3
JAAF

O JAAF-S é um *framework* que basea-se na experiência adquirida durante o desenvolvimento do seu antecessor (JAAF 1.0) a fim de fornecer suporte ao desenvolvimento de agentes auto-adaptativos orientados a serviços. A principal diferença entre o JAAF-S e o JAAF 1.0 esta na adição de um conjunto de mecanismos para monitoramento, descoberta, seleção e disponibilização de serviços em tempo de execução.

Neste capítulo, inicialmente a idéia geral do *framework* JAAF 1.0 e seus respectivos módulos é apresentada, seguida pela descrição do *framework* JAAF-S.

3.1
JAAF 1.0

O JAAF 1.0 é uma extensão do JADE [1] (Bellifemine et al. 2007) e tem como objetivo fornecer mecanismos que possibilitem o desenvolvimento de agentes capazes de realizar auto-adaptações guiadas por *control-loops* onde cada um deles representa a sequência de execução de um conjunto de atividades.

A realização de auto-adaptação no JAAF 1.0 é possibilitada através da implementação do completo *control-loop* proposto pela IBM (IBM 2003), o qual é composto de 4 atividades (Figura 3.1 apresenta a sequência de execução das atividades):

Collect coleta, filtra e estrutura os dados coletados a partir do monitoramento de uma aplicação;

Analyze analisa os dados coletados pela atividade *Collect* a fim de detectar problemas e sugerir soluções;

Decision seleciona dentre as várias soluções sugeridas pela atividade *Analyze* aquela que será utilizada;

[1] Um framework, em conformidade com o padrão FIPA (Odell 1997), para implementar sistemas multi-agentes (MAS) desenvolvidos em Java

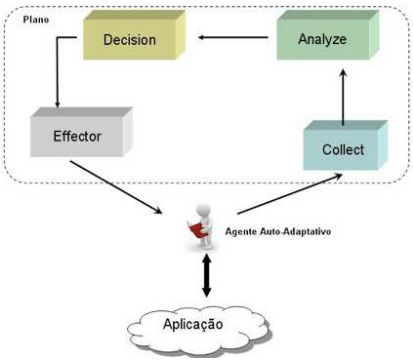

Figura 3.1: *Control-loop* do JAAF 1.0

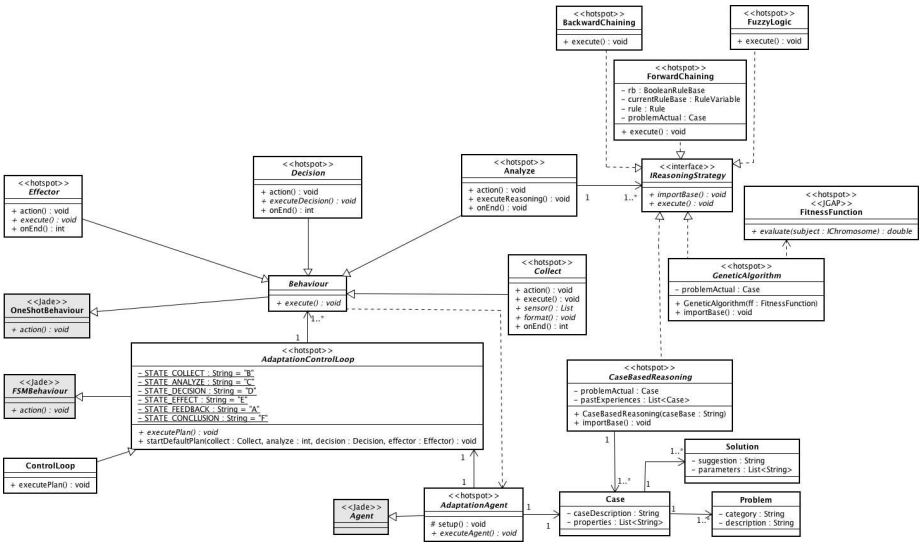

Figura 3.2: Classes do JAAF 1.0

Effector realizar as ações necessárias para efetivação da solução selecionada pela atividade *Decision*.

3.1.1 Detalhes do JAAF 1.0

A Figura 3.2 representa as principais classes do JAAF 1.0. Os agentes auto-adaptativos são representados pela classe *AdaptationAgent* e os *control-loops* de auto-adaptação pela classe *AdaptationControlLoop*. Esta última estende o comportamento *FSMBehaviour* do Jade (Ver Seção 2.2) que permite a implementação de autômatos finitos compostos de atividades representadas pela classe *Behaviour*.

Como mencionado na seção 3.1, o JAAF 1.0 fornece um *control-loop*, representado pela classe *ControlLoop*, que é composto pelas atividades *Collect* (representada pela classe *Collect*), *Analyze* (representada pela classe *Analyze*),

Decision (representada pela classe *Decision*) e *Effector* (representada pela classe *Effector*), cada uma destas atividades estende a classe *Behaviour*.

Note que apesar do JAAF fornecer o *control-loop* supracitado, ele oferece, como pode ser verificado em (Neto et al. 2009a) (Costa et al. 2010), a flexibilidade necessária para implementação de diferentes *control-loops* bastando para isto utilizar as atividades já disponibilizadas pelo JAAF 1.0 ou criar outras estendendo a classe *Behaviour*, e estenter a classe *AdaptationControlLoop* visando implementar um autômato finito que defina a sequência de execução de tais atividades. Desta forma, tornando a abordagem ainda mais generalizável e reutilizável em diferentes classes de sistemas.

Abaixo descrevemos em mais detalhes cada uma das atividades fornecidas pelo JAAF 1.0.

Collect Esta atividade, representada pela classe *Collect*, basea-se em dois conceitos para realizar a coleta, filtragem e estruturação dos dados coletados a partir do monitoramento de uma aplicação: *sensor* (representado pelo método *sensor*) e *format* (representado pelo método *format*). No *sensor* deve ser definido onde os dados necessários para realização da auto-adaptação deverão ser coletados (banco de dados, *log*, etc.). No *format* deve ser definido como os dados coletados a apartir do *sensor* devem ser estruturados de forma que os mesmo possam ser entendidos pelas atividades seguintes. Especificamente, o *format* deve utilizar os dados coletados para povoar os atributos *caseDescription* e *properties* de uma instância da classe *Case*. O primeiro atributo representa uma descrição geral sobre a aplicação e o segundo representa diferentes informações sobre a execução da aplicação, a exemplo quais módulos da aplicação foram utilizados.

Para utilizar a atividade *Collect* é necessário estender a classe *Collect* e implementar os métodos *sensor* e *format* como descrito acima.

Analyze A atividade *Analyze*, representada pela classe *Analyze*, utiliza métodos de raciocínio a fim de analisar os dados coletados, detectar problemas e sugerir soluções para a atividade *Decision*. O JAAF 1.0 prover a infra-estrutura necessária para implementação de três mecanismos de raciocínio:

Raciocínio Baseado em Regras (RBR) (Costa et al. 2008): Visando prover suporte para implementação do RBR, o módulo de regras fornecido pelo *framework* DRP-MAS (Costa et al. 2008) foi incorporado ao JAAF 1.0. Mas, embora três diferentes algoritmos de raciocínio baseado em regras sejam oferecidos: *foward chainning* (representado pela

classe *Foward Chaining*), *Backward chainning* (representado pela classe *BackwardChaining*) e *fuzzy logic* (representado pela classe *FuzzyLogic*).

Nosso foco principal está no algoritmo *foward chaining*. Acreditamos que essa abordagem seja muito útil pois a mesma basea-se em informações disponíveis para inferir novos fatos. No JAAF 1.0 tal algoritmo utiliza uma base de regras que são instanciadas com os dados coletados a partir do monitoramento da aplicação a fim de detectar a existência de diferentes problemas . Quanto mais informações forem coletadas, mais precisa será a detecção.

