um de seus precursores. Para os mantenedores do Eiffel, DBC é tão importante quanto classes, objetos, herança, etc. O uso de DBC na concepção de sistemas é uma abordagem sistemática que produz sistemas com mais corretude.

O conceito chave de Design by Contract é ver a relação entre a classe e seus clientes como uma relação formal, que expressa os direitos e as obrigações de cada parte [19]. Se, por um lado, o cliente tem a obrigação de respeitar as condições impostas pelo fornecedor para fazer uso do módulo, por outro, o fornecedor deve garantir que o retorno ocorra como esperado.

As precondições vinculam o cliente, no sentido de definir as condições que o habilitam para acionar o recurso. Corresponde a uma obrigação para o cliente e o benefício para o fornecedor [19] de que certos pressupostos serão sempre respeitados nas chamadas à rotina. As pós-condições vinculam o fornecedor, de modo a definir as condições para que o retorno ocorra. Corresponde a uma obrigação para o fornecedor e o benefício para o cliente de que certas propriedades serão respeitadas após a chamada à rotina.

De forma indireta, Design by Contract estimula um cuidado maior na análise das condições necessárias para, de forma consistente, se ter o funcionamento correto da relação de cooperação clinte-fornecedor. Essas condições são expressas em cada contrato, o qual especifica as obrigações a que cada parte está condicionada e, em contraponto, os benefícios garantidos.

Nesse contexto, o contrato é um veículo de comunicação, por meio do qual os clientes tomam conhecimento das condições de uso, em especial das precondições. É fundamental que as precondições estejam disponíveis para todos os clientes para os quais as rotinas estão disponvéis, pois, sem que isso ocorra, o cliente corre o risco de acionar a rotina fora de suas garantias de funcionamento.

Segundo Bertrand Meyer [19], Design by Contract é um ferramental para análise, projeto, implementação e documentação, facilitando a construção de *softwares* cuja confiabilidade é embutida, no lugar de buscar essa característica por meio de depuração. Meyer utiliza uma expressão de Harlan D. Mills [20] para afirmar que Design by Contract permite construir programas corretos e saber que eles estão corretos.

Com o uso de Design by Contract, cada rotina é levada a realizar o trabalho para o qual foi projetada e fazer isso bem: com corretude, eficiência e genericamente suficiente para ser reusada. Por outro lado, especifica de forma clara o que a rotina não trata. Esse paradigma é coerente, pois para que a rotina realize seu trabalho bem, é esperado que se estabeleça bem as circunstâncias de execução.

Outra característica da aplicação de Design by Contract é que o recurso tem sua lógica concentrada em efetivamente cumprir com sua função principal, deixando para as precondições o encargo de validar as entradas de dados. Essa abordagem é o oposto à ideia de programação defensiva, pois vai de encontro à realização de checagens redundantes. Se os contratos são precisos e explícitos, não há necessidade de testes redundantes [18].

Todos esses aspectos são fundamentais para se possibilitar o reuso eficiente de componentes, que é o pilar da orientação da objetos e se aplica de forma análoga à orientação a serviços. Componentes reusáveis por várias aplicações devem ser robustos pois as consequências de falhas ou comportamentos incorretos são muito piores que as de aplicações que atendem a único propósito [18].

Há de se registrar ainda que, em orientação a objetos, existe outro tipo de asserção além das pré e pós-condições. Em vez de cuidar das propriedades de cada rotina individualmente, elas expressão condições globais para todas as instâncias de uma classe [19]. Esse tipo de asserção é chamada de invariante. Uma vez que em orientação a serviço se preconiza a ausência de estado, o conceito de invariante não é explorado neste trabalho.

2.3.1 Implementações de DbC

Eiffel - A linguagem Eiffel foi desenvolvida em meados dos anos 80 por Bertrand Meyer [17] com o objetivo de criar ferramentas que garantissem mais qualidade aos *softwares*. A ênfase do projeto de Eiffel foi promover reusabilidade, extensibilidade e compatibilidade. Características que só fazem sentido se os programas forem corretos e robustos.

Foi essa preocupação que incorporou à linguagem Eiffel o conceito de contratos. A partir desse estilo de projeto se criou a noção de Design by Contract, concretizada na linguagem por meio das precondições, pós-condições e invariantes [17]. Esta obordagem influenciou outras linguagem de programação orientadas a objeto.

```
1  class interface ACCOUNT create
2      make
3  feature
4      balance: INTEGER
5      ...
6      deposit (sum: INTEGER) is
7              -- Deposit sum into the account.
8          require
9              sum >= 0
10         ensure
11             balance = old balance + sum
12
13 end -- class ACCOUNT
```

Figura 2.1: Exemplo de pré e pós-condições em Eiffel

JML - *Java Modeling Language* é uma extensão da linguagem Java para suporte a especificação comportamental de interfaces, ou seja, controlar o comportamento de classes em tempo de execução. Para realizar essa função, JML possui amplo suporte a Design by Contract. As asserções (precondição, pós-condição e invariantes) são incluídas no código Java na forma de comentários (//@ ou /*@...@*/).

