Contratos REST robustos e leves: uma abordagem em Design by Contract com NeoIDL

Lucas F. Lima[1], **Rodrigo Bonifácio de Almeida**[2], **Edna Dias Canedo**[1]

[1]Departamento de Engenharia Elétrica – Universidade de Brasília – UnB
CEP 70910-900 – Campus Darcy Ribeiro – Asa Norte – Brasília – DF – Brasil

[2]Departamento de Ciência da Computação – Universidade de Brasília – UnB
CEP 70910-900 – Campus Darcy Ribeiro – Asa Norte – Brasília – DF – Brasil

lucas.lima@aluno.unb.br, rbonifacio@cic.unb.br, ednacanedo@unb.br

Abstract. *The research, summarized in this paper, aims strengthen contract specification for solutions based on service-oriented computing. Robustness is fetched adding design by contract to neoidl language , so it may specify contracts with guarantees. The proposal will be submitted to Empirical validation by establishing rules of transformation into a new plugin neoidl .*

Resumo. *O trabalho de pesquisa de mestrado, sumarizado neste artigo, objetiva fortalecer a especificação de contratos para soluções de baseadas em computação orientada a serviços. A robustez é buscada com construções que agregam design by contract à linguagem NeoIDL , de modo que sejam especificados contratos com garantias. A proposta será submetida a validação empírica, verificando regras de transformações em um novo plugin para a NeoIDL .*

1. Introdução e Caracterização do Problema

O desenvolvimento de soluções de software para suportar as atividades das organizações tem como importante desafio o alinhamento às rápidas mudanças nos processos de negócio, sendo esse um dos objetivos da computação orientada a serviços (*Service-oriented computing, SOC)* – a qual tem se mostrado uma solução de *design* que favorece o alinhamento a mudanças constantes e urgentes [Chen 2008].

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 definidos e independentes da implementação.

Por outro lado, as linguagens de especificação de contratos para SOA apresentam algumas limitações. Por exemplo, a linguagem WSDL (*Web-services description language*)[Zur Muehlen et al. 2005] 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. 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*) que vem se tornando uma tendência para o desenvolvimento de soluções para SOC. Outras

alternativas para REST, como Swagger e RAML[1], usam linguagens de propósito geral (em particular JSON) adaptadas para especificação de contratos. Mesmo levando a 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.

Tentando 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 IDLs (*interface description languages*) 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 de comportamento como os presentes em linguagens que suportam DBC (*Design by Contract*) [Meyer 1992], como JML, Spec# e Eiffel. Dessa forma, os objetivos dessa pesquisa envolve:

- Investigar o uso de construções DBC no contexto da computação orientada a serviços e conduzir uma revisão sistemática da literatura para identificar os principais trabalhos que lidam com a relação entre DBC e SOC.
- Especificar e implementar novas construções para a linguagem NeoIDL , de tal forma que seja possível especificar contratos mais precisos; além de definir regras de transformação das novas construções NeoIDL para diferentes tecnologias (como Twisted) que suportam a implementação de serviços em REST.
- Conduzir uma validação empírica da proposta usando uma abordagem mais exploratória, possivelmente usando a estratégia *pesquisa-ação*.

2. Fundamentação Teórica

2.1. SOC - Computação Orientada a Serviço

SOC (ou SOA) é um estilo arquitetural cujo objetivo é prover baixo acoplamento por meio da iteração entre agentes de software, chamados de serviços [He 2003]. A chave para que a solução baseada em SOC tenha custo-benefício favorável é o reuso, o qual somente é possível se os serviços possuírem interfaces ubíquas, com semânticas genéricas e disponíveis para seus consumidores.

A comunicação com os serviços, e entre eles, é feita por meio da troca de mensagens, em geral, com uso de *webservices*, que são padrões abertos de comunicação que atuam sobre o protocolo http. Os padrões mais utilizados são o SOAP [Box et al. 2000] e REST [Fielding 2000], este cada vez mais utilizado pela industria.

[Erl 2008] descreveu oito princípios para desenvolvimento SOA. Dentre eles, há especial interesse, no contexto do presente trabalho, no princípio *contrato padronizado* (*Standardized Service Contract*). Serviços de um mesmo inventário devem seguir os mesmos padrões de *design*, de modo a prover o reuso e composição. Este princípio prega a abordagem *Contract First*, em que a concepção do serviço parte da especificação do contrato e não com a geração do contrato a partir código.