Para utilizar o RBR supracitado é necessário estender a classe *Foward-Chaining*, implementar o método *importBase* especificando nele o conjunto de regras que servirá como base de conhecimento. Neste caso, classes como *BooleanRuleBase*, *Rule* e *RuleVariable* oferecidas por (Bigus and Bigus 2001) deverão ser utilizadas. O JAAF 1.0 executará no método *execute* o algoritmo *foward chaining* e utilizará o conjunto de problemas detectados para povoar o conjunto de problemas (cada problema é representado pela classe *Problem*) da instância da classe *Case* mencionada na atividade *Collect*.

Raciocínio Baseado em Casos (RBC) : Como mencionado na seção 2.3, o RBC resolve um problema atual baseando-se em experiências passadas. No JAAF 1.0 o problema atual é composto pelos dados coletados pela atividade *Collect* e os problemas detectados pela aplicação do RBR.

Para utilizar o RBC é necessário estender a classe *CaseBasedReasoning*, passando como parâmetro o endereço de um arquivo XML que contém o conjunto de experiências passadas onde cada uma delas possui a estrutura da classe *Case*. Ou seja, uma descrição da aplicação, um conjunto de características da aplicação, o conjunto de problemas detectados e o conjunto de soluções (cada solução é representada pela classe *Solution*) utilizadas para solucionar tais problemas.

Por fim, implementar no método *execute* o algoritmo que irá verificar a similaridade entre o problema atual e cada uma das experiências passadas, e deve retornar o conjunto de soluções das experiências passadas mais semelhantes. Tais soluções devem ser utilizadas para povoar o conjunto de soluções da instância da classe *Case* mencionada na atividade *Collect*.

Algoritmo Genético (AG) : Ao incorporar a biblioteca (Chen et al. 2001), o *framework* consegue prover uma infra-estrutura para aplicação do

algoritmo genético. Para a utilização do AG é necessário estender as classes *FitnessFunction* e *GeneticAlgorithm* . Na primeira no método *evaluate* deve ser implementada a função de avaliação (Ver Seção 2.5) do algoritmo genético e na segunda o método *execute* deve verificar quais das soluções deverão povoar o conjunto de soluções da instância da classe *Case* mencionada na atividade *Collect*.

Decision Esta atividade, representada pela classe *Decision*, é responsável por receber as soluções sugeridas pela atividade *Analyze* e selecionar uma delas para ser executada pelo agente. Para isto, é necessário estender a classe *Decision* e implementar a estratégia de seleção no método *executeDecision*.

Effector É responsvel por receber a solução sugerida pela atividade *Decision*, e executá-la. Para isso, é necessário estender a classe *Effector* e implementar a lógica de execução no método *execute*.

3.1.2 *Hot-spots* e *frozen-spots* do JAAF 1.0

Considerando que o JAAF 1.0 estende o JADE, os *frozen-spots* e *hot-spots* do JADE são também *hot-spots* e *frozen-spots* do JAAF 1.0. Segue abaixo os *hot-spots* específicos do JAAF 1.0:

Agente (classe *AdaptationAgent*) : estendendo tal classe e implementando o método *executeAgent* é possível definir diferentes sequências de comportamentos incluindo o comportamento que representa o *control-loop*;

Planos de auto-adaptação (classe *AdaptationControlLoop*) : é possível definir novos *control-loop*. JAAF 1.0 já fornece um *control-loop* padrão (Figura 3.1) composto por um conjunto de atividades (*Collect*, *Analyze*, *Decision* e *Effector*) que podem ser instanciadas;

Atividades (classe *Behaviour*) : é possível definir novas atividades a serem chamadas pelo *control-loop* de auto-adaptação;

Mecanismos de Raciocínio : Como já descrito na Seção 3.1.1, o JAAF 1.0 oferece a infra-estrutura necessária para três tipos de mecanismos de raciocínio: RBC, AG e RBR. Mas, outros mecanismos podem ser adicionados implementando a interface *IReasoningStrategy*.

Em suma, para implementar um agente auto-adaptativo utilizando o JAAF 1.0, os seguintes passos devem ser realizados: (i) definir o *control-loop* de auto-adaptação estendendo a classe *AdaptationControlLoop* ou utilizando o *control-loop* padrão; (2) definir as atividades que compõem tal *control-loop* utilizando as atividades já providas pelo framework ou criando outras a partir da classe *Behaviour*; e, finalmente, (3) criar um agente auto-adaptativo estendendo a classe *AdaptationAgent* e implementar no método *executeAgent* o conjunto de comportamentos a serem executados incluindo o *control-loop* de auto-adaptação.

3.2 JAAF-S

O JAAF-S é uma evolução do *framework* JAAF 1.0 e tem como objetivo fornecer mecanismos que possibilitem a implementação de agentes auto-adaptativos orientados a serviços. Para tanto, o JAAF-S possibilita a implementação de agentes capazes de realizar as seguintes tarefas relacionadas a um WS: (i) monitoramento, (ii) detecção de falhas provenientes da sua execução, (iii) descoberta de novos WS capazes de solucionar tais falhas, (iv) seleção do melhor dentre os vários descobertos e, finalmente, (v) notificações e configurações necessárias para disponibilização do WS selecionado.

Assim como o JAAF 1.0, o JAAF-S utiliza uma implementação do completo *control-loop* de auto-adaptação proposto pela IBM (IBM 2003), mas agora suas atividades possuem a capacidade de atuar sobre WSs como descrito abaixo (Figura 3.3 apresenta a sequência de execução das atividades do JAAF-S onde é importante notar que agora temos as atividades atuando sobre *Web Services* ao invés de uma aplicação como apresentado na Figura 3.1):

Collect : Utiliza tecnologias para monitoramento de eventos, especificamente mensagens enviadas por outros agentes, e WS objetivando extrair informações que sirvam de base para detecção de falhas e auto-adaptação. Adicionalmente, estrutura tais informações de forma que elas possam ser entendidas pelas atividades seguintes;

Analyze : Esta fase basea-se nas informações coletadas na atividade *Collect* e utiliza um conjunto de mecanismos de raciocínio para detecção de problemas e descoberta de novos WS.

Decision : Esta atividade basea-se na execução de um conjunto de mecanismos de seleção para escolher, dentre os vários descobertos pela atividade *Analyze*, o serviço que melhor solucione os problemas que levaram à necessidade da auto-adaptação;

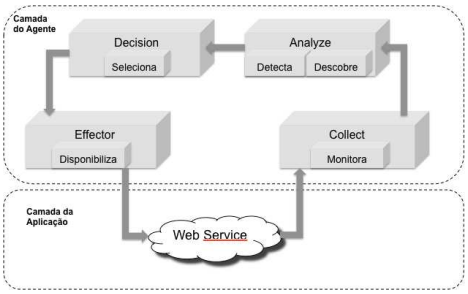

Figura 3.3: *Control-loop* do JAAF-S

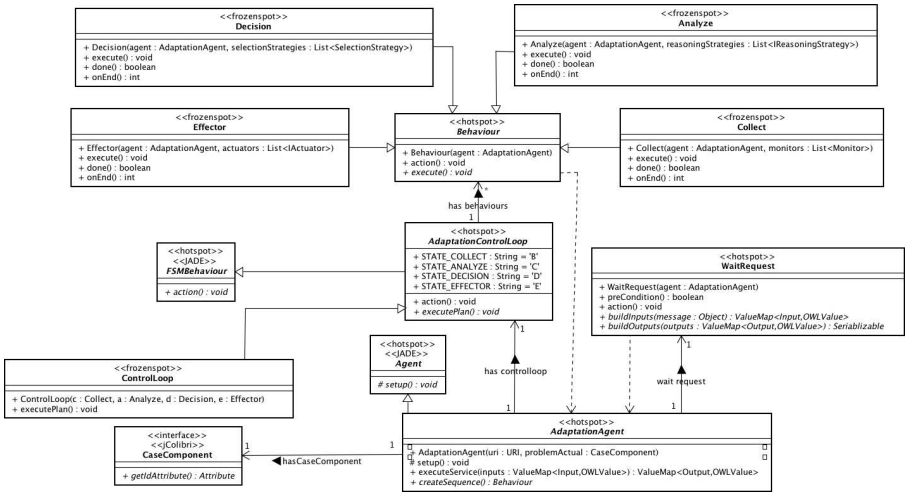

Figura 3.4: Diagrama de Classes do JAAF-S

Effector : Realiza as notificações e configurações necessárias para disponibilização do serviço selecionado pela atividade *Decision*.

