UNIVERSIDADE FEDERAL DA BAHIA

INSTITUTO DE MATEMÁTICA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Anderson Tiago Barbosa de Carvalho

Simulador de Serviços Web Semânticos com Injeção e Recuperação de Falhas

Salvador

2011.1

Anderson Tiago Barbosa de Carvalho

Simulador de Serviços Web Semânticos com Injeção e Recuperação de Falhas

Monografia apresentada ao Curso de graduação em Ciência da Computação, Departamento de Ciência da Computação, Instituto de Matemática, Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profº . Daniela Barreiro Claro

Salvador

2011.1

RESUMO

Serviços Web constituem uma forma de disponibilização de funcionalidades de um sistema de informação na Web por meio de tecnologias padronizadas. A utilização de uma linguagem de descrição semântica e computacionalmente interpretável facilita a descoberta e seleção destes serviços. Serviços descritos através destas linguagens são denominados Serviço Web Semânticos. Em muitos casos, serviços isolados possuem uma utilidade limitada, sendo incapaz de desempenhar uma função desejada. Surgiu então a ideia de composições de Serviços, visando o alcance de determinado objetivo ou funcionalidade. Porém, em ambientes instáveis como a Internet, há uma susceptibilidade a falhas no processo de execução destes serviços ou composições. Com isso, é importante que análises de comportamento e de recuperação do serviço em caso de falha sejam realizadas. Assim, o presente trabalho propõe o desenvolvimento de um Simulador de Serviços Web Semânticos e Composições, facilitando a simulação de cenários reais. Este trabalho visa integrar a funcionalidade de recuperação automática dos serviços em caso de falha e permitirá a injeção de falhas automaticamente para que os serviços possam ser submetidos a situações adversas.

Palavras-chave: Serviços Web Semânticos, OWL-S, Simulador, Auto-cura, Injeção de Falhas.

ABSTRACT

Web services are a way of providing functionality of a web information system using standard technologies. Using a semantic description language and computationally interpretable make the discovery and selection of these services easier. Services described using these languages are called semantic Web services. In many cases, isolated services have limited utility, being unable to perform a desired function. Then, appeared the idea of the composition of services, aiming to reach a particular purpose or functionality. However, in unstable environments such as the Internet, there is a susceptibility to failures in the implementation of these services or compositions. Thus, it is important for analysis of behavior and service recovery in case of fault to be conducted. So, this paper proposes the development of a Semantic Web Services and Compositions Simulator, allowing to simulate real scenarios. This paper aims to integrate the functionality of automatic recovery services in case of failure and allow the automatic injection of faults so that services can be subjected to adverse situations.

Keywords: Semantic Web Services, OWL-S, Simulator, Self-healing, Fault-Injection.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Asynchronous Javascript and XML,	p. 31
API	Application Programming Interface,	p. 9
FARM	Falhas, Ativadores, Resultados e Medidas,	p. 16
HTTP	Hypertext Transfer Protocol,	p. 11
Java EE	Java Enterprise Edition,	p. 30
JSF	Java Server Faces,	p. 30
JSP	Java Server Pages,	p. 30
MVC	Model-View-Controller,	p. 30
OWL	Web Ontology Language,	p. 13
OWL-S	Ontology Web Language for Services,	p. 7
RPC	Remote Procedure Call,	p. 11
SAWSDL	Semantic Annotations for WSDL,	p. 7
SOA	Service Oriented Architecture,	p. 12
SOAP	Simple Object Access Protocol,	p. 11
SWS	Serviços Web Semânticos,	p. 13
UDDI	Universal Description, Discovery and Integration,	p. 12
UI	Interface de Usuário,	p. 31
URI	Uniform Resource Identifier,	p. 52
WSDL	Web Service Description Language,	p. 11
WSMO	Web Service Modeling Ontology,	p. 7
XML	eXtensible Markup Language,	p. 11

1 INTRODUÇÃO

Serviços Web constituem uma forma de disponibilização de funcionalidades de um sistema de informação na web por meio de tecnologias padronizadas (ALONSO et al., 2003). Esse tipo de serviço tem se proliferado rapidamente (MARTIN et al., 2004) devido ao crescente desenvolvimento de soluções criadas para disponibilização via rede mundial de computadores, a Internet. Além disso, o crescimento da utilização de Serviços Web pode ser atribuído também ao fato de que essa tecnologia permite a comunicação entre aplicações independentes e distintas, com baixo acoplamento, pois faz uso de tecnologias padronizadas para descrição de serviços e troca de mensagens. Muitos destes serviços, quando isolados, não são capazes de atender aos requisitos do usuário. Assim, estes serviços podem ser combinados dando origem à Composição de Serviços. Esta composição de serviços traz benefícios empresariais, visto que serviços podem ser reutilizados, minimizando custos e ampliando o potencial de otimização de negócios e de integração de aplicações organizacionais (CHAFLE et al., 2007). A grande publicação destes serviços na Web vem motivando o aprimoramento de novas técnicas de descoberta destes serviços. Esta descoberta necessita cada vez mais de um processo automatizado, minimizando a intervenção humana. Para isso, estes serviços passam a ser descritos por meio de linguagens semânticas, tais como *Ontology Web Language for Services* (OWL-S) (MCGUINNESS; HARMELEN, 2004) , WSMO (*Web Service Modeling Ontology*) (LAUSEN; POLLERES; ROMAN, 2005) e SAWSDL (*Semantic Annotations for* WSDL) (KOPECKY et al., 2007). Atualmente, a Internet é um ambiente complexo visto que os serviços publicados são gerenciados por seus respectivos provedores e assim não se torna um ambiente controlado. Dessa forma, a indisponibilidade de serviços pode inviabilizar o uso de composições pois não se pode garantir que o serviço indisponível será substituído. Neste contexto, é importante que os serviços disponíveis na Internet sejam dotados de características de autogerenciamento, especialmente auto-recuperação, com o intuito de auto gerir estas indisponibilidades e assim tornar o uso transparente para o usuário final. Neste sentido, o trabalho de FERREIRA, CLARO e LOPES (2011) propôs a utilização de mecanismos de Auto-cura (*Self-Healing*), porém testes efetivos não foram realizados.

1.1 MOTIVAÇÃO

Mesmo com as facilidades da descrição semântica, ao se criar um Serviço Web Semântico (ou uma composição de serviços desse tipo) é muito difícil prever o seu real funcionamento, de acordo com as condições de operação as quais ele será submetido. Além disso, também não é simples verificar se ele realmente atende aos requisitos propostos. Logo, torna-se importante a existência de um ambiente onde esses serviços ou composições possam ter o seus ciclos de vida testados e avaliados, antes de serem levados aos ambientes onde realmente serão utilizados. Especialmente no caso de composições de Serviços Web Semânticos, o seu desempenho deve ser medido somente após a invocação da composição, pois trata-se de algo complexo devido aos vários fluxos de execução que podem estar contidos nela.

Em um sistema autonômico de funcionamento de composições de Serviços Web, no qual as composições possuem a propriedade de auto-cura que é responsável por auxiliar no aumento da disponibilidade de serviços ao prover mecanismos para detectar, emitir diagnósticos e corrigir falhas autonomicamente, mantendo o sistema consistente, torna-se ainda mais complicado prever o funcionamento dessas composições. Para cada cenário de falha, existem inúmeras possibilidades de recuperação possíveis e isso faz com que a árvore de possibilidades do ciclo de vida de uma composição seja bastante ramificada. A simulação pode ajudar a verificar todas essas possibilidades.

Adicionalmente, a simulação de serviços pode trazer informações estatísticas relevantes acerca da execução destes. Tempo médio de execução (ou resposta), desvio padrão e variância são dados pertinentes na hora de avaliar a viabilidade de veiculação de um serviço. Essas métricas também poderiam ser medidas a partir de amostras coletadas em tempo de execução, mas caso se tenha a vontade de conhecer esses números antes de colocar o serviço num ambiente real de execução, a simulação pode ajudar nessa tarefa. Tudo isso pode servir como uma ferramenta de apoio para a melhora da estrutura das composições de serviços pelos desenvolvedores, tornando-as melhores e aumentando seu grau de confiabilidade e usabilidade.

1.2 PROPOSTA

O custo de colocar uma composição de Serviços Web em operação sem antes fazer testes de execução e de atendimento aos requisitos pode ser alto, pois assim os seus defeitos (se existirem) só poderão ser descobertos da pior forma, isto é, quando o sistema estiver em operação e for interrompido pelo surgimento de um deles. Tratando-se da autonomicidade de serviços, esse

custo operacional pode ser ainda maior. Caso uma composição venha a falhar e não se recupere como previsto, o solicitante ficará sem a possibilidade de obter a resposta que deseja ao realizar uma consulta. Além disso, será difícil descobrir que tipo de falha causou a indisponibilidade do serviço, qual serviço falhou e porque ele não foi capaz de se recuperar sozinho.

Dessa forma, o presente trabalho propõe a implementação de um simulador de Serviços e Composições Web Semânticas, visando possibilitar aos desenvolvedores uma ferramenta prática de simulação, com injeção e recuperação de falhas. Constitui-se em uma continuação do trabalho apresentado por FERREIRA, CLARO e LOPES (2011) e que visa avaliar o mecanismo proposto neste através da injeção de falhas em uma ferramenta de simulação. O presente trabalho visa especificamente simular uma composição e permitir o seu monitoramento com condições normais e adversas de execução. Com ele é possível simular cenários nos quais falhas normalmente eventuais podem ser forçadas a aparecer e induzir a execução dos testes dos mecanismos de recuperação existentes no sistema. Essa última característica é relevante pelo fato de que esperar que uma falha ocasional ocorresse para que os mecanismos de auto-cura pudessem ser testados poderia ser inviável, já que existem falhas não determinísticas, podendo inclusive nunca vir a ocorrer.

1.3 CONTRIBUIÇÕES

Entre as contribuições obtidas com o desenvolvimento deste trabalho estão:

- Implementação de um mecanismo de injeção de falhas para composições de serviços.
- Modificações no módulo de auto-cura da OWL-S API (Application Programming Interface), permitindo que os tipos de recuperação de falhas a serem executadas e o endereço alternativo para substituição de serviços possam ser escolhidos em tempo de execução e não pré-determinados no código. Além disso, a modificação permite a execução individual dos métodos de recuperação.
- Desenvolvimento de um simulador capaz de executar composições de Serviços Web Semânticos e que possui sistemas de injeção e recuperação de falhas. O sistema de injeção possibilita a adição forçada de falhas ao sistema e a recuperação de falhas permite que o sistema se recupere automaticamente das falhas apresentadas em tempo de execução.

1.4 ESTRUTURA DO TRABALHO

Este trabalho encontra-se estruturado da seguinte forma: o capítulo 2 apresenta a fundamentação teórica acerca de Serviços Web e suas tecnologias básicas, Serviços Web Semânticos e a sua linguagem de descrição utilizada no nosso contexto e composição de Serviços Web Semânticos; no capítulo 3 são apresentados os conceitos relacionados a injeção de falhas e o porque da sua importância, relacionando-a com os mecanismos de recuperação de falhas; no capítulo 4 está a apresentação do ambiente utilizado e todas as tecnologias necessárias para implementação; o capitulo 5 descreve o processo de implementação do simulador, explicitando os passos do seu desenvolvimento e mostrando a sua estrutura arquitetural; o capítulo 6 apresenta um conjunto de trabalhos relacionados, tanto no aspecto de simulação como no aspecto de injeção de falhas e recuperação de falhas; o capítulo 7 trata dos experimentos realizados, descrevendo-os de acordo com as funcionalidades do simulador; o capítulo 8 traz as conclusões a que chegamos, as dificuldades encontradas e trabalhos que podem ser desenvolvidos futuramente com base neste.

2 SERVIÇOS WEB

De acordo com Tanenbaum e Steen (2007), um sistema distribuído é um "conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente". Nesse contexto, a computação tem evoluído bastante no sentido da resolução de problemas que envolvam a computação distribuída, pois esta se apresenta como um caminho para colaboração mútua de sistemas autônomos que possuam um mesmo objetivo, como Serviços Web. Aplicações distribuídas necessitam de um padrão de desenvolvimento que permita a interoperabilidade e comunicação entre aplicações isoladas. Nos últimos anos, vários padrões foram sugeridos, como CORBA (OMG, 1995), DCOM (MICROSOFT, 1995) e Java RMI(ORACLE, 1995), porém nenhum deles conseguiu satisfazer completamente os requisitos para desenvolvimento destes sistemas, pois foram desenvolvidos para domínios específicos que atendiam aos interesses de cada empresa desenvolvedora.

2.1 TECNOLOGIAS BÁSICAS

As tecnologias básicas associadas aos Serviços Web são SOAP (*Simple Object Access Protocol*) e WSDL (*Web Service Description Language*).

SOAP

Protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída. Se baseia no XML (*eXtensible Markup Language*) para seu formato de mensagem. Trabalha junto aos protocolos RPC (*Remote Procedure Call*) e HTTP (*Hypertext Transfer Protocol*) para negociação e transmissão de mensagens. Permite atravessar *firewalls*, diminuir o acoplamento do sistema que o utilize e projetar uma aplicação distribuída neutra em relação à plataforma, sistema operacional e linguagem, o que tornaria o uso de componentes remotos mais viável (SAMPAIO, 2007).

WSDL

Documento, baseado em XML, que descreve formalmente um serviço web, permitindo que

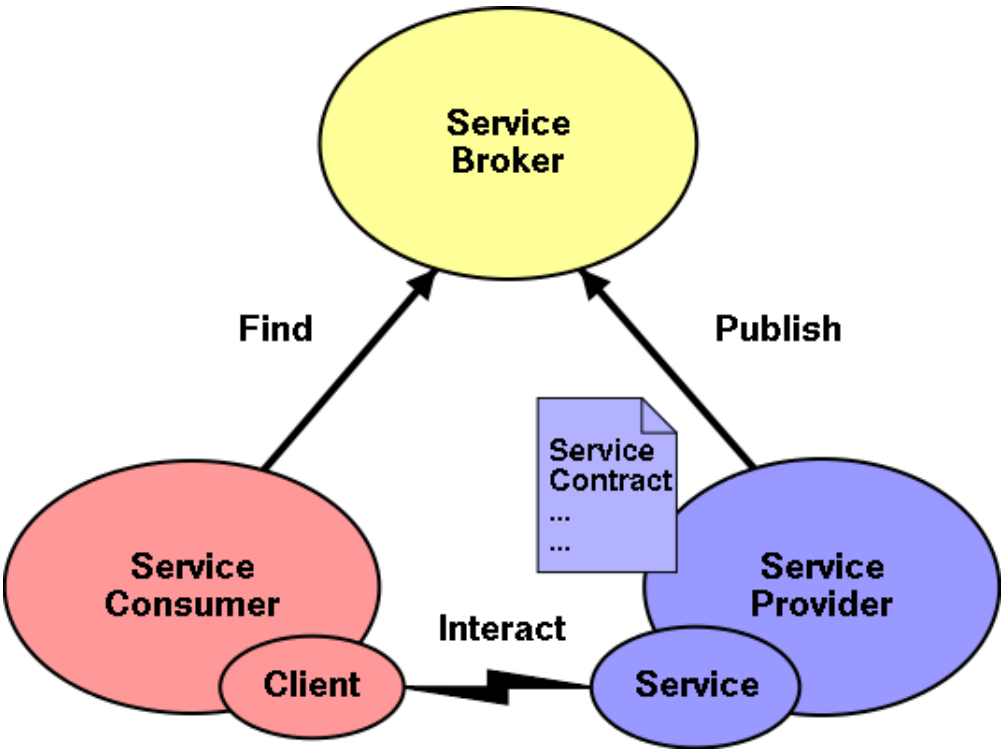

Figura 1: Arquitetura SOA (HAAS, 2003)

uma aplicação que deseje utilizar o serviço possa ter conhecimento sobre como trocar informações com ele. Contém todos os metadados úteis relacionados aos serviços.

