4
JAAF+T: Framework de Autoteste para Agentes de Software

Neste capítulo é apresentado o framework Java self-Adaptive Agent Framework for self-Test (JAAF+T), que permite a criação de agentes de software autoadaptáveis capazes de coordenar seus testes a partir de informações mencionadas no capítulo anterior. Essa preocupação em coordenar testes deve-se a tendência do desenvolvimento de sistemas cada vez mais complexos capazes de autoadaptar seus comportamentos quando necessário. Relacionado a isso, o paradigma de sistemas multiagentes tem sido utilizado especialmente quando entidades pró-ativas, autônomas e distribuídas são representadas.

Atualmente, diversas abordagens já foram propostas para ajudar na construção de sistemas autoadaptativos. No entanto, poucos trabalhos, como, (Denaro et al., 2007; Stevens et al., 2007; Wen et al., 2005) proveem alguma solução para negociar com as seguintes preocupações de sistemas autoadaptativos: garantir a confiabilidade e segurança dos comportamentos autoadaptados. Uma das principais técnicas aplicadas para tratar essa preocupação é o uso do conceito autoteste. Autoteste permite ao agente testar a si mesmo a fim de garantir boas autoadaptações. A partir disso, torna-se possível avaliar se alguma adaptação sugerida é adequada ou não para atingir algum objetivo desejado.

Após analisar diversas abordagens oferecidas na literatura que realizam autotestes, percebeu-se que elas não tratam preocupações importantes, sendo parte delas relacionadas às informações apresentadas no capítulo anterior. Veja a seguir.

- Possibilitar a criação de diferentes processos de autoadaptação (*control-loops*).
- Testar a comunicação entre agentes que façam parte de um mesmo sistema, como, por exemplo, usando a API Jade Agent Testing Framework (JAT) (Coelho et al., 2007a; Coelho et al., 2007b), que permite a construção e execução de cenários de teste para SMA.

- Permitir a aplicação do conceito de autoteste em diferentes processos de autoadaptação.
- Permitir a execução de testes desenvolvidos por diferentes ferramentas ou APIs.
- Explicitar a relação dos artefatos que serão testados com seus respectivos testes.
- Definir condições de teste, isto é, diferentes conjuntos de entrada e saída de dados para um mesmo teste.
- Definir ordens de execução dos testes para validar algum artefato do sistema. Tal preocupação deve ser considerada já que podem existir dependências entre esses testes.
- Definir critérios de execução que devem ser respeitados na execução dos testes.
- Definir diferentes formatos de log para armazenar os resultados dos testes executados (ex: arquivos de texto, XML, etc.). Com isso, cada sistema pode definir diferentes maneiras de manipulá-los.

Portanto, o objetivo deste capítulo é apresentar o Java self-Adaptive Agent Framework for self-Test (JAAF+T) (Costa et al., 2010a; Costa et al., 2011a) que permite a criação de agentes de software capazes de realizar autoteste. Tais agentes usam como base as informações apresentadas no capítulo 3 para realizar a coordenação das execuções dos testes. O capítulo está organizado da seguinte maneira. Na Seção 4.1 é apresentada a visão geral do JAAF+T. Na Seção 4.2 são descritos os arquivos XML utilizados pelo framework. Tais arquivos representam as informações identificadas no capítulo 3 para ajudar na coordenação da execução dos testes, e servem como dados de entrada para o JAAF+T. Já na Seção 4.3 os pontos fixos e flexíveis do framework são apresentados. E finalmente a Seção 4.4 apresenta um passo a passo de como utilizar e instanciar o framework.

4.1. Visão Geral do JAAF+T

JAAF+T considera que antes de um agente concluir alguma autoadaptação, ele deve executar um conjunto de testes para validá-la, em especial o novo comportamento ou ação escolhida. Visando contemplar tal ideia, o JAAF+T permite: (i) a criação de diferentes processos de autoadaptação

executados por agentes JADE com o conceito de autoteste; (ii) a criação de diferentes atividades, que compõem os processos/planos de autoadaptação; (iii) oferece um conjunto de atividades pré-definidas, dentre elas a atividade de Teste e Validação que podem ser reusadas em diferentes processos; e (iii) utiliza um conjunto de arquivos XML pré-configurados, responsáveis por representar informações úteis para a coordenação dos testes.