3.3 Detalhes do JAAF-S

O diagrama de classes apresentado na figura 3.4 apresenta as principais classes e métodos do *framework* JAAF-S. Assim como o JAAF 1.0, no JAAF-S os agentes auto-adaptativos são representados pela classe *AdaptationAgent* e os *control-loop* de auto-adaptação pela classe *AdaptationControlLoop*. Esta última estende a classe *FSMBehaviour* do JADE que fornece suporte para implementação de autômatos finitos compostos de atividades (entenda comportamentos do JADE - Ver Seção 2.2) representadas pela classe *Behaviour*.

Como mencionado na seção 3.2, o JAAF-S também oferece o *control-loop* de auto-adaptação proposto pela IBM, o qual é representado pela classe *Controlloop* onde está definido a ordem de execução das atividades (Figura 3.3): *Collect* (Seção 3.3.1), *Analyze* (Seção 3.3.2), *Decision* (Seção 3.3.3) e

Effector (Seção 3.3.4). Para utilizar o *control-loop* supracitado é necessário instanciar a classe *ControlLoop* passando como parâmetros as instâncias das classes que representam as atividades que o compõem.

Note que diferentemente do JAAF 1.0, no JAAF-S tais atividades são *frozen-spots* e possuem algumas diferenças que serão descritas nas seções seguintes.

Adicionalmente, o JAAF-S fornece um comportamento chamado *WaitRequest* (Ver Figura 3.4) que é responsável por receber os dados enviados por outros agentes, formatá-los, executar o WS utilizando o método *executeService* presente na classe *AdaptationAgent* e formatar o resultado proveniente da execução do WS para que o mesmo possa ser enviado para o agente que requisitou o serviço. Para tanto, os dois métodos *buildInputs* e *buildOutputs* do comportamento *WaitRequest* precisam ser implementados. No primeiro os dados enviados pelos agentes requisitantes do serviço devem ser formatados de forma que os mesmo possam servir como entrada para execução do WS. No segundo o resultado proveniente da execução do serviço deve ser formatado de forma que o mesmo possa ser enviado e entendido pelo agente requisitante.

Em suma, para implementar um agente auto-adaptativo utilizando o JAAF-S é necessário estender a classe *AdaptationAgent* passando os seguintes parâmetros: (i) a URI [2] da ontologia OWL-S que descreve o serviço que deverá ser utilizado inicialmente pelo agente JAAF-S; (ii) uma implementação da interface *CaseComponent* (Ver Figura 3.4), a qual representa o cenário que pode levar a necessidade de auto-adaptação, ou seja, os dados coletados a partir do monitoramento de eventos e WS (como mencionado na Seçãsao 3.2), e (iii) implementar o método *createSequence* (Ver classe *AdaptationAgent*) o qual deve retornar um comportamento do tipo *Behaviour* (Ver Seção 2.2) que contém a sequência de execução dos comportamentos que o agente deve executar, incluindo os comportamentos *WaitRequest* e o *control-loop* de auto-adaptação.

Para facilitar o entendimento das atividades do *control-loop*, estas são explicadas em mais detalhes nas seções seguintes junto com seus respectivos diagramas de classes.

3.3.1 Collect

Este é o primeiro passo executado pelo *control-loop* provido pelo JAAF-S. Ele é responsável por monitorar a execução de serviços, coletar mensagens enviadas por agentes que requisitaram a execução dos mesmos e, por fim, filtrar

[2] *(Uniform Resource Identifier)*

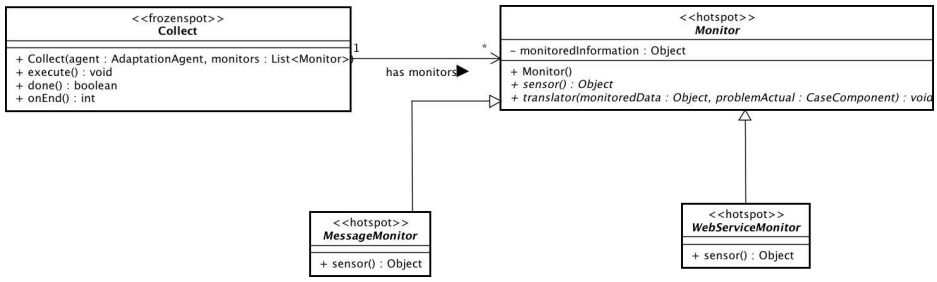

Figura 3.5: Atividade *Collect*

e estruturar os dados coletados a partir de tais tarefas de forma que eles possam ser entendidos pelas atividades seguintes.

A Figura 3.5 apresenta o diagrama de classes da atividade *Collect* onde temos a classe *Collect* responsável por gerenciar a execução do conjunto de monitores representados pela classe abstrata *Monitor*, a qual possui os métodos abstratos *sensor* e *translator* representando duas importantes tarefas no que diz respeito à monitoramento:

(sensor) No *sensor* deve ser especificado o que monitorar e quais dados devem ser coletados durante o monitoramento. Neste caso, o JAAF-S oferece: (i) um mecanismo (representado pela classe *MessageMonitor*) que coleta a última mensagem enviada pelo agente que requisitou a execução do serviço e (ii) utiliza a API (Perry 2002) para coletar informações sobre o WS em uso, tais como: tempo de respota, número de execuções realizadas com sucesso ou falha, etc. Tal funcionalidade é disponibilizada pela classe *WebServiceMonitor*;

(translator) No método *translator* devem ser implementados mecanismos que recebem os dados coletados pelo sensor atual (representados pelo parâmetro *monitoredData* do tipo *Object*) e os dados coletados pelos sensores anteriormente executados (representados pelo parâmetro *problemActual* do tipo *CaseComponent*), e os estruturem de forma que os mesmos possam ser entendidos pelas atividades seguintes.

Após os monitores finalizarem sua execução a atividade *Analyze* inicia sua execução visando detectar problemas e, se necessário, descobrir serviços que possam solucionar tais problemas. Na prática, o raciocínio baseado em regras é aplicado visando detectar problemas e caso algum problema seja encontrado, o raciocínio baseado em casos é aplicado a fim de descobrir novos serviços. Caso nenhum problema seja detectado, o resultado proveniente da execução do WS será enviado para o agente requisitante.