As duas tecnologias descritas anteriormente formam a base da arquitetura SOA (*Service Oriented Architecture*), que surgiu em decorrência da necessidade de integrar sistemas heterogêneos, facilitando a comunicação e o reuso.

De acordo com a Figura 1, a arquitetura SOA funciona da seguinte maneira: o Provedor do Serviço (*Service Provider*) desenvolve uma funcionalidade e a publica no Repositório (*Service Broker*) (1). O solicitante ou cliente (*Service Consumer*) pode então localizar o serviço e fazer uma chamada diretamente pelo seu provedor (3).

A UDDI (*Universal Description, Discovery and Integration*) também aparece no cenário de tecnologia associada a Serviços Web, implementado mecanismos de registro e descoberta desses serviços, porém não será descrito neste trabalho por se limitar a descrições sem características semânticas.

2.2 SERVIÇOS WEB SEMÂNTICOS

A Web Semântica surgiu devido à necessidade de adicionar significado aos dados disponibilizados na Internet, para tornar viável a sua indexação e extração. Ela pode ser considerada uma extensão da web convencional e foi criada com o objetivo principal de estruturar dados, criando um ambiente em que as informações fossem interpretadas automaticamente por máquinas (LEE; HENDLER; LASSILA, 2001). Os Serviços Web proveem um *framework* baseado em padrões para a troca de informações de forma transparente e dinâmica entre aplicações desenvolvidas nas mais diversas plataformas. Existe uma preocupação com esses padrões justamente para tornar cada vez mais simples integrações padronizadas entre aplicações desenvolvidas sobre a base de Serviços Web e permitir a viabilização de computação autonômica. Para solucionar esses problemas, baseados na Web Semântica, surgiram os Serviços Web Semânticos (SWS), que são serviços descritos por uma linguagem de descrição semântica que permite uma descrição não ambígua e interpretável por máquinas dos dados presentes nesses serviços. Os SWS utilizam o WSDL para prover acesso aos serviços e descrever suas funcionalidades e possuem um documento de descrição semântica. Neste trabalho há a utilização da OWL-S como linguagem de descrição semântica por ser uma linguagem amplamente difundida na academia, já possuir trabalhos em andamento abordando composições nesta linguagem e ter fácil integração com uma ontologia de domínio, a OWL (*Web Ontology Language*).

2.2.1 OWL-S

A linguagem de descrição de Ontologias para Serviços Web (OWL-S) foi criada com base na OWL, que é uma linguagem para descrição de ontologias (MCGUINNESS; HARMELEN, 2004), com a intenção de suprir a necessidade de inserção de informações semânticas em documentos WSDL, surgindo para descrever de forma semântica um serviço (MARTIN et al., 2004), especificando suas propriedades e capacidades, fluxo de execução e detalhes de implementação. A OWL-S é composta por uma ontologia e três sub-ontologias que servem para representar as informações referentes à estrutura de um serviço Web:

- *Service*:
  É a ontologia principal das descrições da OWL-S e todo serviço descrito com base nesta deve obrigatoriamente possuir uma instância do *Service*. Faz referências às sub-ontologias *Profile*, *Model* e *Grounding*, permitindo o acesso às informações como habilidades, limitações, qualidade e pré-requisitos para execução do serviço. A Figura 2 mostra a relação dessa ontologia com as sub-ontologias.

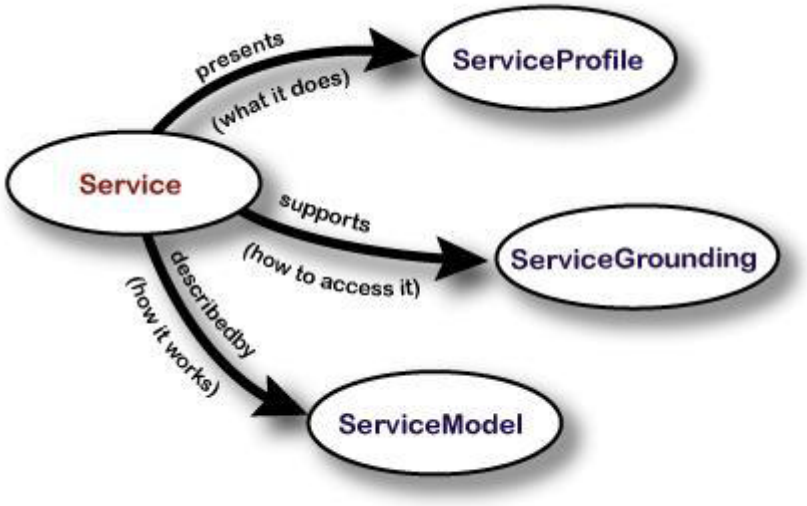

Figura 2: Ontologias do OWL-S (MARTIN et al., 2004)

- *Profile*:
  Ontologia onde estão descritas as informações referentes às habilidades do serviço. É responsável por representar elementos funcionais do serviço, como interfaces de entrada e saída, pré-condições e efeitos e elementos não funcionais como categoria e QoS. Em resumo, essa sub-ontologia é utilizada para publicação e descoberta do serviço.

- *Model*:
  Descreve a execução, o fluxo de dados e informações de invocações de serviços e troca de mensagens. Possui uma subclasse denominada *Process*, que contém a descrição do comportamento do serviço. Esse serviço descrito pode ser atômico ou composto. Um serviço composto é a junção de mais de um serviço trabalhando em conjunto e possui informações necessárias para a invocação dos serviços contidos nessa sub-ontologia.

- *Grounding*:
  Responsável por descrever a localização do WSDL e a forma de acesso ao serviço, explicitando como outras aplicações podem interagir com o serviço ao descrever os protocolos de comunicação e formatos de mensagens.

2.3 COMPOSIÇÕES DE SERVIÇOS WEB SEMÂNTICOS

Muitos serviços não possuem complexidade suficiente a ponto de suprir as necessidades do alcance de funcionalidades elaboradas, como por exemplo, busca de preços de um produto e posterior comparação para retornar o valor mais baixo encontrado. As composições de Ser-

viços Web tem a intenção de preencher essa lacuna, pois, elas tem um enorme potencial de otimização de negócios e integração de aplicações, facilitando assim o alcance de objetivos mais complexos. As composições de serviços também pode utilizar características semânticas (CLARO; MACEDO, 2008) e usufruir das vantagens que a adição de significado traz aos serviços e podem ser compostas por serviços atômicos e/ou compostos.

As composições de Serviços Web Semânticos, assim como os serviços atômicos, possuem parâmetros de entrada, saída, pré-condições e efeitos e também podem ser perfeitamente representadas pela arquitetura SOA, apresentada na Figura 1. Uma composição adequada pode economizar dinheiro, tempo, recursos humanos, esforço e produzir resultados com maior qualidade (GARG; MISHRA, 2008). Assim, a ordem em que a composição será executada é de suma importância e deve ser bastante estudada antes da sua construção, para que os recursos sejam bem aproveitados. Existem vários tipos de fluxos possíveis: execuções em sequência, em paralelo, condicionais e repetições, dentre outros.

Por causa da grande complexidade a qual uma composição de serviços pode chegar, devido aos fluxos citados e a junção de vários serviços que podem já possuir certo grau de complexidade, a possibilidade de ocorrer uma falha na sua execução é probabilisticamente maior do que a ocorrência de falha em um serviço atômico. Dessa forma, é interessante saber a quais tipos de falhas essas composições estão sujeitas e quais modelos de recuperação podem ser aplicados a cada uma delas. Porém, esse conhecimento pode ser mais bem aproveitado se for usado não somente após a ocorrência de uma falha, para recuperá-la, mas também de forma antecipada, para prever as consequências dessa ocorrência antes que ela ocorra e mensurar os efeitos. Para isso é necessário um mecanismo que force o aparecimento intencional de falhas conhecidas, de forma que se possa verificar o comportamento do sistema em tais situações. Esse mecanismo é denominado injeção de falhas e serve para testar a auto-cura de um sistema sem precisar esperar que falhas arbitrárias (bizantinas) ocorram. O próximo capítulo aborda os conceitos de injeção de falhas, modelos de falhas e de recuperação.

3 INJEÇÃO DE FALHAS

Injeção de Falhas pode ser definida como a introdução intencional e controlada de falhas em uma aplicação alvo para observar seu comportamento (ARLAT et al., 1990). Como falhas ocasionais podem demorar muito tempo para ocorrer (ou nunca ocorrer), seria inviável esperar por determinado tempo e recolher dados estatísticos sobre as ocorrências de falhas em um sistema, a fim de validar o seu funcionamento. Assim, a injeção de falhas acelera o processo de aparecimento de uma falha, injetando-a artificialmente em um sistema em execução. Essa técnica é importante para diminuir os esforços na validação de um sistema que implementa auto-cura, pois torna controlável o acionamento deste mecanismo. Essa validação é muito importante quando se trata principalmente de sistemas críticos, pois custos e consequências relacionados com as falhas podem ser altos. Ao simular a ocorrência de falhas, eventuais defeitos no sistema de tratamento de falhas podem ser identificados e corrigidos previamente à operação real do sistema.

Segundo Arlat et al. (1990), quando se considera a injeção de falhas o domínio de entrada corresponde a um conjunto F de falhas a serem injetadas e um conjunto A de dados utilizados na ativação das falhas, que especificam o domínio usado para testar funcionalmente o sistema. O domínio de saída corresponde a um conjunto R de resultados de comportamentos do sistema na presença de falhas e um conjunto M de medidas derivadas da análise e processamento dos conjuntos F, A e R. Juntos, os elementos da FARM (Falhas, Ativadores, Resultados e Medidas) constituem os principais atributos para caracterizar um processo de injeção. Especificamente, FARM significa:

- F: O conjunto de falhas a injetar depende das características do sistema e do nível de abstração no qual as falhas são injetadas (portas lógicas, chips, sistemas ou rede). Em qualquer abordagem de injeção é importante garantir que o conjunto de falhas produzido pela injeção seja o mais próximo possível dos erros produzidos por falhas reais. Tal conjunto é formado por meio de um processo estocástico no qual os parâmetros são caracterizados por distribuições probabilísticas.

- A: O conjunto de dados de entrada deve ser cuidadosamente determinado, pois ele é o responsável por ativar as falhas injetadas. Padrões de dados que geram a ativação de falhas devem ser encontrados e o conjunto será construído baseando-se nesses padrões.

- R: Quanto aos dados de saída, deve-se fazer uma análise em relação à execução do sistema em teste pois a execução pode ser interrompida antes de atingir o ponto final esperado por força da injeção de falhas, pelo sistema operacional ou pelo mecanismo de tolerância a falhas, caso tenha detectado uma falha que não é capaz de tratar. Por outro lado, se um programa termina normalmente no ponto esperado, seus resultados ainda assim podem não estar corretos devido a um erro não detectado. A solução empregada normalmente consiste em comparar os resultados obtidos após as injeções com aqueles obtidos durante a execução sem injeção, que serve como referência.

- M: Corresponde às medidas finais de confiabilidade (*dependability*) obtidas pelo experimento. Na maioria dos estudos o objetivo visado é a avaliação de parâmetros como latência e fator de cobertura (porcentagem de falhas que são detectadas). Eles podem ser usados na construção de modelos de análise para cálculo de confiabilidade ou na validação de modelos preexistentes.

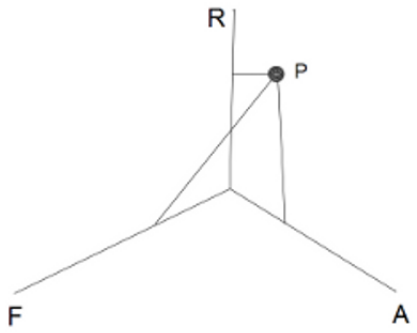

Figura 3: Processo de Injeção de Falhas representado pelo ponto P no espaço FxAxR

Dessa forma, um processo de injeção de falhas consiste em uma coleção de experimentos na qual cada um deles requer a injeção de uma ou mais falhas do conjunto F, sujeição do sistema a um dos dados do conjunto A, análise dos dados do conjunto R que, por sua vez, será fonte para as medidas obtidas no conjunto M. A sequência de testes por injeção de falhas é composta por experimentos caracterizados por um ponto P no espaço F x A x R, como mostra a Figura 3.

Para entender o que será tratado pelo mecanismo de injeção, é importante esclarecer a diferença entre falha, erro e defeito. Segundo Avižienis et al. (2004), esses conceitos são definidos

como segue:

- Falha (*Fault*): É o elemento que ocasiona o erro, que provoca uma transição de estado não planejada do sistema. O sistema pode apresentar falhas e não apresentar erros.
- Erro (*Error*): Estado do sistema que pode levar a um defeito, caso altere propriedades funcionais do sistema.
- Defeito (*Failure*): Manifestação de um erro. Sistema se torna incapaz de fornecer um ou mais serviços pois suas propriedades não estão mais de acordo com o especificado.

Na Figura 4 está ilustrado o processo desde o aparecimento de uma falha até a manifestação de um defeito.

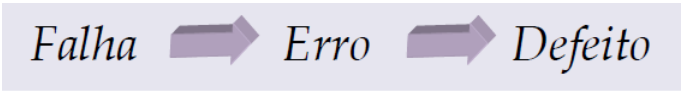

Figura 4: Sequência de eventos para ocorrência de defeito

Para que seja possível uma injeção de falhas pertinente, também é necessário saber quais são os tipos de falhas mais comumente encontrados na execução de sistemas distribuídos. As falhas podem ser classificadas de maneiras diferentes, acordo com diversos enfoques (SELIC, 2004): falhas de duração, falhas de causa e falhas de comportamento.