JML combina a praticidade de Design by Contract de linguagens como Eiffel com a expressividade e formalismo de linguagens de especificação orientadas a modelo [14].

```
1  public class IntMathOps {
2
3    /*@ public normal_behavior
4      @ requires y >= 0;
5      @ ensures \result * \result <= y && y < (Math.abs(\result) + 1)
6      @ * (Math.abs(\result) + 1);
7      @*/
8
9    public static int isqrt(int y)
10   {
11     return (int) Math.sqrt(y);
12   }
13 }
```

Figura 2.2: Exemplo de pré e pós-condições em JML

Spec# - é uma extensão da linguagem C#, à qual agrega o suporte para distinguir referência de objetos nulos de referência a objetos possivelmente não nulos, especificações de pré e pós-condições, um método para gerenciar exceções entre outros recursos [4].

```
1  class CircularList {
2
3    // Construct an empty circular list
4    public CircularList()
5      require true;
6      ensure Empty();
7
8    // Return my number of elements
9    public int Size()
10     require true;
11     ensure size = CountElements() && noChange;
12 }
```

Figura 2.3: Exemplo de pré e pós-condições em Spec#

3 NEOIDL: LINGUAGEM PARA ESPECIFICAÇÃO DE CONTRATOS REST

3.1 APRESENTAÇÃO

Nevertheless, these languages do not target to the REST architectural style and lack support for language extensibility. In this paper we present the design and implementation of NeoIDL, an extensible domain specific language and program generator for writing REST based contracts that are further translated into service's implementations.

Besides describing REST contracts in terms of resources, methods, and media types, NeoIDL specifications also include the definition of the data types used in the visible interface of a service. We considered the following requirements when designing NeoIDL. First, the language should be concise and easy to learn and understand. Second, the language should present a well-defined type system and support single inheritance of user defined data types. In addition, developers using NeoIDL should be able to specify concepts related to the *REST architectural style for service-oriented computing* [**?**], in order to simplify the translation of a NeoIDL specification into basic components tailored to that architectural style. Finally, both NeoIDL and the program generator should be extensible. For that reason, we designed NeoIDL to support extensibility through annotations;

3.1.1 Histórico e motivação

NeoIDL has been developed to enable the specification of REST services and to allow the code generation for the implementation of those services for specific platforms. It aims to simplify the development of services, by generating code from a service specification. Figure **??** illustrates the main components of our approach, which consists of: (i) a domain-specific language (NeoIDL) for specifying REST services with their respective contracts; and (ii) a program generator that enables the code generation of REST services in different platforms. The NeoIDL generator is structured as a set of core modules, which are responsible for the parsing, syntax definition, and processing

of NeoIDL specifications; as well as modules for the definition and management of NeoIDL plugins. Each NeoIDL plugin defines specific extensions for the code generator that enables the generation of REST services for different platforms or programming languages.

The current implementation of NeoIDL has been already used to enable the generation of services for the NeoCortex platform and the open-source Python Twisted Framework [**?**]. The NeoCortex platform is a proprietary framework designed by the Brazilian Army that implements a service-oriented architecture based on REST. NeoCortex has been developed using NodeJS—a cross-platform runtime environment for server-side and networking applications. The main reasons for developing NeoCortex, instead of using existing frameworks for SOC, are related to specific needs to deploy services in different platforms, ranging from well structured environments to small devices (such as tablets and mobile phones), as well as different networking environments (including eventually-connected half-duplex radio channels).

The polyglot requirement of NeoCortex motivated us to implement the program generator of NeoIDL as a pluggable architecture (see Figure **??**)—so that we are able to evolve the code generation support in a modular way. For instance, implementing a C++ program generator from NeoIDL specifications does not require any change in the existing code of the program generator. It is only necessary to implement a new NeoIDL plugin.

We have developed nine services that implement operations related to the domain of Command and Control (C2) [**?**]. These services comprehend almost 50 resources and 3000 lines of Python code. Therefore, all these services have been implemented in Python, though other projects have been implemented in Java as well. Here we concentrate our analysis to the services implemented in Python.

Approximately, the number of lines of Python code related to our service repository increases according to the function $sloc = 330 \times numberOfServices$— since, in average, each service requires about 330 lines of Python code (with a standard deviation of 119). It is important to note that services are often implemented as a thin layer on top of existing components that implement reusable tasks or business logic. Accordingly, to understand the impact of NeoIDL accurately, here we do not consider lines of code related to (a) existing tasks and business logic implementations and (b) libraries that might be reused through different services. The boxplot of Figure 3.1 summarizes the

SLOC distribution among the services implementations.

sloc-boxplot.pdf

Figura 3.1: SLOC distribution among services implementations

3.1.2 *Framework*

As shown in Figure **??**, the implementation of NeoIDL consists of a core (split into several Haskell modules) and several plugins, one for each target language (such as Swagger, Python, or Java). The core module includes a tiny application that loads plugins definition and processes the program arguments, which specify the input NeoIDL file, the output directory, and the languages that should be generated code from the input file. Moreover, the core module contains a parser and a type checker for NeoIDL specifications. We have developed the parser for NeoIDL using `BNFConverter` [25], an easy to use parser generator that takes as input a syntax specification and generates code (both abstract syntax and parser) for different languages, including Haskell. Figure 3.2 represents an architectural abstraction of NeoIDL program generator.

