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

**Lucas F. Lima**[1]
**Orientadores: 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

Nível: Mestrado
Ano de Ingresso: 2014
Previsão de conclusão: Dezembro/2015
Aprovação da Proposta: Janeiro de 2014
Eventos Relacionados: SBES

***Resumo.*** *O trabalho de pesquisa de mestrado, sumarizado neste artigo, objetiva fortalecer a especificação de contratos para soluções concebidas sob o paradigma de computação orientada a serviços. A robustez é buscada por meio de construções que agregam Design-by-Contract à linguagem NeoIDL, de modo que os serviços especificados assegurem as pós-condições, desde que satisfeitas suas pré-condições. A proposta será submetida a validação empírica, verificando regras de transformações em um novo plugin para a NeoIDL.*

## 1. Introdução e Caracterização do Problema

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 insituições [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*). Outras alternativas para REST, como Swagger e RAML[1], usam linguagens de propósito geral (em particular JSON) 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 NeoIDLfoi 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 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 envolvem inicialmente 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. Em seguida 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 NeoIDLpara diferentes tecnologias (como Twisted) que suportam a implementação de serviços em REST. Por fim, conduzir uma validação empírica da proposta usando uma abordagem mais exploratória, possivelmente usando a estratégia *pesquisa-ação*.

## 2. Fundamentação Teórica

SOC é um estilo arquitetural cujo objetivo é prover baixo acoplamento por meio da interação 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

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

e disponíveis para seus consumidores. Usualmente, a comunicação com os serviços, e entre eles, é feita por meio da troca de mensagens com uso de *serviços web*, que seguem padrões abertos de comunicação e que atuam sobre o protocolo HTTP. SOAP e REST [Fielding 2000] são as tecnologias mais usadas para implementação de serviços web, sendo que REST tem atraído um maior interesse nos últimos anos.

Entre os oito princípios para desenvolvimento SOA descritos em [Erl 2008], existe um especial interesse no *contrato padronizado* (*Standardized Service Contract*), que sugere que serviços de um mesmo inventário devem seguir os mesmos padrões de *design*, de modo a favorecer 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. É importante destacar que não existe um padrão para especificar contratos em REST, o que motivou o desenho da NeoIDL [Bonifácio 2015], uma linguagem específica do domínio para especificação de contratos REST e que, diferente das linguagens existents, provê recursos de modularidade e herança de tipos de dados customizados. Utilizando uma abordagem transformacional, a NeoIDL oferece suporte ferramental que gera código fonte para diferentes linguagens e plataformas REST, a partir de um conjunto de módulos NeoIDL onde são especificados os tipos de dados e os serviços.

A Figura 1 apresenta um exemplo de módulo NeoIDL com a definição de um tipo de dados `ItemDoCatalogo` e duas operações (chamadas de capacidades na terminologia REST): `atualizarItem` (operação do tipo POST associada ao *endpoint* `catalogo`) e `pesquisarItem` (operação do tipo GET, do mesmo *endpoint*).

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

**Figure 1. Exemplo de um módulo escrito em NeoIDL**

Conforme mencionado, o suporte ferramental da NeoIDL processa um conjunto de módulos escritos na linguagem NeoIDL, gerando o código com a estrutura para implementação dos serviços descritos. A versão atual do ambiente NeoIDL suporta geração de códigos em Java e Python para as plataformas *neoCortex*[2] e *Twisted*. Entretanto, esse ambiente é extensível por meio da implementação de novos plugins [Bonifácio 2015].

*Limitação da NeoIDL.* Atualmente a NeoIDL permite apenas a especificação de *weak contracts*, o que não permite estabelecer obrigações entre fornecedores e clientes de

[2]proprietária do Exército Brasileiro

serviços. Esse tipo de obrigação pode ser estabelecida com alguma técnica de *Design-by-contract* [Meyer 1992] (DBC) – na qual o consumidor e o fornecedor de serviços firmam entre si um conjunto de garantias. Em um contexto mais simples, de um lado o consumidor deve garantir que, antes da chamada a um serviço (ou um método de uma classe), os parâmetros de entrada devem ser respeitados (essas garantias são denominadas de pré-condições). Do outro lado, o fornecedor deve garantir que, uma vez respeitadas as pré-condições, as propriedades relacionadas ao sucesso da execução (pós-condições) são preservadas. DBC tem o objetivo de aumentar a robustez do sistema e tem na linguagem Eiffel [Meyer 1991] um de seus precursores. O uso de DBC na concepção de sistemas, segundo [Software 2015] é uma abordagem sistemática que tende a reduzir a quantidade de erros observados nos sistemas. Mais recentemente, algumas técnicas de *design-by-contract* foram especificadas e implementadas para outras linguagens de programação, como JML para a linguagem Java [Leavens et al. 2006] e `Spec#` para a linguagem `C#` (e demais linguagens da plataforma .NET) [Barnett et al. 2005].