1. Falhas de duração: Quando o problema detectado é causado por lentidão na operação. Podem ser transientes, intermitentes ou permanentes. Falhas transientes são caracterizadas por uma única ocorrência e posterior desaparecimento. Falhas intermitentes ocorrem e desaparecem não deterministicamente, podendo voltar a aparecer a qualquer momento. Falhas permanentes são aquelas que continuam a existir até que o componente faltoso seja substituído.
2. Falhas de projeto: São o resultado de falhas de projeto e/ou de implementação. São resultantes de falhas humanas causadas normalmente por erros no momento de construção do sistema.
3. Falhas operacionais: Ocorrem durante a vida útil do sistema e são, invariavelmente, devido a causas físicas, tais como falhas de processador ou falhas de disco.

   Baseando-se em como um componente faltoso se comporta uma vez que uma falha tenha ocorrido, as falhas podem ser classificadas como segue (SELIC, 2004):

- Falha por queda: quando ocorre uma interrupção permanente na execução do servidor sendo, entretanto, possível detectar a indisponibilidade do processo.
- Falha por omissão: caracterizada por uma falha de omissão de recepção e/ou omissão de envio de mensagens.
  (a) Recepção: ocorre quando um processo deixa de receber do subsistema de comunicação algumas mensagens enviadas através de um meio de comunicação sem falhas.
  (b) Envio: ocorrem quando um processo deixa de enviar mensagens por um meio de comunicação sem falhas.
- Falha de temporização: a resposta do servidor se encontra fora do intervalo de tempo previamente definido.
- Falha de resposta: a resposta do servidor está incorreta. Se o valor da resposta estiver incorreto, caracteriza-se falha de valor. Se o servidor de desvia do fluxo de controle correto, caracteriza-se falha de transição de estado.
- Falha arbitrária (bizantina): um servidor produz respostas arbitrárias em momentos arbitrários.

Várias pesquisas que já foram realizadas sobre injeção de falhas indicam que essa é uma técnica reconhecida e bastante útil para análises de níveis de confiança de um sistema (LYU, 1995). Existem técnicas de injeção baseadas em hardware e em software. Injeções em hardware são baseadas na modificação de propriedades ou danificação de elementos de hardware que fazem parte do sistema. Injeções via software baseiam-se em modificações em conteúdos de registradores, código-fonte e/ou recursos utilizados por um sistema. Neste trabalho foi utilizado apenas o tipo de injeção via software pois essa categoria não necessita de processos físicos complexos e é de baixo custo, não necessitando da utilização de elementos muitas vezes caros de hardware.

3.1 INJEÇÕES DE FALHAS VIA SOFTWARE

As injeções de falhas via software permitem que sistemas sejam testados em pontos específicos, sem que a injeção interfira em outras partes do sistema. Além disso, esse tipo de injeção tem as vantagens de que não é preciso utilizar o hardware onde o sistema irá rodar, é portável, adaptável a outros tipos de falhas, tem baixo custo de desenvolvimento e não há risco de danificar o sistema. No entanto, o processo via software possui algumas limitações importantes (LOOKER; MUNRO; XU, 2004):

(a) Não tem acesso a recursos escondidos do software, como memória. Isso poderia ser útil para alguns tipos de testes.

(b) Manipulação do código a ser testado pode perturbar o funcionamento de todo o sistema.

(c) Captura de eventos pode ser imprecisa porque os temporizadores disponíveis para um sistema de software pode não ter resolução alta o suficiente para capturar falhas de curta latência.

A injeção via software pode ser categorizada de acordo com os seguintes tipos (LOOKER; MUNRO; XU, 2004): injeção em tempo de compilação, injeção em tempo de execução e injeção a nível de rede.

- Injeção em tempo de Compilação

  Injeção em tempo de compilação (também conhecida como mutação de código) é uma técnica onde o código fonte de um sistema é modificado de forma a gerar uma falha. Assim, quando o mesmo for compilado, diferirá do original e ficará suscetível à ocorrência de um defeito. Um exemplo simples da aplicação dessa técnica pode ser a modificação do código:

  ```
  a = a + 1
  ```

  para:

  ```
  a = a - 1
  ```

  Essa técnica pode simular tanto falhas de software quanto de hardware e produz falhas que se aproximam daquelas produzidas por erros de programação. Como as falhas são codificadas e compiladas junto com o código fonte, é possível emular falhas permanentes e transientes. No entanto, como se trata de modificação de código, ao injetar uma falha há a possibilidade de outra falha não prevista ser introduzida no sistema, por uma falha do programador. Ainda, esse tipo de injeção não deve ser usado com a intenção de certificação de propriedades, pois o sistema testado difere do original.

- Injeção em tempo de Execução

  Segundo (ROSA, 1998), quando a injeção de falhas é realizada durante a execução da aplicação alvo, dois elementos são necessários: um software ou módulo extra para injetar as falhas e algum mecanismo para ativar o injetor de falhas. O injetor corresponde ao software que produz as alterações necessárias para emular os tipos

de falhas considerados pela ferramenta. Um injetor pode ser implementado das seguintes formas:

(a) Como um código extra anexado ao código da aplicação alvo. Ou seja, injetor e aplicação alvo constituem o mesmo processo;

(b) Como um processo separado que executa concorrentemente com o processo da aplicação alvo. O processo do injetor deve ter acesso ao estado do processo alvo para que as devidas modificações possam ser realizadas;

(c) Como tratadores de exceções/interrupções. Nesses casos, os mecanismos de injeção são ligados ao vetor de tratamento de interrupções. Esses tratadores têm acesso ao estado do processo em execução e podem alterá-lo, simulando a ocorrência de falhas.

Quanto aos mecanismos de ativação dos injetores, as seguintes abordagens são possíveis:

(a) Gatilhos baseados no Tempo (*Time based triggers*): quando determinado instante de tempo é atingido, uma interrupção é gerada e um gerenciador de interrupções associado ao relógio injeta a falha no sistema.

(b) Gatilhos baseados em Interrupções (*Interrupt based triggers*): utiliza exceções de hardware e mecanismos de *trap* (chamadas de sistema via software) para gerar uma interrupção num local específico do código do sistema. Esse método é capaz de associar a injeção de falhas ocorrência de eventos.

(c) Inserção de Código: consiste em inserir código em um sistema alvo no momento imediatamente anterior à execução de uma dada instrução. A diferença dessa técnica em relação à injeção em tempo de compilação são os fatos de que ela não influencia no código existente no sistema, somente adiciona um código que causará uma falha e é inserida no momento da execução. Pelos mesmos motivos do método de tempo de compilação, não pode ser utilizado para certificações de sistemas. Possui a vantagem de que o injetor de falhas pode ser compilado junto ao sistema como uma biblioteca, que será requisitada na hora que for necessário.

- Injeção a nível de rede (*Network Level Fault Injection*)

  Injeção a nível de rede é uma técnica de injeção de falhas baseada em corrupção, perda ou reordenação de pacotes numa comunicação de rede.

3.2 RECUPERAÇÃO DE FALHAS

Para recuperar uma falha, após injeção ou ocorrência natural, é necessário um mecanismo de Auto-cura (*Self-Healing*). A pesquisa que antecede essa (FERREIRA; CLARO; LOPES, 2011) implementou esse mecanismo e foi focada na utilização dos seguintes métodos para recuperação: reexecução (*retry*), substituição (*replace by equal* ou *replace by equivalent*) e salto (*skip*). A primeira técnica simplesmente provoca a reexecução de um processo que falhou. A segunda faz a substituição do processo que falhou por uma réplica do mesmo ou por um processo semanticamente equivalente. Já o terceiro método tem como objetivo deixar de executar aqueles serviços que não são essenciais ao alcance do objetivo de execução de uma composição e só pode ser utilizado se for possível identificar que o processo que falhou foi facultativo na sequência de execução de uma composição. No trabalho citado, esses métodos são acionados obedecendo a uma ordem pré-estabelecida (que pode ser modificada via código), mostrada nos diagramas da Figuras 5 e 6. Esses diagramas também dão uma visão geral do funcionamento de todo o mecanismo de auto-cura. Neles, se pode perceber a sequência de eventos para ativação dos métodos de auto-cura: (1) a falha no sistema é detectada pelo sistema de monitoramento de falhas, que solicita e execução do método de recuperação *retry* (1.1). Este verifica se pode executar (2), em caso positivo (2.1), realiza a recuperação e retorna o valor esperado pela execução do processo, em caso negativo ou se ocorrer falhas durante o processo de recuperação, faz uma requisição (4) para acionamento do próximo método. Esse processo ocorre sucessivamente até que o serviço que falhou seja recuperado ou não existam mais métodos de recuperação na fila.

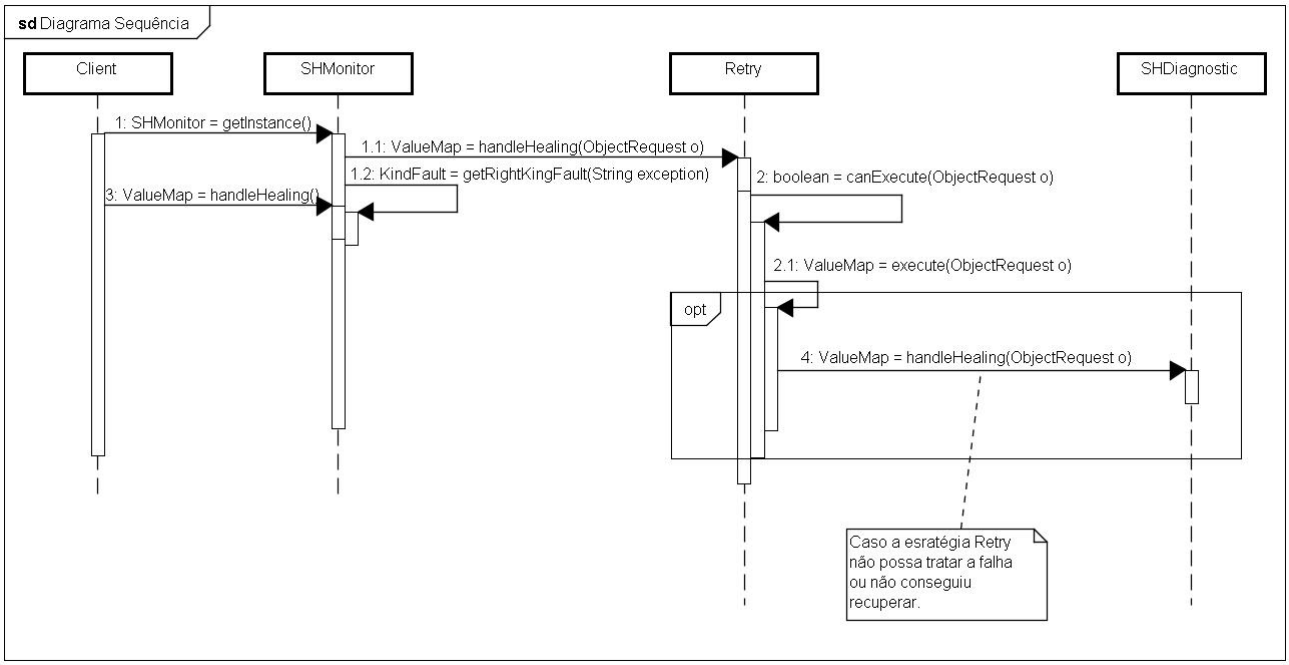

Figura 5: Diagrama de sequência do mecanismo de Auto-cura com início da sequência (FERREIRA; CLARO; LOPES, 2011)

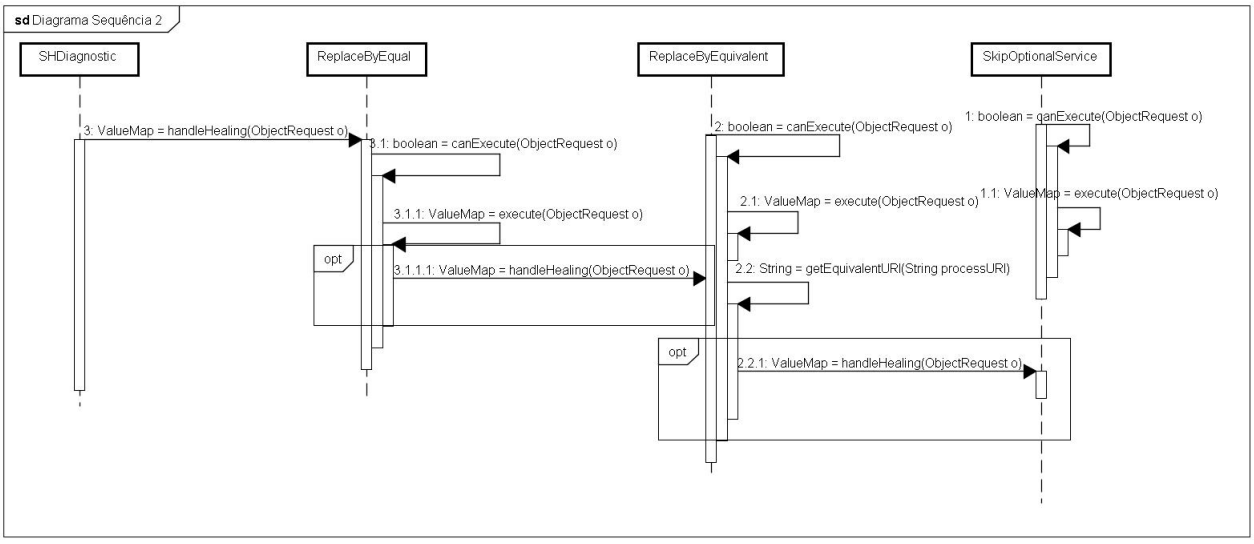

Figura 6: Diagrama de sequência do mecanismo de Auto-cura com continuação da sequência (FERREIRA; CLARO; LOPES, 2011)

4 IMPLEMENTAÇÃO DO SIMULADOR

Este capítulo apresenta os principais aspectos do desenvolvimento do Simulador de Composições de Serviços Web Semânticos com Injeção e Recuperação de Falhas. O propósito principal deste simulador é avaliar os mecanismos de recuperação propostos por FERREIRA, CLARO e LOPES (2011) através da injeção de falhas e consequentemente do desenvolvimento de uma ferramenta de simulação.

4.1 PROJETO ESTRUTURAL

Esta seção descreve como o simulador está logicamente estruturado, apresentando os módulos presentes, suas inter-relações e suas funções no sistema.