In the remaining of this section we present details about the implementation of two NeoIDL Haskell modules: `PluginDef` and `PluginLoader`. The first states the organization of a NeoIDL plugin and the second is responsible for loading all available plugins. The details here are particularly useful for those who want to develop extensible architectures using Haskell.

3.1.2.1 PluginDef component

NeoIDL plugins must comply with a few design rules that `PluginDef` states. `PluginDef` is a Haskell module that declares two data types (`Plugin` and `GeneratedFile`) and a type signature (`Transform = Module -> [GeneratedFile]`) defining a family of

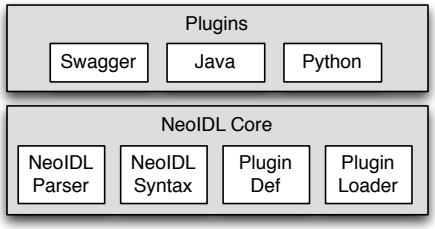

Figura 3.2: Major architectural components of NeoIDL program generator.

functions that map a NeoIDL module into a list of files whose contents are the results of the transformation process.

According to these design rules, each NeoIDL plugin must declare an instance of the `Plugin` data type and implement functions according to the `Transform` type signature. Moreover, the `Plugin` instance must be named as `plugin`, so that the `PluginLoader` component will be able to obtain the necessary data for executing a given plugin. Indeed, the execution of a plugin consists of applying the respective `transformation` function for a NeoIDL module, producing as result a list of files that consists of a name and a `Doc` as file content.[1]

As an alternative, we could have implemented a Haskell type class [**?**] exposing operations for obtaining the necessary data for a given plugin. Although this approach might seem more natural for specifying design rules for a pluggable architecture in Haskell, in the end it would lead to a cumbersome approach to our problem. The main reason for discarding this alternative approach was the need to (a) implement a data type, (b) make this data type an instance of the mentioned type class, and (c) create an instance of that data type. All those steps would be necessary for each plugin. Using our approach, the obligation of a plugin developer is just to provide an instance of the `Plugin` datatype, taking into account the name convention we mentioned above. The `language` attribute of the `Plugin` datatype is used for UI purpose only, so that the users will be able to obtain the list of available plugins and select which plugins will be used during a program generation.

3.1.2.2 PluginLoader component

Based on the design rules discussed in the previous section, the `PluginLoader` component is able to dynamically load the available NeoIDL plugins. This is a Haskell module (see Figure **??**) that exposes the `loadPlugins` function, which returns a list with all available plugins. This list is obtained by compiling the Haskell plugin modules during the program execution and dynamically evaluating an expression that yields a list of `Plugin` datatype instances.

We assume that all Haskell modules within the top level `Plugins` directory must have a plugin definition, according to the design rules of Section 3.1.2.1. In Figure **??**, the

[1] The `Doc` data type comes from the John Hughes Pretty Printer library.

`loadPlugins` function lists all files within the `Plugins` directory, filters the Haskell files (files with the ‘‘`.hs`’’ extension), creates a qualified name to these files, and applies the `compile` function to the resulting list of qualified names. In the next step, the `compile` function uses the GHC API [**?**] for compiling the Haskell modules with plugin definitions and to evaluate an expression that produces a list with the available plugins.

Our dynamic approach for loading plugins relies on the GHC API, using a specific idiom to compile Haskell modules and execute expressions. Figure **??** shows that idiom in the definition of the `compile` function, although we omit some boilerplate code that is necessary to compile Haskell modules using the GHC API. The last four lines of `compile` are specific to the program generator of NeoIDL. First, we build a string representation of a Haskell `list` comprising all instances of the `Plugin` datatype, obtained from the different NeoIDL plugins. Then, we evaluate this string representation of a plugin list using the *meta-programming* ability of the `compileExpr` function, which is available in the GHC API. Thus, `compileExpr` dynamically evaluates a string representation of an expression, which leads to a value that could be used by other functions of a program. The call to `compileExpr` also checks the design rule that requires (a) a `plugin` definition within all NeoIDL plugins; and (b) that definition must be an instance of the `Plugin` data type. In the cases where a plugin (exposed as a Haskell module on the top-level Plugins directory) does not comply with this design rule, a runtime error occurs. Accordingly, we use the default error handler of GHC API to report problems when loading a plugin. This is a new approach of using the GHC API to dynamically check Haskell modules in pluggable architectures.

As an example of design rule violation in the NeoIDL architecture, if there is no `plugin` definition within a plugin, the following error is reported at runtime:

```
$./neoIDL
neoIDL: panic! (the 'impossible' happened)
  (GHC version 7.6.3 for x86_64-darwin):
    Not in scope: ‘Plugins.Python.plugin’
```

In the last line of the above interactive section, NeoIDL program generator reports that `plugin` is not defined within the Python plugin module. In a similar way, if the `plugin` definition is available but it is not an instance of the `Plugin` data type, a type

error occurs during the execution of the NeoIDL program (see the reported message bellow, using a `plugin` definition assigned to a String value).

```
$./neoIDL
neoIDL: panic! (the 'impossible' happened)
(GHC version 7.6.3 for x86_64-darwin):
 Couldn't match expected type 'Plugin'
  with actual type '[GHC.Types.Char]'
```

Therefore, we are using GHC API to type check Haskell modules (note that Haskell is a statically typed language) during the loading phase of a NeoIDL execution. Next section presents the assessment of NeoIDL considering different goals.

