UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RAFAEL SCORTEGAGNA

Avaliação da compatibilidade de serviços com uma abordagem orientada ao uso

Trabalho de Graduação.

Prof. Dr. Karin Becker
Orientadora

Porto Alegre, janeiro de 2013.

AGRADECIMENTOS

Agradeço aos meus pais Jair e Fátima pelo amor, carinho e dedicação que eles têm por mim, assim como a oportunidade que eles proporcionaram de estar hoje concluindo o ensino superior em uma universidade de excelência como esta. Ao meu irmão, que apesar de nossas diferenças, sempre foi meu amigo e companheiro nas horas de dificuldade.

À minha namorada Mariele Froner Nogueira pelo carinho, apoio e atenção nos momentos difíceis que tive durante a realização deste trabalho, assim como a paciência e compreensão nos momentos em que não pude dedicar toda minha atenção a ela.

À professora Karin pela paciência, compreensão e apoio na conclusão deste último trabalho nesta faculdade.

A todos os professores que fizeram parte da minha trajetória nesta faculdade sendo todos sempre muito atenciosos e corretos nas suas decisões.

Agradeço ainda a todos aqueles que fizeram parte deste trabalho direta ou indiretamente.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS **6**
LISTA DE FIGURAS **7**
LISTA DE TABELAS **8**
RESUMO **9**
ABSTRACT **10**
1 INTRODUÇÃO **11**
2 REVISÃO DE LITERATURA **13**
2.1 Web Services **13**
2.1.1 Linguagem WSDL 13
2.2 Métodos, estratégias e identificação de mudanças **14**
2.2.1 Regras de compatibilidade entre serviço 15
2.3 Projeto WS - Evolv **18**
2.3.1 Framework de apoio à evolução 18
2.3.2 Modelo de versões orientado a *feature* 18
2.3.3 Gerenciador de versões 20
2.3.3.1 Conversão de WSDL para feature 20
2.3.3.2 Avaliação de compatibilidade 22
2.3.4 Perfis de Uso 23
2.4 Considerações finais **24**
3 PROPOSTA DO PROBLEMA **25**
3.1 Problema e objetivo **25**
3.2 Visão Geral **26**
3.3 Modelo de Versões orientado a perfis de uso **27**
3.4 Flexibilização de regras para avaliação de compatibilidade **27**
3.5 Proposta de algoritmo **29**
4 IMPLEMENTAÇÃO **36**
4.1 Arquitetura **36**
4.1.1 Repositório 36
4.1.1.1 Representação do repositório no banco de dados 37
4.1.2 Perfil de uso 38
4.1.3 Funções relacionadas ao algoritmo 39
4.2 Estudo de caso **40**
4.2.1 Serviço StockQuote 40
4.2.1.1 O Serviço 40
4.2.1.2 Perfis de uso 46
4.2.1.3 Execução do algoritmo 46

5 CONCLUSÃO..... 53
6 REFERÊNCIAS 54

LISTA DE ABREVIATURAS E SIGLAS

KDD	Knowledge Discovery in Databases
NoSQL	Not Only SQL
SQL	Structured Query Language
WSD	Web Service Description
WSDL	Web Service Description Language
XML	eXtensible Markup Language
XSD	XML Schema Definition

LISTA DE FIGURAS

Figura 2.1 – Framework para gestão de mudanças 18
Figura 2.2 – Modelo de verão em nível de *feature* 19
Figura 2.3 – Descrição WSDL 1.1 e representação proposta 20
Figura 2.4 – Exemplo de versionamento do serviço StockQuote 21
Figura 2.5 - Algoritmo versionamento 22
Figura 2.6 – Estrutura de um perfil de uso 23
Figura 3.1 – expansão do framework de apoio à evolução 26
Figura 3.2 - Modelo de versões orientado à perfis de uso 27
Figura 4.1 – Exemplo de execução de query 37
Figura 4.2 – Exemplo de execução de uma consulta 38
Figura 4.3 – Exemplo de consulta da função *dependents* 39
Figura 4.4 – Exemplo de consulta da função *getMinOccurs* 40
Figura 4.5 – Serviço StockQuote 41
Figura 4.6 – Serviço *StockQuote* (continuação) 42
Figura 4.7 – Grafo de dependência da primeira versão do serviço *StockQuote* 42
Figura 4.8 – Segunda versão do serviço *StockQuote* 43
Figura 4.9 – Segunda versão do serviço *StockQuote* (continuação) 44
Figura 4.10 – Segunda versão do serviço *StockQuote* (continuação) 45
Figura 4.11 – Grafo de dependência da segunda versão do serviço *StockQuote* 45
Figura 4.12 – Construção grafo relacionado ao perfil 47
Figura 4.13 – Grafo de representação do perfil 1 47
Figura 4.14 – Execução do algoritmo com perfil1 e versões 1 e 2 do serviço *StockQuote* 48
Figura 4.15 – Construção grafo relacionado ao perfil 49
Figura 4.16 – Grafo de representação do perfil 2 49
Figura 4.17 – Execução do algoritmo com perfil2 e versões 1 e 2 do serviço *StockQuote* sem flexibilização das regras 50
Figura 4.18 – Log de execução do algoritmo 50
Figura 4.19 - Execução do algoritmo com perfil2 e versões 1 e 2 do serviço *StockQuote* com flexibilização das regras 51

LISTA DE TABELAS

Tabela 2.1 – Avaliação de mudanças em um serviço 16
Tabela 2.2 – Relação de autores com casos de compatibilidade 17
Tabela 2.3 - Casos de mudança para compatibilidade de versões 22
Tabela 3.1 – Regras de compatibilidade entre features 29

RESUMO

Atualmente, com a evolução dos sistemas de informações temos conectados na internet uma heterogeneidade muito grande de sistemas que interagem entre si. Essa interação, muitas vezes, é realizada com a utilização de web services. Um web service pode ser desenvolvido por um grupo de desenvolvedores que o disponibilizam em um servidor para ser utilizado por pessoas que assim desejar.

A disponibilização de novas versões de um web service pode tornar versões antigas incompatíveis com novas versões. Essa incompatibilidade faz com que um web service necessite de manutenção como qualquer outro componente de software. A manutenção de versões de web services requer estratégias para tornar a disponibilização de versões o menos impactante possível.

O trabalho "Measuring Change Impact Based on Usage Profiles" de Yamashita et al. (2012) propõe um framewok de apoio à evolução de serviços onde são podem ser realizadas verificações de compatibilidade, criação de perfis de uso e análise de impacto do uso de um serviço a partir de novas versões do mesmo.

Este trabalho tem como objetivos expandir o framewok proposto no trabalho de Yamashita et al. (2012) agregando um componente de avaliação de compatibilidade considerando um perfil de uso na avaliação e realizar a flexibilização das regras de uso utilizadas para a verificação de compatibilidade entre versões de um serviço. Este trabalho também apresenta experimentos utilizando versões de um serviço, perfis de uso e a flexibilização de regras para explorar a eficiência e realizar a validação de sua utilidade.

Palavras-Chave: *web services*, versionamento, *framework*.

Services compatibility evaluating by an use oriented approach

ABSTRACT

Nowadays, with the evolution of information systems we have connected on the internet a very large heterogeneity of systems that interact with each other. This interaction, often, is performed with the use of web services. A web service can be developed by a group of developers that provide it on a server to be used by people who desire.

The availability of new versions of a web service can make older versions incompatible with new versions. This mismatch makes that a web service requires maintenance just like any other software component. Maintaining versions of web services requires strategies to make available versions the least impactful possible.

The paper “Measuring Change Impact Based on Usage Profiles” from Yamashita et al. (2012) propose a framework to support the evolution of services where can be performed compatibility checks, creation of usage profiles and impact analysis from a service based on new versions of the same.

This paper aims to expand the framework proposed in the Yamashita et al. (2012) paper adding a component related to the evaluation of compatibility considering an usage profile on the evaluation and perform flexible use rules used to verify the compatibility between two versions of a service. This paper also presents experiments using versions of a service, usage profiles and easing of rule to explore the efficiency and perform validation of its usefulness.

Keywords: web services, versioning, framework

1 INTRODUÇÃO

Atualmente, com a evolução dos sistemas de informações, tem-se, na internet, uma heterogeneidade muito grande de sistemas que interagem entre si. Essa interação, muitas vezes, é realizada com a utilização de *web services.* Um *web service* é definido como um software projetado para suportar a interoperabilidade entre máquinas em uma rede (W3C, 2004).

Web services necessitam de manutenção como qualquer outro componente de software. Conforme Papazoglou et al. (2011), a gestão da evolução de um serviço engloba a criação, manutenção e desativação de versões diferentes em um ambiente provedor de serviços. Operações de correção, atualização ou criação de novas funcionalidades são constantes na manutenção de *web services.* Por exemplo, serviços providos pela *Google, Amazon* e *Ebay* são atualizados de forma regular uma ou duas vezes por mês. O serviço eBay *Trading*, por sua vez, disponibiliza uma nova versão de seu serviço a cada duas semanas. Essa manutenção nos leva ao problema da criação de múltiplas versões do serviço que devem ser suportadas durante a distribuição deste para garantir a evolução correta do mesmo.

Papazoglou (2008) afirma que é importante a adoção de uma estratégia de versionamento robusta para o suporte de múltiplas versões do serviço, e que para isso é necessário introduzir a noção de compatibilidade. Existem dois tipos de compatibilidade entre serviços: a compatibilidade com versões anteriores e a compatibilidade com versões futuras. A compatibilidade com versões anteriores garante que os clientes que utilizam um serviço antigo não necessitam realizar atualizações para utilizar o serviço modificado. A compatibilidade com versões futuras garante que os novos clientes não necessitam de adaptações com relação a serviços antigos. (PAPAZOGLOU, 2008);(FANG et al. 2007). Neste trabalho será abordada a compatibilidade com versões anteriores, portanto quando for citada a compatibilidade entre serviços é a compatibilidade com versões anteriores que está sendo referida.

Avaliações de compatibilidade podem suportar a evolução de um serviço provendo informações relevantes em relação aos efeitos das mudanças na aplicação do cliente. Abordagens tradicionais de compatibilidade avaliam a compatibilidade do pior caso para a determinação da compatibilidade de versões do serviço, ou seja, se um elemento do serviço na versão S é incompatível com o mesmo elemento na sua versão S', tal serviço será considerado incompatível por completo (FANG et al., 2007);(BECKER et al. 2008);(BROWN E ELLIS. 2004).

A compatibilidade é um problema central na evolução de um serviço, pois sua avaliação pode prover informações relevantes em relação aos efeitos das mudanças nas

aplicações do cliente (BECKER et al., 2008). Estabelecendo um veredicto sobre a compatibilidade ou não de um serviço baseado no pior caso não é uma visão verdadeira dessa compatibilidade. Isso ocorre pois as aplicações do cliente não utilizam, necessariamente, todas as funcionalidades de um serviço, podendo utilizar somente algumas delas.

Se levarmos em conta o provedor de serviços *eBay* por exemplo, e sua vasta gama de produtos, temos que o número de requisições por mês atinge a casa dos bilhões. Uma mudança em um serviço pode influenciar de forma diferenciada muitos desses clientes, que realizam essas requisições de maneiras diferentes. Uma pequena alteração em um serviço pode ser essencial para o cliente *x*, mas para o cliente *y* essa alteração nada influencia na sua utilização.

No trabalho de Yamashita et al. (2012) é apresentada a ideia de criação de perfis de uso de um serviço onde um perfil descreve as *features* atualmente usadas por um conjunto de aplicações quantificadas por métricas tais como: número de aplicações e número de requisições de uma operação. Esses perfis são gerenciados por um gerenciador de perfis que monitora as requisições dos serviços, agrupa os clientes de acordo com um padrão de utilização e elabora um perfil de aplicação enriquecido com as métricas citadas.

Este trabalho propõe a implementação de um algoritmo para a análise de compatibilidade de serviços com uma abordagem orientada ao uso do serviço, que, ao contrário das abordagens tradicionais, possibilite a avaliação da compatibilidade conforme o uso deste, representada por perfis de uso. Deverá ser possível verificar se o perfil analisado foi impactado pela mudança do serviço e, se o mesmo for impactado, a apresentação de quais *features* sofreram mudanças e quais mudanças foram executadas.

A principal motivação deste trabalho é a necessidade de melhoria na avaliação de compatibilidade de serviços com a utilização do cliente sobre o mesmo. Ainda agregar para o trabalho de pesquisa que é realizado junto a esta universidade o qual será citado posteriormente.

O resto deste trabalho está estruturado da seguinte forma. No Capítulo 2 serão apresentadas as regras de compatibilidade e os algoritmos existentes assim como o modelo de versionamento de algoritmo de compatibilidade existente. No Capítulo 3 será apresentada a proposta do problema assim como as alterações realizadas no modelo de versões atual juntamente com uma proposta de algoritmo. No Capítulo 4 será apresentado detalhes da implementação do algoritmo com a exibição dos resultados. Conclusão e trabalhos futuros serão apresentados no Capítulo 5.

2 REVISÃO DE LITERATURA

Este capítulo irá apresentar os conceitos fundamentais concernentes a este trabalho, sendo eles: web services e sua interface de apresentação, regras de compatibilidade para avaliação de compatibilidade de um serviço, algoritmos existentes para avaliação de compatibilidade e o modelo de versionamento utilizado para fundamentação deste trabalho.

2.1 Web Services

Web services fornecem um serviço padrão de interoperabilidade entre diferentes aplicações, rodando em uma variedade de plataformas e/ou sistemas. Mais especificamente pode-se afirmar que um web service é um sistema de software que suporta interação entre máquinas em uma rede de computadores (W3C, 2004).

A interoperabilidade entre as aplicações sobre uma rede é realizada através da troca de mensagens. A funcionalidade do serviço disponível a seus clientes é documentada em uma descrição do web service (*Web Service Description* - WSD), que é a especificação da interface do web service, escrita em WSDL (*Web Service Description Language*). Na descrição do serviço que são definidos: o formato da mensagem, tipo de dados, protocolo e o formato de transporte entre as máquinas que se comunicam (W3C, 2004).

A construção de um web service envolve muitas tecnologias correlacionadas. De todas essas tecnologias abordaremos aquela que realiza a descrição de um web service, a WSDL, pois é, neste trabalho, sobre a descrição do serviço que será avaliada a (in) compatibilidade do serviço .

2.1.1 Linguagem WSDL

Conceitualmente, definimos o WSDL como a notação para a descrição de um *web service* a partir das mensagens que são trocadas entre máquinas que se comunicam (W3C, 2004). Mais tecnicamente um documento WSDL é uma documento em formato XML (eXtensible Markup Language) para descrição de serviços de rede como um conjunto de terminais que operam em mensagens que contêm informações orientadas a documento ou orientadas a processo (W3C, 2001).

Um documento WSDL define um serviço como uma coleção de endereços de rede, ou portas. A definição abstrata de portas e as mensagens são separadas do uso concreto de instâncias, permitindo o reuso de definições (W3C, 2001).

No WSDL que descreve um serviço de rede, existe uma divisão por elementos do documento, a saber:

1. *Types* – um recipiente para as definições do tipo de dados utilizando algum sistema (ex. XSD – XML Schema Definition)
2. *Message* – uma definição abstrata dos dados da comunicação.
3. *Operation* – uma descrição abstrata de uma ação suportada pelo serviço.
4. *Port Type* – um conjunto abstrato de operações suportado por um ou mais terminais.
5. *Binding* – um protocolo concreto e uma especificação do formato de dados para um tipo particular de *port type.*
6. *Port* – um único terminal definido como a combinação de um *binding* e um endereço de rede.
7. *Service* – uma coleão de terminais relacionados.