4.1.1 ESTRUTURA DO SIMULADOR

A estrutura do simulador está logicamente dividida em um módulo de funcionamento da *interface* gráfica, um módulo de configuração e execução de Serviços Web, um módulo de injeção de falhas, um módulo de monitoramento, diagnóstico e recuperação de falhas, um módulo de descoberta de serviços por similaridade semântica e um módulo de estatísticas. Essa estrutura geral pode ser vista na Figura 7, que mostra a organização dos módulos, suas inter-relações (arestas de relacionamento) e a direção dos fluxos de dados do sistema (orientação das arestas de relacionamentos).

O módulo de funcionamento da *interface* gráfica contêm somente os arquivos necessários para a construção e configuração da interface gráfica do simulador. Esses arquivos são responsáveis pela interação com o usuário, recebendo deste dados de entrada (endereço de hospedagem da descrição semântica do Serviço Web, número de execuções a serem procedidas, escolha de execução com ou sem injeção de falhas, método de recuperação de falhas, endereço do repositório de substituição, valores dos parâmetros de entrada)

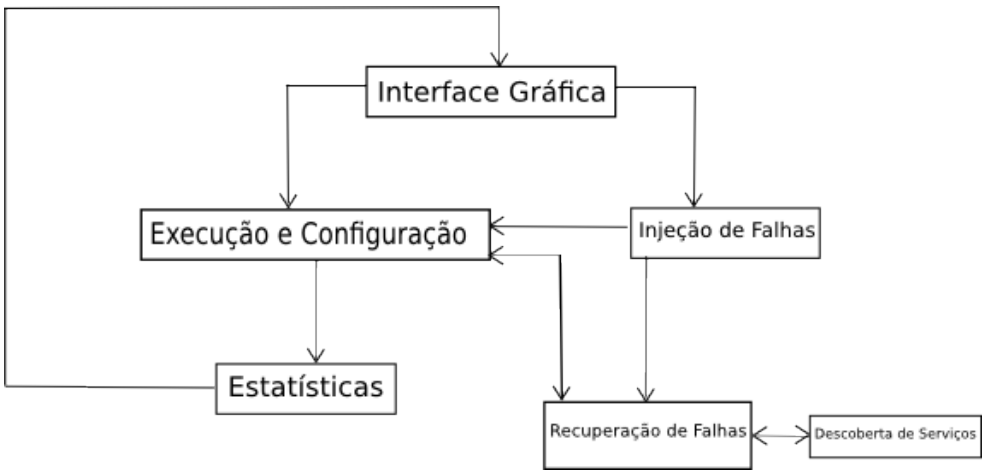

Figura 7: Módulos e seus relacionamentos

retornando dados de saída (parâmetros de saída) e mostrando informações (estatísticas). Eles transportam os dados inseridos até os módulos de configuração e execução e de injeção de falhas.

O módulo de configuração e execução de Serviços Web é responsável por construir toda a lista de parâmetros de entrada de um serviço, de acordo com os valores inseridos pelo usuário para cada parâmetro, e por fazer a requisição de execução do serviço. Caso tenha sido configurada a opção de execução com injeção e recuperação de falhas, esse módulo passa para o módulo de recuperação os dados correspondentes às escolhas do usuário em relação ao método de recuperação e ao repositório de substituições. Ao fim das execuções, esse módulo retorna para o módulo de estatísticas as medidas de tempo relacionadas às execuções.

O módulo de injeção de falhas é responsável por injetar uma falha no processo de execução de um serviço, caso essa ação tenha sido requisitada pelo usuário. Além disso, esse módulo identifica qual o método de recuperação de falhas escolhido e o informa ao módulo de recuperação de falhas.

O módulo de recuperação de falhas faz o monitoramento de uma falha, emite o diagnóstico e executa as ações necessárias para recuperação de uma falha. No caso de recuperação por substituição por similaridade semântica, ele passa para o módulo de descoberta os dados necessários para proceder a busca por similaridade semântica. Esse módulo retorna o resultado de uma recuperação para o módulo de execução.

O módulo de descoberta de serviços por similaridade semântica é responsável por fazer a busca por similaridade semântica caso o método de recuperação de falhas escolhido funcione por meio de substituição do serviço que falhou por outro semanticamente similar.

Ele retorna o resultado de uma busca para o módulo de recuperação.

O módulo de estatísticas faz cálculos estatísticos baseados nos tempos de execuções de serviços ou composições e envia para a *interface* gráfica o tempo médio, a variância e o desvio padrão calculados de acordo esses tempos.

4.1.2 INJEÇÃO DE FALHAS

O método de injeção de falhas utilizado neste trabalho é de inserção de código. Para injetar a falha, parâmetros de entrada necessários para a execução dos serviços atômicos são omitidos no momento imediatamente anterior à invocação de um serviço presente numa composição. Com isso o sistema se torna incapaz de executar o serviço, pois esses parâmetros constituem condição necessária para essa ação. Isso faz com que o sistema fique aguardando pela inserção desses parâmetros e essa espera acaba gerando uma falha por temporização, pois o sistema é configurado para esperar por determinado período de tempo pelos parâmetros e caso esse tempo passe e estes não sejam inseridos, o sistema dispara a falha, evitando-o de ficar esperando eternamente. Uma vez disparada essa falha, o sistema de monitoramento de falhas (pertencente ao mecanismo de auto-cura) a identifica e procede a recuperação, de acordo com o método de recuperação escolhido previamente pelo usuário na tela do simulador. Uma falha por temporização pode ser recuperada por todos os métodos de recuperação disponíveis no simulador, fazendo com que seja possível testar todos eles com esse mesmo tipo de injeção descrito. O módulo de injeção de falhas se encontra fisicamente no mesmo pacote onde está o módulo de configuração e execução de Serviços Web. O injetor e a aplicação alvo constituem o mesmo processo, porém o injetor é um elemento logicamente separado, que tem seu funcionamento próprio, que só é chamado se for necessário e que não é condição necessária para o funcionamento do simulador. Seu funcionamento é acionado somente no caso de o usuário marcar a opção de execução com falhas na tela do simulador.

4.1.3 ESTRATÉGIAS DE RECUPERAÇÃO DE FALHAS DO SIMULADOR

As técnicas de recuperação inseridas no simulador foram adaptadas de FERREIRA, CLARO e LOPES (2011). Em tal trabalho, foram implementadas as técnicas de recuperação já introduzidas na seção 3.2. A escolha dessas técnicas foi feita dando prioridade à utilização de estratégias que não necessitassem de tratamento transacional, visando diminuir a complexidade na implementação dessas estratégias. Privilegiou-se, também, a não modi-

ficação da estrutura atual do padrão do OWL-S, permitindo que o usuário possa utilizar qualquer arquivo OWL-S (FERREIRA; CLARO; LOPES, 2011). Explicando melhor as estratégias de recuperação:

- Reexecutar (*Retry*): Reexecuta o processo novamente.
- Substituição

  Por uma réplica (*ReplaceByEqual*): A estratégia utilizada visa efetuar uma busca em outros servidores pré-definidos visando selecionar um serviço alternativo que seja uma réplica daquele onde foi identificada uma falha. Esse novo serviço selecionado será executado e o seu resultado utilizado no fluxo de execução da composição.

  Por um serviço equivalente (*ReplaceByEquivalent*): Essa estratégia tem como objetivo selecionar serviços que são semanticamente similares àquele que foi identificada uma falha. É utilizada a ferramenta OWL-S *Discovery* (AMORIM et al., 2011) para identificação, em tempo de execução, de serviços semanticamente similares. Posteriormente a seleção de um novo serviço, esse será executado e o seu resultado retornado ao fluxo de execução da composição.
- Saltar (*SkipOptionalService*): Essa estratégia tem como objetivo deixar de executar aqueles serviços que não são essenciais ao alcance do objetivo de execução de uma composição. É identificado se o serviço que falhou é obrigatório ou facultativo na sequência de execução de uma composição. Essa identificação é baseada em uma análise dos seus *outputs* (parâmetros de saída). Se a quantidade de *outputs* for igual a zero ou não houverem interligações entre esses *outputs* e outros serviços, esse será considerado facultativo e não será executado. Caso contrário, é obrigatório e sua execução é obrigatória.

O mecanismo de auto-cura que implementa essas técnicas tem seu funcionamento baseado em três fases básicas, de acordo com a Figura 8: Monitoramento, onde é feito o processo de avaliação e identificação de uma falha gerada pelo elemento que está sendo gerenciado; Diagnóstico, tem como objetivo avaliar as falhas identificadas e selecionar uma estratégia de recuperação e a Recuperação, onde é executada a estratégia de recuperação que irá atuar no elemento gerenciado. O processo é automático, sem intervenção humana e o elemento gerenciado é uma composição de Serviços Web.

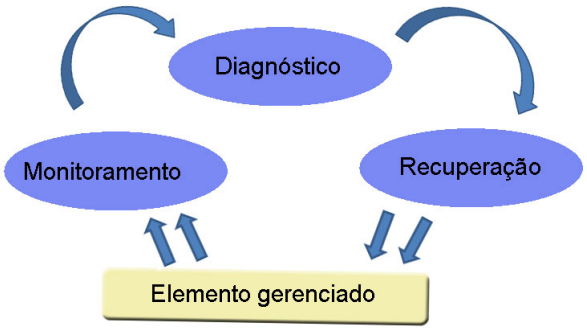

Figura 8: Estrutura MDR (FERREIRA; CLARO; LOPES, 2011)

4.1.4 MODIFICAÇÕES NA ESTRUTURA DO MECANISMO DE RECUPERAÇÃO

Como a OWL-S API suporta execução de composições de Serviços Web Semânticos e as falhas são identificadas pelo mecanismo de auto-cura na execução, FERREIRA, CLARO e LOPES (2011) criou um módulo dentro da API que permite identificação, diagnóstico e recuperação de falhas. No fluxo de execução do mecanismo de auto-cura contido nesse módulo, havia a definição de um servidor alternativo de serviços, que seria chamado quando uma falha fosse detectada em um serviço, para servir como repositório de busca de outro serviço que pudesse substituir o primeiro. Neste trabalho essa definição de servidor alternativo foi modificada. Como se trata de um simulador que visa facilitar e tornar transparente ao usuário os processos de execução de serviços (ou composições), injeção e recuperação de falhas, julgou-se que o usuário deveria poder ter a liberdade de escolher em qual repositório gostaria de procurar pelo serviço que poderá substituir um que teve falha de execução. Dessa forma, ao escolher que deseja injetar uma falha, o usuário deve inserir o endereço do servidor alternativo no campo destinado a esse fim, caso isso seja requerido pelo método de recuperação escolhido. Além disso, este trabalho modificou a estrutura das chamadas de métodos de recuperação de falhas, para que elas não fossem chamadas necessariamente de forma sequencial, mas sim de acordo com a escolha efetuada no menu de método de recuperação da *interface* gráfica do simulador. Assim, os métodos podem ser acionados de forma independente.

4.2 DESENVOLVIMENTO DO SIMULADOR

4.2.1 AMBIENTE DE DESENVOLVIMENTO

O simulador foi desenvolvido sobre a Plataforma Eclipse, versão 3.5.0. Como ferramenta de auxílio, foi utilizada a OWL-S API versão 3.0 (BASEL, 2004), que é responsável por prover o suporte pela leitura, escrita e execução de descrições OWL-S através de código Java. A versão da API utilizada foi modificada em relação à original, visando a integração do mecanismo de auto-cura ao processo de execução de um Serviço Web. Essa adição foi feita pelo trabalho que antecedeu a esse (FERREIRA; CLARO; LOPES, 2011). Como pode ser visto no diagrama de pacotes da Figura 9, a OWL-S API possui três pacotes principais: *examples*, onde existem os arquivos com exemplos de uso da API, *impl*, que contêm as classes implementadas e *org*, as classes abstratas e interfaces relacionadas ao domínio da OWL-S. Dentro de *impl* existem quatro pacotes *jena*,*owl*, *owls* e *swrl*. No pacote *owls*, responsável por manipular arquivos OWL-S e por executar os serviços descritos nesses arquivos, foi adicionado um novo pacote, destacado através da cor verde na Figura 9, denominado *sh*, onde foram implementadas as classes referentes ao mecanismo de auto-cura.

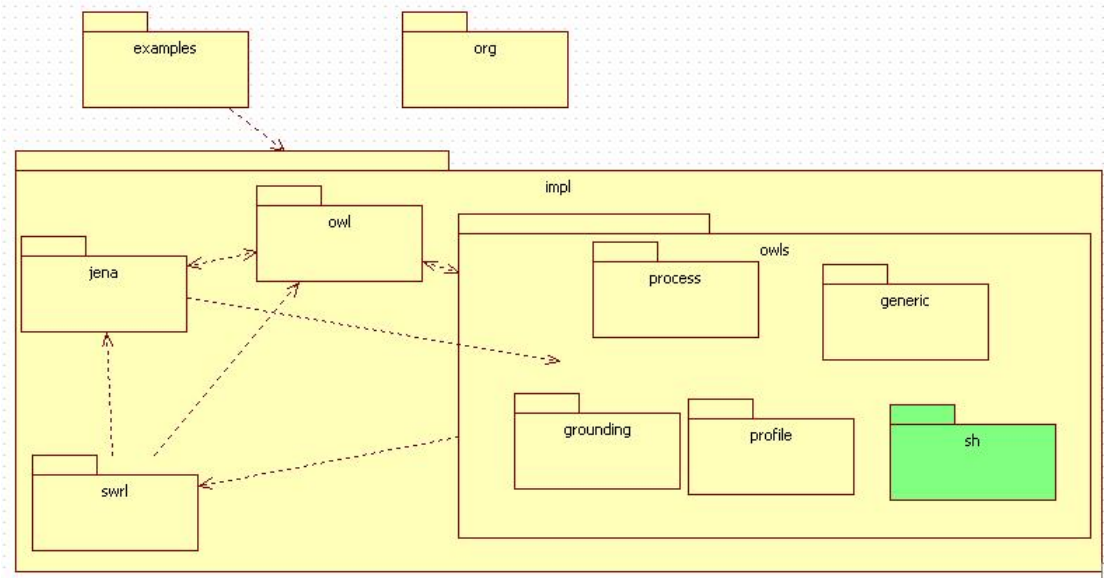

Figura 9: Nova Estrutura da OWL-S API

Como ferramenta de descoberta de serviços semanticamente similares (etapa necessária para funcionamento de um dos métodos de auto-cura) foi utilizada a OWL-S *Discovery* (AMORIM et al., 2011). Para cada serviço de uma composição, o algoritmo implementado nessa ferramenta é capaz de realizar uma busca no diretório indicado para encontrar serviços similares.