## 3. Estado Atual do Trabalho

As construções para especificar pré-condições e pós-condições na linguagem da NeoIDL estão sendo definidas, de forma a agregar às especificações mecanismos para estabelecer as condições de execução e as garantias dos serviços. Note que, além da validação dos parâmetros de entrada e valores de retorno típicos do DBC, a abordagem proposta nessa dissertação permitirá a inclusão de chamadas a serviços REST como pré e pós condições. Para ilustrar algumas decisões de projeto, a seguir são apresentados dois exemplos do uso da abordagem que está sendo proposta.

O trecho de módulo NeoIDL constante da Figura 2 destaca a capacidade `incluirItem` (linha 7). Para que se tenha o item incluído no catálogo, é necessário fornecer um valor para o atributo descrição (obrigatório). A pré-condição (linha 5) 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 análoga, está previsto o prefixo `new`, que indica o valor do atributo após a execução do serviço. Na Figura 2 ainda é estabelecida uma cláusula `/@otherwise` (linha 6), estabelecendo que a falha no atendimento da pré-condição deve levar a uma resposta com o código HTTP 412 (falha de pré-condição).

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

**Figure 2. Exemplo da notação DBC na linguagem NeoIDL**

A Figura 3 apresenta a capacidade `excluirItem` (linha 8). Nesse caso são estabelecidas pré e 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 3. Exemplo da notação DBC na linguagem NeoIDL com chamada a serviço**

Ou seja, a proposta envolve elementos concebidos na linguagem JML [Leavens et al. 2006] (como as construções `old` e `new`) com elementos típicas do estilo arquitetural REST (chamadas a serviços utilizando métodos HTTP, semântica de retorno das operações observando os códigos de retorno HTTP, etc.), o que habilita não apenas o estabelecimento de contratos mas também uma forma de guiar a composição de serviços. Dessa forma, para permitir as chamadas a serviços a partir das pré e pós condições, novos plugins NeoIDL devem ser implementados.

*Validação.* Em paralelo à especificação das extensões de linguagem NeoIDL e implementação de plugins, está sendo feito o planejamento da validação da proposta – que deve seguir uma abordagem de pesquisa-ação considerando cenários reais (possivelmente com exemplos do Exército Brasileio, parceiro na concepção da primeira versão da NeoIDL). Um cenário candidato é de verificação de controle de acesso aos serviços, a partir de pré-condições com chamada ao serviço de autorização. Trata-se um problema real que, a medida em que se aprofunda na investigação de seus requisitos, se constroi a solução do problema (características da abordagem pesquisa-ação).

## 4. Trabalhos Relacionados

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 aderentes a que se pretende atingir com esta pesquisa.

Durante a pesquisa bibliográfica, muito embora o primeiro princípio SOA estabelecido por [Erl 2008] recomende *Contract First*, não se identificou publicação que associasse diretamente *Design-by-Contract* à especificação de contratos SOA.

Os trabalhos que mais se aproximam são os de [Ling et al. 2003] e [Heckel and Lohmann 2005]. O primeiro define uma forma de *Design-by-Contract* para arquiteturas de *workflow*. O autor considera, porém, que para grandes arquiteturas, a complexidade aumenta o risco de falha de *design*. Já [Heckel and Lohmann 2005] foca na especificação de modelos para DbC em Web Services, não tratando a especificação concreta do contrato. Além disso, se restringe a Web Services SOAP.

## References


Barnett, M., Leino, K., and Schulte, W. (2005). The `Spec#` programming system: An overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., and Muntean, T., editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg.

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.

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.

Heckel, R. and Lohmann, M. (2005). Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 116:145–156.

Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38.

Ling, S., Poernomo, I., and Schmidt, H. (2003). Describing web service architectures through design-by-contract. In *Computer and Information Sciences-ISCIS 2003*, pages 1008–1018. Springer.

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.

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.