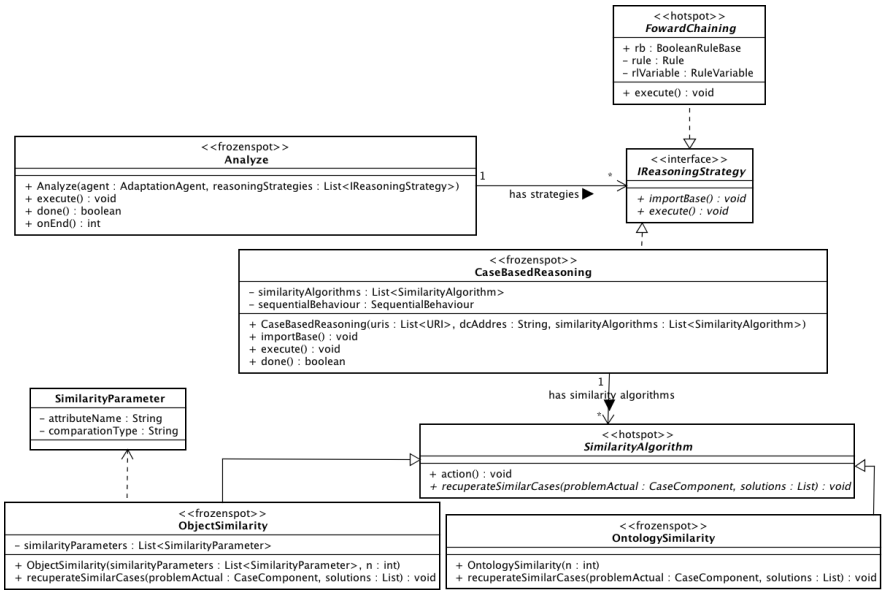

Figura 3.6: Atividade Analyze

3.3.2 Analyze

A atividade *Analyze*, representada pelo diagrama de classes apresentado na Figura 3.6, é responsável por analisar os dados coletados na atividade *Collect* visando detectar problemas e descobrir novos serviços.

No diagrama de classes apresentado na figura 3.6, temos a classe *Analyze* responsável por gerenciar a execução do conjunto de raciocínios representados pela interface *IReasoningStrategy*, a qual possui o método *importBase*, onde deve ser definido a base de conhecimento (por exemplo, base de casos ou regras) que servirá como base para o seu respectivo raciocínio, e método *execute*, onde a lógica de execução de cada raciocínio deve ser especificada. O JAAF-S utiliza o mecanismo de raciocínio baseado em regras já provido pelo JAAF 1.0 e evolui o mecanismo de raciocínio baseado em casos. Tais raciocínios são representados pelas classes *FowardChaining* e *CaseBasedReasoning*, respectivamente, e serão explicados em mais detalhes abaixo.

Raciocínio Baseado em Regras

Como citado anteriormente o JAAF-S incorporou o mecanismo de raciocínio baseado em regras provido pelo JAAF 1.0. No JAAF-S tal mecanismo ajuda detectar problemas como os descritos no cenário apresentado abaixo.

Considere que a atividade *Collect* esteja monitorando um determinado serviço e as seguintes informações são coletadas: (i) Número de falhas ocorridas durante a sua execução (representada pela variável *serviceFailure*), (ii) tempo

de resposta (representada pela variável *responseTime*) e (iii) autenticações realizadas com sucesso (representada pela variável *authenticationErro*).

Então, a partir de tais informações, é possível detectar problemas como: (Ver **R1**) ocorrência de falhas durante a execução do serviço; (Ver **R2**) tempo de resposta alto, e (Ver **R3**) ocorrência de erro de autenticação.

R1 : If *serviceFailure* $>$ 0 Then Problem =“serviceFail”;

R2 : If Problem !=“serviceFail” AND *responseTime* $>$ 10 Then Problem =“highResponseTime”;

R3 : If Problem !=“serviceFail” AND *authenticationError* == true Then Problem =“authenticationErro”;

Os passos a serem seguidos para utilizar o RBR no JAAF-S são semelhantes aos passos no JAAF 1.0. Ou seja, é necessário estender a classe *Foward-Chaining*, implementar o método *importBase* especificando nele o conjunto de regras que servirá como base de conhecimento. Neste caso, classes como *BooleanRuleBase*, *Rule* e *RuleVariable* oferecidas por (Bigus and Bigus 2001) deverão ser utilizadas. O JAAF-S executará no método *execute* o algoritmo *foward chaining* e obterá o conjunto de problemas detectados.

Por fim, como mencionado na seção 3.3.1, se algum problema é encontrado, o raciocínio baseado em casos é executado. Caso contrário, o resultado proveniente da execução do serviço é disponibilizado.

Raciocínio Baseado em Casos

Como mencionado na seção 2.3, o RBC resolve um problema atual baseando-se em experiências passadas. Embora as quatro fases que compõem o RBC sejam importante, aqui focamos em duas delas: Recuperação e Reuso.

No JAAF-S, o RBC é representado pela classe *CaseBasedReasoning*, a qual recebe como parâmetro: (i) um conjunto de serviços atualmente *on-line* representados pelo parâmetro *uris* do tipo *List<URI>*; (ii) o endereço de um arquivo *databaseconfig.xml* (mencionado na seção 2.4), este arquivo contém as configurações necessárias para acessar o banco de dados que armazena experiências passadas, o endereço de tal arquivo é passado pelo parâmetro *dcAddress* do tipo *String*; e (iii) o conjunto de algoritmos de similaridade que devem ser executados, representados pelo parâmetro *similarityAlgorithms* do tipo *List<SimilarityAlgorithm>*.

Adicionalmente, a classe *CaseBasedReasoning* implementa os métodos *importBase* e *execute* da interface *IReasoningStrategy*. No método *importBase* a API jColibri2 (Garcia 2008) é utilizada a fim de recuperar os casos (entenda

experiências passadas) armazenados no banco de dados cuja as configurações de acesso estão no arquivo XML *databaseconfig.xml* supracitado, é importante notar que utilizando API jColibri2 (Garcia 2008) o JAAF-S possibilita a recuperação de experiências passadas a partir de diferentes tipos de banco de dados, tais como Oracle (Oracles 2010), MySQL (Oracle 2010), etc. No método *execute* é realizada a execução dos algoritmos de similaridade representados pela classe *SimilarityAlgorithm*, a qual possui o método abstrato *recuperateSimilarCases* que recebe os dados coletados pela atividade *Collect*, juntamente, com os problemas detectados a partir da aplicação do RBR (ambos são passados através do parâmetro *problemActual* do tipo *CaseComponent*) e o conjunto de soluções já encontradas pelos algoritmos de similaridade anteriormente executados (representado pelo parâmetro *solutions* do tipo *List*), e deve utilizar tais dados, problemas e soluções para encontrar WSs que possam solucionar os problemas que levaram a necessidade de auto-adaptação.

Na seção 2.3 foi mencionado que que quando um novo problema surge, mecanismos de similaridade são aplicados entre o problema e os casos na BC, a fim de encontrar os casos mais similares com o tal problema. No JAAF-S dois mecanismos para verificação de similaridade são oferecidos: **Similaridade entre Objetos** e **Similaridade entre OWL-S**.

Similaridade entre Objetos: O objetivo deste mecanismo é encontrar as experiências passadas mais similares com o problema atual. Para tanto, ele estende a API jColibri2(Garcia 2008), a fim de fazer uso dos algoritmo de similaridade local e global, descritos na seção 2.4. Tais algoritmos são aplicados entre os dados coletados na atividade *Collect*, juntamente, com os problemas detectados a partir da aplicação do RBR (ver seção 3.3.2), e o conjunto de casos (ou experiências passadas) recuperados a partir do banco de dados mencionado anteriormente.

Para melhor entendimento, imagine um agente A que utiliza um serviço, chamado *Travel Package*, monitorado por um Agente B implementado a partir do JAAF-S. A função do serviço é fazer reservas de pacotes de viagens para usuários. Para tanto, tal serviço recebe informações como: cidade de origem, cidade de destino, tipo de carro desejado para locação na cidade destino, o mês que pretende fazer a viagem , dia da partida, dia do retorno e tipo de acomodação. E retorna informações referentes as reservas do hotel, passagens aéreas e locação do carro.

Entretanto, após uma requisição do agente A. O Agente B detectou que o serviço *Travel Package* falhou durante sua execução. Então, o Agente B realiza a auto-adaptação tomando como problema atual (Ver seção 2.3) os dados enviados pelo Agente A, juntamente, com as inferências provenientes

da aplicação do RBR na atividade *Analyze*. O agente verifica quais casos recuperados do banco de dados são mais semelhantes com tal problema.