Como havia a intenção de produzir uma aplicação web independente de plataforma, foi feito o uso também da tecnologia *Java Server Faces* (JSF) (BURNS; SCHALK; GRIFFIN, 2009) . Ela estabelece o padrão para construção de interfaces sediadas no servidor e possui APIs de utilização desenvolvidas para facilitar cada vez mais o desenvolvimento de aplicações web sobre a plataforma Java EE (*Java Enterprise Edition*) . Essa APIs são disponibilizadas para representação de componentes de interface de usuário e controle de seus estados, manipulando eventos e validando dados de entrada. O JSF faz uso do *Java Server Pages* (JSP) (BERGSTEN, 2003) que são páginas de representação da interface gráfica dos componentes JSF. Como o JSF se baseia na arquitetura MVC (*Model*, *View*, *Controller*) (BURBECK, 1987), as aplicações produzidas com ela também seguem esse padrão, separando controle de fluxo, lógica do negócio e apresentação e provendo fácil integração entre essas camadas, como pode ser visto na Figura 10.

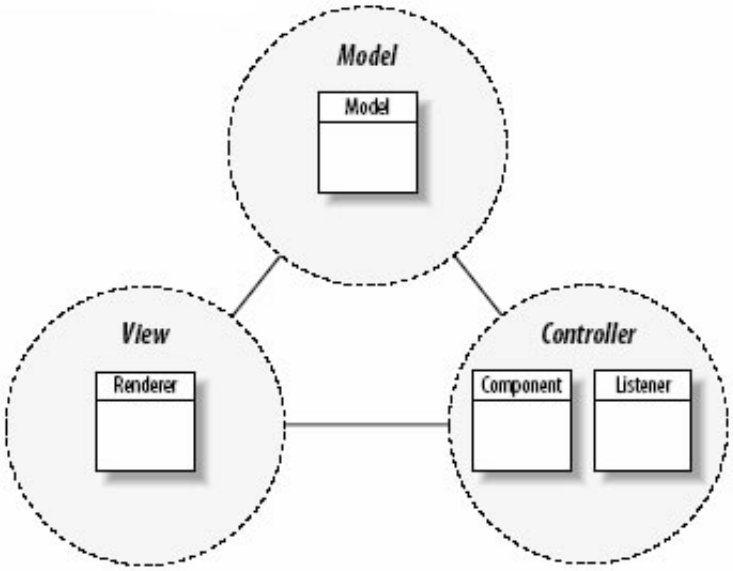

Figura 10: JSF na arquitetura MVC (BERGSTEN, 2004)

Nessa arquitetura, o *Model* é o componente onde ficam as regras de negócio, o *Controller* é responsável pelo controle do fluxo e a *View* cuida dos aspectos de interface e visualização da aplicação. É importante ressaltar o valor de separar lógica de negócio, controle de fluxo e apresentação. Isso permite que a lógica de negócio possa ser acessada e visualizada através de várias interfaces, sem saber quais nem como o usuário está acessando. Essa separação facilita o reuso, cria responsabilidades mais definidas e reduz esforços. No contexto JSF, as regras de negócios ficam localizadas nas classes java, o controle de fluxo e navegação está nos arquivos XML responsáveis pela configuração. Já a apresentação da aplicação é feita nas páginas JSP.

Outro componente utilizado no desenvolvimento deste trabalho foi o *RichFaces* (KATZ,

2008), hoje desenvolvido pela *Jboss*. Ele é um *framework* avançado de desenvolvimento de interface de usuário (UI) que possibilita fácil integração de características AJAX (*Asynchronous Javascript and* XML) dentro da aplicação JSF. A implementação do *RichFaces* procura dar ênfase a aspectos como usabilidade, desempenho da aplicação, recursos dinâmicos, customização de *layouts* e desenvolvimento de componentes. Isso faz com que a curva de aprendizagem do desenvolvedor seja reduzida, trazendo ganhos de produtividade. Sua estrutura é dividida basicamente em duas bibliotecas, que contém um conjunto completo dos componentes AJAX:

- a4j: utilizado para adicionar funcionalidades AJAX aos componentes JSF já existentes
- rich: palheta de componentes que estende os componentes JSF que já possuem funcionalidades AJAX

O *RichFaces* provê validação do lado do cliente, recursos avançados de enfileiramento de eventos, que é importante para atender aos requisitos de alto desempenho de aplicações do mundo real, ferramentas de testes para os próprios componentes e para as páginas nas quais estão inseridos e suporte a maioria dos navegadores de internet existentes.

5 TRABALHOS RELACIONADOS

Nesta seção, serão apresentados os trabalhos relacionados a este projeto, tanto no aspecto de simulação de Serviços Web como no de injeção de falhas por software.

5.1 MB-XP

O MB-XP (Mecanismo de Busca eXPerimental) (BELISARIO et al., 2005) é uma ferramenta construída para simular um serviço web baseado em Web Semântica e Ontologia. Sua implementação foi realizada utilizando o *Framework* .NET, por privilegiar a convergência entre diferentes equipamentos como *desktops*, *palms*, impressoras, servidores, entre outros. Sua aplicação consiste em um *front-end* encarregado de efetuar a requisição e obtenção da resposta vinda do serviço web, dentro do navegador Internet.

Figura 11: Tela do Simulador MB-XP (BELISARIO et al., 2005)

Na Figura 11 está a imagem da interface gráfica dessa ferramenta. Ao acionar o botão "Buscar" um arquivo XML é criado com as informações pertinentes à consulta, baseada no contexto, que é serializada e enviada através de um "envelope" SOAP encaminhado

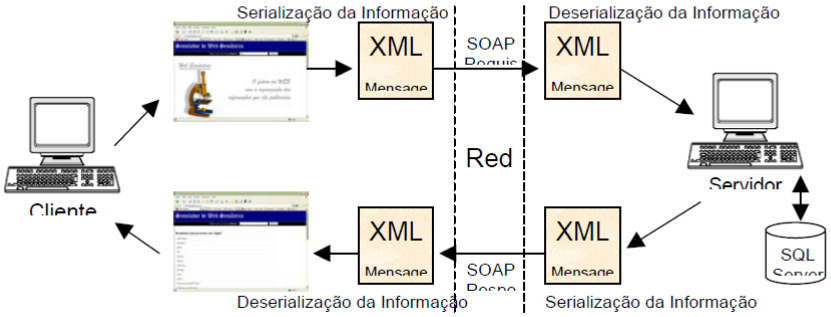

Figura 12: Funcionamento lógico do MB-XP (BELISARIO et al., 2005)

sobre o protocolo HTTP. Na outra extremidade, tem-se o receptor deste arquivo XML, na verdade a aplicação servidora, que recebe e deserializa a informação para extrair os parâmetros necessários para realização do processamento da consulta pelo serviço web. Esta consulta retornará a resposta dentro de um arquivo XML também parametrizado. Um "envelope" SOAP é enviado sobre o protocolo HTTP contendo a resposta da consulta. Esse processo é representado pela Figura 12.

Esta ferramenta foi construída como um modelo experimental, com o objetivo de demonstrar a importância e atual necessidade de organização semântica de conteúdo da web. Ela foi desenvolvida para testar um serviço web específico de busca de palavras. Ou seja, sua utilização não é de propósito geral, não servindo assim para simular o funcionamento de qualquer outro serviço web. Além disso, não possui características de recuperação nem de injeção de falhas.

5.2 FIRE

FIRE (*Fault Injection using a REflective architecture*) (ROSA, 1998), criado na Universidade Estadual de Campinas (UNICAMP), se baseia em programação reflexiva para injetar falhas no sistema sob teste. Reflexão significa que, além dos objetos normais de um sistema orientado a objeto, chamados objetos de nível base, existem ainda os meta-objetos, que podem modificar os objetos de nível base. Essas modificações são definidas e controladas por um protocolo de meta-objeto (*Meta-Object Protocol* MOP). Tal mecanismo de associação entre objetos e meta-objetos é utilizado para mudar o valor de atributos e de retorno de alguns dos métodos dos objetos de nível base, injetando assim as falhas nos objetos. O FIRE é capaz de simular falhas de hardware, porém seu objetivo principal é simular falhas de software transientes ou permanentes. Para isso, utiliza a técnica

de inserção de código por meio do meta-nível, fazendo com que a injeção de falhas seja completamente independente das funcionalidades do sistema alvo.

Vale ressaltar que essa ferramenta é capaz de injetar falhas apenas na aplicação e não no sistema inteiro, sendo assim possível utilizá-la no processo de teste especificamente da aplicação e não só do sistema onde a aplicação reside. Ela é capaz de trabalhar com sistemas escritos em C++ e sua utilização aumenta o tamanho do programa original, já que ela adiciona código extra necessário para os meta-objetos. Seu funcionamento necessita da disponibilidade de um arquivo de falhas (que especifica a quantidade de falhas a serem injetadas e os atributos das falhas), de um arquivo da aplicação alvo do experimento e de um arquivo onde será guardado o histórico da execução. A Figura 13 mostra uma tela de execução de uma injeção de falha.

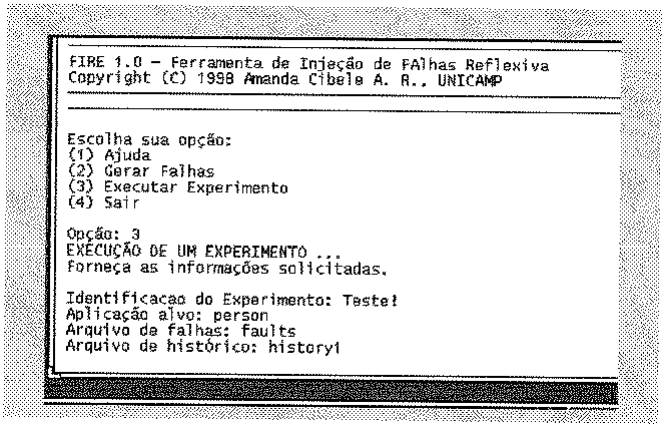

Figura 13: FIRE - Definição de um experimento (ROSA, 1998)

A FIRE se limita a injetar as falhas no sistema alvo e monitorar os seus efeitos, sem prover nenhum mecanismo de recuperação automático que possa fazer com que o sistema continue a funcionar no caso de ocorrência de um defeito. Dessa forma, os próprios sistemas a serem avaliados devem ter características de auto-cura se for desejável que uma falha seja tratada e recuperada na sua execução.

5.3 JACA

JACA (LEME, 2001) é uma ferramenta desenvolvida em Java para uso prático na realização de testes de injeção de falhas por software em sistemas. Ela pode ser utilizada para injeções de falhas em tempo de compilação, em alto nível: a nível da linguagem de programação, sem intervenções em *Assembly*. Para isso, é capaz de injetar falhas ao modificar atributos e métodos de um objeto de um programa Java. Também monitora o

sistema que está sendo testado para verificar se apresenta a resposta esperada mesmo na presença de falhas. Foi desenvolvido para ser portável, funcionando em qualquer plataforma que tenha a JVM (*Java Virtual Machine*).

Na sua utilização, o usuário escolhe o sistema no qual será injetado a falha, o conjunto de falhas que podem ser injetadas, o conjunto de classes que serão monitoradas para verificação da ocorrência de falhas e a execução dada pela relação entre conjunto de falhas e monitores. Oferece também a opção de execução do sistema sem a injeção de falhas, servindo assim como um simulador. A Figura 14 mostra a tela de configuração para um experimento.

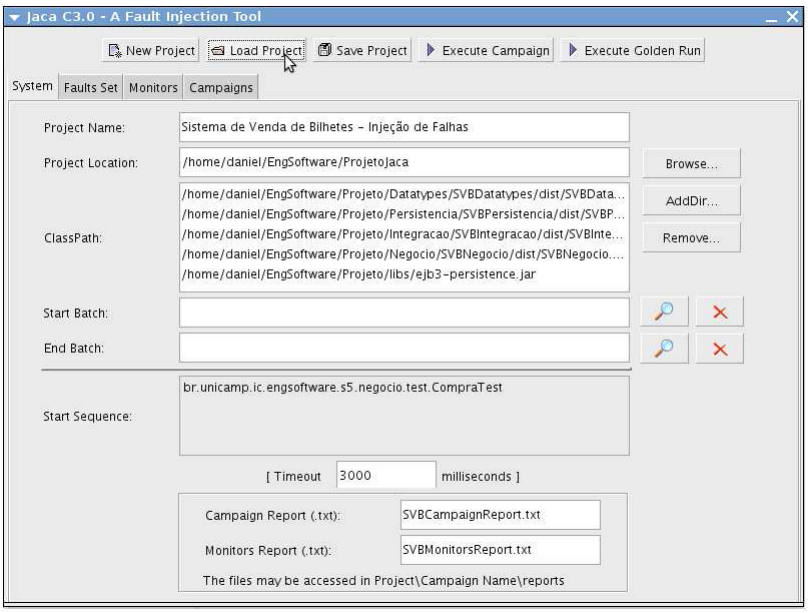

Figura 14: Tela da Ferramenta JACA (LEME, 2001)

Essa ferramenta é capaz de injetar diversos tipos de falhas, como especificado na Tabela 1.

Tipo de Falha	*Identificador da Falha*	*Descrição*
Falha de Atributo	`FalhaAtributo`	Altera o valor de um atributo de uma dada classe, no momento em que este é lido.
Falha de Retorno de Método	`FalhaRetMetodo`	Altera o valor de retorno de uma chamada de um método de uma dada classe.
Falha de Parâmetro	`FalhaParametro`	Altera o valor de um parâmetro dentro de uma chamada de um método de uma dada classe.

Tabela 1: Tipos de Falhas Ferramenta JACA (LEME, 2001)

Sua implementação não inclui mecanismos de recuperação de falhas. Isto faz com que o seu uso se restrinja ao monitoramento e observação do funcionamento de sistemas no

caso da ocorrência de uma falha, não provendo a funcionalidade de recuperá-lo em tempo de execução.

5.4 FIAT

FIAT (*Fault Injection based Automated Testing environment*) (CLARK; PRADHAN, 1995) é uma das primeiras ferramentas de injeção de falhas e foi desenvolvida pela Universidade Carnegie-Mellon. Utiliza inserção de código para corromper *bytes* em imagens de memória de programas. O código extra pode ser inserido no nível de aplicação ou de sistema operacional. Falhas são disparadas quando determinadas posições na memória são atingidas durante a execução. Seu método permite injetar falhas em código e dados de aplicação do usuário.

Seu propósito é avaliar a eficiência de detecção de erros em aplicações distribuídas. Além disso, busca analisar também se os mecanismos de recuperação dessas aplicações funcionam corretamente. A própria ferramenta não é capaz de prover mecanismos de recuperação para as falhas que injeta, somente verifica se o sistema por si só foi capaz de se recuperar de uma falha injetada.