O WSDL não introduz um novo tipo de definição de linguagem (W3C, 2004). Contudo o WSDL suporta a especificação de esquemas XML (XSD) como o seu sistema de tipo canônico (W3C, 2004b) para a descrição do formato das mensagens. O WSDL define, também, um mecanismo de ligação comum. Este mecanismo é utilizado para anexar um formato de protocolo, formatos de dados ou estruturas a uma mensagem abstrata, operação ou *endpoint.*

2.2 Métodos, estratégias e identificação de mudanças

A necessidade de identificar e avaliar as mudanças realizadas ao longo da vida de um serviço nos leva a utilização de métodos de versionamento como melhor solução para este problema. Na visão do cliente, uma versão do serviço se refere ao “contrato” estabelecido entre a interface do mesmo e a funcionalidade que ele entrega (CASTAGNA et al., 2008). Já, na visão do provedor de serviço, versionamento se refere a uma implementação particular do serviço e como esta implementação evoluiu ao longo do tempo, levando em conta as suas mudanças.

Cada provedor de serviços constrói a sua estratégia de versionamento a fim de gerar soluções, ou não, para os clientes impactados pelas mudanças. Algumas estratégias apresentam uma solução em que não é considerado aonde o serviço impacta seus clientes, sendo que a identificação de onde as mudanças impactam os clientes é de responsabilidade dos desenvolvedores (PELTZ e ANAGOL-SUBBARAO apud ANDRIKOPOULOS et al., 1992). Temos também estratégias de compatibilidade orientada para realização do versionamento, ou seja, a manutenibilidade é realizada mantendo-se ativas várias versões do serviço (BROWN e ELLIS, 2004).

Apesar das diferenças apresentadas, o objetivo em ambos os casos é diminuir o número de versões ativas para a disponibilização do serviço. Deve ser observado que antes da desativação de um serviço é proporcionado um tempo de carência para a

adaptação e, dependendo do modelo adotado, ou os clientes são avisados da realização da mudança ou eles devem descobrir por conta própria a mudança realizada (ANDRIKOPOULOS et al., 2011). Pode-se citar como exemplo o *eBay Trading Service* onde, para cada nova versão do serviço, são disponibilizadas notas no website do eBay que apresentam explicitamente as principais mudanças da nova versão com relação a anterior (EBAY TRADING API, 2012), sendo que cada versão fica disponível por pelo menos 18 meses. Com isto, os desenvolvedores são responsáveis por detectar onde estas mudanças irão afetar suas aplicações.

A identificação destas mudanças pelos clientes de serviços é muito importante para a manutenção de seu funcionamento. Na literatura encontramos três modelos de identificação de mudanças: modelo cliente, modelo notificação e o modelo transparente. No modelo cliente (BROWN E ELLIS, 2004), todas as mudanças resultam em uma nova versão e a identificação da existência desta versão e a sua adaptação é responsabilidade do utilizador do serviço. No modelo de identificação (FANG et al., 2007) o utilizador do serviço é explicitamente notificado da existência de uma nova versão e requisitado a realizar ajustes. Por fim, o modelo transparente os utilizadores do serviço não necessitam ser notificados nem precisam fazer ajustes pois as mudanças são transparentes, como o próprio nome já diz.

2.2.1 Regras de compatibilidade entre serviços

Após o estabelecimento de uma estratégia para realizar a evolução de um serviço são estabelecidas regras para a avaliação da sua compatibilidade. Um dos problemas críticos na evolução de um serviço é a compatibilidade da nova versão com as suas versões antigas do serviço. Para avaliarmos a compatibilidade de uma nova versão com as versões antigas devemos analisar os tipos de compatibilidade existentes. Existem dois tipos de compatibilidade: a compatibilidade com versões anteriores e a compatibilidade com versões futuras (BROWN e ELLIS,2004);(FANG et al., 2007);(PAPAZOGLOU, 2008).

A compatibilidade com versões anteriores garante que os clientes que utilizam um serviço antigo não necessitam atualizações para utilizar o serviço modificado, ou seja, podem continuar utilizando o sistema normalmente sem a ocorrência de erros após a atualização do mesmo. Já a compatibilidade futura garante que os novos clientes não necessitam adaptações com relação a serviços antigos para utilizar os mesmos (BROWN e ELLIS,2004);(FANG et al., 2007);(PAPAZOGLOU, 2008).

Na realização deste trabalho será abordada a compatibilidade com versões anteriores. Endrei et al. (2006) afirmam que a compatibilidade com versões anteriores é altamente relacionada com a mudança ou a gestão de mudanças e apresentam os estágios típicos na manutenção da mudança, que são:

- A avaliação se a mudança é compatível com versões anteriores;
- Se não for compatível, é avaliada a necessidade de suporte para a interface antiga;
- Se for compatível, projetar e implementar a mudança para que tanto a velha como a nova interface sejam suportadas simultaneamente;

- Por fim, a desativação do serviço antigo é realizada.

Dentre esses estágios o que interessa no contexto deste trabalho é a avaliação se a mudança é compatível ou não com versões anteriores do serviço. Para esta avaliação temos conjuntos de alterações no serviço que determinam a (in)compatibilidade do mesmo (BROWN e ELLIS, 2004);(ENDREI et al., 2006);(FANG et al., 2007);(BECKER et al.,2008).

As mudanças de um *web service* podem ser divididas em três grupos: mudanças de implementação, mudanças de interface e mudanças de *binding*. Esta divisão não é rígida e alguns autores como Endrei et al. (2006) apresentam somente as mudanças que consideram compatíveis ou incompatíveis. Não serão consideradas as mudanças de implementação nem as mudanças de *binding* apresentadas em Fang et al. (2007) pois essas não estão relacionadas com a descrição do serviço.

Tabela 2.1 – Avaliação de mudanças em um serviço

Caso	Mudança	Descrição	Compatível?
1	Adição de operação	Nova operação é disponibilizada no serviço	Sim
2	Renomeação de uma operação	Renomear uma operação	Não
3	Remoção de uma operação	Remover uma operação do serviço	Não
4	Adição de um tipo	Adição de tipos não relacionado a parâmetros existentes	Sim
5	Adição de um tipo	Adição de um tipo como parâmetro opcional de entrada de uma operação.	Sim*
6	Adição de um tipo	Adição de um tipo obrigatório como parâmetro de operação	Não
7	Renomeação de um tipo	Renomeação de parâmetros ou tipo de dados complexo	Não
8	Alteração de um tipo primitivo	Alterar um tipo primitivo de um parâmetro sem a perda de informações	Sim*
9	Alteração de um tipo primitivo	Alterar um tipo primitivo de um parâmetro com a perda de informações	Não
10	Mudança de cardinalidade de um parâmetro	Mudanças de cardinalidade inferior	Sim*
11	Mudança de cardinalidade de um parâmetro	Mudança de cardinalidade superior	Sim*

A Tabela 2.1 apresenta as principais mudanças relacionadas a interface e o veredicto sobre sua compatibilidade. A primeira observação a ser feita com relação a esta tabela é que somente dois casos são consenso na literatura, com relação a compatibilidade, a

saber: a adição de nova operação, caso 1, e adição de um tipo não relacionado a nenhum tipo já existente, caso 4. As marcações de compatibilidade com * significam a falta de consenso sobre esta afirmação, na tabela 2.2 é apresentada a relação dos autores com os casos onde as mudanças são consideradas compatíveis.

Tabela 2.2 – Relação de autores com casos de compatibilidade

Autor	**Casos**
Brown e Ellis (2004)	1, 4
Fang et al. (2008)	1, 4
Endrei et al. (2006)	1, 4, 5, 10, 11
Becker et al. (2008)	1, 4, 5, 10, 11
Andrikopoulos et al. (2011)	1, 4, 5, 8, 10, 11

Becker et al. (2008), Andrikopoulos (2001) e Endrei et al. (2006) apresentam algumas regras para avaliação de compatibilidade mais flexíveis com relação a adição de parâmetros e mudança de cardinalidade.

Com relação a alterações de cardinalidade de parâmetros são considerados compatíveis os seguintes casos:

1. Alterar a cardinalidade inferior de obrigatória para opcional será compatível somente se for relacionado a um parâmetro de entrada de uma operação. (Caso 10 da Tabela 2.1);
2. Alterar a cardinalidade superior de n para y (aonde y > n) será compatível somente se for relacionado a um parâmetro de entrada de uma operação. (Caso 11 da Tabela 2.1);
3. A combinação dos casos acima somente será compatível se for relacionado a um parâmetro de entrada de uma operação;
4. Alterar a cardinalidade inferior opcional para obrigatória será compatível somente ser for relacionado a um parâmetro de saída de uma operação. (Caso 10 da Tabela 2.1);
5. Alterar a cardinalidade superior de n para y (aonde y < n) será compatível somente se for relacionado a um parâmetro de saída de uma operação. (Caso 11 da Tabela 2.1);
6. A combinação dos dois casos acima somente será compatível se for relacionada a um parâmetro de saída de uma operação;
7. Alterar a cardinalidade de um parâmetro de obrigatória para opcional.

Quando uma nova versão do serviço é publicada e essa versão apresenta mudanças incompatíveis, ambas as versões (antiga e atual) podem estar disponíveis ou não. Para a avaliação da compatibilidade dessas versões são utilizados métodos e algoritmos de versionamento, os quais são apresentados na sessão a seguir.

2.3 Projeto WS - Evolv

2.3.1 Framework de apoio à evolução

No trabalho de Yamashita et al. (2012) é apresentada uma proposta de gestão de mudanças onde os provedores de serviço podem avaliar e quantificar o impacto das mudanças baseados em análise de uso. A partir deste ponto do texto qualquer referência a *framework* deve se considerar uma referência a esta proposta de gestão de mudanças.

O *framework* é baseado em um modelo de versionamento orientado a *features*, onde cada *feature* corresponde a uma porção da descrição de um serviço WSDL/XSD. Para isso, um serviço é caracterizado como a composição de operações onde dados são trocados de acordo com tipo pré-definidos (ex.: mensagens, elementos do esquema). O *framework* é composto por três componentes principais, que podem ser observados na Figura 2.1, cada componente tem uma função específica (YAMASHITA et al.,2012).

Version Manager – este componente extrai as versões das *features* da descrição do serviço, mantendo as relações, e avalia a compatibilidade no que diz respeito as versões existentes.

Profile Manager – este componente monitora a requisição de serviço, agrupa clientes conforme padrões de uso, e produz perfis de uso das aplicações enriquecidas com métricas relevantes para análise de uso.

Usage Manager – este componente possibilita a análise do impacto das mudanças levando em conta as características de uso de acordo com as métricas estabelecidas.

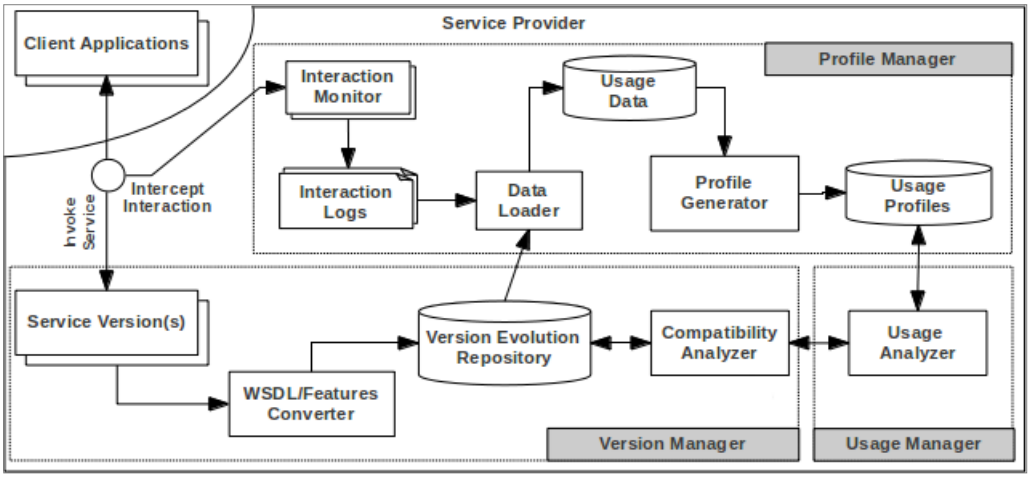

Figura 2.1 – Framework para gestão de mudanças

Fonte: YAMASHITA et al., 2012

Até o momento do desenvolvimento deste trabalho, estão sendo realizadas as implementações dos componentes: *Version Manager* (YAMASHITA et al. 2011) e *Profile Manager* (em desenvolvimento).

2.3.2 Modelo de versões orientado a *feature*

No modelo de versões apresentado por Yamashita et al. (2012) o objetivo é prover um gerenciamento abstrato da descrição da interface avaliando somente as *features* que

sofreram alterações, ao invés de todo serviço. Essa representação com um nível de detalhamento maior permite um controle mais amplo das partes modificadas na descrição de um serviço, podendo detectar quais partes que foram modificadas e sua compatibilidade.

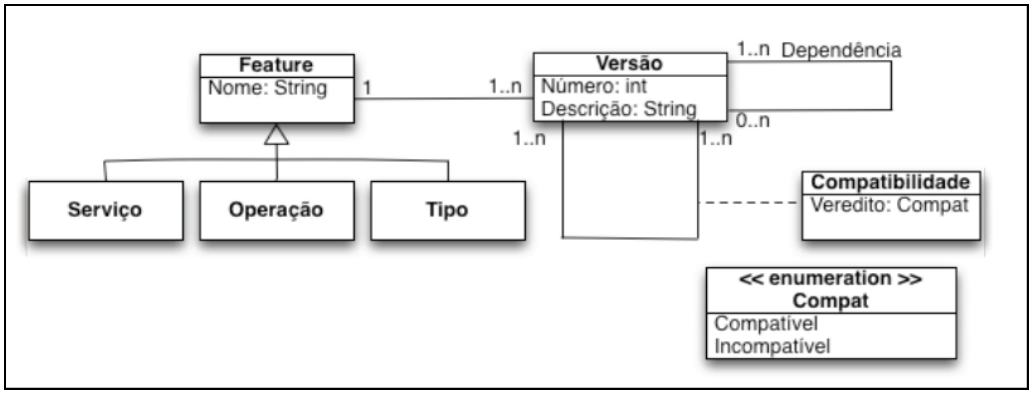

Figura 2.2 – Modelo de versão em nível de *feature*

Fonte: YAMASHITA et al. (2012)

No modelo de versões proposto por Yamashita et. al (2012) apresentado na Figura 2.2, uma *feature* é a generalização de serviço, operação e tipo. Cada *feature* tem pelo menos uma versão, a qual pode depender de outras versões de diferentes *features*. Versões são identificadas por um par <Feature.Nome, Versão.Número>. O atributo Versão.Descrição corresponde à descrição textual do documento WSDL.

Yamashita et al. (2012) estabelecem a correspondência entre a representação textual do serviço *WSDL/XSD* e o modelo de versões apresentado, da seguinte forma:

Operação: relacionado ao conteúdo da *tag* <*operation*> juntamente com as *tags* <*portType*> e <*binding*>.

Tipo: relacionado ao conteúdo das *tags* < *element* >, < *complexType*> ou <*simpleType*> dentro da *tag* <*schema*>, ou do conteúdo da *tag* < *message* >. Com relação aos tipos, somente são considerados para versionamento definidos fora do contexto dos elementos complexos XSD, ou seja, somente são versionados tipos próprios para reuso. Consequentemente, não são versionados os tipos primitivos (ex.: string, double, etc.) nem tipos complexos que não podem ser referenciados em outros lugares.