3.1.3 Linguagem

NeoIDL simplifies service specifications by means of (a) mechanisms for modularizing and inheriting user defined data types, and (b) a concise syntax that is quite similar to the *interface description languages* of Apache Thrift or CORBA. A NeoIDL specification might be split into modules, where each module contains several definitions. In essence, a NeoIDL definition might be either a data type (using the `entity` construct) or a service describing operations that might be reached by a given pair (URI, HTTP method). Figures 3.3 and 3.4 present two NeoIDL modules: (i) the data-oriented `MessageData` module; and the service-oriented `Message` module.

The `MessageData` module (Figure 3.3) declares an enumeration (`MessageType`), which states the two valid types of messages (a message must be either a *message sent* or a *message received*); and a data type (`Message`), which details the expected structure of a message. We use a *convention over configuration approach*, assuming that all attributes of a user defined data type are mandatory, though it is possible to specify an attribute as being optional using the syntax `<Type> <Ident> = 0;`, as exemplified by the `subject` field of the `Message` data type.

The `Message` module of Figure 3.4 specifies one service resource (`sentbox`). As explained, we send requests for the methods of a given resource using a specific path. In the example, the `sentbox` resource's methods are available from the relative path

```
module MessageData {
    enum MessageType { Received, Sent };

    entity Message {
      string id;
      string from;
      string to;
      string subject = 0;
      string content;
      MessageType type;
    };
}
```

Figura 3.3: Message data type specified in NeoIDL

`/messages/sent`. This resource declares two operations: one `POST` method that might be used for sending messages and one `GET` method that might be used for listing all messages sent from a given sequential number.

Also according to our *convention over configuration* approach, we assume that the arguments of `POST` and `PUT` operations are sent in the request body, whereas arguments of `GET` operations are either sent enclosed with the request URL or enclosed with the URL path (in a similar way as `DELETE` operations). We are able to change these conventions by using specific annotations attached to an operation parameter. In these examples, conventions are used to reduce the size of services' specifications.

```
module  Message {
 import MessageData;

 resource sentbox {
  path = "/messages/sent";
  @post void sendMessage(Message message);
  @get [Message] listMessages(string seq);
 };
};
```

Figura 3.4: Sent message service specification in NeoIDL

To support language extensibility, NeoIDL specifications can be augmented through annotations. The main reason for introducing annotations in NeoIDL was the possibility to extend the semantics of a specification without the need to change the concrete

syntax of NeoIDL. For instance, suppose that we want to express security policies for a service resource. A developer could change the concrete syntax of NeoIDL for this purpose, defining new language constructs for specifying the authentication method (based on tokens or user passwords), the cryptographic algorithm used in the resource request and response, and the role-based permissions to the resource capabilities. However, changing the concrete syntax to allow the specification of unanticipated properties of a resource often breaks the code of the program generator.

Instead, using annotations, developers might extend the language within NeoIDL specifications. Therefore, apart from the NeoIDL definitions discussed before, it is also possible to define new annotations that might be attached to the fundamental constructs of NeoIDL (i.e. `module`, `enum`, `entity`, and `resource`). Each annotation consists of a name, a target element that indicates the NeoIDL constructs the annotation might be attached to, and a list of properties. When transforming a specification, the list of annotations attached to a NeoIDL element is available to the plugins, which could consider the additional semantics during the program generation. Figure 3.5 presents a NeoIDL example that attaches an user defined annotation (`SecurityPolicy`) to specify security policies on the `agent` resource. In the example, using the `SecurityPolicy` annotation we specify that the operations of the `agent` resource (i) must use a basic authentication mechanism, (ii) the arguments and return values must be encoded using the AES algorithm, and (iii) only authenticated users having the *admin* role are authorized to request the resources.

In the next sections we present the lexical and syntactic structures of NeoIDL, considering the revision of August 23, 2015. Both sections were (almost) automatically generated by the BNF-Converter [**?**] parser generator.

3.1.3.1 The lexical structure of NeoIDL

1. Identifiers

   Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded.

2. Literals

   String literals ⟨*String*⟩ have the form ‘‘x‘‘, where x is any sequence of any characters except ‘‘ unless preceded by

```
1  module Agente {
2
3   entity Agent {
4      ...
5   };
6
7   annotation SecurityPolicy for resource {
8      string method;
9      string algorithm;
10     string role;
11  };
12
13  @SecurityPolicy(method = "basic",
14                  algorithm="AES",
15                  role = "admin");
16  resource agent {
17     path = "/agent";
18     @post void persistAgent(Agent agent);
19  };
20 };
```

Figura 3.5: NeoIDL specification using annotations

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + '.'⟨*digit*⟩ + ('e'-?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

3. Reserved words and symbols The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

   The reserved words used in NeoIDL are the following:

`annotation`	`call`	`entity`
`enum`	`extends`	`float`
`for`	`import`	`int`
`module`	`path`	`resource`
`string`		

   The symbols used in NeoIDL are the following:

`{`	`}`	`;`
`=`	`.`	`@`
`(`	`)`	`0`
`==`	`<>`	`>`
`>=`	`<`	`<=`
`[`	`]`	`@get`
`@post`	`@put`	`@delete`
`/@require`	`/@ensure`	`/@invariant`
`/@otherwise`	`/**`	`*/`
`*`	`@desc`	`@param`
`@consume`	`,`	

3.1.3.2 The syntactic structure of NeoIDL

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the following BNF notation. All other symbols are terminals.