5.5 ANÁLISE COMPARATIVA

Existem muitas diferenças entre as ferramentas aqui analisadas e o presente trabalho. Primeiramente, nenhuma delas incorpora mecanismos de recuperação de falhas. Isso faz com que ao ser simulada uma injeção de falhas, seja possível somente monitorar os efeitos causados. Caso o sistema que estiver sendo testado venha a parar por causa dessa falha, o experimento tem que ser interrompido devido a não existência de outra ação a ser tomada. Como não há a implementação de recuperação, nenhuma das ferramentas leva em consideração a análise de características semânticas dos sistemas testados, que serviria para uma possível substituição por similaridade em caso de falha.

Existem algumas outras diferenças evidentes quando comparamos o atual trabalho com as outras ferramentas analisadas. Em relação ao MB-XP, o trabalho desenvolvido apresenta as vantagens de poder simular a execução de qualquer Serviço Web que esteja semanticamente definido, além de prover mecanismos de injeção e recuperação de falhas e fornecer dados estatísticos a cerca dos tempos de execução.

Comparado ao FIRE e ao JACA, o presente trabalho apresenta a vantagem de não necessitar que os códigos de implementação do sistema a ser testado precisem ser disponibiliza-

dos ao ambiente de simulação. É suficiente que o simulador tenha acesso aos documentos de descrição semântica de um Serviço Web para que sua execução possa ser realizada.

Confrontado com o injetor de falhas FIAT, o método de injeção de falhas utilizado pelo atual trabalho não é tão invasivo a nível de corromper imagens de memória de programas a fim de que a falha gerada resulte da falha de *bits* corrompidos. Dessa forma, o método de injeção de falhas utilizado pela FIAT é de mais baixo nível e de difícil controle.

Na Tabela 2 é possível visualizar a comparação entre características analisadas do atual trabalho e das outras ferramentas relacionadas nesse capítulo. A primeira coluna apresenta a característica analisada, e as seguintes mostram a presença (coluna marcada com um X) ou ausência (coluna vazia) da característica em cada ferramenta. A última coluna é referente ao simulador implementado neste trabalho.

Característica	MB-XP	FIRE	JACA	FIAT	Simulador
Simulação de sistemas de propósito geral		X	X	X	X
Injeção de Falhas		X	X	X	X
Injeção de Falhas de Alto Nível		X	X		X
Recuperação de Falhas					X
Análise de similaridade semântica					X
Necessidade de código fonte do sistema avaliado	X	X	X	X	

Tabela 2: Tabela comparativa das ferramentas

6 TESTES E RESULTADOS

Neste capítulo serão apresentados os testes realizados visando validar o funcionamento do Simulador de Serviços Web Semânticos, incluindo a execução de serviços e os processos de injeção a recuperação de falhas.

6.1 TESTES

Esta seção apresenta os testes do simulador. Nesta trabalho, três tipos de testes foram realizados para validar o simulador:

(a) Funcionamento da Interface: objetiva avaliar o funcionamento correto de todos os elementos que fazem parte da interface (painéis, botões e campos de texto), verificando se cada um deles funciona de acordo com o especificado.

(b) Execução de Composições de Serviços Web Semânticos

- Execução sem falhas: visa avaliar a execução de uma Composição de Serviços Web Semânticos.
- Execução com injeção e recuperação de falhas: visa avaliar os mecanismos de injeção e recuperação de falhas existentes no simulador.

(c) Escalabilidade: objetiva medir a capacidade do simulador de suportar o aumento no número de requisições. Na área de Ciência da Computação, escalabilidade pode ser definida como capacidade de um sistema de se manter disponível e funcional à medida que o número de requisições cresce, mesmo chegando a números grandes (BARISH, 2001).

6.1.1 CENÁRIO

A base de serviços utilizada neste trabalho foi proposta pelos autores do Online Portal for Semantic Services (KRUG; KüSTER; RUHMER, 2009). Para todos os testes realizados foi utilizada a composição de serviços *NameAutoPriceTechnology*, modelada no

OWL-S Composer 3.0 (FERREIRA; CLARO; LOPES, 2011). Essa composição possui como entrada um parâmetro do tipo *Model* e como saída dois parâmetros, um *Price* e um *Technology*. Para intermediar a comunicação entre os serviços atômicos usados por essa composição, é utilizado o parâmetro *Auto*. Cada um desses parâmetros é especificado por meio de uma ontologia. A Figura 15 mostra a hierarquia da classe *Auto*. *Auto* possui *Car* como classe filha e *WheeledVehicle* como pai. *Model* é uma propriedade que pertence à classe *DesignedThing* relacionada a *Auto*.

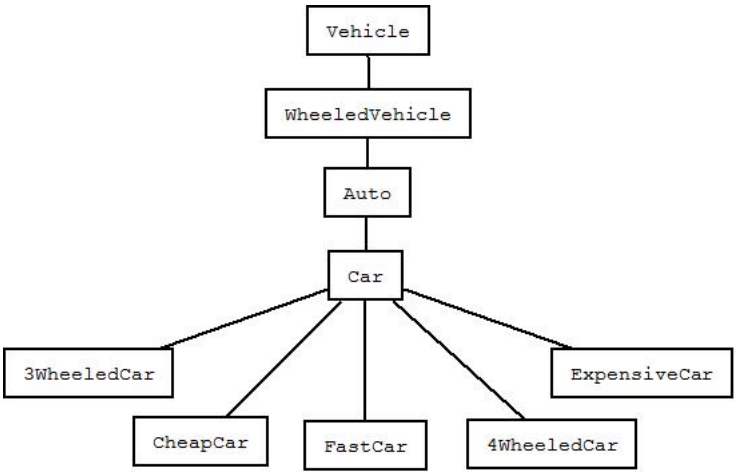

Figura 15: Ontologia *Auto*

Na Figura 16 está representada a hierarquia da classe *Technology*. *Technology* é um *Thing* e têm *InformationTechnology* e *ComputingTechnology* como filhos.

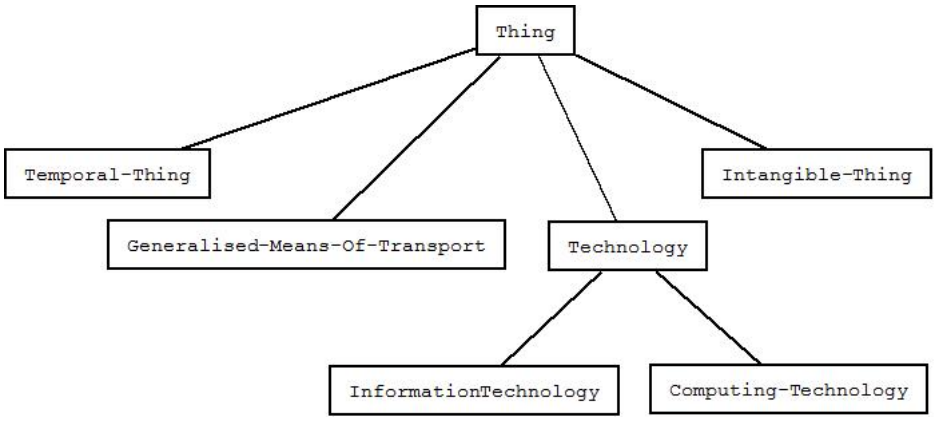

Figura 16: Ontologia *Technology*

Na Figura 17 está representada a hierarquia da classe *Price*. A classe *Price* é filha de *UntangibleObjects* e *MaxPrice*, *RecommendedPrice*, *TaxFreePrice* e *TaxedPrice* são seus filhos.

O serviço *NameAutoPriceTechnology* é composto pelos serviços atômicos: *getNameCar_NameCar*, *getCarPrice_CarPrice* e *getAutoTechnology_AutoTechnology*. O primeiro

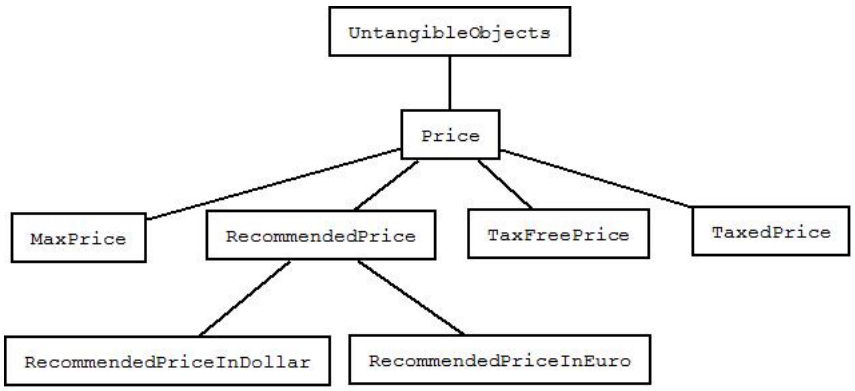

Figura 17: Ontologia *Price*

serviço possui *Model* como parâmetro de entrada e *Auto* como parâmetro de saída. O segundo tem *Auto* (saída do primeiro serviço) como parâmetro de entrada e *Price* como parâmetro de saída. O terceiro possui *Auto* como parâmetro de entrada (saída do primeiro serviço) e *Technology* como parâmetro de saída.

6.1.2 TESTES TIPO (A) - FUNCIONAMENTO DA INTERFACE

Nesse teste, foi simulado uma execução de Composição de Serviço com injeção e recuperação de falhas, de forma a explorar todas as funcionalidades disponibilizadas na interface do simulador. Iniciando o teste, foi inserido o endereço onde estava hospedado o serviço no campo correspondente do simulador. Foi então construída pelo sistema a lista de parâmetros de entrada do serviço. Esse lista continha apenas o elemento *Model*. Foi inserido no campo correspondente a esse parâmetro de entrada a palavra 'Celta'. Também foi configurado o valor 5 no campo que serve para configurar o número de execuções, para que o serviço fosse executado 5 vezes seguidas. Foi escolhida a opção para que as execuções fossem feitas com injeção de falhas. Foi inserido um endereço de servidor alternativo no campo referente (com o endereço do serviço no qual seria injetada a falha) e foi escolhido o método de recuperação "Replace by Equal"no menu de escolha de métodos, como mostra a Figura 18.

Após feitas as configurações acima descritas, foi dado o comando de execução e o serviço foi executado por 5 vezes, procedendo a injeção de falhas e a posterior recuperação pelo método escolhido. Após as execuções, foram disponibilizados na tela do simulador os valores referentes a tempo médio de execução, variância e desvio padrão e os valores dos parâmetros de saída do serviço, como pode ser visto na Figura 19.

Após solicitação de eliminação da primeira execução, o simulador eliminou os dados referentes à primeira execução e exibiu os novos valores na tela, como pode ser visto na

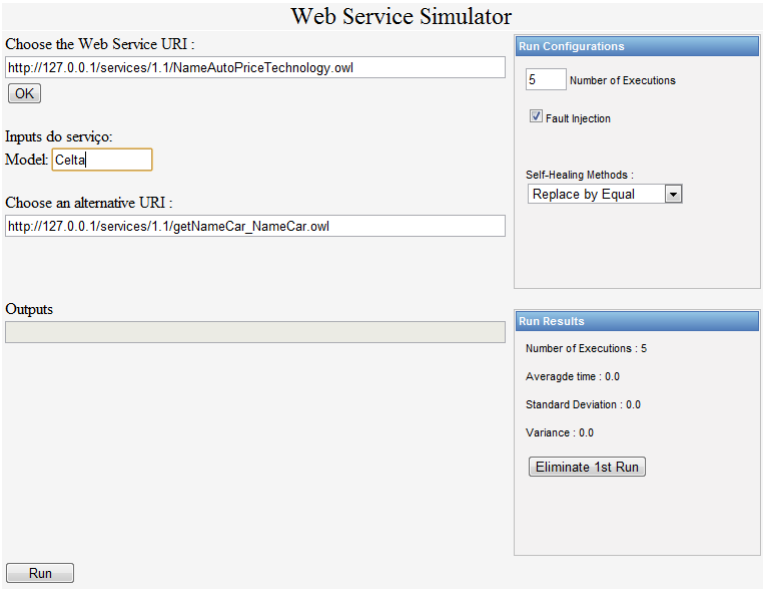

Figura 18: Tela antes das execuções

Figura 19: Tela depois das execuções

Figura 20.

Processo análogo foi repetido por mais 4 vezes, variando somente o método de recuperação de falhas. A intenção dessas repetições era verificar se o menu de escolha de método de recuperação realizava o endereçamento do método escolhido para as funções de controle do simulador.

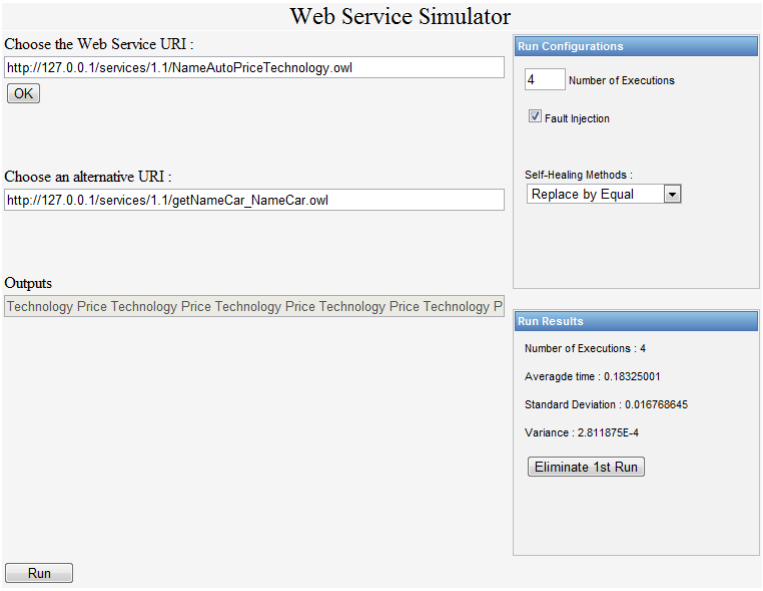

Figura 20: Tela depois da eliminação de dados da primeira execução

6.1.3 TESTES TIPO (B) - EXECUÇÃO DE COMPOSIÇÕES DE SERVIÇOS WEB SEMÂNTICOS

Esse tipo de testes foi subdividido em 6 testes diferentes. O primeiro avalia um sequência de execuções sem injeção de falhas e os outros 5 tem o objetivo de avaliar a execução com injeção de falhas e verificar separadamente cada um dos mecanismos de recuperação de falhas presentes no simulador: *Retry*, *Replace By Equal*, *Replace By Equivalent*, *Skip* e *Sequence*.