Serviço: relacionado a todo conteúdo restante do documento de descrição de interface, como a *tag* < *service* > e o conteúdo restante das *tags* < *schema* >,<*portType*> e <*binding*>.

A Figura 2.3 apresenta da correspondência da descrição WSDL e do modelo proposto. Neste exemplo, Yamashita et al. (2012) apresentam como os fragmentos da descrição de um WSDL v1.6 foram mapeados para a representação proposta, utilizando o serviço *StockQuote*. Foram separadas as descrições da Figura 2.3(i) em fragmentos para a representação do serviço, das operações e dos tipos. O resultado desta fragmentação é um grafo contendo as versões das *features* (Figura 2.3(ii)). Cada versão da *feature* é associada com a correspondente descrição, como os exemplos das Figuras 2.3(iii), 2.3(iv) e 2.3(v).

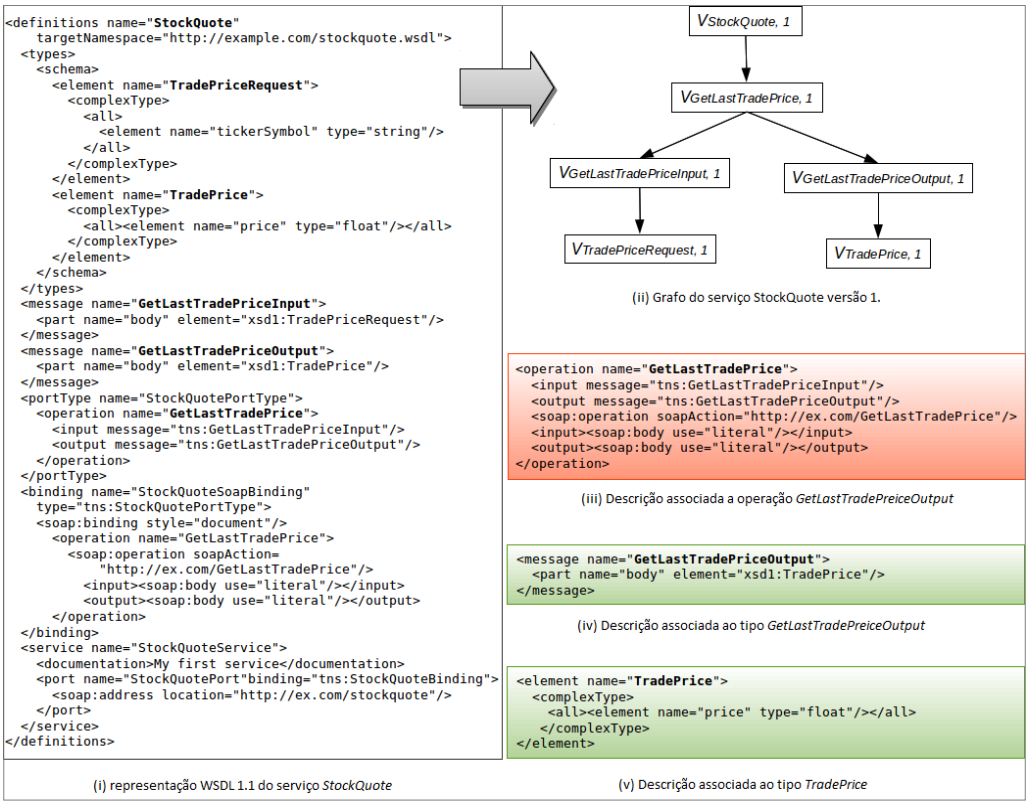

Figura 2.3 – Descrição WSDL 1.1 e representação proposta

Fonte: YAMASHITA et al. (2012).

2.3.3 Gerenciador de versões

Yamashita et al. (2012) dividem o componente *Version Manager* em duas etapas, a conversão do WSDL para *features* e a avaliação de compatibilidade. Estas etapas são apresentadas a seguir.

2.3.3.1 Conversão de WSDL para feature

A conversão de WSDL para *features* no modelo de Yamashita et al. (2012) identifica as *features* em um documento WSDL, relaciona elas à versão apropriada, possivelmente criando novas, e armazena esta representação abstrata em um Repositório de Evolução de Versões.

Neste modelo somente são avaliadas *features* explicitamente alteradas ou *features* indiretamente afetadas pelas mudanças. Uma *feature* alterada é uma *feature* que teve seu fragmento de descrição alterado, dependendo de uma *feature* que não dependia antes, ou que não depende de uma *feature* a qual era dependente antes. E quando uma *feature* é considerada afetada é considerado que ela não mudou mas que depende de uma *feature* que foi alterada ou afetada.

Yamashita et al. (2012) apresentam os passos para a conversão do WSDL para *features.* Primeiramente, o documento de descrição de interface é convertido para o nível de representação de *feature* (ex.: Figura 3(i)), o qual resulta em um grafo representando as versões das *features* e suas relações de dependência (ex.: Figura 3(ii)).

Para cada *feature*, é realizada uma pesquisa no repositório para comparar com versões existentes da *feature* em questão. Essa análise é realizada de uma maneira *bottom-up* no grafo de *features* para verificar corretamente as mudanças de dependência, seguindo as seguintes regras:

- Se a *feature* não existe, então ela é criada junto com a sua primeira versão;
- Se a *feature* já existe, já foi avaliada, e a sua descrição difere de todas as versões de sua *feature* particular, então ela é marcada como alterada no grafo e uma nova versão para esta *feature* é criada;
- Se a feature já existe e sua descrição é igual a uma versão existente, temos duas possibilidades:
  - Se ela depende de outra *feature* que já foi marcada como alterada, então uma nova versão é criada devido a efeitos de propagação;
  - Qualquer outra *feature* que dependa desta é referenciada a uma versão igual já existente.

Ao final da conversão do WSDL para uma *feature* temos um grafo como o da Figura 2.4.

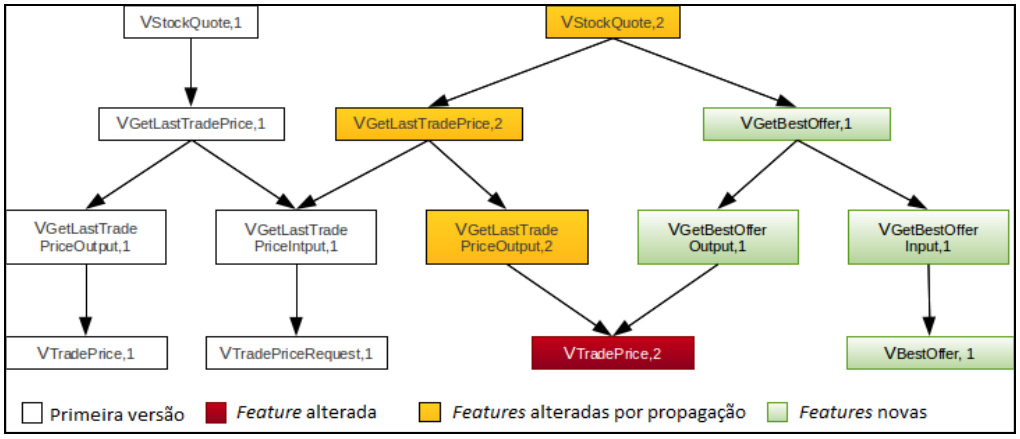

Figura 2.4 – Exemplo de versionamento do serviço *StockQuote*

Fonte: YAMASHITA et al. (2011)

2.3.3.2 Avaliação de compatibilidade

Na composição do algoritmo Yamashita et al. (2011) utiliza uma tabela de regras (Tabela 2.3) de compatibilidade para avaliar a relação de compatibilidade entre duas versões de *features* e estabelecer a relação de compatibilidade entre ambas.

Tabela 2.3 - Casos de mudança para compatibilidade de versões

Mudança	**Tipo de *feature***	**Descrição**	**Compatibilidade**
Adição	Operação	Adicionar nova operação ao serviço	Compatível
Adição	Tipo	Acionar novo tipo como dependente de uma nova operação ou novo tipo	Compatível
Adição	Tipo	Adicionar novo tipo como dependente de uma operação ou tipo existente	Incompatível
Atualização	Tipo	Mudança na descrição conforme ordem, cardinalidade ou tipo	Incompatível
Remoção	Operação	Remover a dependência de uma operação	Incompatível
Remoção	Tipo	Remover a dependência d e um tipo	Incompatível

Fonte: YAMASHITA ET. AL, 2011

Como podemos observar na Figura 2.5, o algoritmo avalia recursivamente a compatibilidade entre duas versões da mesma *feature* de acordo com as regras da Tabela 2.3, com o intuito de estabelecer a relação de compatibilidade entre eles, com o veredito correspondente. O algoritmo recebe duas versões de *features* como entrada Vfeature,p e Vfeature,q, e avalia a compatibilidade do último no que diz respeito ao primeiro. O grafo com raiz em Vfeature,q é percorrido em profundidade, o que possibilita a propagação das incompatibilidades detectadas para as versões dependentes.

Listing 1 *compatibilityAssessment*($v_{featurep}$, $v_{featureq}$)

```
1  boolean compat ← true;
2  compat ← evaluateRemovedDependencies(v_featurep ,v_featureq);
3  compat ← compat ∧ evaluateDescription(v_featurep , v_featureq );
4  foreach v_depQj ∈ setOfDependencies(v_featureq) do
     // If there is a dependency feature version with the same name but different version
5    if exists v_depPj ∈ setOfDependencies(v_featurep) ∧ (depP = depQ) ∧ (i ≠ j) then
       // Verify recursively if these two versions of a same feature are in turn compatible
6      compat ← compat ∧ compatibilityAssessment(v_depPj, v_depQj)
7    end if
8  end foreach
9  setVerditct(v_featureq, v_featurep , compat);
10 return compat;
```

Figura 2.5 - Algoritmo versionamento

Fonte: YAMASHITA et al. (2011)

2.3.4 Perfis de Uso

Yamashita et al. (2012) definem perfis de uso como representações abstratas de grupos de aplicações de clientes com um padrões de uso semelhante. Ainda definem que métricas podem ser associadas à perfis de uso, podendo assim avaliar o impacto das mudanças na evolução do serviço. Essa associação pode ser observada na Figura 2.6.

A estrutura de um perfil de uso, Figura 2.6, é composta pelo perfil de uso, *Usage Profile.* Este perfil de uso é uma generalização de uma versão do serviço, o qual pode ser associado a métricas de avaliação, como por exemplo: número de aplicações que utiliza determinadas *features* e número de requisições à determinadas *features*. A versão do serviço associado ao perfil de uso, por sua vez, é um conjunto de features daquele serviço onde uma *feature* é uma generalização de serviço, operação e tipo.

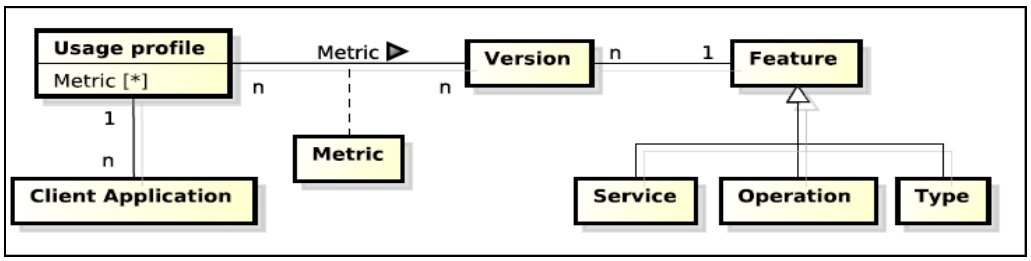

Figura 2.6 – Estrutura de um perfil de uso

Fonte: Yamashita et al. (2011)

Para um melhor entendimento dessa estrutura será apresentado um exemplo prático. Supõe-se a utilização do serviço *StockQuote* nas versões representadas na Figura 2.4, é analisada a versão 2 do serviço para determinado cliente com a aplicação de uma métrica simples: *features* com maior número de requisições para aquele cliente. Após realizar o processo de mineração de dados se chega a conclusão que as *features* com mais requisições para aqueles clientes são: *VLastTradePrice, VGetLastTrade-PriceOutput e VTradePrice*, com isso pode ser estruturado o perfil de uso que utilize a versão escolhida 2 do serviço StockQuote com as *features VLastTradePrice, VGetLastTradePriceOutput e VTradePrice*.

Um modelo para a criação de perfis de uso, Figura 2.1 parte destacada como *Profile Manager*, é proposto por Yamashita et al. (2012). Nesse modelo a criação de perfis de uso é realizada a partir do monitoramento e do registro das requisições de um determinado serviço. Esse monitoramento produz uma quantidade muito grande de dados de uso. Para resolver este problema foi utilizada uma técnica de KDD (Knowledge Discovery in Databases) para a identificação de padrões de uso.

KDD é o processo iterativo e interativo, não trivial de identificar padrões válidos, potencialmente úteis e inteligíveis a partir de massas de dados (FAYYAD et al., 1996). Esse processo pode conter várias etapas, a saber: seleção de dados, processamento, transformação de dados, mineração de dados e avaliação e pós-processamento. A realização dessa etapa é realizada nos componentes inerentes ao modelo apresentado na Figura 2.1 (parte destacada como *Profile Manager*).

A primeira etapa realizada é a seleção de dados que é realizada pelo componente *"Interaction Monitor"* o qual é responsável pela interpretação da troca de mensagens

entre a aplicação do cliente e as versões do serviço que eles estão ligados (YAMASHITA et al., 2012). Estes dados são armazenados em um log de interação. Cada serviço possui o seu "*Interaction Monitor"* o qual grava os dados em seu próprio arquivo de log.

Com a coleta de dados completa pode ser realizada a extração destes dados. O componente *"Data Loader"* é responsável por esta etapa, os dados são extraídos e processados a partir dos arquivos de logs de diferentes versões do serviço. Após o processamento dos dados eles são integrados à base de dados de uso ("*Usage Database"*). A base de dados de uso é utilizada como um facilitador para a preparação de dados na realização dos diferentes tipos de análise.

A última etapa é a geração de perfis de uso que é realizada pelo componente *"Profile Generator"*. Este componente é responsável por agrupar aplicações baseadas no uso de *features* semelhantes, utilizando algoritmos de *clustering*, e após o pós-processamento dos resultados agrupados são criados os perfis de uso (YAMASHITA et al., 2012).

2.4 Considerações finais

Yamashita et al. (2011) adotam um modelo de versionamento com uma granularidade menor onde, mais facilmente, são encontradas e avaliadas as mudanças na descrição de um serviço.

No Capítulo 3 serão apresentadas alterações no modelo de versões para que exista a possibilidade de avaliar as mudanças com relação a um perfil de uso, ou seja, orientado ao uso do serviço. Também será apresentada uma alteração no *framework* para agregar a avaliação de compatibilidade com relação a um perfil de uso no mesmo.

Essas alterações objetivam tornar possível a verificação do real impacto da mudança do serviço no cliente, ou seja, somente serão apresentadas mudanças nas *features* que são utilizadas pelo cliente.

3 PROPOSTA DO PROBLEMA

Este capítulo irá relembrar os problemas com relação à evolução de serviços web, assim como o objetivo deste trabalho. Serão apresentadas as alterações realizadas no modelo de versões apresentado por Yamashita et al. (2011), também será apresentado o modelo de perfil de uso utilizado no desenvolvimento e, por fim, uma proposta de algoritmo para a resolução do problema.