Como exemplo, considere que os dados enviados pelo agente *A*, juntamente, com o tipo de problema detectado pelo RBR na atividade *Analyze*, são como descritos abaixo em **Problema Atual**:

Problema Atual

- *Cidade de Origem*: São Paulo
  - *Cidade de Destino*: Maceió
  - *Transporte*: Honda Civic
  - *Mês*: Janeiro
  - *Dia da Partida*: 10
  - *Dia do Retorno*: 20
  - *Tipo de Acomodação*: 4 estrelas
  - *Problema*: "serviceFail"

Considere também que o conjunto de casos (onde cada caso, como mencionado na seção 2.3, são representados pelo par problema-solução) recuperados do banco de dados possuem a descrição do "Problema" idêntica a do **Problema Atual** apresentado anteriormente e como "Solução" um *Profile OWL-S* que descreve um serviço que no passado solucionou tal problema. Então, considerando que um dos casos tem informações como as apresentadas abaixo:

Caso (1) Problema – *Cidade de Origem*: São Paulo
- *Cidade de Destino*: Maceió
- *Transporte*: Civic
- *Mês*: Fevereiro
- *Dia da Partida*: 12
- *Dia do Retorno*: 25
- *Tipo de Acomodação*: 5 estrelas
- *Problema*: "serviceFail"

Solução – Esta solução contém o *Profile* OWL-S, apresentado na Figura 3.7, que descreve um serviço com as seguintes características:

Nome Pacote Viagem NordesteTurismo;

Entradas Os objetos dos tipos *Trecho* (representa os atributos cidade de origem e destino), *Carro* (representa o tipo de carro), *Periodo* (representa os atributos mês, dia da partida e retorno) e *TipoAcomodacao* (representando o tipo de acomodação);

Saídas O objeto do tipo *Confirmacao* que representa as reservas do hotel, passagens aéreas e locação do carro;

```
– <rdf:RDF xml:base="http://192.168.1.3:8080/OWLSRepository/PVNT.owl">
  – <service:Service rdf:ID="Service">
    – <service:presents>
        <profile:Profile rdf:ID="Profile"/>
      </service:presents>
    – <service:describedBy>
        <process:AtomicProcess rdf:ID="MyProcess"/>
      </service:describedBy>
    </service:Service>
  – <profile:Profile rdf:about="#Profile">
      <profile:serviceName>Pacote Viagem Nordeste Turismo</profile:serviceName>
    – <profile:hasInput>
      – <process:Input rdf:ID="Carro">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Carro
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
    – <profile:hasInput>
      – <process:Input rdf:ID="Periodo">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Periodo
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
  – <profile:hasInput>
    – <process:Input rdf:ID="TipoAcomodacao">
      – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#TipoAcomodacao
        </process:parameterType>
      </process:Input>
    </profile:hasInput>
  – <profile:hasInput>
    – <process:Input rdf:ID="Trecho">
      – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Trecho
        </process:parameterType>
      </process:Input>
    </profile:hasInput>
  – <profile:hasOutput>
    – <process:Output rdf:ID="Confirmacao">
      – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Confirmacao
        </process:parameterType>
      </process:Output>
    </profile:hasOutput>
      – <j.1:serviceParameter>
        – <j.1:ServiceParameter>
            <j.1:serviceParameterName>Tempo de Resposta</j.1:serviceParameterName>
          – <j.1:sParameter>
              <j.0:Qualidade rdf:about="/OWLRepository/TravelPackage.owl#bom"/>
            </j.1:sParameter>
          </j.1:ServiceParameter>
        </j.1:serviceParameter>
      – <j.1:serviceParameter>
        – <j.1:ServiceParameter>
            <j.1:serviceParameterName>Escalabilidade</j.1:serviceParameterName>
            <j.1:sParameter rdf:resource="/OWLRepository/TravelPackage.owl#excelente"/>
          </j.1:ServiceParameter>
        </j.1:serviceParameter>
      – <j.1:serviceParameter>
        – <j.1:ServiceParameter>
            <j.1:serviceParameterName>Seguranca</j.1:serviceParameterName>
          – <j.1:sParameter>
              <j.0:Qualidade rdf:about="/OWLRepository/TravelPackage.owl#excelente"/>
            </j.1:sParameter>
          </j.1:ServiceParameter>
        </j.1:serviceParameter>
    </profile:Profile>
  – <process:AtomicProcess rdf:about="#MyProcess">
      <process:hasOutput rdf:resource="#Confirmacao"/>
      <process:hasInput rdf:resource="#Carro"/>
      <process:hasInput rdf:resource="#TipoAcomodacao"/>
      <process:hasInput rdf:resource="#Periodo"/>
      <process:hasInput rdf:resource="#Trecho"/>
      <service:describes rdf:resource="#Service"/>
    </process:AtomicProcess>
  </rdf:RDF>
```

Figura 3.7: OWL-S: Pacote de Viagem Nordeste Turismo

Propriedades Tempo de resposta, segurança e escalabilidade com qualidades *bom*, *excelente* e *excelente*, respectivamente;

Então, a similaridade entre o **Problema Atual** e o problema descrito no Caso (1) é calculada como apresentada na tabela 3.1. Onde a coluna **Similaridade Local** nas linhas 2, 3 ,4 e 9 apresenta o resultado da comparação do tipo *Equal* (Ver Seção 2.4) aplicada nos atributos **Cidade Origem**, **Cidade Destino**, **Transpote** e **Problema**. Nas linhas 5 e 8 temos o resultado da comparação do tipo *EnumDistance* (Ver Seção 2.4) aplicada nos atributos **Tipo de Acomodação** (1,2,3,4 e 5 estrelas) e **Mês**(Janeiro, Fevereiro, Março, Abril, Maio, Junho, Julho, Agosto, Setembro, Outubro, Novembro e Dezembro). E nas linhas 6 e 7 temos o resultado da comparação do tipo *Interval* (Ver Seção 2.4), sendo o intervalo entre 1 e 31, aplicada nos atributos **Dia da Partida** e **Dia do Retorno**. Por fim, a linha **Similaridade Global** apresenta o valor retornado pelo algoritmo de similaridade global (Ver Seção 2.4) aplicado sobre as similaridades locais.

Tabela 3.1: Similaridade entre o Problema Atual e o problema descrito no Caso (1)

1	**Atributos**	**Problema Atual**	**Problema do Caso (1)**	**Similaridade Local**
2	**Cidade Origem**	São Paulo	São Paulo	1
3	**Cidade Destino**	Maceió	Maceió	1
4	**Transporte**	Civic	Civic	1
5	**Mês**	Janeiro	Fevereiro	0,917
6	**Dia da Partida**	10	12	0.94
7	**Dia do Retorno**	20	25	0.84
8	**Tipo de Acomodação**	4 estrelas	5 estrelas	0.8
9	**Problema**	“serviceFailure”	“serviceFailure”	1
Similaridade Global				0,94

Por fim, considerando que o “Caso 1” é o mais semelhante com o “Problema Atual”, a solução do “Caso 1” será enviada para o segundo mecanismo disponibilizado pelo JAAF-S, o qual verifica a similaridade entre os *Profiles* especificados pelas soluções dos casos mais similares com o problema atual e o conjunto de *Profiles* que descrevem serviços atualmente *on-line*.

Para utilizar o mecanismo que verifica similaridade entre objetos, é necessário instanciar a classe *ObjectSimilarity* passando como parâmetros o nú-

mero máximo de casos similares que devem ser enviados para o mecanismos de similaridade seguinte, e o nome dos atributos a serem verificados durante a aplicação do algoritmo de similaridade local juntamente com o tipo de comparação a ser aplicada em cada um deles, por exemplo *Equal*, *EnumDistance*, etc. Tais atributos e seus respectivos tipos de comparação podem ser especificado a partir da classe *SimilarityParameter*.