⟨*Modulo*⟩ ::= `module` ⟨*Ident*⟩ {
⟨*ListImport*⟩
⟨*MPath*⟩
⟨*ListEnum*⟩
⟨*ListEntity*⟩
⟨*ListResource*⟩
⟨*ListDecAnnotation*⟩
}

⟨*Import*⟩ ::= `import` ⟨*NImport*⟩ `;`

⟨*MPath*⟩ ::= ϵ
| `path` = ⟨*String*⟩ `;`

⟨*NImport*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `.` ⟨*NImport*⟩

⟨*Entity*⟩ ::= ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ { ⟨*ListProperty*⟩ } `;`
| ⟨*ListDefAnnotation*⟩ `entity` ⟨*Ident*⟩ `extends` ⟨*Ident*⟩ { ⟨*ListProperty*⟩ } `;`

⟨*Enum*⟩ ::= `enum` ⟨*Ident*⟩ { ⟨*ListValue*⟩ } `;`

⟨*DecAnnotation*⟩ ::= `annotation` ⟨*Ident*⟩ `for` ⟨*AnnotationType*⟩ { ⟨*ListProperty*⟩ } `;`

⟨*DefAnnotation*⟩ ::= `@` ⟨*Ident*⟩ `(` ⟨*ListAssignment*⟩ `)` `;`

⟨*Parameter*⟩ ::= ⟨*Type*⟩ ⟨*Ident*⟩ ⟨*Modifier*⟩

⟨*Assignment*⟩ ::= ⟨*Ident*⟩ = ⟨*Value*⟩

⟨*Modifier*⟩ ::= ϵ
| = `0`

⟨*AnnotationType*⟩ ::= `resource` | `enum` | `entity` | `module`

⟨*Resource*⟩ ::= ⟨*ListDefAnnotation*⟩ `resource` ⟨*Ident*⟩ { `path` = ⟨*String*⟩ `;` ⟨*ListCapacity*⟩ } `;`

⟨*Capacity*⟩ ::= ⟨*NeoDoc*⟩ ⟨*ListDefNAnnotation*⟩ ⟨*Method*⟩ ⟨*Type*⟩ ⟨*Ident*⟩ `(` ⟨*ListParameter*⟩ `)` `;`

⟨*Method*⟩ ::= `@get` | `@post` | `@put` | `@delete`

3.2 AVALIAÇÃO EMPÍRICA

3.2.1 Expressividade

In Section **??** we presented the comparison in SLOC between the contracts specification in NeoIDL and the resultant code generated by the framework. Another relevant issue is a comparison of expressiveness of contracts written in NeoIDL and other popular languages with the same purpose (specifying contracts for REST services). In this context, we compared NeoIDL specifications with Swagger specifications, a language that has been increasily used by the industry. Swagger [**?**] contracts may by written in JSON and Yaml, both based on key-value structure.

In a collaborative work with the Brazilian Army, we obtained a portion of their contracts' specifications in Swagger (44 in total), specified with version 1.2. Our first step was to rewrite these specifications in NeoIDL and thus compare the total number of lines of code (which might serve as a metric of expressiveness). The 44 contracts

in Swagger amount to 13921 lines of specification, while the same set of contracts in NeoIDL comprises 5140 lines of specification. Thus, the average reduction was about 63%. In others words, it means that 10 lines of structured Swagger specification require about 4 lines of NeoIDL specification. In this analysis we only considered *physical lines of code*, ignoring blank lines and lines consisting of delimiters only. The Appendix A shows a sample contract we analised.

The reduction in number of lines is not the same in all contracts. For instance, a given service[2] required 367 lines of Swagger specification and 112 lines of NeoIDL specification. This case represents a reduction of about 69%. On the other hand, another service contract required 81 lines of specification in Swagger and 42 lines of NeoIDL specification. In this case, the SLOC decrease was slightly less than 50%.

The size of the original contract has only a small influence in the observed expressiveness. Therefore, we cannot assume that *the bigger the contract is in Swagger the bigger is the improvement (with respect to the smaller specification size) of NeoIDL*. We also realized that the use of a more descritive documentation, the number of entities, and the number of capacities do not correlate to the advantageous reduction of lines of code during a transformation of Swagger specifications into NeoIDL specifications. Therefore, it seems that the benefits do not relate to the size of the original specifications. Table 3.1 presents the correlation between the improvement of expressiveness (measured as the percentage of reduction obtained after transforming Swagger specifications into NeoIDL specifications) and some metrics related to the size of the original Swagger specifications.

Tabela 3.1: Correlation of the Expressiveness Improvement with the size of the Swagger specifications

Metric	Pearson's correlation	*p-value*
LOC of Swagger specification	0.19	0.20
Number of services	0.14	0.35
Number of capacities	0.14	0.34
Number of entities	0.20	0.18

[2]For confidentiality reasons, the real names of contracts were omitted.