3.1 Problema e objetivo

Analisando as propostas citadas no Capítulo 2, pode-se observar que estas resolvem o problema de avaliação de compatibilidade entre versões. Entretanto, existem pontos a serem melhorados para trazê-las mais perto da realidade do uso efetivo de serviços por seus clientes, levando-se em conta a evolução de serviços.

Os autores discutidos apresentam soluções onde consideram o pior caso para a avaliação de compatibilidade (FANG et al., 2007);(BECKER et al. 2008);(BROWN E ELLIS. 2004). As soluções que se baseiam no pior caso são soluções importantes pois, apesar de não existir um detalhamento maior sobre as mudanças que são feitas na evolução do serviço, o cliente que utiliza o mesmo sabe que foram realizadas mudanças possivelmente incompatíveis no serviço que ele utiliza, e que elas podem ou não afetá-lo.

Uma solução para a melhoria do controle da evolução do serviço é apresentada por Yamashita et al. (2012). Os autores apresentam um algoritmo mais flexível que apresenta uma granularidade menor na avaliação da mudanças e de sua compatibilidade, levando em conta cada mudança realizada em partes específicas de um serviço. Ainda é proposto um framework de apoio à evolução que possui componentes de avaliação de compatibilidade e construção de perfis de serviços. Em seu estágio atual, os perfis de uso são utilizados apenas para quantificar o impacto de mudanças, no componente *Usage Analyzer* (Figura 2.1).

Analisando-se este *framewok* verificou-se a possibilidade da avaliação da compatibilidade de duas versões de um serviço levando-se em conta a utilização de perfis de uso. Realizando-se a avaliação de compatibilidade desta forma teremos uma avaliação mais específica da compatibilidade levando em conta o uso de um conjunto representativo de clientes.

Outro aspecto importante é que observando as regras utilizadas para a avaliação de compatibilidade do modelo de versionamento da proposta de Yamashita et al. (2012), baseadas no consenso sobre literatura (ex: Brown e Ellis (2004), Endrei et al. (2006),

Fang et al. (2008)) conclui-se que as regras de compatibilidade utilizadas por Yamashita et al. (2012) podem ser mais flexíveis, considerando as propostas de Becker et al. (2008) e Andrikopoulos et al. (2011), as quais consideram que determinadas mudanças em tipos de dados são possíveis, desde que se considere seu contexto de uso como parâmetro de entrada ou saída.

Como este trabalho é realizado a partir do trabalho de Yamashita et al. (2012) será proposta a expansão do framework de apoio à evolução com dois grandes objetivos:

1- Realizar a avaliação de compatibilidade orientada a utilização de um perfil de uso;

2- Levar em conta regras de compatibilidade mais flexíveis na avaliação de compatibilidade entre as versões do serviço.

Com a realização desses objetivos teremos um *framework* mais robusto e com uma maior flexibilização para o cliente, pois ele terá mais opções de avaliação quando realizar a evolução do serviço que utiliza.

3.2 Visão Geral

A realização deste trabalho propõe a expansão do framework de apoio à evolução de serviços, com a adição de um novo componente, a saber: o "Analisador de Compatibilidade Orientado a Perfis de Uso" como mostra em destaque a Figura 3.1.

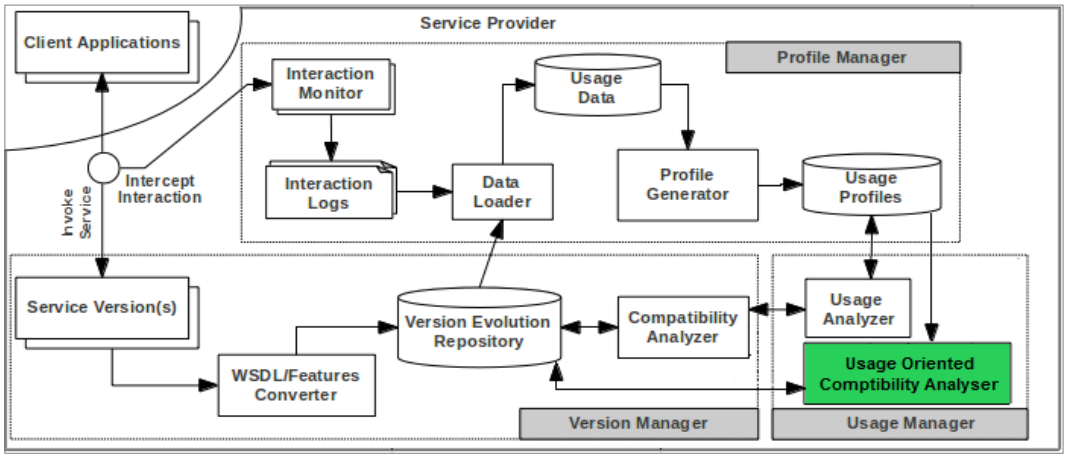

Figura 3.1 – expansão do framework de apoio à evolução

Este novo componente está incluso no módulo de Gerenciamento de Uso (User Manager) juntamente com o componente de análise de uso pré-existente, entretanto ambos são independentes. O componente de análise de compatibilidade orientado ao uso está relacionado ao repositório de versões e ao repositório de perfis de uso. Esta associação permite a recuperação das versões e de perfis de uso requisitados para a avaliação da compatibilidade levando-se em conta estes fatores.

Um segundo objetivo deste trabalho é a flexibilização das regras de avaliação de compatibilidade, estas regras não são representadas visualmente no *framework*. As regras de compatibilidade mais flexíveis são baseadas nas regras encontradas na literatura (Tabela 2.1 - casos 5, 8, 10 e 11).

Para o desenvolvimento deste novo componente será proposta uma extensão ao modelo de versões existente, para que este leve em conta a utilização de perfis de uso, e uma proposta de algoritmo que implementa a análise de compatibilidade com a aplicação de perfis de uso e utilizando regras de avaliação mais flexíveis nesta análise.

3.3 Modelo de Versões orientado a perfis de uso

A primeira etapa do desenvolvimento do componente de avaliação de compatibilidade orientado ao uso é a alteração do modelo de versões atual, Figura 2.2, para um modelo que compreenda também os perfis de uso. Estas alterações são apresentadas na Figura 3.2.

Nesta alteração a avaliação de compatibilidade deixa de ser exclusivamente entre versões (CompatVersão) e passa a contemplar também a possível utilização de um perfil de uso (CompatPerfil). Para isto são adicionados os perfis de uso na avaliação de compatibilidade, os quais possuem a estrutura apresentada na Seção 2.3.4. Neste trabalho somente será abordada a compatibilidade com a utilização de perfis de uso (CompatPerfil).

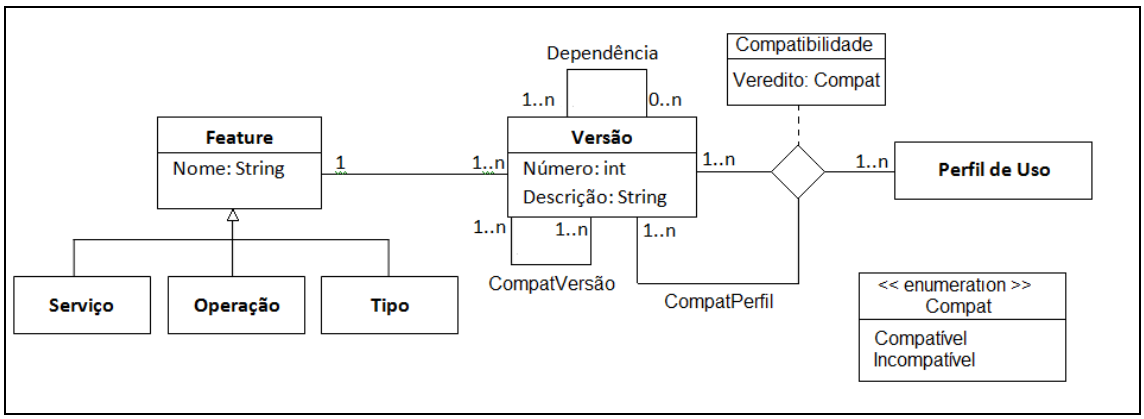

Figura 3.2 - Modelo de versões orientado à perfis de uso

3.4 Flexibilização de regras para avaliação de compatibilidade

Para o desenvolvimento deste trabalho são levadas em conta as regras de avaliação de compatibilidade apresentadas na Tabela 3.1. Estas regras são uma flexibilização das regras apresentadas por Yamashita et al. (2012) a partir de regras encontradas na literatura (ENDREI et al., 2006);(BECKER et al.,2008);(ANDRIKOPOULOS et al.(2011)).

Algumas regras apresentadas na Tabela 3.1 são dependentes de alguns fatores para um veredicto sobre a compatibilidade ou não da alteração, a saber:

- Caso 2: a alteração será considerada compatível somente se for adicionado um tipo dependente como parâmetro opcional de uma operação/tipo de entrada. Se a dependência for obrigatório a alteração é considerada incompatível;

- Caso 4: a alteração será considerada compatível se a ordem não for obrigatória;
- Caso 5: a alteração será considerada compatível se:
  - Alterar a cardinalidade inferior de obrigatória para opcional será compatível somente se for relacionado a um parâmetro de entrada de uma operação;
  - Alterar a cardinalidade superior de n para y (onde y > n) será compatível somente se for relacionado a um parâmetro de entrada de uma operação;
  - A combinação dos dois casos anteriores será compatível somente se for relacionado a um parâmetro de entrada de uma operação;
  - Alterar a cardinalidade inferior opcional para obrigatória será compatível somente se for relacionado a um parâmetro de saída de uma operação;
  - Alterar a cardinalidade superior de n para y (aonde y < n) será compatível somente se for relacionado a um parâmetro de saída de uma operação;
  - A combinação dos dois casos anteriores será compatível somente se for relacionado a um parâmetro de saída de uma operação.
- Caso 6: a alteração será considerada compatível se o tipo de dado for alterado por um tipo que é abrangido pelo tipo original. Exemplo: alteração de “*Postive Integer*” para “*Integer*”.

Tabela 3.1 – Regras de compatibilidade entre features

Casos	Mudança	Tipo de Feature	Descrição	Compatibilidade
1	Adição	Operação	Adição de nova operação	Compatível
2	Adição	Tipo	Adição de um tipo dependente de uma operação/tipo existente	Depende
3	Adição	Tipo	Adição de um tipo não relacionado a parâmetros existentes	Compatível
4	Alteração	Tipo	Alteração na descrição com relação à ordem	Depende
5	Alteração	Tipo	Alteração na descrição com relação à cardinalidade	Depende
6	Alteração	Tipo	Alteração na descrição com relação ao tipo de dado	Depende
7	Remoção	Operação	Remoção de uma operação	Incompatível
8	Remoção	Tipo	Remoção de um tipo	Incompatível

As regras apresentadas na Tabela 3.1 serão utilizadas no algoritmo de avaliação de compatibilidade orientado à perfil de uso. Todas as alterações não explicitadas na Tabela 3.1 são consideradas incompatíveis.

3.5 Proposta de algoritmo

O algoritmo proposto neste trabalho tem como objetivo avaliar a compatibilidade entre duas versões quaisquer de um serviço em função de um perfil de uso construído previamente. A utilização deste perfil na avaliação da compatibilidade implica em examinar a compatibilidade considerando somente as *features* que estão contidas no mesmo.

O algoritmo para avaliação da compatibilidade orientada ao uso proposto neste trabalho é inspirado no algoritmo de avaliação de compatibilidade proposto por Yamashita et al. (2012).

O algoritmo foi adaptado para recuperar as versões do serviço do repositório de versões existente e avaliar a compatibilidade destas versões aplicando um perfil de uso do serviço. Na avaliação de compatibilidade serão consideradas as mudanças apresentadas na Tabela 3.1. Todas as mudanças não explicitadas serão consideradas incompatíveis.

O pseudo-algoritmo é apresentado nas Listagens 1 - 8. O algoritmo recebe como entrada o nome do serviço que deve ser avaliado, o número das versões relacionadas a este serviço e o perfil de uso que deve ser considerado para a avaliação da compatibilidade entre as duas versões de entrada. Assume-se como verdade que o nome do serviço informado é relacionado a um serviço válido, as duas versões informadas são versões existentes deste serviço e o perfil é um perfil válido. Não são realizadas verificações sobre a integridade destas informações no algoritmo proposto.

```
profileEvaluator(string serviceName, int v1, int v2, Profile profile)
1 begin
2       Boolean compatibility;
3       version svcProfile, service2;
4       svcProfile = buildProfileSvc(serviceName, v1, profile);
5       service2 = buildSvc(serviceName, v2);
6       compatibility = evaluateService(svcProfile, service2, profile, null);
7       return compatibility;
8 end
```

Listagem 1 – Pseudo-algoritmo

O primeiro passo do algoritmo (linha 4 na Listagem 1) realiza a construção de um grafo de versões baseado no nome do serviço (serviceName), na primeira versão passada como parâmetro (v1) e no perfil de uso informado (profile). A construção do grafo baseado no perfil de uso recupera do repositório de versões o grafo relacionado ao serviço com versão v1 e aplica o perfil de uso nele. Isso quer dizer que as *features* que não estão presentes no perfil não são inseridas no grafo e não serão avaliadas no restante da execução. Com isto, é construído um grafo que possui somente as versões das *features* que estão incluídas no perfil de uso.

Após a construção do grafo relacionado ao perfil, o algoritmo recupera do repositório de versões o grafo da segunda versão do serviço, v2, (linha 5 na Listagem 1) baseado no nome do serviço e na segunda versão passada como parâmetro. É importante destacar que é recuperado todo o grafo da segunda versão do serviço e não é aplicado o perfil de uso sobre o mesmo.

Com a construção dos grafos a partir dos dados informados, estão disponíveis as estruturas de dados necessárias para a avaliação da compatibilidade entre as duas versões com a utilização de um perfil. Para a avaliação da compatibilidade, propriamente dita, é invocada a função *evaluateService* (linha 6 na Listagem 1). Após a avaliação da compatibilidade pela função *evaluateService* é retornada a compatibilidade do serviço com relação as avaliações realizadas (retorno *true*: todas alterações realizadas na segunda versão do serviço são compatíveis com a primeira versão do mesmo levando-se em conta o perfil de uso; retorno *false*: pelo menos uma das alterações na segunda versão do serviço é incompatível com a primeira versão do mesmo levando-se em conta o perfil de uso).

```
evaluateService(version vp, version nv, profile prof, context cont)
1 begin
2    boolean compat <- true;
3    if(vp.version != nv.version)
4        compat = evaluateRemovedDependencies(vp, nv);
5        if(nv.isType())
6            compat = compat ^ evaluateRelation(nv,vp, cont)
7        endif
8        foreach e in nv.dependents() do
9            if nv.isOperation()
10               cont <- nv.inputOutput(e);
11           endif
12           pe = vp.findCorrespondent(e);
13           if (pe = null)
14               compat = compat ^ evaluateAddedDependencies(e,cont,nv);
15           elseif
16               compat = compat ^ evaluateService(pe,e,prof,cont);
17           endif
18       endforeach
19   setCompatibility(vp,nv,compat,prof);
20   endif
21   return compat;
22 end
```

Listagem 2 – avaliação de compatibilidade

A função *evaluateService*, detalhada na Listagem 2, tem como objetivo avaliar, recursivamente, as relações de compatibilidade dos dois nodos dos grafos construídos previamente. Ela recebe como parâmetro a raiz dos dois grafos citados (*svcProfile* e *service2),* o perfil de uso e o contexto relacionado a *feature* avaliada (relação de entrada ou saída de um parâmetro com sua operação). A função avalia a compatibilidade do último com relação ao primeiro. O grafo *svcProfile* é percorrido por um algoritmo de busca por profundidade a partir do nodo raiz.