TESTE 1 - EXECUÇÃO SEM FALHAS

Configurações semelhantes às dos testes da seção 6.1.2 foram feitas para esse teste, com a diferença que a opção para execução com injeção de falhas não foi escolhida, pois desejava-se realizar execuções sem injeção de falhas. Inserindo o parâmetro de entrada e dando o comando de execução, o serviço foi executado por 5 vezes seguidas. Os tempos de execuções obtidos podem ser vistos na Tabela 3. Tais valores levam a uma média de tempos de execução 0.3786s, uma variância de 0.1713s e um desvio padrão de 0.4138s. Após solicitação de eliminação de dados da primeira execução, o valor da média diminuiu para 0.1717s. O valor da média sem eliminação da primeira execução é 120% maior do que a média após a eliminação da primeira execução. O gráfico da Figura 21 mostra a variação dos tempos de execução para esse teste.

Execução	Tempo (segundos)
1	1.206
2	0.187
3	0.166
4	0.183
5	0.151

Tabela 3: Tempos de execução

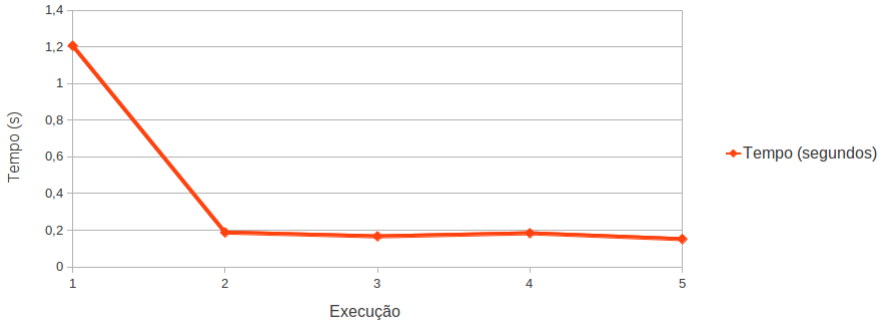

Figura 21: Gráfico de tempos de execução

TESTE 2 - EXECUÇÃO COM FALHA E RECUPERAÇÃO DO TIPO *RETRY*

Configurações semelhantes às dos testes da seção 6.1.2 foram feitas para esse teste, com a diferença que o método de recuperação escolhido no menu correspondente foi *Retry*. Como se trata da reexecução do serviço, o endereço alternativo não teria nenhuma influência no processo, podendo o campo correspondente ser preenchido ou não. Foi dado então o comando de execução. Como houve a injeção de falhas, o sistema de recuperação identificou a falha por temporização e realizou o procedimento de reexecução do serviço. Esse processo se repetiu por 5 vezes (em todas as execuções houve a injeção e a recuperação de falhas). Os tempos de execução obtidos foram: 1.246s, 0.212s, 0.191s, 0.169s e 0.155s, respectivamente. Esse valores levam a uma média de tempos de execução de 0.3946s, uma variância de 0.1815s e um desvio padrão de 0.4261s. Após eliminação dos dados da primeira execução, o valor da média diminuiu para 0.1817s.

TESTE 3 - EXECUÇÃO COM FALHA E RECUPERAÇÃO DO TIPO *REPLACE BY EQUAL*

Para esse teste foram feitas no simulador configurações similares às do teste 2 desta seção. As diferenças foram que o método de recuperação escolhido foi *Replace By Equal* e como nesse caso se trata da substituição do serviço por um equivalente, o endereço alternativo

fez-se necessário pois o mecanismo de auto-cura utiliza-o na sua busca por um serviço igual ao que falhou. Feitas as configurações requeridas (endereço do serviço, parâmetros de execução, endereço alternativo, número de execuções, opção de injeção com falhas e tipo de recuperação escolhido igual a *Replace By Equal*), foi dado o comando de execução. O sistema de recuperação identificou a presença de falha por temporização e realizou a substituição do serviço, executando o serviço indicando pelo endereço alternativo. Ao fim da execução, a tela disponibilizou os dados estatísticos referentes e os parâmetros de saída do serviço. Os tempos de execução obtidos foram: 1.314s, 0.193s, 0.198s, 0.187s e 0.155s, respectivamente. Esse valores levam a uma média de tempos de execução de 0.4094s, uma variância de 0.2048s e um desvio padrão de 0.4525s. Após eliminação dos dados da primeira execução, o valor da média diminuiu para 0.1832s.

TESTE 4 - EXECUÇÃO COM FALHA E RECUPERAÇÃO DO TIPO *REPLACE BY EQUIVALENT*

Nesse teste também foram feitas configurações similares às do teste 2 desta seção. A diferença foi que o método de recuperação escolhido foi *Replace By Equivalent*. Nesse caso, a inserção do endereço alternativo também se faz necessária pois o mecanismo de auto-cura utiliza-o na sua busca por um serviço semanticamente equivalente ao que falhou. Para que fosse possível o sucesso na busca por um serviço semanticamente equivalente, que é pré-requisito para substituição por equivalência semântica, foi colocado no servidor alternativo um serviço que possui exatamente a mesma estrutura do serviço no qual seria inserida a falha, porém com outra nomenclatura. Dessa forma os dois serviços possuíam grau de similaridade de 100%. Feitas as configurações, foi dado o comando de execução. O sistema de recuperação identificou a falha e o mecanismo de recuperação foi acionado para realizar a substituição por serviço equivalente, no entanto, esse processo não foi completado.

Ao analisar o motivo da interrupção, foi descoberto que o *OWL-S Discovery* possuía a limitação de ter estabelecido diretamente em um trecho de código o endereço do repositório onde o serviço similarmente equivalente deveria ser procurado. Ou seja, o endereço colocado na interface do simulador não chegava até o mecanismo de descoberta. Isso fez com que esse tipo de teste se tornasse inviável, pois para funcionar corretamente toda a estrutura do *OWL-S Discovery* teria que ser modificada em função de conseguir passar o endereço de uma URI desde a tela do programa até esse mecanismo. Esse ponto fugia ao escopo do presente trabalho.

TESTE 5 - EXECUÇÃO COM FALHA E RECUPERAÇÃO DO TIPO *SKIP*

Para esse teste foram feitas no simulador configurações similares às do teste 2 desta seção. A diferença foi que o método de recuperação escolhido foi *Skip*. Nesse caso, também não há necessidade de inserção de endereço alternativo. Nesse teste era necessário que o serviço fosse considerado opcional, isto é, não fosse considerado essencial ao alcance do objetivo de execução de uma composição. Dessa forma a composição foi modificada, retirando-se o parâmetro de saída do serviço *getAutoTechnology_AutoTechnology* (no qual foi inserida a falha), condição suficiente para que esse serviço fosse considerado facultativo na execução da composição. Assim, foram feitas as configurações de execução e foi dado o comando de execução. O mecanismo de auto-cura identificou a falha e procedeu a execução do método de recuperação escolhido. Após verificar que o serviço não possuía parâmetro de saída, o serviço não foi executado e o resto da composição foi executada normalmente, exibindo na tela os resultados. Como pode ser visto na Figura 22, somente o parâmetro de saída *Price* foi retornado (por 5 vezes) na tela do simulador, pois o parâmetro de saída *Technology* não existia mais.

Figura 22: Tela depois de execuções com injeção de falha e recuperação do tipo *Skip*

Ao fim da execução, a tela disponibilizou os dados estatísticos referentes e os parâmetros de saída do serviço. Os tempos de execução obtidos foram: 1.222s, 0.169s 0.185s, 0.135s e 0.212s, respectivamente. Esse valores levam a uma média de tempos de execução de 0.3845s, uma variância de 0.1759s e um desvio padrão de 0.4194s. Após eliminação dos dados da primeira execução, o valor da média diminuiu para 0.1752s.

TESTE 6 - EXECUÇÃO COM FALHA E RECUPERAÇÃO DO TIPO *SEQUENCY*

Esse teste foi feito somente para validação do funcionamento da opção de recuperação de falhas do tipo *Sequency*, que funciona tentando recuperar uma falha pelos diversos métodos descritos nesse trabalho, acionando-os de acordo com a sequência: reexecutar, substituir por igual, substituir por equivalente e saltar. Essa opção de recuperação aciona primeiramente o método reexecutar. Caso este se mostre incapaz de recuperar a falha, o próximo método da sequência é acionado. Essas chamadas ao próximo método da sequência são feitas até que o serviço que falhou seja recuperado ou que acabem os métodos de recuperação disponíveis.

Pelo método de injeção de falhas utilizado, sempre será possível reexecutar o serviço, pois o motivo da falha não está nem no serviço nem no seu servidor. Dessa forma, ao escolher como forma de recuperação a opção *Sequency* no simulador e dar o comando de execução, o mecanismo de auto-cura identificou a falha e a recuperou pelo processo de reexecução do serviço, mostrando os resultados na tela. Os tempos de execução obtidos foram: 1.203s, 0.169s, 0.182s, 0.172s e 0.193s, respectivamente. Esse valores levam a uma média de tempos de execução de 0,3838s, uma variância de 0,2098s e um desvio padrão de 0,4580s. Após eliminação dos dados da primeira execução, o valor da média diminuiu para 0,1790s.

Essa opção foi criada porque a depender da falha ocorrida, não será possível que ela seja recuperada por reexecução e assim o mecanismo tentará a recuperação pelo próximo método da sequência até encontrar um método que recupere o processo ou então chegar num ponto em que acabem os métodos disponíveis. Nesse último caso significa que o mecanismo de auto-cura não foi capaz recuperar o serviço que falhou.

6.1.4 TESTES TIPO (C) - ESCALABILIDADE

Nesse teste foram realizados testes com a intenção de medir a escalabilidade do simulador. Para realização do teste foram realizadas execuções simples (sem injeção e recuperação de falhas) com a composição *NameAutoPriceTechnology*. Primeiramente foi feita a configuração de 500 execuções seguidas. O simulador procedeu as execuções sem acusar nenhum tipo de problema. A tela do simulador após essa sequência de execuções é mostrada na Figura 23. Foram realizadas então as solicitações de sequências de 1000 e 2000 execuções e o simulador também as realizou sem nenhum problema. As telas dessas sequências são mostradas nas Figuras 24 e 25 respectivamente. As médias de tempo de execução foram 0.2048s, 0.7539s e 1.471s, respectivamente.

Figura 23: Tela depois de 500 execuções

Figura 24: Tela depois de 1000 execuções

6.2 AVALIAÇÃO DOS RESULTADOS

De acordo com os resultados alcançados pelos tipos de testes descritos nas seções anteriores, é possível observar que a *interface* gráfica do simulador funciona de acordo com os requisitos especificados. Em relação às execuções, pode-se concluir que o simulador é capaz de realizar a execução de Composições de Serviços Web, recebendo os parâmetros de entrada e retornando os parâmetros de saída. Além disso, o simulador se mostrou hábil a injetar falhas e se recuperar das mesmas, de acordo com os tipos de injeção e recuperação implementados (à exceção do método *Replace by Equivalent*). Percebe-se

Figura 25: Tela depois de 2000 execuções

também que as execuções feitas com injeção e posterior recuperação de falhas possuem médias de tempos diferentes, refletindo a complexidade de cada método de recuperação específico. A Figura 26 mostra um gráfico com as médias totais e após eliminação da primeira execução (lembrando que em todos os casos foram realizadas 5 execuções).

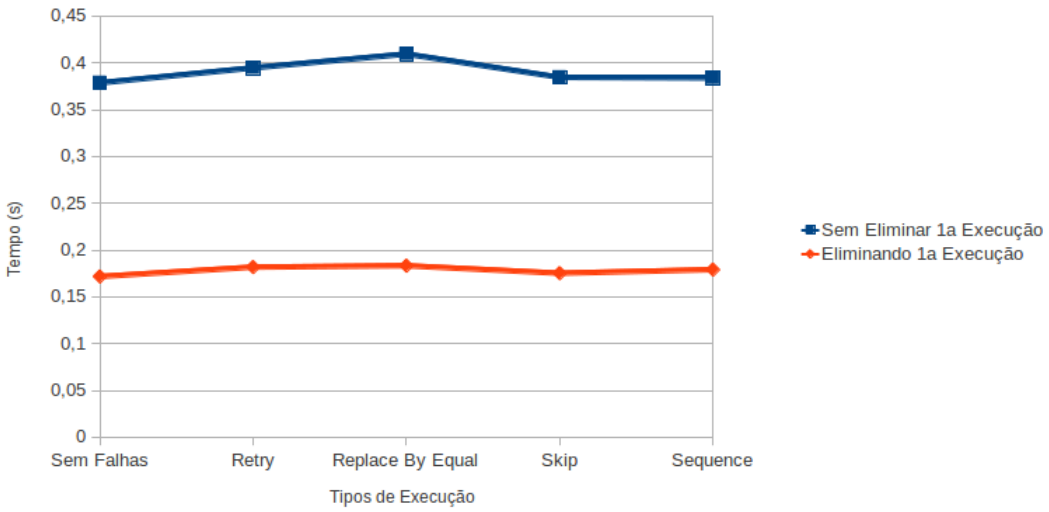

Figura 26: Gráfico de médias de tempos de execução (antes e após eliminação da primeira execução)

Em relação à escalabilidade, pode-se perceber que os valores das médias de tempos vão aumentando à medida que aumenta o número de requisições de execuções em sequência, mostrando que o sistema vai ficando cada vez mais sobrecarregado de acordo com esse

aumento. Pode-se ver no gráfico da Figura 27 essa variação.

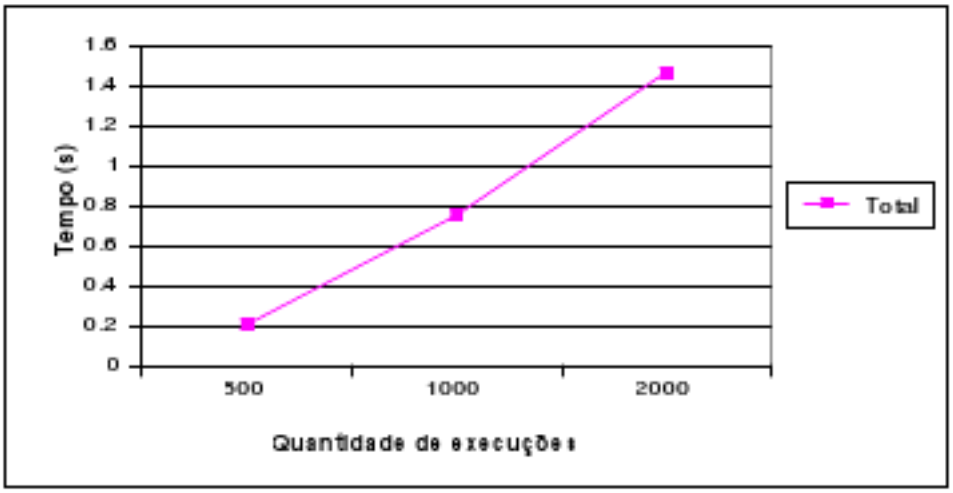

Figura 27: Gráfico de médias de tempos de execução - Escalabilidade

7 CONCLUSÃO

As contribuições que ambientes de simulação de *softwares* podem trazer são inúmeras, desde a validação de aplicações em ambientes que se aproximam do qual elas realmente irão operar até testes funcionais e de desempenho que ajudam na detecção de defeitos estruturais e operacionais e fazem com que correções possam ser realizadas previamente. Num ambiente autonômico de funcionamento de Serviços Web, é importante que hajam ferramentas que possam ajudar a prever a operação desses sistemas de forma que seus ciclos de vida possam ser testados e avaliados.