3.2.2 Potencial de reuso

Similar to NeoIDL, Swagger presents some mechanisms to reuse user defined structures. Nevertheless, this feature is almost ignored in the set of contracts we analysed, which leads to the duplication of entities' definition across different Swagger specifications. This might have occurred either due to the nonintuitive construct for reusing definitions in Swagger (based on references to JSON files) and the difficulties to identify that one entity had already been specified in another contract. After analysing the 44 Swagger specifications, we realized that 40 entities have been specified in more than one contract. Actually, one specific entity is present in 12 distinct Swagger contracts.

3.3 EXTENSÃO DA NEOIDL PARA DESIGN BY CONTRACT

Influência de Eiffel, JML e Spec#

Eiffel assertions are Boolean expressions, with a few extensions such as the old notation. Since the whole power of Boolean expressions is available, they may include function calls. Because the full power of the language is available to write these functions, the conditions they express can be quite sophisticated. [18]

3.3.1 Proposta: Serviços com Desing-by-Contract

Os benefícios esperados pela adoção da arquitetura orientada a serviços somente serão auferidos com a concepção adequada de cada serviço. Por essa razão, é necessário planejar o projeto dos serviços criteriosamente antes de lançar mão do desenvolvimento, com preocupação especial em garantir um nível aceitável de estabilidade aos consumidores de cada serviço. Nessa etapa do projeto de desenho da solução, a especificação do contrato do serviço (Web API) exerce uma função fundamental.

Na sociedade civil, contratos são meios de se formalizar acordo entre partes a fim de definir os direitos e deveres de cada parte e buscar atingir o objetivo esperado dentro de determinadas regras. Cada parte espera que as outras cumpram com suas obrigações. Por outro lado, sabe-se que o descumprimeto das obrigações costuma implicar de penalizações até o desfazimento do contrato.

Contratos entre serviços Web seguem em uma linha análoga. O desenho das capacidades (operações) e dos dados das mensagens correspondem aos termos do contrato no sentido do que o consumidor deve esperar do serviço provedor. Porém identificou-se, após ampla pesquisa realizada sobre o tema, que as linguagens disponíves para especificação de contratos atingem apenas esse nível de garantias. No contexto de web-services em REST, conforme descrito na seção 2.2.2, há ainda a ausência de padrão para especificação contratos, tal como ocorre com o WSDL adotado em SOAP.

A proposta deste trabalho é extender os níveis de garantias, de modo a promover um patamar adicional com obrigações mutuas entre os serviços (consumidor e provedor). Isso se dá para adoção do conceito de Design-by-Contract (debatido na seção 2.3) em que a execução da capacidade do serviço garantirá a execução, desde que satisfeitas as condições prévias. O detalhamento do processo é exposto nas seções que se seguem.

3.3.1.1 Modelo de operação

As garantias para execução dos serviços são estabelecidas em duas etapas: pré- e pós-condições. Nas pré-condições o provedor do serviço estabelece os requisitos para que o serviço possa ser executado. A etapa de pós-condições tem o papel de validar se a mensagem de retorno do serviço possui resultados válidos.

O diagrama da Figura 3.6 descreve como ocorre a operação das pré- e pós-condições. O processo se inicia com a chamada à capacidade do serviço e a identificação da existência de uma pré-condição. Caso tenham sido estabelecidas pré-condições, essas são avaliadas. Caso alguma delas não tenham sido satisfeitas, o serviço principal não é processado e o provedor do serviço retornar o código de falha definido no contrato correspondente.

Caso tenham sido definidas pós-condições, essas são acionadas após o processamento da capacidade, porém antes do retorno ao consumidor do serviço. Assim, conforme Figura 3.6, visando não entregar ao cliente uma mensagem ou situação incoerente, as pós-condições são validadas. Caso todas as pós-condições tenham sido satisfeitas, a mensagem de retorno é encaminhada ao cliente. Caso contrário, será retornado o código de falha.

3.3.1.2 Verificação das pré-condições

As pré-condições podem ser do tipo baseado nos parâmetros da requisição ou do tipo baseado na chamada a outro serviço. Denominamos, para o contexto desta dissertação,