O primeiro passo do algoritmo é criar a variável de compatibilidade para a avaliação de compatibilidade das versões de *feature* em questão (linha 2 na Listagem 2). Então, o algoritmo verifica se as versões das features são diferentes (linha 3 na Listagem2).

Quando temos versões diferentes de *features* com o mesmo nome, significa que esta *feature* sofreu alterações e foi criada uma outra versão. Portanto se as versões das *features* avaliadas forem diferentes são realizadas as verificações do impacto desta alteração no serviço que utiliza o perfil de uso em questão.

```
evaluateRemovedDependencies(version vp, version nv)
1 begin
2     Boolean compatibility <- true;
3     foreach e in vp.dependents() do
4         if !(nv.dependents() contains e)
5             compatibility <- false;
6         endif
7     endforeach
8     return compatibility;
9 end
```

Listagem 3 – avaliação de dependências removidas

O primeiro passo desta avaliação é verificar se alguma das dependências da primeira versão da *feature*, vp, foram removidas na nova versão, nv (linha 4 na Listagem 2). A função *evaluateRemovedDependencies*, detalhada na Listagem 3, verifica se todas as *features* no conjunto de dependentes da versão da *feature* vp ainda existem no conjunto de dependentes da versão da *feature*, nv. Se identificado que alguma *feature* foi removida das versões, vp e vn são consideradas incompatíveis, de acordo com os casos 7 ou 8 da Tabela 3.1. Desta função destaca-se a função *dependents()* que retorna todos os nodos dependentes da *feature* analisada.

O segundo passo da avaliação (linha 6 na Listagem 2) é verificar se foram efetuadas alterações compatíveis com relação a alterações na ordem ou cardinalidade do relacionamento de dependência e alterações relacionadas ao tipo básico de dados. Essa verificação somente é realizada se estivermos avaliando um tipo (linha 5 na Listagem 2)

A função *evaluateRelation*, detalhada na Listagem 4, recebe como parâmetro as duas versões da *feature* e o contexto. O contexto é a relação de dependência de um parâmetro relacionado a sua operação podendo ser entrada ou saída, a atribuição do contexto será detalhada mais adiante.

A primeira verificação realizada na função *evaluateRelation* é a ocorrência de alguma alteração relacionada à cardinalidade no relacionamento de dependência da *feature* (linha 3 na Listagem 4). A segunda verificação é a ocorrência de alguma alteração relacionada à ordem no relacionamento de dependência da *feature* (linha 4 na Listagem 4) e, por fim, é verificado se o tipo de dados foi alterado da primeira versão, vp, com relação a segunda, vn. As funções que apresentam o modo como são feitas estas avaliações serão detalhadas adiante. As verificações realizadas na função *evaluateRelation* buscam avaliar se foram realizadas alterações que compreendam os casos 4, 5 e 6 da Tabela 3.1.

```
evaluateRelation(version vp, version nv, context cont)
1 begin
2     Boolean compatibility <- true;
3         compatibility <- evaluateCardinality(vp, nv, cont);
4         compatibility <- compatibility ^ evaluatOrder(vp,nv);
5         compatibility <- compatibility ^ evaluateType(vp,nv);
6     return compatibility;
7 end
```

Listagem 4 – Avaliação de descrição

Após estas verificações o algoritmo continua percorrendo o grafo relacionado ao perfil de uso para realizar as verificações em todas as *features* relacionadas (linhas 7 a 12 da Listagem 2). A avaliação dos dependentes é realizada da nova versão com relação a anterior, é realizada desta forma para uma correta avaliação das dependências adicionadas.

O primeiro passo na avaliação dos dependentes realiza a atribuição de contexto da operação com o tipo relacionado ao parâmetro. Esta relação é propagada na chamada da função *evaluateService* para avaliação de relação na avaliação de tipos.

Após a atribuição do contexto é procurado o correspondente de e na primeira versão, vp (linha 12 na Listagem 2). Se for encontrado um correspondente o algoritmo continua avaliando o serviço de forma recorrente (linha 16 na Listagem 2).

Se não for encontrado significa que foi adicionada uma dependência e é invocada a função *evaluateAddedDependencies* (linha 14 na Listagem 2).

```
evaluateAddedDependencies(version e, context cont, version nv)
1 begin
2     Boolean compatibility <- true;
3     if nv.isMandatory(e) and e.isType()
4         compat <- incompatible
5     endif
6     if !nv.isMandatory(e) and e.isType() and context = output
7         compat <- incompatible
8     endif
9     return compatibility;
10 end
```

Listagem 5 – Avaliação de dependências obrigatórias adicionadas

A função *evaluateAddedDependencies*, detalhada na Listagem 5, realiza duas verificações que podem ser consideradas incompatíveis com relação a adição de dependências, a saber:

- Se for um tipo e for obrigatório;
- Se for um tipo relacionado a um parâmetro opcional de saída.

Se alguma destas condições for atingida é retornado que esta adição é incompatível. É importante destacar a função *isMandatory* na função *evaluateAddedDependencies*, esta função verifica se a *feature* em questão é declarada como obrigatória, se for é retornado *true*.

O último passo realizado pelo algoritmo é atribuir a compatibilidade das versões que estão sendo avaliadas, linha 20 da Listagem 2, e retornar a compatibilidade (linha 21 da Listagem 2), registrando o relacionamento de compatibilidade entre np, nv e perfil.

A seguir serão detalhadas algumas funções que não foram devidamente explicadas anteriormente.

```
evaluateCardinality(version vp, version nv, context cont)
1 begin
2     Boolean compatibility <- true;
3    if vp.getMinOccurs() > nv.getMinOccurs() then
4       compatibility <- false;
5    end if
6    if vp.getMaxOccurs() > nv.getMaxOccurs() then
7       compatibility <- false;
8    else if (vp. getMaxOccurs()  < nv. getMaxOccurs()) and (cont = output)
9       compatibility <- false;
10    end if
11    else if (vp. getMaxOccurs()  > nv. getMaxOccurs()) and (cont = input)
12       compatibility <- false;
13    end if
14    return compatibility;
15 end
```

Listagem 6 – Avaliação de cardinalidade

A função evaluateCardinality, Listagem 7, tem como objetivo avaliar se a cardinalidade do relacionamento de dependência do tipo em questão foi alterada para um valor incompatível na nova versão da *feature*, nv. Para isso, são realizadas verificações sobre a cardinalidade mínima (*minOccurs*) e a cardinalidade superior (*maxOccurs*) de cada tipo.

O algoritmo utiliza 4 funções básicas para realizar as avaliações, a saber:

- *getMinOccurs*() -> retorna a cardinalidade mínima do relacionamento de dependência do tipo analisado;
- *getMaxOcurrs*() -> retorna a cardinalidade máxima do relacionamento de dependência do tipo analisado.

O algoritmo ainda utiliza a informação de contexto recuperada na função evaluateService.

As verificações realizadas na função *evaluateCardinality* compreendem o caso 5 da Tabela 3.1.

```
evaluateOrder(version vp, version nv)
1 begin
2     Boolean compatibility <- true;
3     if vp.isOrderMandatory() ^ vp.getOrder() != nv.getOrder() then
4        compatibility <- false;
5     end if
6     return compatibility;
7 end
```

Listagem 7 – avaliação de ordem

A função evaluateOrder realiza a avaliação se a ordem do relacionamento de dependência dos parâmetros foi alterada. Na linha 3 da Listagem 6 temos a verificação da necessidade de ordem dos parâmetro pela função *isOrderMandatory*. A função *isOrderMandatory* avalia se a ordem de declaração é obrigatória para esta *feature*. Se a ordem for obrigatório é buscada qual a ordem de declaração da *feature* na primeira versão, *vp.getOrder()*, e comparada com a ordem de declaração da *feature* na segunda versão, *vp.getOrder() != nv.getOrder()*, se a ordem for alterada a compatibilidade entre as versões da *feature* é considerada incompatível. Essa verificação compreende o caso 6 da Tabela 3.1.

```
evaluateType(version vp, version nv)
1 begin
2    Boolean compatibility <- true;
3    if vp.type != nv.type then
4        compatibleTypes  = getCompatibleTypes(vp.type);
5            if nv.type ∉ compatibleTypes then
6                compatibility <- false;
7            end if
8    end if
10   return compatibility;
11 end
```

Listagem 8 – Avaliação de tipo básico

A função *evaluateType*, Listagem 8, realiza uma verificação se o tipo de dados da primeira versão da *feature* ,vp, foi alterado para um tipo compatível na nova versão, nv.

O primeiro passo realizado por esta função é a verificação se o tipo de dados foi alterado (linha 3 na Listagem 8). Se o tipo não for alterado não é necessário realizar a avaliação e é retornado que as *features* são consideradas compatíveis com relação ao tipo de dados. Se o tipo de dado for alterado é realizada a verificação se foi alterado para um tipo incompatível, para isso a função recupera uma lista de tipos de dados compatíveis com relação ao tipo de dado da versão da *feature* vp através do método getCompatibleTypes. Com a lista de tipos de dados compatíveis com a primeira versão da *feature* é verificado se o novo tipo está contido nesta lista (linha 5 na Listagem 8). Se o novo tipo de dado não estiver contido quer dizer que ele não é compatível e é retornado que esta alteração é incompatível. Esta verificação compreende o caso 6 da Tabela 3.1.

4 IMPLEMENTAÇÃO

Neste capítulo serão abordados detalhes sobre a implementação do algoritmo proposto assim como exemplos de excução do mesmo.

4.1 Arquitetura

A proposta desenvolvida por Yamashita et al. (2012) se apresenta hoje como um trabalho de pesquisa nesta universidade onde são desenvolvidos os módulos propostos e também a melhoria dos módulos já existentes. Como estes trabalhos estão todos sendo desenvolvidos na linguagem Java utilizando-se de um banco de dados NoSQL (Not Only SQL) orientado a grafos, optou-se por continuar utilizando este conjunto de tecnologias. A utilização de outras tecnologias acarretaria em migrações de código e banco dos dados à medida que a manutenção das tecnologias utilizadas possibilitaria o reuso dos componentes já desenvolvidos.

Tendo a definição das tecnologias a serem utilizadas foi realizado um estudo sobre as tecnologias utilizadas pelo repositório de versões para o desenvolvimento do algoritmo. A seguir são apresentadas estas tecnologias assim como os detalhes pertinentes ao desenvolvimento do algoritmo em si. Não será realizado o detalhamento da linguagem Java por ser uma tecnologia conhecida e com uma quantidade grande de documentação disponível.

4.1.1 Repositório

A primeira proposta de implementação do repositório de versões criada pelo grupo de pesquisa se apresentava em um arquivo XML que segue um esquema pré-definido. Verificou-se ao longo do tempo que a utilização deste modelo era ineficiente e resultavam em duas grandes dificuldades: a dificuldade de extrair as informações das versões dos serviços para outros componentes do framework e a grande ineficiência para criação e recuperação destas informações.

Para suprir estas dificuldades foi realizada a implementação de uma segunda proposta com a utilização de um banco de dados NoSQL baseado em grafo. Neste tipo de banco de dados os dados são armazenados em nós de um grafo com associações entre os dados representadas através de suas arestas (BRITO, 2010). Este tipo de representação facilitou o acesso às informações do repositório, tendo a eficiência aumentada significativamente. O repositório continuou sendo representado conforme o modelo proposto acrescido de uma eficiência muito melhor.

O grupo de pesquisa optou pelo banco de dados OrientDB. O OrientDB é disponibilizado sob a licença Apache2 e é um banco de dados leve onde o servidor ocupa um espaço muito pequeno. Este banco de dados ainda apresenta uma facilidade muito interessante: é possível realizar consultas no banco com a linguagem SQL (*Structured Query Language)*. A realização de consultas com a linguagem SQL facilitou muito o trabalho de desenvolvimento, pois não foi necessário aprender a utilização de novas sintaxes.

4.1.1.1 Representação do repositório no banco de dados

No banco de dados as estruturas do grafo são representadas por objetos que, por sua vez, representam alguma propriedade do grafo. A seguir será apresentado o objeto VertexBaseClass. Este objeto possui informações sobre todo o grafo do serviço. A partir deste objeto podem ser recuperadas todas as informações pertinentes para avaliação da compatibilidade entre versões.

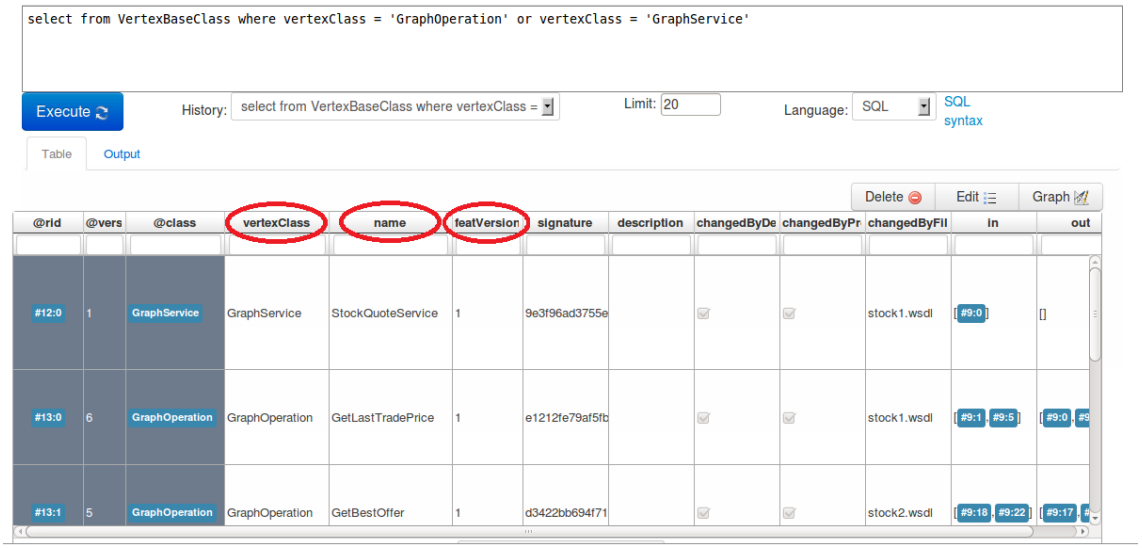

Figura 4.1 – Exemplo de execução de uma consulta

Como pode ser observado na Figura 4.1, como resultado de uma consulta por todas as informações contidas no objeto VertexBaseClass, são apresentados as informações sobre o serviço que foram mapeadas para o repositório. Nesta figura destacam-se as informações marcadas em vermelho, a saber:

- vertexClass: classe do vértice em questão, que pode ser serviço, operação ou tipo;
- name: o nome do objeto, que pode ser o nome do serviço, da operação ou da descrição do tipo;
- featVersion: a versão à qual este vértice está associado.

Um exemplo prático que pode ser apresentado na representação neste banco de dados é a recuperação das operações do serviço StockQuote relacionados a segunda versão do serviço. Este exemplo segue a representação da Figura 4.2.