Similaridade entre OWL-S:O objetivo do mecanismos em questão é verificar o nível de similaridade entre as soluções enviadas pelo mecanismo anteriormente descrito e um conjunto de *Profiles* descrevendo serviços atualmente *on-line*. Para tanto, tal mecanismo estende o algoritmo proposto por (Paolucci et al. 2002) e disponibiliza sua implementação utilizando a API (Sirin and et. al. 2009). Tal extensão considera dois parâmetros adicionais além dos parâmetros *hasInput* e *hasOutput* (Ver Seção 2.1) já considerados em (Paolucci et al. 2002). O primeiro parâmetro é *serviceName* (Ver Seção 2.1), que descreve o nome do serviço. O segundo parâmetro considerado pela extensão é *serviceParameter*, onde pode ser definido uma lista de propriedades (tais como tempo de resposta, segurança, escalabilidade, etc.) informando a qualidade do serviço sendo provido. Este parâmetro é composto de dois atributos: *serviceParameterName* que descreve o nome da propriedade, e *sParameter* que representa o valor do parâmetro (Ver Seção 2.1). Tal mecanismo é implementado utilizando o algoritmo 5, o qual recebe dois serviços como entrada e retorna a similaridade entre eles. O primeiro representa a descrição do serviço desejado e o segundo a descrição de um dos serviços atualmente *on-line*.

A linha 3 verifica a similaridade entre os atributos *serviceName* e as linhas de 7 a 16 verificam a similaridade entre os atributos *serviceParameter*. Onde primeiro é verificado a similaridade entre os atributos *serviceParameterName* na linha 10, caso sejam semelhantes o valor do parâmetro é verificado na linha 11.

A similaridade entre os *Inputs* é verificada entre as linhas 17 e 22, e os *Outputs* entre as linhas 23 e 28. O grau de similaridade entre dois *Inputs* ou *Outpus* depende da relação entre os conceitos associados aos *Outputs* ou *Inputs*. Por exemplo, considere um serviço que possui *Car* como *Input* e outro serviço possui *Vehicle* como *Input*. Dado o fragmento de uma ontologia apresentada na figura 3.8 , notamos que existe uma relação de herança entre eles.

O grau de similaridade é determinado pela distância mínima entre conceitos na árvore de taxonomia. A abordagem proposta em (Paolucci et al. 2002) é utilizada nesta dissertação, a qual diferencia entre quatro graus de similaridade de acordo com as regras apresentadas na função *degreeOfMatch* (Ver Algoritmo 6) que recebe dois parâmetros *IOOntReq*, representando o parâmetro *Input* ou

Algoritmo 5 similarity(Service OntReq, Service OntProv)

```
1: similarity = 0
2: attributeNumber = 0
3: if (OntReq.serviceName == OntProv.serviceName) then
4:    similarity = similarity + 1
5:    attributeNumber = attributeNumber + 1
6: end if
7: for all (ServiceParameter) sPReq in (OntReq.getServiceParamters()) do
8:    attributeNumber = attributeNumber + 1
9:    for all (ServiceParameter) sPProv in (OntProv.getServiceParamters())
      do
10:      if (sPReq.serviceParameterName == sP-
         Prov.serviceParameterName) then
11:         if sPReq.sParameter.equal(sPProv.sParameter) then
12:            similarity = similarity + 1
13:         end if
14:      end if
15:   end for
16: end for
17: for all (Input) iReq in (OntReq.getInputs()) do
18:    attributeNumber = attributeNumber + 1
19:    for all (Input) iProv in (OntProv.getInputs()) do
20:       similarity = similarity + degreeOfMatch(iReq, iProv)
21:    end for
22: end for
23: for all (Output) oReq in (OntReq.getOutputs()) do
24:    attributeNumber = attributeNumber + 1
25:    for all (Output) oProv in (OntProv.getOutputs()) do
26:       similarity = similarity + degreeOfMatch(oReq, oProv)
27:    end for
28: end for
29: return similarity/attributeNumber
```

Algoritmo 6 degreeOfMatch(OWLType IOOntReq, OWLType IOOntProv)

```
1: if IOOntReq.equal(IOOntProv) then
2:    return 1
3: end if
4: if IOOntProv.plugIn(IOOntReq) then
5:    return 0.75
6: end if
7: if IOOntReq.subsumes(IOOntProv) then
8:    return 0.5
9: end if
10: return 0
```

Output do serviço requisitado, e *IOOntProv*, representado o parâmetro *Input* ou *Output* do serviço atualmente *on-line*. A lógica de tais regras é apresentada

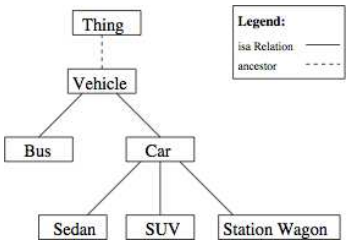

Figura 3.8: Fragmento de Ontologia

abaixo:

equal : Se *IOOntReq* e *IOOntProv* são equivalentes.

plugIn : Se *IOOntProv plugIn IOOntReq* então *IOOntProv* é um conjunto que inclui *IOOntReq*, ou seja, ele pode ser utilizado para substituir *IOOntReq* (Zaremski and Wing 1997). Por exemplo, um serviço que fornece qualquer tipo de veículos poderia ser utilizado por outro serviço que espera *Sedan*. Entretanto, estamos diante de um relação fraca entre *IOOntProv* e *IOOntReq*. Pois, nós podemos esperar que um serviço que possui um *output* de veículos fornece alguns tipos de veículos, mas não todo tipo de veículo;

subsumes : Se *IOOntReq subsumes IOOntProv*, estamos diante da lógica do inversa do *plugIn*, ou seja, *IOOntReq plugIn IOOntProv*. Então o provedor do serviço não satisfaz completamente a requisição;

fail : Ocorre *fail* quando nenhum das regras supracitadas ocorrem. Neste caso, a função *degreeOfMatch* retorna zero.

Para melhor entendimento considere a comparação entre a OWL-S descrita na solução do *Caso 1*, com o *Profile* OWL-S apresentado na Figura 3.9 que descreve um serviço com as seguintes características:

Nome Pacote Viagem Nordeste Paraíso;

Entradas Os objetos dos tipos *Trecho* (representa os atributos cidade de origem e destino), *Veiculo* (representa o tipo de carro), *Periodo* (representa os atributos mês, dia da partida e retorno) e *TipoAcomodacao* (representando o tipo de acomodação);

Saídas O objeto do tipo *Confirmacao* que representa as reservas do hotel, passagens aéreas e locação do carro;