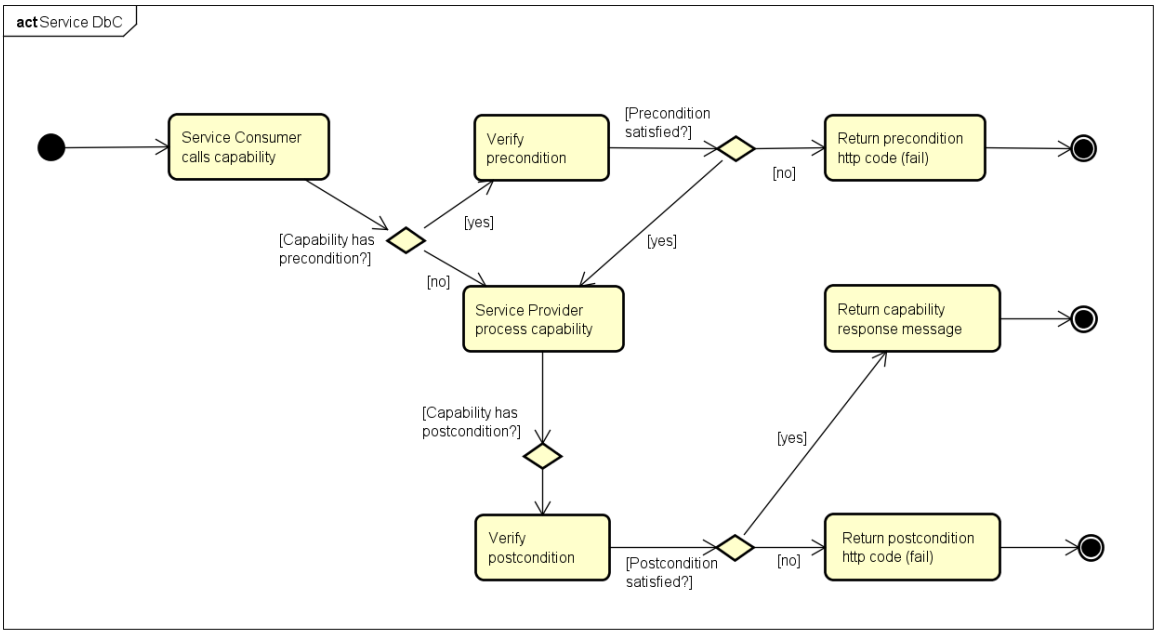

Figura 3.6: Digrama de atividades com verificação de pré e pós condições

de básica a pré-condição baseada apenas nos parâmetros da requisição (atributos da chamada ao serviço). Nessa validação é direta, comparando os valores passados com os valores admitidos.

No caso das pré-condições baseadas em serviços, é realizada chamada a outro serviço para verificar se uma determinada condição é satisfeita. Este modo de funcionamento, que se assemelha a uma composição de serviço, é mais versátil, pois permite validações de condições complexas sem que a lógica associada seja conhecida pelo cliente. Assim, os contratos que estabelecem esse tipo de pré-condição se mantem simples.

A Figura 3.7 detalha as etapas de verificação de cada pré-condição. Nota-se que a saída para as situações de desatendimento às pré-condições, independentemente do tipo, é o mesmo. O objetivo desta abordagem é simplificar o tratametno de exceção no consumidor.

3.3.1.3 Verificação das pós-condições

A verificação das pós-condições acontece de modo muito similar a das pré-condições. Há também os dois tipos, baseado em valores e em chamadas a outros serviços. O diferencial está em que a validação dos valores passa a ocorrer a partir dos valores contidos na mensagem de retorno. A Figura 3.8 descreve as etapas necessárias para

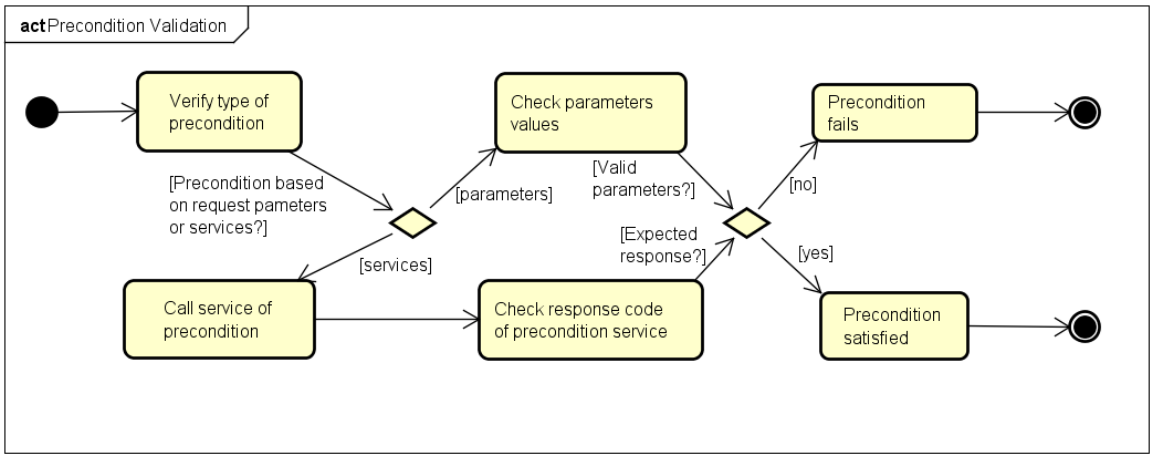

Figura 3.7: Diagrama de atividades do processamento da pré-condição

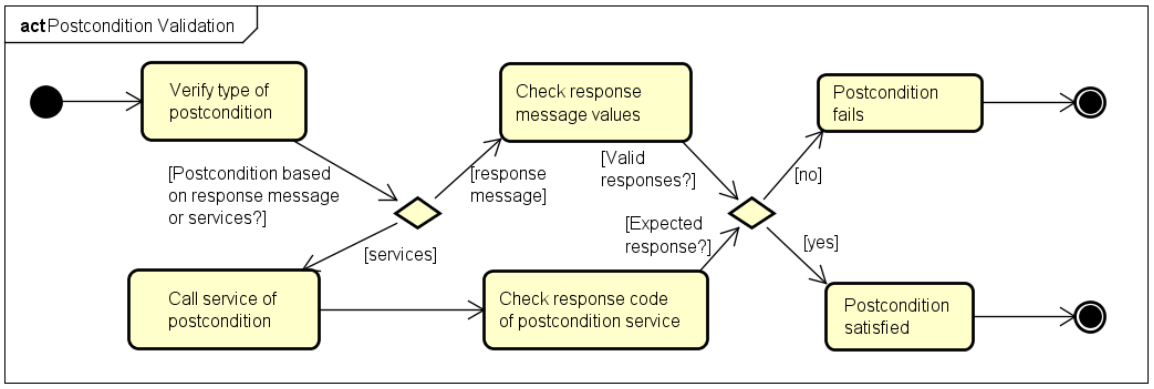

Figura 3.8: Diagrama de atividades do processamento da pós-condição

validação de cada pré-condição.