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

Por fim, em razão da popularização dos dispositivos móveis, um tipo de serviço tem ganhado relavância: *micro-service*. O microsserviço consiste de um serviço atômico, especializado e que requer baixo processamento do consumidor.

2.2. *Design by Contract*

Design by Contract [Meyer 1992] – DBC – é um conceito aplicado à utilização de módulos de *software*, no qual consumidor e fornecedor firmam entre si garantias. De um lado o consumidor deve garantir que, antes da chamada a um módulo, os parâmetros de entrada devem ser respeitados (denominadas de pré-condições). Do outro lado o fornecedor deve garantir, se respeitadas as pré-condições, as propriedades relacionadas ao sucesso da execução (pós-condições).

DBC tem o objetivo de aumentar a robustez do sistema e tem na linguagem Eiffel [Meyer 1991] 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, segundo [Software 2015], é uma abordagem sistemática que produz sistemas sem erros.

2.3. NeoIDL

A NeoIDL [Bonifácio 2015] é uma linguagem para especificação de serviços SOA. NeoIDL simplifica a especificação provendo modularidade e herança de tipos de dados customizados além de uma sintaxe concisa. O *framework* NeoIDL recebe um conjunto de módulos onde são especificados os tipos de dados e as capacidades dos serviços. A Figura 1 apresenta um exemplo de um módulo simples com a uma estrutura chamada itemCatalogo e duas capacidades: atualizarItem e pesquisarItem.

```
1  module Catalogo {
2    path = "/catalogo";
3
4    struct ItemCatalogo {
5      string id;
6      string descricao;
7      string produto;
8      float valor;
9    };
10
11   service Catalogo {
12     path = "/catalogo";
13     @post Catalogo atualizarItem(string id, ItemCatalogo
           itemCatalogo);
14     @get Catalogo pesquisarItem(string id);
15   };
16
17 }
```

Figure 1. Exemplo de um módulo escrito em NeoIDL

O *framework* NeoIDL processa um módulo escrito na linguagem NeoIDL e produz o código com a estrutura para implementação dos serviços descritos, conforme ilustra

a Figura 2. A versão atual da NeoIDL suporta geração de códigos em Java, Python ou Swagger. Entretanto, o *framework* é extensível por meio da implementação de novos plugins.

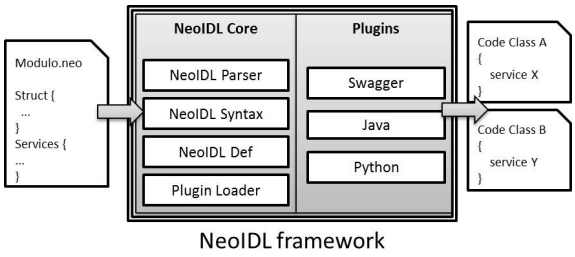

Figure 2. Entradas (lado esquerdo) e saídas (lado direito) do *framework* NeoIDL

3. Exemplo de DBC para SOC

Nesta seção apresentamos dois exemplos do uso de DBC para SOC. O primeiro caso, mais simples, consiste em estabelecer um contrato que restringe o argumento passado para o serviço. O segundo exemplo é mais elaborado, no qual se estabelece como pré-condição o resultado positivo na chamada de um outro serviço. Satisfeita a pré-condição, o serviço principal garante, por meio de outra chamada, que a necessidade do cliente é atendida.

3.1. Exemplo simples

O trecho de módulo NeoIDL constante da figura 3 destaca a capacidade *incluirItem* (linha 8). Para que se tenha o item incluído no catálogo, é necessário fornecer um valor para o atributo descrição (obrigatório). A pré-condição (linha 6) se inicia com a notação */@pre* e possui validação sobre o atributo descrição. Note que o atributo é precedido da palavra *old*, que indica o valor do atributo antes do processamento do serviço. De forma inversa, o prefixo *new* indicaria o valor do atributo após a execução do serviço. É estabelecido ainda a cláusula */@otherwise* na linha 7. Sua função é informar ao cliente do serviço a falha no atendimento da pré-condição, retornando o código HTTP 412 (Falha de pré-condição).

```
1  module Catalogo {
2      (...)
3      service Catalogo {
4      path = "/catalogo";
5
6      /@pre old.descricao != null
7      /@otherwise HTTP_Precondition_Failed
8      @post Catalogo incluirItem (string id, string descricao,
           string produto, float valor);
9
10     (...)
11 }
```