Esse trabalho propôs a implementação de um simulador de Serviços Web com características semânticas, que possui mecanismos de injeção e recuperação de falhas. Tais mecanismos trazem a oportunidade de simulação em condições adversas e automática recuperação caso ocorra um defeito na execução do serviço. A proposta é ser um facilitador para os desenvolvedores de Serviços Web Semânticos, tornando os testes mais práticos e verificando sob quais condições o serviço é capaz de operar sem ser interrompido definitivamente. Vários experimentos foram realizados como forma de validação do projeto, analisando os aspectos de funcionalidades, mecanismos de injeção e recuperação de falhas e escalabilidade. Os códigos-fonte do simulador podem ser encontrados em `https://code.google.com/p/web-service-simulator`.

7.1 DIFICULDADES ENCONTRADAS

A primeira e mais representativa dificuldade encontrada foi o fato de a OWL-S API possuir pouca documentação oficial. Isso fez com que muitas vezes fosse difícil saber a forma de utilização e o objetivo de diversas funções disponibilizadas pela API. Essa API possui 203 classes e o seu código não possui muitos comentários. Além disso, as modificações feitas por FERREIRA, CLARO e LOPES (2011) na mesma API também carecem de comentários e explicações.

Outro fator de dificuldade foi a utilização de JSF, *RichFaces* e JSP pois eram tecnologias

não conhecidas pelo desenvolvedor do simulador. Isso demandou um alto investimento de tempo no entendimento da arquitetura e da forma de funcionamento desses *frameworks*.

7.2 TRABALHOS FUTUROS

Dentro do contexto de simulação de Serviços Web Semânticos, muitas possibilidades de trabalhos futuros podem ser citadas para melhoria do simulador:

- Estudo de outros métodos de injeção de falhas que poderiam ser utilizados, estendendo essa função do simulador.
- Estudo de outros métodos de recuperação de falhas que podem ser integrados aos mecanismos de auto-cura presentes no simulador.
- Melhora nos elementos gráficos da tela de apresentação do simulador, para melhoria nos aspectos de usabilidade. Além disso, o simulador poderia ter o seu funcionamento baseado em navegação entre telas.
- Efetuar testes de desempenho do simulador, comparando com outras ferramentas.
- Mudança no *OWL-S Discovery* para que seja possível que ele considere endereços inseridos na tela do simulador, tornando possível a execução do método de recuperação *Replace by Equivalent*.

ANEXO A – FUNCIONAMENTO DO SIMULADOR

O simulador é composto por apenas uma tela, representada na *View* pelo arquivo index.jsp. A tela de utilização do simulador possui um campo de inserção de texto identificado pelo rótulo '*Choose the Web Service URI*' onde o usuário deve inserir a URI (*Uniform Resource Identifier*) do serviço que deseja executar e um botão 'OK' logo abaixo que serve para o sistema localizar o serviço e construir sua lista de parâmetros de entrada (*inputs*). Possui também outro campo rotulado '*Choose an alternative URI*' onde deve ser inserido o endereço do servidor alternativo, para o caso de execução com injeção de falhas, e um terceiro campo que mostrará os valores de resposta do serviço (*outputs*). No lado direito existem dois painéis:

- Superior, denominado '*Run Configurations*', é onde o usuário pode escolher, por meio do campo '*Number of Executions*', o número de vezes que deseja que o seu serviço seja executado e marcar a caixa de seleção '*Fault Injection*' caso deseje que o sistema proceda uma injeção de falhas na execução. Essa caixa de seleção habilita o menu '*Self-Healing Method*' de escolha de método de recuperação, localizado no mesmo painel, e a caixa de inserção de endereço de servidor alternativo;
- Inferior, denominado 'Run Results', é o responsável por disponibilizar as informações estatísticas após a execução do serviço: quantidade de execuções (*Number of Executions*), média de tempo (*Time Average*), variância (*Variance*) e desvio padrão (*Standard Deviation*). Esse mesmo painel apresenta o botão '*Eliminate 1st Run*', que tem a função de desprezar os dados da primeira chamada à execução do serviço. Isso se justifica pelo fato de que a primeira chamada para execução de um serviço costuma demorar mais, pois é quando os objetos *proxy* do serviço são criados na memória do computador. Dessa forma, as chamadas subsequentes demoram substancialmente menos. De acordo com essa característica, esse botão faz com que os dados estatísticos da primeira execução não interfiram na análise de desempenho.

Na parte de baixo da tela existe o botão '*Run*' que é o comando para execução do serviço. Esse botão só deve ser pressionado após todas as configurações de execução serem feitas corretamente. Na Figura 28 é possível ver a tela principal do simulador, com todos os painéis, caixas de texto e botões citados anteriormente.

Figura 28: Simulador de Serviços Web

Para proceder a execução de um serviço, o usuário deve primeiramente inserir o endereço URI do serviço e clicar em 'OK'. Este botão acionará o configurador de execução do serviço. Ele irá verificar se a URI foi realmente fornecida e em caso positivo começará a ler o arquivo de descrição do serviço, a fim de construir uma lista de parâmetros de entrada de acordo com o número de parâmetros que o serviço requer. Após terminada essa leitura, aparecerá na tela do configurador o número de campos correspondente ao número de parâmetros de entrada requisitados pelo serviço. O usuário deve então preencher esses campos com os valores de entrada para o serviço. A partir desse ponto, existem dois fluxos de execução possíveis: execução sem falhas e execução com falhas.

Na execução sem falhas, o usuário irá clicar no botão 'Run' e o simulador irá chamar a classe responsável pela execução do serviço. Essa classe irá adicionar os valores inseridos pelo usuário à lista de parâmetros de entrada do serviço, fazendo a correspondência correta entre eles (o ordem de inserção é a mesma ordem dos parâmetros do serviço).

Após isso, a OWL-S API é chamada e cria o mecanismo de execução do serviço, que irá fazer todos os procedimentos de execução e, caso não haja nenhum problema, irá retornar os parâmetros de saída do serviço. No caso de haver algum problema relacionado à falhas do serviço, o sistema de auto-cura presente na API indicará a sua ocorrência.

Caso o usuário marque a opção de execução com falhas, será habilitado na tela o campo de inserção do endereço do servidor alternativo onde o mecanismo de auto-cura irá procurar por uma solução para a falha que ocorrerá. Também será habilitado o menu onde o usuário deverá escolher qual tipo de recuperação de falha deseja que o simulador execute. Cada uma das quatro primeiras opções corresponde a um método de recuperação e a quinta opção indica que os métodos de recuperação serão executados em sequência, com uma ordem pré-estabelecida, como descrito em 3.2. Nesse momento, o usuário já pode clicar em 'Run'. Feito isso, o módulo injetor de falhas será imediatamente chamado e irá desviar o fluxo original de execução. No fluxo alternativo, não há o mapeamento dos valores inseridos pelo usuário com os parâmetros de entrada do serviço. A execução do serviço é então chamada. Como ela necessita que esses valores estejam mapeados, o mecanismo de execução ficará esperando esse valores e não encontrará, decorrendo uma falha por temporização. Quando isso acontecer o mecanismo de auto-cura identificará o problema e um aviso surgirá na tela para informar ao usuário que houve uma falha na execução do serviço. Para continuar, o usuário deve clicar no botão 'OK'. Baseando-se no endereço de servidor alternativo fornecido, o mecanismo de Auto-cura começará os seus procedimentos, executando o tipo de recuperação escolhida pelo usuário.

Em qualquer um dos casos de execução (com ou sem falhas), quando a etapa de execução terminar, serão mostrados no painel inferior do lado direito da tela os dados estatísticos relacionados: número de execuções, tempo médio de execução, desvio padrão e variância desses tempos.

REFERÊNCIAS BIBLIOGRÁFICAS

ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V. *Web Services - Concepts, Architectures and Applications*. 1. ed. [S.l.]: Springer, 2003.

AMORIM, R.; CLARO, D. B.; LOPES, D.; ALBERS, P.; ANDRADE, A. International conference on web services. In: *IEEE 9th International Conference on Web Services*. [s.n.], 2011. Disponível em: `http://conferences.computer.org/icws/2011/`.

ARLAT, J.; AGUERA, M.; AMAT, L.; CROUZET, Y.; FABRE, J.-C.; LAPRIE, J.-C.; MARTINS, E.; POWELL, D. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, IEEE, 1990.

AVIŽIENIS, A.; LAPRIE, J.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, Published by the IEEE Computer Society, v. 01, n. 1, p. 11–33, 2004.

BARISH, G. *Building Scalable and High-Performance Java Web Applications Using J2EE Technology*. [S.l.]: Addison-Wesley Professional, 2001. 416 p.

BASEL, U. *OWL-S API*. 2004. Último acesso em 29 de junho de 2011. Disponível em: `http://on.cs.unibas.ch/owls-api/`.

BELISARIO, G.; KAGUEYAMA, R. C.; LIMA, I.; SILVA, J. R.; TOLEDO, M. Mb-xp - um mecanismo de busca experimental simulador de web semÂntica. *FACULDADE DE CIÊNCIAS JURÍDICAS E GERENCIAIS DE GARÇA/FAEG*, 2005. ISSN 1807-1872.

BERGSTEN, H. *JavaServer Pages*. 3. ed. [S.l.]: Oreilly and Associates Inc, 2003.

BERGSTEN, H. *JavaServer Faces*. [S.l.]: Oreilly and Associates Inc, 2004.

BURBECK, S. Applications programming in smalltalk-80: How to use model view controller. 1987.

BURNS, E.; SCHALK, C.; GRIFFIN, N. *JavaServer Faces 2.0, The Complete Reference*. [S.l.]: McGraw-Hill, 2009.

CHAFLE, G.; DAS, G.; DASGUPTA, K.; KUMAR, A.; MITTAL, S.; MUKHERJEA, S.; SRIVASTAVA, B. An integrated development enviroment for web service composition. *IEEE International Conference on Web Services*, p. 839–847, 2007.

CLARK, J.; PRADHAN, D. Fault injetction: A method for validanting computer system dependability. *IEEE Computer Society*, p. 47–56, 1995.

CLARO, D.; MACEDO, R. J. de A. Dependable web service compositions using a semantic replication scheme. In: *26 Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2008)*. [S.l.: s.n.], 2008.

FERREIRA, M. R.; CLARO, D. B.; LOPES, D. C. P. Integração do mecanismo de self-healing no tratamento das composição de sistemas de informação através dos serviços web semânticos. In: *VII Simpósio Brasileiro de Sistemas de Informação*. [S.l.: s.n.], 2011.

GARG, S.; MISHRA, R. Trs: system for recommending semantic web service composition approaches. In: *Information Technology, 2008. ITSim 2008. International Symposium on*. [S.l.: s.n.], 2008. v. 2.

HAAS, H. *Designing the architecture for Web services*. Maio 2003. Último acesso 08 de Julho de 2011. Disponível em: `http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html`.

KATZ, M. *Rich Faces*. 1. ed. [S.l.]: Apress, 2008. ISBN 9781430210559.

KOPECKY, J.; VITVAR, T.; BOURNEZ, C.; FARRELL, J. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Computing*, v. 11, n. 6, p. 60–67, 2007.

KRUG, A.; KüSTER, U.; RUHMER, C. *OPOSSum Online Portal for Semantic Services*. Agosto 2009. Último acesso 08 de Julho de 2011. Disponível em: `http://fusion.cs.uni-jena.de/opossum/index.php`.

LAUSEN, H.; POLLERES, A.; ROMAN, D. Web service modeling ontology (wsmo). *W3C Member Submission*, Junho 2005. Último acesso em 07 de Junho de 2011. Disponível em: `http://www.w3.org/Submission/WSMO/`.

LEE, T.; HENDLER, J.; LASSILA, O. The semantic web. *Scientific American*, v. 284, n. 5, p. 34–43, 2001.

LEME, N. G. M. Um sistema de padrões para injeção de falhas por software. Dissertação de Mestrado - Universidade Estadual de Campinas. 2001.

LOOKER, N.; MUNRO, M.; XU, J. Simulating errors in web services. *International Journal of Simulation Systems, Science & Technology*, Citeseer, v. 5, n. 5, p. 29–37, 2004.

LYU, M. R. *Software Fault Tolerance*. [S.l.]: John Wiley & Sons, 1995.

MARTIN, D.; BURSTEIN, M.; HOBBS, J.; LASSILA, O.; MCDERMOTT, D.; MCILRAITH, S.; NARAYANAN, S.; PAOLUCCI, M.; PARSIA, B.; PAYNE, T.; SIRIN, E.; SRINIVASAN, N.; SYCARA, K. *OWL-S Semantic Markup for Web Services*. Novembro 2004. Último acesso em 09 de Julho de 2011. Disponível em: `http://www.w3.org/Submission/OWL-S/`.

MCGUINNESS, D. L.; HARMELEN, F. van. *OWL Web Ontology Language*. 2004. Último acesso em 29 de junho de 2011. Disponível em: `http://www.w3.org/TR/owl-features/`.

MICROSOFT. *DCOM: Distributed Component Object Model*. 1995. Último acesso em 09 de Julho de 2011. Disponível em: `http://www.microsoft.com/com/default.mspx`.

OMG. *CORBA: Common Object Request Broker Architecture*. 1995. Último acesso em 07 de Julho de 2011. Disponível em: `http://www.corba.org/`.

ORACLE. *JAVA RMI : Remote Method Invocation*. 1995. Último acesso em 18 de Junho de 2011. Disponível em: `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`.

ROSA, A. C. A. Uma arquitetura reflexiva para injetar falhas em aplicações orientadas a objetos. Dissertação de Mestrado - Universidade Estadual de Campinas. 1998.

SAMPAIO, C. *SOA e Web Services em Java*. [S.l.]: Brasport, 2007.

SELIC, B. *Fault tolerance techniques for distributed systems*. 2004. Último acesso em 30 de junho de 2011. Disponível em: `http://www-128.ibm.com/developerworks/rational/library/114.html`.

TANENBAUM, A. S.; STEEN, M. V. *Sistemas Distribuídos: Princípios e Paradigmas*. [S.l.]: Prentice Hall Brasil, 2007.