Propriedades Tempo de resposta, segurança e escalabilidade com qualidades *medio*, *excelente* e *medio*, respectivamente;

```
– <rdf:RDF xml:base="http://192.168.1.3:8080/OWLRepository/PVNP.owl">
  – <service:Service rdf:ID="Service">
    – <service:presents>
        <profile:Profile rdf:ID="Profile"/>
      </service:presents>
    – <service:describedBy>
        <process:AtomicProcess rdf:ID="MyProcess"/>
      </service:describedBy>
    </service:Service>
  – <profile:Profile rdf:about="#Profile">
      <profile:serviceName>Pacote Viagem Nordeste Paraiso</profile:serviceName>
    – <profile:hasInput>
      – <process:Input rdf:ID="Veiculo">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Veiculo
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
    – <profile:hasInput>
      – <process:Input rdf:ID="Periodo">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Periodo
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
    – <profile:hasInput>
      – <process:Input rdf:ID="TipoAcomodacao">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#TipoAcomodacao
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
    – <profile:hasInput>
      – <process:Input rdf:ID="Trecho">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Trecho
          </process:parameterType>
        </process:Input>
      </profile:hasInput>
    – <profile:hasOutput>
      – <process:Output rdf:ID="Confirmacao">
        – <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://192.168.1.3:8080/OWLRepository/TravelPackage.owl#Confirmacao
          </process:parameterType>
        </process:Output>
      </profile:hasOutput>
    – <j.1:serviceParameter>
      – <j.1:ServiceParameter>
          <j.1:serviceParameterName>Tempo de Resposta</j.1:serviceParameterName>
        – <j.1:sParameter>
            <j.0:Quality rdf:about="TravelPackage.owl#medio"/>
          </j.1:sParameter>
        </j.1:ServiceParameter>
      </j.1:serviceParameter>
    – <j.1:serviceParameter>
      – <j.1:ServiceParameter>
          <j.1:serviceParameterName>Seguranca</j.1:serviceParameterName>
        – <j.1:sParameter>
            <j.0:Quality rdf:about="TravelPackage.owl#excelente"/>
          </j.1:sParameter>
        </j.1:ServiceParameter>
      </j.1:serviceParameter>
            – <j.1:serviceParameter>
              – <j.1:ServiceParameter>
                  <j.1:serviceParameterName>Escalabilidade</j.1:serviceParameterName>
                  <j.1:sParameter rdf:resource="TravelPackage.owl#medio"/>
                </j.1:ServiceParameter>
              </j.1:serviceParameter>
          </profile:Profile>
        – <process:AtomicProcess rdf:about="#MyProcess">
            <process:hasOutput rdf:resource="#Confirmacao"/>
            <process:hasInput rdf:resource="#Veiculo"/>
            <process:hasInput rdf:resource="#TipoAcomodacao"/>
            <process:hasInput rdf:resource="#Periodo"/>
            <process:hasInput rdf:resource="#Trecho"/>
            <service:describes rdf:resource="#Service"/>
          </process:AtomicProcess>
        </rdf:RDF>
```

Figura 3.9: OWL-S: Pacote de Viagem Nordeste Paraíso

Neste caso, a similaridade entre as ontologias pode ser verificada na Tabela 3.2. Onde a coluna **Similaridade** apresenta a similaridade entre os atributos das ontologias e a linha **Similaridade Total** apresenta a soma da similaridade dos atributos dividido pela quantidade de atributos.

Tabela 3.2: Similaridade a solução do Caso (1) e a *Profile* OWL-S da Figura 3.9

Atributos	**Solução do Caso (1)**	**OWL-S da Figura 3.9**	**Similaridade**
Service Name	Pacote Viagem Nordeste Turismo	Pacote Viagem Nordeste Paraíso	0
Parameter	Tempo de Resposta:bom	Tempo de Resposta:medio	0
	Segurança:excelente	Segurança:excelente	1
	Escalabilidade:excellent	Escalabilidade:medio	0
Inputs	Trecho	Trecho	1
	Periodo	Periodo	1
	TipoAcomodacao	TipoAcomodacao	1
	Carro	Veiculo	0.75
Outputs	Confirmacao	Confirmacao	1
Similaridade Total			0.64

Para utilizar tal mecanismo basta instanciar a classe *OntologySimilarity* passando como parâmetro o número máximo de *Profiles* que devem ser enviados para a atividade ou algoritmo seguinte.

3.3.3 Decision

A atividade *Decision*, representada pelo diagrama de classes apresentado na Figura 3.10, é responsável por decidir qual serviço será sugerido para atividade *Effector*.

No diagrama de classes apresentado na figura 3.10 temos a classe *Decision* responsável por gerenciar a execução do conjunto de mecanismos de seleção representados pela classe abstrata *SelectionStrategy* a qual declara o método abstrato *selectSolutions* que recebe a lista de serviços selecionados pelo mecanismo de seleção anteriormente aplicado e escolhe os melhores serviços. O JAAF-S oferece dois diferentes métodos de seleção: funções de utilidade (Petrucci 2008) e reputação (Maximilien and Singh 2001).

Função de Utilidade

O mecanismo de função de utilidade, representado pela classe abstrata *UtilityFunction*, avalia e seleciona cada serviço levando em consideração dife-

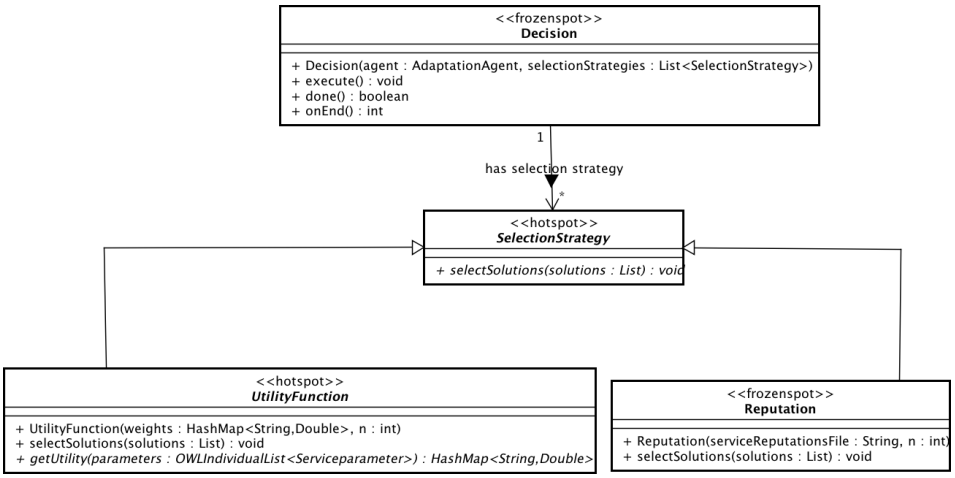

Figura 3.10: Atividade *Decision*

rentes dimensões de qualidade como tempo de resposta, custo, performance, segurança, escalabilidade, etc. Como descrito na Seção 2.1.2, tais características de um serviço são especificadas através do parâmetro *ServiceParameter* de um *Profile* OWL-S.

Neste contexto, considerando D como um conjunto de dimensões de qualidade de um serviço, nós adotamos uma abordagem baseada em utilidade, onde as preferências sobre as dimensões de qualidade são representadas como um número real w_d para cada $d \in D$, tal que $\sum_{\forall d \in D} w_d = 1$. Ainda para cada categoria ou valor (representado por c) de uma dimensão definimos uma função $u_d : c \rightarrow [0, 1]$, a fim de representar, num intervalo entre zero e um, o grau de utilidade de cada dimensão em um determinado contexto. Então, a importância de um serviço é dada aplicado a equação apresentada abaixo.

Para melhor entendimento tomemos como exemplo o serviço da figura 3.9, onde:

Serviço Pacote Viagem Nordeste Paraíso : Possui tempo de resposta (t) *medio*, segurança (s) *excelente* e escalabilidade (e) *medio*, com utilidade $u_t(medio) = 0.5$, $u_s(excelente) = 1$ e $u_e(medio) = 0.5$, respectivamente;

Então, considerando que o tempo de resposta tem peso $w_t = 0.2$, segurança $w_s = 0.4$ e escalabilidade $w_e = 0.4$ a importância do serviço é calculada da seguinte forma:

Serviço Pacote-Viagem-Nordeste-Paraíso : 0.2*0.5 + 0.4*1 + 0.4*0.5 = 0.7;

Por fim, considerando que o serviço **Pacote Viagem Nordeste Paraíso** é o de maior importância. Ele seria enviado para o próximo mecanismo de seleção.