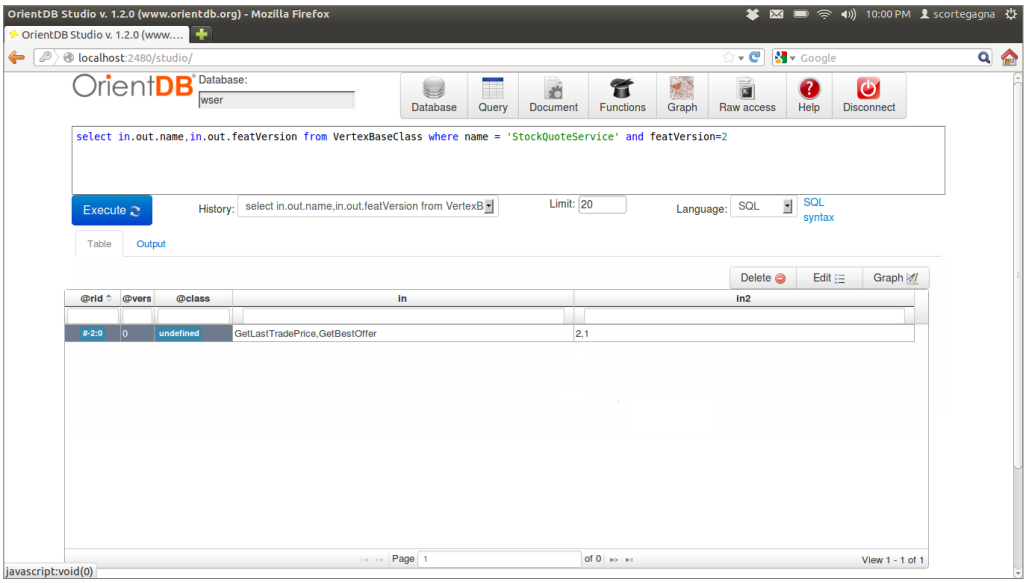

Figura 4.2 – Exemplo de consulta de dependência

Na parte superior desta figura temos a consulta realizada no banco para a recuperação das operações relacionadas ao serviço StockQuote. Na parte inferior da figura é apresentada uma tabela com a lista de operações que são relacionadas ao serviço e uma lista com a versão destas operações. Cada elemento na primeira lista é um nodo do vértice e cada elemento na segunda é a versão associada ao nodo correspondente. Neste momento é importante salientar que foi realizado um *parser* destes dados para a utilização dos mesmos.

Podemos observar, a partir da Figura 4.2, que a recuperação de informações do banco de dados é relativamente simples e esta facilidade auxiliou na recuperação de informações de informações relacionadas a compatibilidade de versões.

4.1.2 Perfil de uso

A criação de perfis de uso pelo módulo de gerenciamento de perfis proposto por Yamashita et al. (2012) apresenta todo o processo para a criação de um perfil de uso, assim como também apresenta uma estrutura para a representação do mesmo. Até a data de conclusão deste trabalho o módulo de gerenciamento de perfis ainda estava em desenvolvimento e, portanto, optou-se por desenvolver uma estrutura própria para a representação do perfil de uso.

nomeDoPerfil.perfil
StockQuote,2,GetLastTradePrice,2,TradePriceRequest,2,TradePriceResponse,2

Listagem 8 – Representação do perfil de uso

A estrutura desenvolvida é uma estrutura simples contendo somente as informações básicas necessárias para a representação do perfil. O modelo apresentado na Listagem 8 é a representação da estrutura utilizada. A estrutura é composta de uma entrada com o nome do perfil, definido pelo usuário, que é o nome do arquivo do perfil de uso.

O perfil é uma lista aonde o primeiro valor será o nome do serviço utilizado e o segundo valor o número da versão do serviço utilizado. O restante da lista que consiste o perfil de uso são as versões das *features* relacionadas à aquele serviço selecionadas para fazer parte do perfil, onde cada uma é seguida do seu número de versão.

Tem-se como premissa que os dados inseridos nesta estrutura são dados válidos e de que são inseridos na ordem correta (Nome do serviço, versão, *features*). Não é verificada a integridade dos dados nem a validação dos mesmos perante o serviço que será avaliado.

4.1.3 Funções relacionadas ao algoritmo

Algumas funções relacionadas ao algoritmo serão detalhadas nesta seção. As funções escolhidas são as funções que tem um relacionamento como banco de dados.

1. dependents()

Esta função tem como objetivo buscar os dependentes de uma *feature* para avaliações de compatibilidade sobre estes dependentes. No banco de dados escolhido a busca pelo dependentes é realizada com uma consulta simples.

Supõe-se que em um serviço hipotético seria necessário saber quais operações são dependentes de um serviço. Supõe-se que este serviço tem o nome de *StockQuoteService* e na sua primeira versão possui somente a operação *GetLastTradePrice.*

A consulta a ser realizada no banco é a seguinte:

Select in.out.name from GraphService where name = ‘StockQuoteService’ and featVersion = ‘1’

Esta consulta retorna o nome dos filhos relacionados ao serviço StockQuote como pode ser observado na Figura 4.3.

Figura 4.3 – Exemplo de consulta da função *dependents*

2. getMinOccurs()

Esta função tem como objetivo buscar a relação de cardinalidade mínima da *feature* analisada.

Supõe-se que em um serviço hipotético gostaria de saber qual a cardinalidade mínima de um parâmetro de uma determinada operação. Supõe-se que este serviço tem nome *StockQuoteService,* que exista a operação *GetLastTradePriceRequest* e o parâmetro *tradePrice* tenha cardinalidade mínima igual a 5.

A consulta a ser realizada no banco é a seguinte:

select out[in.name=’TradePrice’].out.name, out.in.minOccurs from GraphOperation where name = ‘GetLastTradePriceRequest’

Esta consulta retorna a cardinalidade mínima do parâmetro *tradePrice* relacionado a operação GetLastTradePriceRequest, como pode ser observado na figura 4.4.

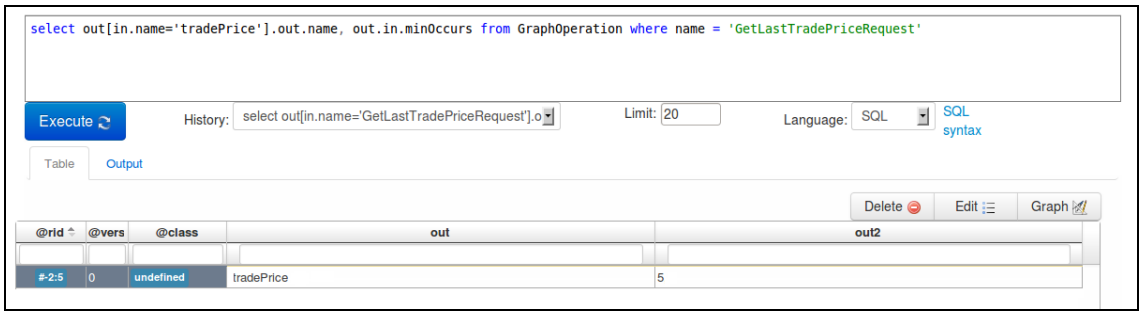

Figura 4.4 – Exemplo de consulta da função *getMinOccurs*

4.2 Estudo de caso

Uma versão do algoritmo proposto foi implementada para realizar a verificação da compatibilidade de versões de um serviço levando-se em conta um perfil de uso, o perfil de uso criado é hipotético e foi criado empiricamente. Esta versão apresenta um relatório com as *features* que foram consideradas incompatíveis e um breve detalhamento do motivo pelo qual foi detectada esta incompatibilidade. Para provar a eficiência do algoritmo e a sua utilização são apresentados exemplos de execução levando-se em conta os objetivos deste trabalho.

4.2.1 Serviço StockQuote

4.2.1.1 O Serviço

O serviço StockQuote é um serviço didático no qual poderemos observar as mudanças realizadas mais facilmente por se tratar de um serviço pequeno e específico. Serão apresentadas duas versões do serviço e apresentadas as alterações realizadas entre as versões. As descrições do serviço utilizam WSDL 1.6 utilizando XML Schema (XSD).

A primeira versão do serviço que será utilizada é a versão representada nas Figuras 4.5 e 4.6. A descrição do serviço StockQuote possui duas operação (*GetLastTradePrice, GetFirstTradePrice*), quatro trocas de mensagens (*GetLastTradePriceRequest, GetLastTradePriceResponse,GetFirstTradePriceRequest, GetFirstTradePriceResponse*) e quatro tipos complexos de dados (*TradePriceRequest, TradePriceResponse, FirstTradeRequest, FirstTradeResponse*), é utilizada uma ponte para vincular as operações (bind) aos seus tipos.

```
<wsdl:definitions name="StockQuote" targetNamespace="http://www.example.org/stockQuote1.wsd
    <wsdl:types>
        <xsd:schema targetNamespace="http://www.example.org/teste/">
            <xsd:element name="TradePriceRequest">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="tickerSymbol" type="xsd:string"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="TradePriceResponse">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd:string"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="FirstTradeRequest">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="tickerSymbol" type="xsd:string"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="FirstTradeResponse">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd:int"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="GetLastTradePriceRequest">
        <wsdl:part name="body" element="tns:TradePriceRequest" />
    </wsdl:message>
    <wsdl:message name="GetLastTradePriceResponse">
        <wsdl:part name="body" element="tns:TradePriceResponse" />
    </wsdl:message>
    <wsdl:message name="GetFirstTradePriceRequest">
        <wsdl:part name="body" element="tns:FirstTradeRequest" />
    </wsdl:message>
    <wsdl:message name="GetFirstTradePriceResponse">
        <wsdl:part name="body" element="tns:FirstTradeResponse" />
    </wsdl:message>
    <wsdl:portType name="StockQuotePortType">
        <wsdl:operation name="GetLastTradePrice">
            <wsdl:input message="tns:GetLastTradePriceRequest" />
            <wsdl:output message="tns:GetLastTradePriceResponse" />
        </wsdl:operation>
        <wsdl:operation name="GetFirstTradePrice">
            <wsdl:input message="tns:GetFirstTradePriceRequest" />
            <wsdl:output message="tns:GetFirstTradePriceResponse" />
        </wsdl:operation>
    </wsdl:portType>
```

Figura 4.5 – Serviço StockQuote

```
    <wsdl:binding name="StockQuoteBinding" type="tns:StockQuotePortType">
        <soap:binding style="document" />
        <wsdl:operation name="GetLastTradePrice">
            <soap:operation soapAction="http://www.example.org/GetLastTradePrice" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="GetFirstTradePrice">
            <soap:operation soapAction="http://www.example.org/GetFirstTradePrice" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuotePort" binding="tns:StockQuoteBinding" >
            <soap:address location="http://www.example.org/stockQuote" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figura 4.6 – Serviço *StockQuote* (continuação)

A representação do serviço na sua primeira versão no repositório é apresentada na forma de um grafo (Figura 4.7).

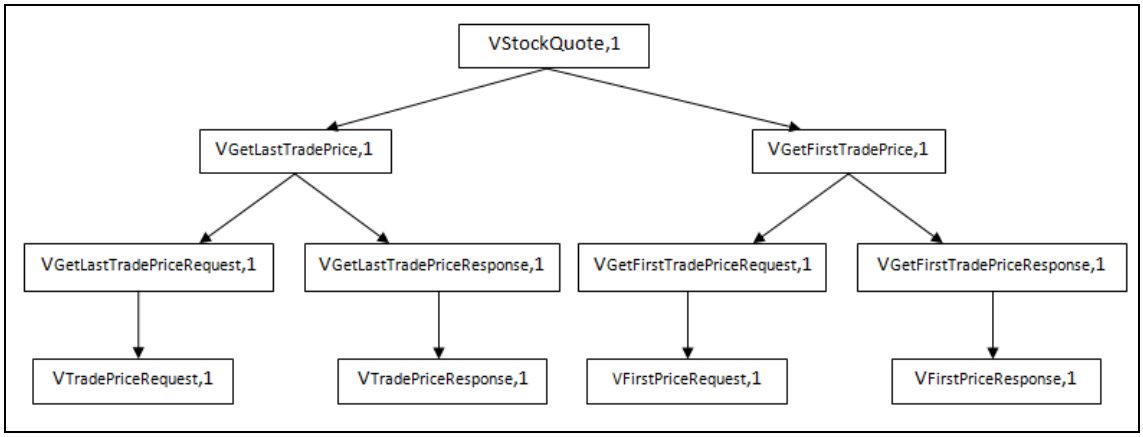

Figura 4.7 – Grafo de dependência da primeira versão do serviço *StockQuote*

Suponhamos que foi apresentada uma nova versão do serviço *StockQuote*, com algumas modificações, a saber:

1. Adição de uma nova operação;
2. Alteração do tipo de dados relacionado ao elemento *GetLastTradePriceResponse*;

3. Alteração da cardinalidade máxima do elemento *tickerSimbol* de 1 para 2 (1 é o valor *default* quando *maxOccurs* não é declarado (W3C, 2004)).

As alterações realizadas no serviço podem ser verificadas nas Figuras 4.8, 4.9 e 4.10, com a seguinte legenda:

1. Amarelo = adição de operação
2. Verde = alteração do tipo de dados
3. Roxo = alteração na cardinalidade

```
<wsdl:definitions name="StockQuote" targetNamespace="http://www.example.org/stockQuote1.wsdl">
    <wsdl:types>
        <xsd:schema targetNamespace="http://www.example.org/teste/">
            <xsd:element name="TradePriceRequest">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="tickerSymbol" type="xsd:string" maxOccurs="2"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="TradePriceResponse">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd:integer"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="FirstTradeRequest">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="tickerSymbol" type="xsd:string"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="FirstTradeResponse">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd:int"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="BestOfferRequest">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd:string"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="BestOfferResponse">
                <xsd:complexType>
                    <xsd:all>
                        <xsd:element name="price" type="xsd: integer"></xsd:element>
                    </xsd:all>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
```

Figura 4.8 – Segunda versão do serviço *StockQuote*

```
<wsdl:message name="GetLastTradePriceRequest">
    <wsdl:part name="body" element="tns:TradePriceRequest" />
</wsdl:message>
<wsdl:message name="GetLastTradePriceResponse">
    <wsdl:part name="body" element="tns:TradePriceResponse" />
</wsdl:message>
<wsdl:message name="GetFirstTradePriceRequest">
    <wsdl:part name="body" element="tns:FirstTradeRequest" />
</wsdl:message>
<wsdl:message name="GetFirstTradePriceResponse">
    <wsdl:part name="body" element="tns:FirstTradeResponse" />
</wsdl:message>
<wsdl:message name="GetBestOfferRequest">
    <wsdl:part name="body" element="tns:BestOfferRequest" />
</wsdl:message>
<wsdl:message name="GetBestOfferResponse">
    <wsdl:part name="body" element="tns:BestOfferResponse" />
</wsdl:message>
<wsdl:portType name="StockQuotePortType">
    <wsdl:operation name="GetLastTradePrice">
        <wsdl:input message="tns:GetLastTradePriceRequest" />
        <wsdl:output message="tns:GetLastTradePriceResponse" />
    </wsdl:operation>
    <wsdl:operation name="GetFirstTradePrice">
        <wsdl:input message="tns:GetFirstTradePriceRequest" />
        <wsdl:output message="tns:GetFirstTradePriceResponse" />
    </wsdl:operation>
    <wsdl:operation name="GetBestOffer">
        <wsdl:input message="tns:GetBestOfferRequest" />
        <wsdl:output message="tns:GetBestOfferResponse" />
    </wsdl:operation>
</wsdl:portType>
```

Figura 4.9 – Segunda versão do serviço *StockQuote* (continuação)

```
    <wsdl:binding name="StockQuoteBinding" type="tns:StockQuotePortType">
        <soap:binding style="document" />
        <wsdl:operation name="GetLastTradePrice">
            <soap:operation soapAction="http://www.example.org/GetLastTradePrice" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="GetFirstTradePrice">
            <soap:operation soapAction="http://www.example.org/GetFirstTradePrice" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="GetBestOffer">
            <soap:operation soapAction="http://www.example.org/GetBestOffer" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuotePort" binding="tns:StockQuoteBinding" >
            <soap:address location="http://www.example.org/stockQuote" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figura 4.10 – Segunda versão do serviço *StockQuote* (continuação)

A representação do serviço *StockQuote* na sua segunda versão no repositório de versões é apresentada na Figura 4.11.