3.3.2 Extensão da linguagem

3.3.2.1 Precondição básica

3.3.2.2 Pós-condição básica

...

```
1 module Catalogo {
2     service Catalogo {
3     path = "/catalogo";
4
5     /@require old.descricao != null
6     /@otherwise HTTP_Precondition_Failed
7     @post Catalogo incluirItem (string id, string descricao, float
          valor);
8     (...)
9 }
```

Figura 3.9: Exemplo da notação DBC básica na NeoIDL

```
1 module Catalogo {
2     (...)
3     service Catalogo {
4     path = "/catalogo";
5     /@require call Catalogo.pesquisarItem(old.id)==HTTP_OK
6     /@ensure call Catalogo.pesquisarItem(old.id)==HTTP_Not_Found
7     /@otherwise HTTP_Not_Found
8     @delete Atividade excluiItem(string id);
9     (...)
10 }
```

Figura 3.10: Exemplo da notação DBC na NeoIDL com chamada a serviço

3.3.2.3 Precondição com chamada a serviço

3.3.2.4 Pós-condição com chamada a serviço

...

3.3.3 Estudo de caso: plugin twisted

...

3.3.3.1 Arquitetura

3.3.3.2 Geração de código

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Swagger - the world's most popular framework for apis. http://swagger.io/. Swagger project oficial site.

[2] World wide web consortium (w3c) - web services description language. https://www.w3.org/TR/wsdl. WSDL W3C oficial site.

[3] Alonso, G., Casati, F., Kuno, H., e Machiraju, V. *Web services*. Springer, 2004.

[4] Barnett, M., Leino, K. R. M., e Schulte, W. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2004.

[5] Chen, H.-M. Towards service engineering: service orientation and business-it alignment. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pages 114–114. IEEE, 2008.

[6] Erl, T. *SOA design patterns*. Pearson Education, 2008.

[7] Erl, T. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[8] Erl, T., Karmarkar, A., Walmsley, P., Haas, H., Yalcinalp, L. U., Liu, K., Orchard, D., Tost, A., e Pasley, J. *Web service contract design and versioning for SOA*. Prentice Hall, 2009.

[9] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. Tese de Doutorado, University of California, Irvine, 2000.

[10] Hadley, M. J. Web application description language (wadl). 2006.

[11] He, H. What is service-oriented architecture. *Publicação eletrônica em 30/09/2003*, 2003.

[12] Jazequel, J.-M. e Meyer, B. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[13] Karthikeyan, T. e Geetha, J. Contract first design: The best approach to design of web services. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 5:338–339, 2014.

[14] Leavens, G. T. e Cheon, Y. Design by contract with jml, 2006.

[15] Leymann, F. Combining web services and the grid: Towards adaptive enterprise applications. In *CAiSE Workshops (2)*, pages 9–21, 2005.

[16] Lima, L., Bonifácio, R., Canedo, E., Castro, T. M.de , Fernandes, R., Palmeira, A., e Kulesza, U. Neoidl: A domain specific language for specifying rest contracts detailed design and extended evaluation. *International Journal of Software Engineering and Knowledge Engineering*, 25(09n10):1653–1675, 2015.

[17] Meyer, B. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[18] Meyer, B. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[19] Meyer, B. *Object-oriented software construction*, volume 2. Prentice hall New York, 1997.

[20] Mills, H. D. The new math of computer programming. *Communications of the ACM*, 18(1):43–48, 1975.

[21] Mumbaikar, S., Padiya, P., e others,. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5).

[22] Papazoglou, M. P., Traverso, P., Dustdar, S., e Leymann, F. Service-oriented computing: State of the art and research challenges. *Computer*, (11):38–45, 2007.

[23] Papazoglou, M. P., Traverso, P., Dustdar, S., e Leymann, F. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[24] Papazoglou, M. P. e Van Den Heuvel, W.-J. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[25] Ranta, A. *Implementing Programming Languages. An Introduction to Compilers and Interpreters*. Texts in computing. College Publications, 2012.

[26] Rao, J. e Su, X. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2004.

[27] Serrano, N., Hernantes, J., e Gallardo, G. Service-oriented architecture and legacy systems. *Software, IEEE*, 31(5):15–19, 2014.

[28] Wideberg, R. Restful services in an enterprise environment - a comparative case study of specification formats and hateoas. Dissertação de Mestrado, Royal Institute of Technology, Stockholm, Sweden, 2015.

APÊNDICES

A CONTRATO NEOIDL COM DBC

...