Figura 3.11: Arquivo de Configuração da Reputação

Para implementar o mecanismos em questão através do JAAF-S, é necessário estender a classe *UtilityFunction* passando como parâmetro o número máximo de serviços que devem ser enviados para o próximo mecanismo de selação e uma *HashMap* onde a *Key* é o nome da dimensão e *Value* o peso dela. É necessário ainda implementar o método *getUtility* que recebe o conjunto de *ServiceParemeter* de cada serviço (tais serviços são passados por parâmetro através do método *selectSolutions* declarado na classe abstrata *SelectionStrategy* e implementado na classe *UtilityFunction*) e deve retornar uma *HashMap* tendo como *Key* o nome da dimensão da dimensão de qualidade e *Value* a utilidade dela.

Reputação

O mecanismo de reputação, representado pela classe *Reputation*, basease na reputação dos serviços para realizar a seleção. Para implementá-lo é necessário instanciar a classe *Reputation* passando como parâmetro o número máximo de serviços que devem ser selecionados e o endereço de um arquivo XML onde cada serviço possui duas *tags* (como apresentado na Figura 3.11): **URI** e **Reputation**. A primeira descrevendo a *URI* do serviço e a segunda a reputação do serviço.

Tal classe implementa no método *selectSolutions* da classe abstrata *SelectionStrategy* um mecanismo que procura no arquivo XML mencionado anteriormente, o nome dos serviços, passados como parâmetro pelo método *selectSolutions*, e irá selecionar aqueles com maior reputação.

É importante observar que o arquivo XML supracitado funciona como uma base onde as reputações podem ser atualizadas por diferentes agentes. Este trabalho não visa abordar as diferentes técnicas existentes para calcular reputação de WS (Maximilien and Singh 2001) (Maximilien and Singh 2002).

3.3.4

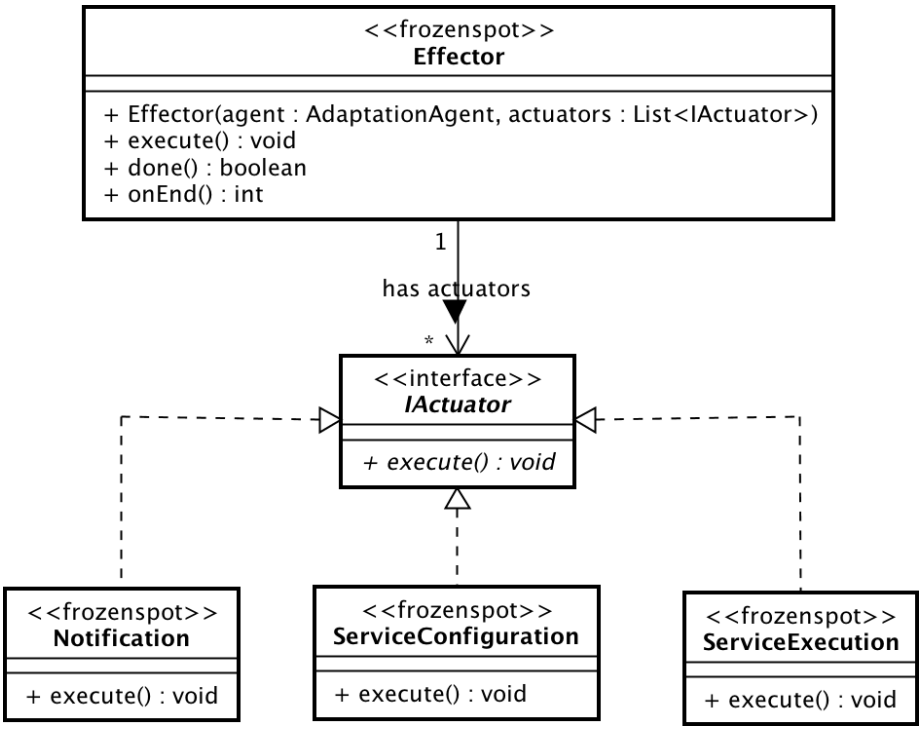

Figura 3.12: Atividade *Effector*

Effector

A atividade *Effector*, representada pelo diagrama de classes apresentado na Figura 3.12, realiza as notificações e configurações necessárias para disponibilização do serviço selecionado pela atividade *Decision*.

Na Figura 3.12 temos a atividade *Effector* representada pela classe *Effector* que é responsável por gerenciar os atuadores representados pela interface *IActuator*, os quais possuem as seguintes funcionalidades: (i)enviar para o agente que requisitou o serviço em uso atualmente, informações descrevendo o novo serviço selecionado, a exemplo nome, descrição, *inputs* e *outputs*. Tal funcionalidade é implementada no método *execute* da classe *Notification*; (ii) realizar as configurações necessárias para que o agente auto-adaptativo esteja apto a receber novas requisições de execução do novo serviço, monitorar, detectar problemas provenientes da execução do mesmo por fim, caso necessário adaptar-se novamente. Tal funcionalidade é implementada no método *execute* da classe *ServiceConfiguration*, e (iii) verificar se o serviço selecionado é diferente do serviço atualmente em uso e executa o comportamento *WaitRequest*, mencionado na seção 3.3, com os dados enviados pelo agente requisitante. Tal funcionalidade é implementada no método *execute* da classe *ServiceExecution*.

3.4 *Hot-spots* e *Frozen-spots* do JAAF-S

Considerando que JAAF-S estende o JADE, o *kernel* do JADE é também o *kernel* do JAAF-S e os *hot-spots* (ou pontos flexíveis) do JADE são os *hot-spots* do JAAF-S.

Os *frozen-spots* especificamente providos pelo JAAF-S são:

- Um *control-loop* de auto-adaptação seguindo a sequência de execução apresentada na Figura 3.3 e representado pela classe *ControlLoop* (Ver Seção 3.2 para maiores detalhes como instanciá-lo).
- Quatro atividades: *Collect* (Seção 3.3.1), *Analyze* (Seção 3.3.2), *Decision* (Seção 3.3.3) e *Effector* (Seção 3.3.4).
- Um mecanismo para monitoramento de WS e outro para interceptação de eventos (Seção 3.3.1).
- Dois mecanismos de similaridade(Seção 3.3.2): (i) Verifica similaridade entre objetos e (ii) Verifica similaridade entre ontologias OWL-S.
- Dois mecanismos de seleção (Seção 3.3.3): (i) Função de Utilidade e (ii) Reputação;
- Três mecanismos relacionados a atividade *Effector*: (i) realiza notificação de mudanças; (ii) realiza as configurações necessárias para disponibilização de novos serviços (Seção 3.3.4) e, por fim, (iii) toma as ações necessáras para execução do serviço.

Os principais *hot-spots* especificamente definidos no JAAF-S são:

- Planos de auto-adaptação (classe abstrata *AdaptationConrolLoop* na Figura 3.4): é possível definir novos *control-loops* e a sequência para executar as atividades dos *control-loops*, como pode ser visto em (Neto et al. 2009a) (Costa et al. 2010).
- Atividades (classe abstrata *Behaviour* na Figura 3.4): é possível definir novas atividades, como pode ser visto em (Neto et al. 2009a) (Costa et al. 2010).
- Monitoramento (classe abstrata *Monitor* na Figura 3.5): mecanismos para monitoramento podem ser adicionados estendendo a classe abstrata *Monitor*.
- Raciocínios (interface *IReasoningStrategy* na Figura 3.6): permite acoplar outros algoritmos de raciocínio a partir da implementação da interface *IReasoningStrategy*.

- Tecnicas para seleção de serviços (classe abstrata *SelectionStrategy* na Figura 3.10): técnicas de seleção podem ser incorporadas a apartir da implementação da classe abstrata *SelectionStrategy*
- Mecanismos para efetivação da solução (interface *IActuator* na Figura 3.12): diferentes efetuadores podem ser incorporados a apartir da implementação da interface *IActuator*.