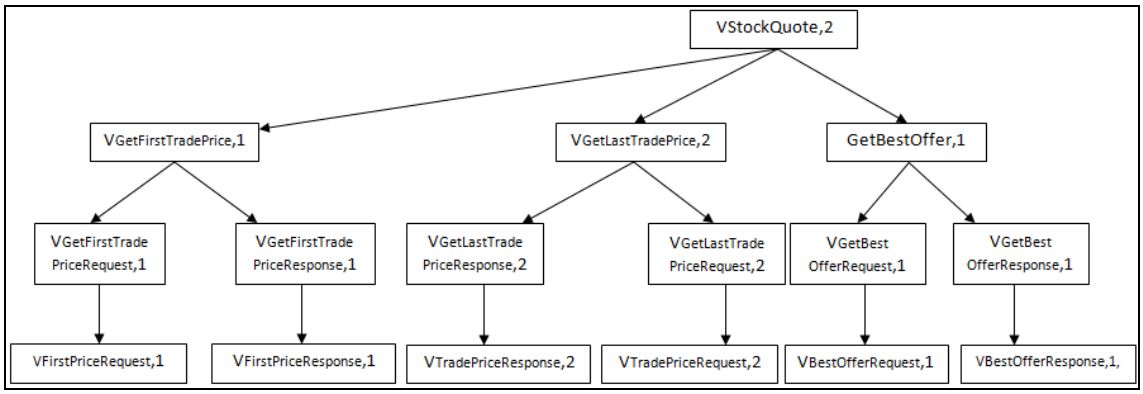

Figura 4.11 – Grafo de dependência da segunda versão do serviço *StockQuote*

Para a elaboração do exemplo serão utilizadas estas duas versões do serviço. A seguir serão apresentados os perfis de uso utilizados na execução do algoritmo.

4.2.1.2Perfis de uso

Para a execução do algoritmo sobre versões do serviço StockQuote serão apresentados perfis de uso sobre a versão 1 do serviço StockQuote. Os perfis de uso foram criados empiricamente somente para este exemplo.

Perfil 1

O primeiro perfil de uso inclui o serviço StockQuote em sua primeira versão, a operação GetFirstTradePrice e os tipos relacionados. Com isso temos:

Perfil1.perfil
StockQuote,1,GetFirstTradePrice,1,GetFirstTradePriceRequest,1, GetFirstTradePriceResponse,1

Perfil 2

O segundo perfil de uso inclui o serviço StockQuote em sua primeira versão, a operação GetLastTradePrice e os tipos relacionados. Com isso temos:

Perfil2.perfil
StockQuote,1,GetLastTradePrice,1,GetLastTradePriceRequest,1,GetLastTradePriceResponse,1

4.2.1.3Execução do algoritmo

4.2.1.3.1 Perfil 1, versão 1 e versão 2

A primeira execução do algoritmo será relacionada ao perfil de uso Perfil1 e às versões 1 e 2 do serviço *StockQuote*.

O primeiro passo que o algoritmo realiza é construção do grafo do perfil e a recuperação de todo o grafo da segunda versão passada como parâmetro. Neste exemplo será utilizado o Perfil 1, apresentado na seção anterior, e as versões 1 e 2 do serviço *StockQuote.*

A construção do grafo para avaliação de compatibilidade do perfil de uso é realizada a partir da recuperação do grafo de dependência da primeira versão considerada. Após a recuperação do grafo da primeira versão é realizada uma avaliação sobre quais versões de *features* estão presentes no grafo e no perfil. As versões que estão presentes no perfil são mantidas e aquelas que não estão são removidas.

Neste primeiro caso o grafo de dependência da primeira versão passada como parâmetro é o grafo da primeira versão do serviço *StockQuote*, Figura 4.7. Como pode ser observado o Perfil 1 não utiliza todas as *features* relacionadas a esta versão do serviço e, portanto, é realizada a eliminação dos vértices que não estão no perfil.

Na Figura 4.12 é apresentada a poda do grafo recuperado com a aplicação do perfil de uso selecionado. Na figura 4.13 é apresentado o grafo gerado a partir das informações do perfil.

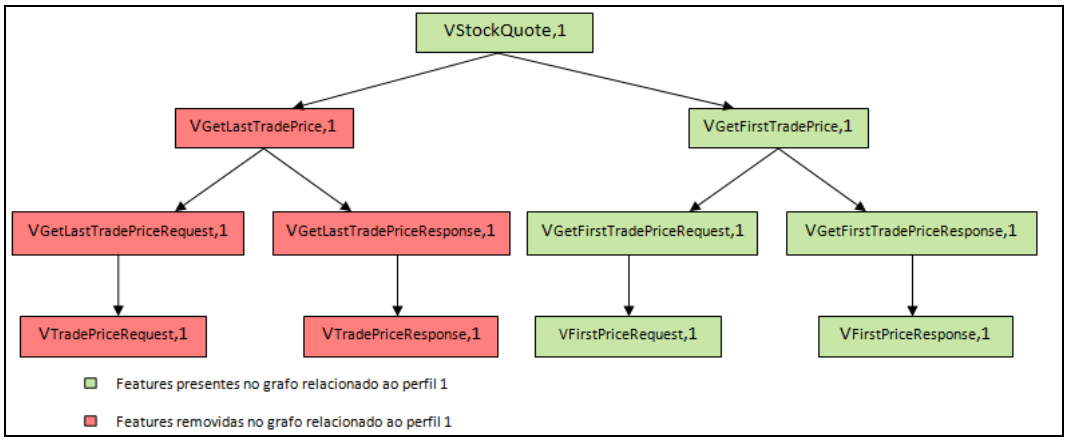

Figura 4.12 – Construção grafo relacionado ao perfil

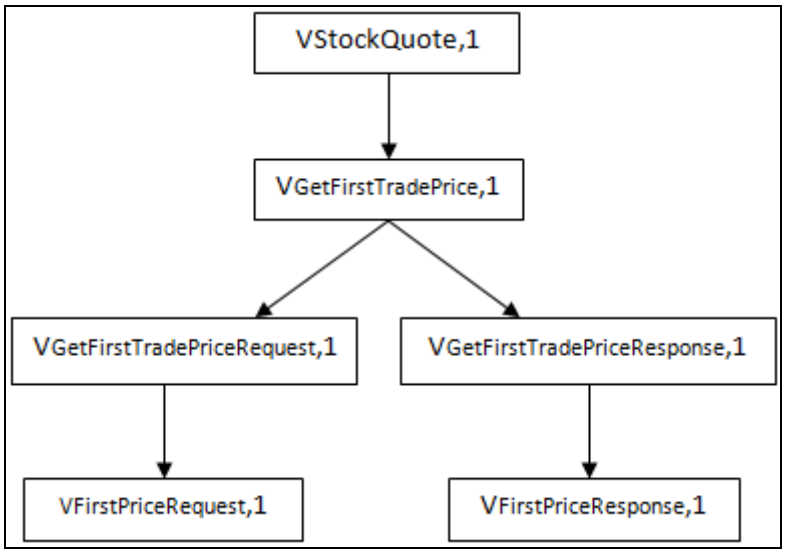

Figura 4.13 – Grafo de representação do perfil 1

Com relação a recuperação do grafo relacionado a segunda versão do serviço não são realizados filtros na recuperação, ou seja, o serviço na versão informada é recuperado em sua totalidade.

Com a recuperação dos dois grafos, o algoritmo invoca a chamada para avaliação de compatibilidade. Como já foi citado, para cada *feature* do serviço relacionado ao perfil é avaliado se existe uma versão diferente desta *feature* na outra versão do serviço (linha 3 na Listagem 2). Observando os grafos de representação dos serviços em questão (Figura 4.11 e Figura 4.13) observa-se que não existem novas versões das *features* do primeiro serviço com relação ao segundo.

Com isso a execução do algoritmo retorna que as versões do serviço são compatíveis levando-se em conta o perfil de uso do cliente informado. O resultado da execução pode ser observado na Figura 4.14.

Analisando esta primeira execução do algoritmo observa-se que a execução levando-se em conta um perfil de uso torna a avaliação mais específica. Se somente fossem verificadas as versões do serviço, sem a utilização de um perfil de uso, possivelmente o

utilizador deste serviço seria informado que as versões são incompatíveis. Se o perfil de uso for levado em conta, Perfil1 no exemplo, as versões são consideradas compatíveis pois as *features* que ele utiliza deste serviço não sofreram alterações na nova versão, logo não afeta os clientes representados por este perfil.

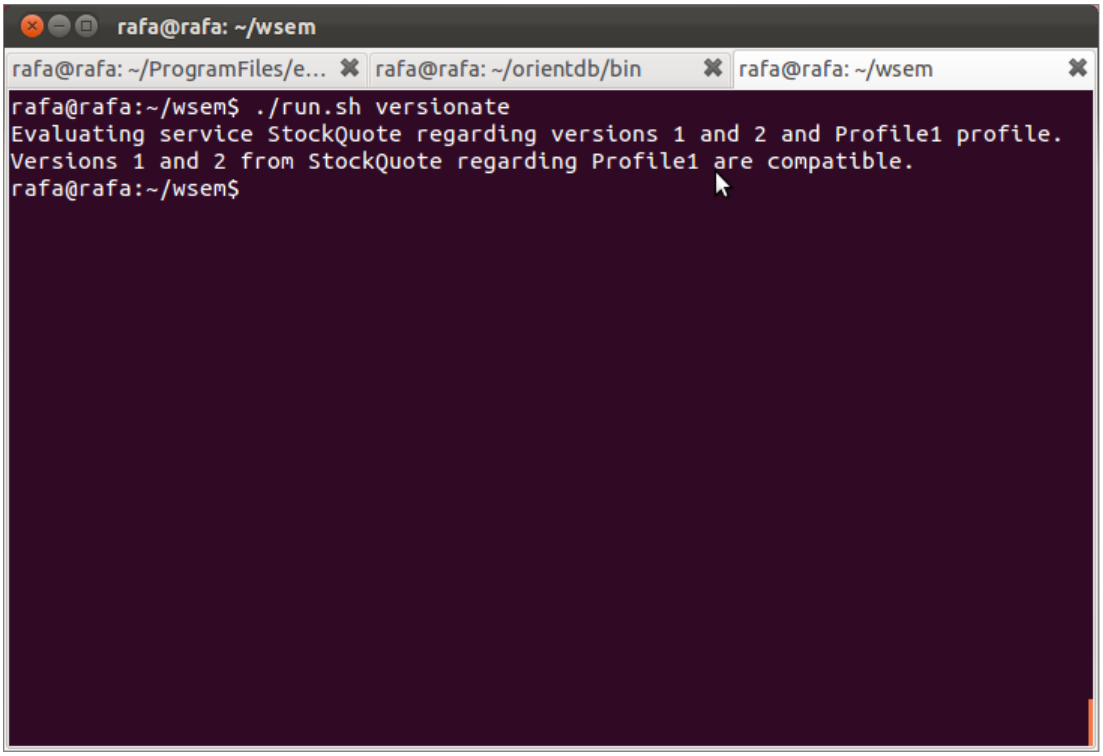

Figura 4.14 – Execução do algoritmo com perfil1 e versões 1 e 2 do serviço *StockQuote*

4.2.1.3.2 Perfil 2, versão 1 e versão 2 sem flexibilização de regras

A segunda execução do algoritmo será relacionada ao perfil de uso Perfil2 e às versões 1 e 2 do serviço *StockQuote*. Para esta execução não será levado em conta a flexibilização das regras de avaliação, Tabela 3.1. Será realizada a aplicação de um perfil de uso na primeira versão do serviço e as regras de avaliação utilizadas serão as propostas por Yamashita et al. (2012), Tabela 2.3.

O primeiro passo que o algoritmo realiza é a construção do grafo da primeira versão do serviço com a aplicação de um perfil de uso sobre o mesmo e a recuperação do grafo da segunda versão do serviço. Primeira e segunda versão citadas aqui são as versões passadas como parâmetros na chamada do algoritmo na ordem correspondente. De acordo com o processo de recuperação dos grafos, os grafos das Figuras 4.11 e 4.16 serão comparados.

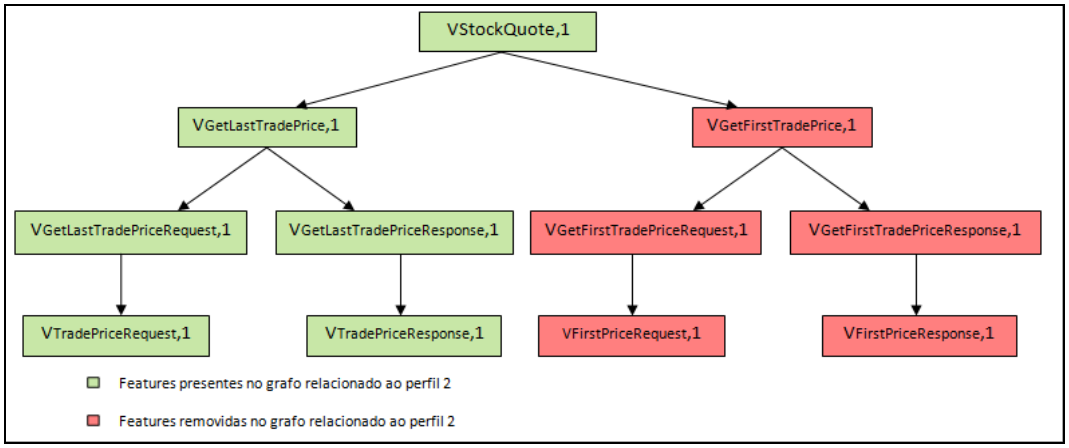

Figura 4.15 – Construção grafo relacionado ao perfil

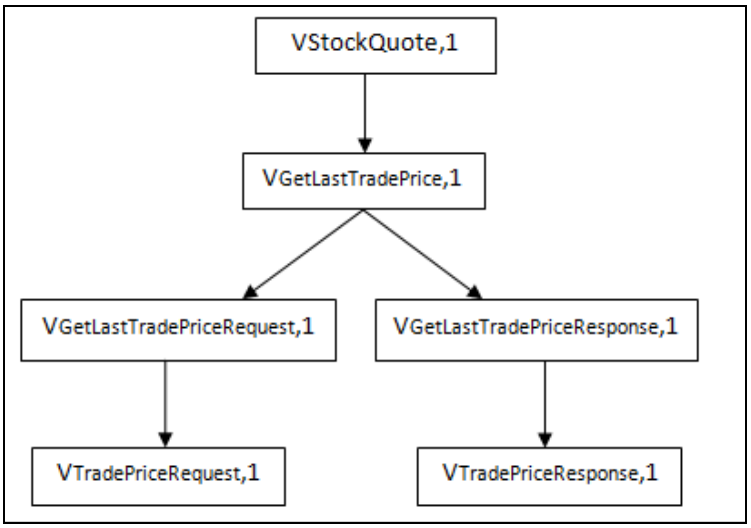

Figura 4.16 – Grafo de representação do perfil 2

Com a recuperação dos dois grafos o algoritmo invoca a chamada para avaliação de compatibilidade. Como já foi citado para cada *feature* do serviço relacionado ao perfil é avaliado se existe uma versão diferente desta *feature* na outra versão do serviço (linha 3 na Listagem 2). Analisando os grafos de representação dos dois serviços em questão (Figura 4.11 e Figura 4.16) observamos que as *features* possuem versões diferentes, portanto é realizada a avaliação de compatibilidade entre elas.