Figure 3. Exemplo da notação DBC no NeoIDL

3.2. Exemplo de DBC com chamada a serviço

A figura 4 apresenta a capacidade *excluirItem* (linha 8). Nesse caso são estabelecidas a pré e a pós condições, ambas com chamadas ao serviço *Catalogo.pesquisarItem*. Antes do processamento da capacidade *excluirItem*, é verificado se o item exite. Caso exista, a pré-condição é satisfeita e o item é excluído. A pós-condição (linha 6) confirma que o item não consta mais do catálogo. Se a pré-condição não for satisfeita, a cláusula */@otherwise* (linha 7) informa ao cliente do serviço que o item não foi localizado (código HTTP 404 - Objeto não encontrado).

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

Figure 4. Exemplo da notação DBC no NeoIDL com chamada a serviço

4. Trabalhos Relacionados

Embora o primeiro princípio SOA estabelecido por [Erl 2008] esteja relacionado a definição do contrato, mais especificamente à abordagem *Contract First*, não se identificou, durante a pesquisa bibliográfica, nenhuma publicação que associasse o conceito de *Design by Contract* à especificação de contratos SOA.

Entretanto verifica-se que, embora definido já há alguns anos, DBC continua sendo objeto de pesquisa [Poyias 2014], [Rubio-Medrano et al. 2013], [Belhaouari et al. 2012]. Nesse dois últimos casos, o estudo de caso está associado a controle de acesso, cenários totalmente aderentes a que se pretende atingir com esta pesquisa.

5. Estado Atual do Trabalho

Estamos refinando a sintaxe das pré-condições e pós-condições na linguagem da NeoIDL, de forma a agregar à especificação mecanismos para verificação das condições de execução e das garantias dos serviços. Além da validação dos parâmetros de entrada e de saída típicos do DBC, conforme descrito na seção 3, nossa abordagem permitirá a inclusão de chamadas a serviços REST como pré e pós condições.

Um novo plugin, com suporte a DBC, está sendo implementado no NeoIDL *framework*. Uma vez que todos os serviços gerados seguirão o paradigma REST, adotamos a tecnologia *Python Twisted*, projetada para tratar os protocolos de rede e flexível para permitir manipulação dos códigos padrão HTTP (Forbiden, Not Found, Ok, etc), informando, assim, sucesso ou insucesso nas pré e pós-condições.

Para permitir essas chamadas a serviços, o plugin deverá ser capaz de gerar o código correspondente, algo que ainda não é possível com os plugins existentes na

NeoIDL . A ideia é que essas chamadas se assemelhem a composição de serviços. Há de se considerar, contudo, que está consolidado na indústria a implementação de composição de serviços com o produto *Enterprise Service Bus* – ESB [Schmidt et al. 2005]. Entretanto, o uso de ESB para *microservices* não é conveniente, em razão da granularidade dos serviços.

Em termos de estudo de caso, o presente trabalho aborda o cenário de catálogo de serviços. O uso de DBC será utilizado para verificação das controle de acesso na inclusão e alterações de itens no catálogo.

References

Belhaouari, H., Konopacki, P., Laleau, R., and Frappier, M. (2012). A design by contract approach to verify access control policies. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 263–272. IEEE.

Bonifácio, R. (2015). Neoidl a generative framework for service-oriented computing. *SEKE*, page 10.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple object access protocol (soap) 1.1.

Chen, H.-M. (2008). 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.

Erl, T. (2008). *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River.

Fielding, R. (2000). Fielding dissertation: Chapter 5: Representational state transfer (rest).

He, H. (2003). What is service-oriented architecture. *Publicação eletrônica em*, 30:50.

Meyer, B. (1991). *Eiffel: The Language*, volume 1. Prentice Hall.

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

Poyias, K. (2014). *Design-by-contract for software architectures*. PhD thesis, Department of Computer Science.

Rubio-Medrano, C. E., Ahn, G.-J., and Sohr, K. (2013). Verifying access control properties with design by contract: Framework and lessons learned. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 21–26. IEEE.

Schmidt, M.-T., Hutchison, B., Lambros, P., and Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797.

Software, E. (2012 (accessed June 6, 2015)). *Building bug-free O-O software: An introduction to Design by Contract(TM)*.

Zur Muehlen, M., Nickerson, J. V., and Swenson, K. D. (2005). Developing web services choreography standards—the case of rest vs. soap. *Decision Support Systems*, 40(1):9–29.