Após a realização da analise de compatibilidade entre as *features* o algoritmo retorna que as versões do serviço são incompatíveis (Figura 4.17) e proporciona um log de execução (Figura 4.18) com informações sobre as incompatibilidades encontradas.

Nesta execução podemos observar que, apesar de utilizar o perfil de uso, o algoritmo encontrou incompatibilidades levando-se em conta as regras de Yamashita et al. (2012). A seguir, são apresentadas as (in)compatibilidades encontradas levando-se em conta as regras utilizadas.

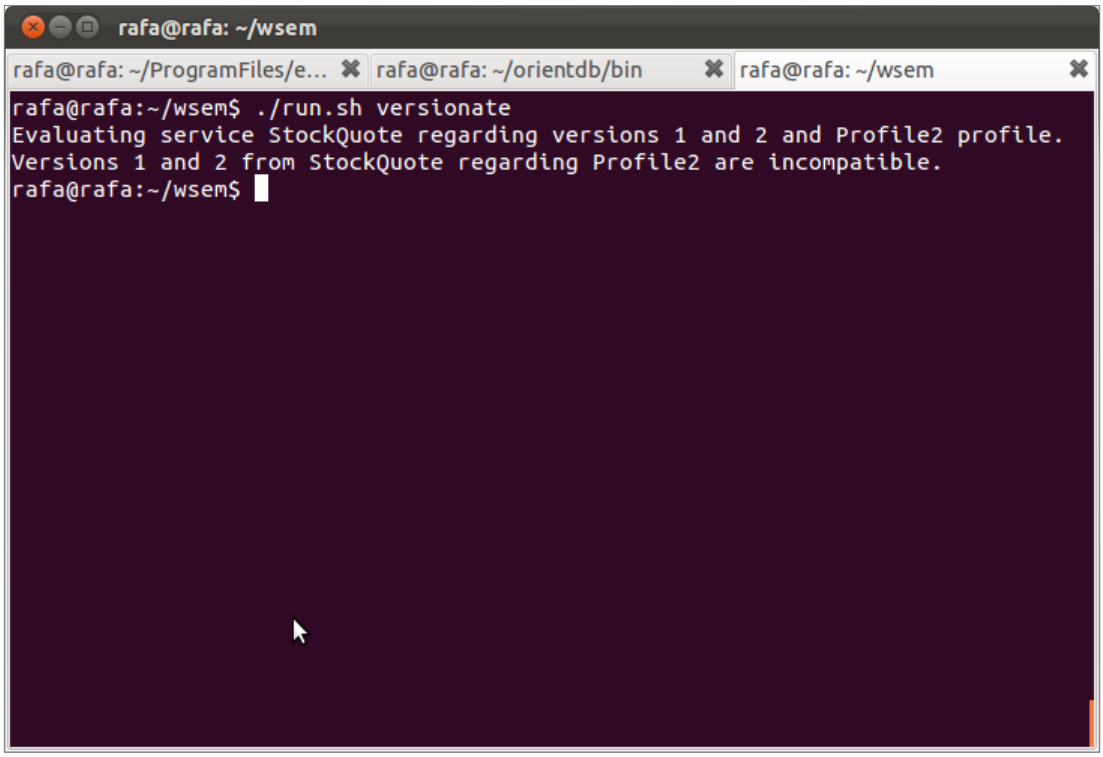

Figura 4.17 – Execução do algoritmo com perfil2 e versões 1 e 2 do serviço *StockQuote* sem flexibilização das regras

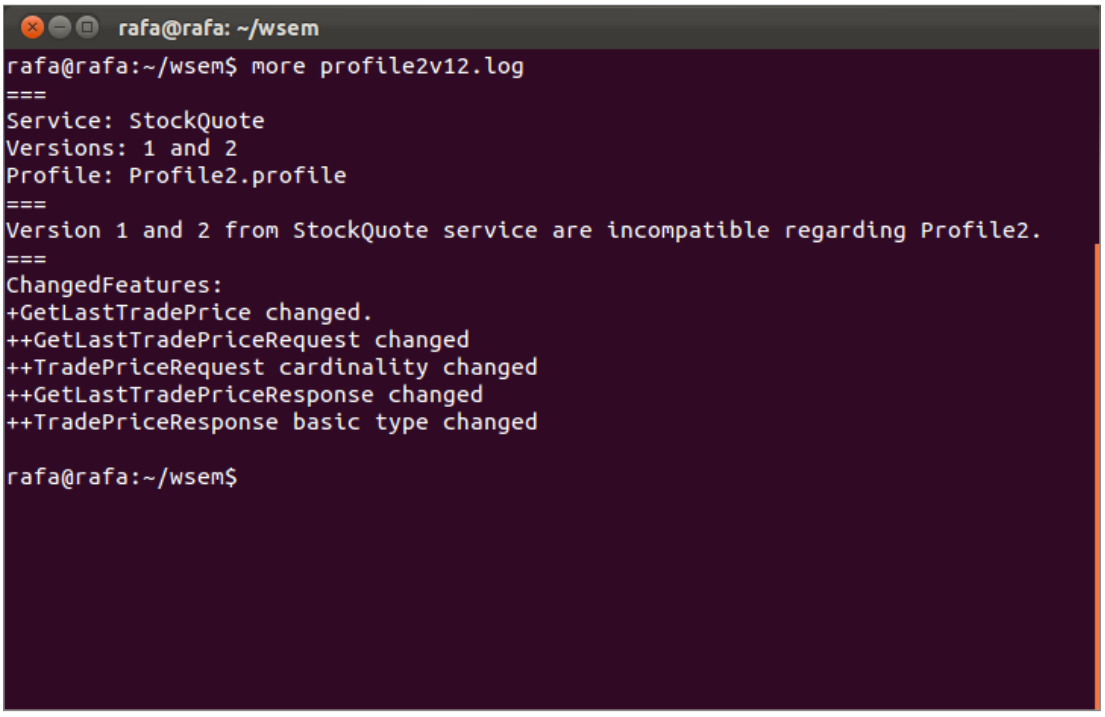

Figura 4.18 – Log de execução do algoritmo

4.2.1.3.3 Perfil 2, versão 1 e versão 2 com flexibilização das regras

A terceira execução do algoritmo será relacionada ao perfil de uso Perfil2 e às versões 1 e 2 do serviço *StockQuote.* Para esta execução será realizada a flexibilização das regras de avaliação, Tabela 3.1, juntamente com a aplicação de um perfil de uso na primeira versão do serviço.

O primeiro passo que o algoritmo realiza é a construção do grafo da primeira versão do serviço com a aplicação de um perfil de uso sobre o mesmo e a recuperação do grafo da segunda versão do serviço. Primeira e segunda versão citadas aqui são as versões passadas como parâmetros na chamada do algoritmo na ordem. O processo de recuperação da primeira versão do serviço com a aplicação do perfil de uso é o mesmo da segunda execução, resultando no serviço representado na Figura 4.16.

Com a recuperação dos dois grafos o algoritmo invoca a chamada para avaliação de compatibilidade. Como já foi citado para cada *feature* do serviço relacionado ao perfil é avaliado se existe uma versão diferente desta *feature* na outra versão do serviço (linha 3 na Listagem 2). Analisando os grafos de representação dos dois serviços em questão (Figura 4.11 e Figura 4.16) observamos que as *features* possuem versões diferentes, portanto é realizada a avaliação de compatibilidade entre elas.

Após a realização da analise de compatibilidade entre as *features* o algoritmo retorna que as versões do serviço são compatíveis, Figura 4.19.

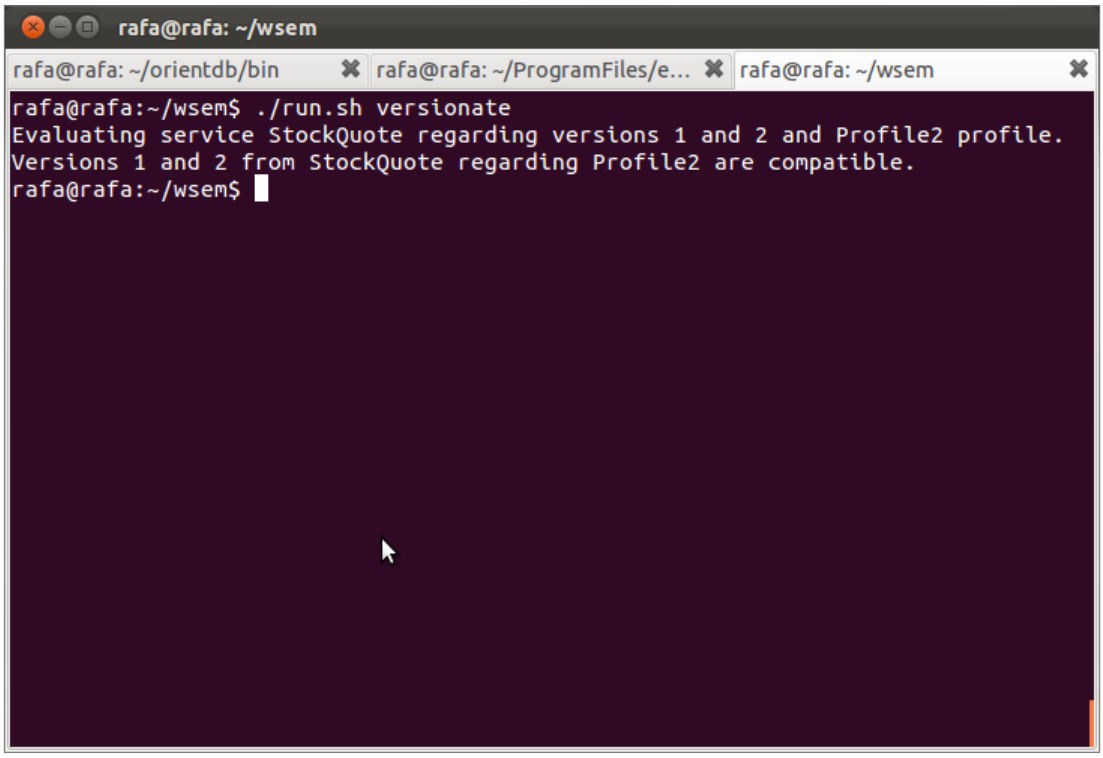

Figura 4.19 - Execução do algoritmo com perfil2 e versões 1 e 2 do serviço *StockQuote* com flexibilização das regras

Nesta execução, observa-se que a flexibilização das regras de compatibilidade se mostram relevantes no momento da avaliação de compatibilidade. Podemos observar

que apesar das mudanças realizadas nas *features*, estas mudanças não são incompatíveis. A saber:

1. Alteração da cardinalidade máxima em GetLastTradePriceRequest: a cardinalidade máxima foi alterada para um valor superior, e por se tratar de um parâmetro de entrada de dados a alteração é considerada compatível, caso 5 da Tabela 3.1;
2. Alteração do tipo básico: alteração do tipo básico de *Positive Integer* para *Integer*, alteração considerada compatível pois o tipo *Integer* engloba o tipo *Positive Integer*, caso 6 da Tabela 3.1.

5 CONCLUSÃO

Este trabalho teve como objetivos traçados a realização da avaliação de compatibilidade orientada a um perfil de uso e a utilização de regras de compatibilidade mais flexíveis na avaliação de compatibilidade entre versões de um serviço. O trabalho desenvolvido para a realização destes objetivos foi baseado na proposta apresentada por Yamashita et al. (2012). A utilização do trabalho de Yamashita et al. (2012) foi muito importante pois ele apresenta os elementos básicos para o desenvolvimento das tarefas relacionadas a este trabalho.

A existência do repositório de versões em uma versão funcional foi vital para o desenvolvimento do trabalho assim como a escolha das tecnologias utilizadas, com um destaque especial para a utilização do banco de dados OrientDB, a utilização da linguagem Java, assim como a utilização da IDE Eclipse para o auxilio no desenvolvimento.

O resultado do desenvolvimento foi a implementação do algoritmo apresentado no Capítulo 3. Algumas funções apresentadas no capítulo 3 não foram implementadas na sua totalidade, pois não houve tempo hábil para o desenvolvimento. Entretanto, o aperfeiçoamento da implementação destas funções ficam como propostas de objetivos para trabalhos a serem realizados no futuro.

Acredito que os objetivos propostos por este trabalho foram alcançados, pois os resultados obtidos pelo algoritmo desenvolvido foram satisfatórios. A utilização de perfis de uso e a flexibilização das regras de avaliação na análise de compatibilidade se mostraram eficientes quando realizada a avaliação de compatibilidade de um serviço levando-se em conta estes fatores.

6 REFERÊNCIAS

EBAY TRADING EPI, 2012. Disponível em: < https://www.x.com/developers/ebay/products/trading-api >. Acesso em: ago/2012.

YAMASHITA, M.; VIOLLINO, B.; BECKER K.; GALANTE R. **Measuring Change Impact Based on Usage Profiles.** Web Services (ICWS), 2012 IEEE 19th International Conference on, p. 226-233, June. 2012.

YAMASHITA, M.; BECKER K.; GALANTE R. **A Flexible Approach for Accessing Service Compatibility at Feature Level**. Simpósio Brasileiro de Banco de Dados, Florianópolis, p. 105-112, Out. 2011.

BECKER, K. et al. **Automatic Determination of Compatibility in Evolving Services**. International Journal of Web Service Research, Hershey, v.8, n.1, p. 21-40, Jan./Mar. 2011.

ANDRIKOPOULOS, V.; BENBERNOU, S.; PAPAZOGLOU, M. **On the Evolution of Services**, IEEE Transactions on Software Engineering, p. 609-628, Mar. 2011.

BRITO, Ricardo W.. **Bancos de Dados NoSQL x SGBDs Relacionais: Análise**

Comparativa. Faculdade Farias Brito e Universidade de Fortaleza, 2010. Disponível em : < http://www.infobrasil.inf.br/userfiles/27-05-S4-1-68840-Bancos%20de%20Dados%20NoSQL.pdf >. Acesso em: set/2012.

PAPAZOGLOU, M. **The challenges of service evolution**. Advanced Information systems Engineering. Springer, pp.1_15, 2008.

FANG, R. et al. **A Version-aware Approach for Web Service Directory**, IEEE International Conference on Web Services, p. 406-413, Jul. 2007.

WORLD WIDE WEB CONSORTIUM. **W3C NOTE-ws-arch-20040211**: Web Services Architecture. 2004.

WORLD WIDE WEB CONSORTIUM. **W3C REC-XMLSCHEMA-2-20041028**: XML Schema Part 2: Datatypes Second Edition. 2004.

ENDREI, M. et al. **Moving forward with Web services backward compatibility**, 2004, Disponível em: < *http://www-128.ibm.com/developerworks/webservices/library/ws-soa-backcomp* >. Acesso em: jun/2012

BROWN, K; ELLIS, M. **Best Practices for Web Service Versioning**, Jan. 2004. Disponível em: < http://www.ibm.com/developerworks/webservices/library/ws-version/ >. Acesso em: jul/2012.

WORLD WIDE WEB CONSORTIUM. **W3C NOTE-wsdl-20010315**: Web Services Description Language (WSDL) 1.1. 2001.

FAYYAD, U; SHAPIRO-PIATETSKY, G; SMYTH, P. **From Data Mining to Knowledge Discovery in Databases**. Al Magazine, [s.l], n.3, p.37-54, 1996