Para facilitar o trabalho dos desenvolvedores e testadores, o JAAF+T provê um novo processo de autoadaptação baseado no control-loop padrão proposto pela IBM (IBM, 2003). Esse novo processo é composto por seis atividades (Coleta, Análise, Decisão, Teste, Validação e Efetuar), assim como ilustrado na Figura 5.

A atividade Coleta é responsável por recuperar dados que ajudam a identificar o problema e qual ação deve ser adaptada pelo agente de software. Além disso, essa atividade formata os dados recuperados para alguma estrutura que permita o entendimento e manipulação das outras atividades. Em seguida, na atividade Análise, é realizado um diagnóstico dos dados recolhidos para avaliar se haverá necessidade de realizar alguma autoadaptação.

Após o diagnóstico, a atividade Decisão procura por alguma ação (comportamento JADE, web-service, etc.) que permita ao agente alcançar o objetivo desejado. No entanto, antes de confirmar a escolha, a ação é enviada para a atividade de Teste, que é responsável por executar testes em tal ação. O agente sabe quais testes e como eles deverão ser executados, devido informações presentes em um conjunto de arquivos XML usados pelo framework que são explicados em detalhe mais a frente. Quando a execução é finalizada, os resultados dos testes são enviados para a atividade Validação, responsável por analisar esses resultados e decidir se a ação candidata é válida ou se outra ação deve ser escolhida. Caso a ação seja aprovada, a atividade Efetuar é executada para realizar a configuração necessária no agente de software para que possa ser executada. Caso contrário, a atividade Decisão é executada novamente para realizar a escolha de outra ação candidata.

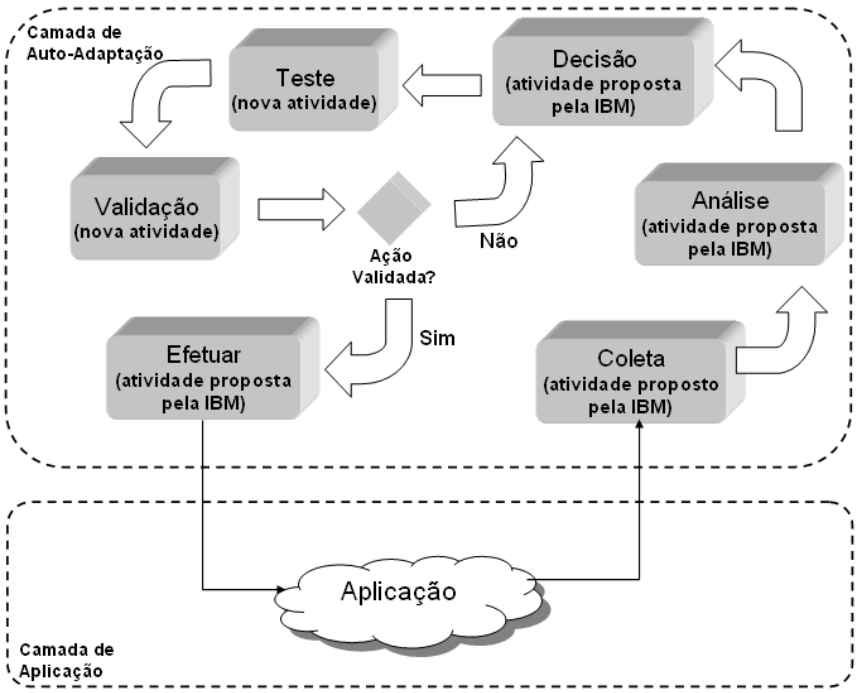

Figura 5. Control-loop provido pelo JAAF+T.

A Figura 6 ilustra um diagrama de classe do framework com suas principais classes. A partir do uso da classe *ControlLoop* torna-se possível definir diferentes planos de autoadaptação. Essa classe estende de *FSMBehaviour*, classe que permite a definição de novas atividades (extensão da classe *Activity*) baseadas no conceito de máquinas de estado finito.

O JAAF+T já provê o control-loop ilustrado na Figura 5 a partir da classe *CLWithSelfTest*, composta pelas seis atividades mencionadas anteriormente e representadas pelas classes *Collect* (Coleta), *Analyze* (Análise), *Decision* (Decisão), *Test* (Teste), *Validation* (validação) e *Effect* (Efetuar). Além disso, para permitir que dados possam ser manipulados pelas diferentes atividades, a classe *HandledData* foi definida.

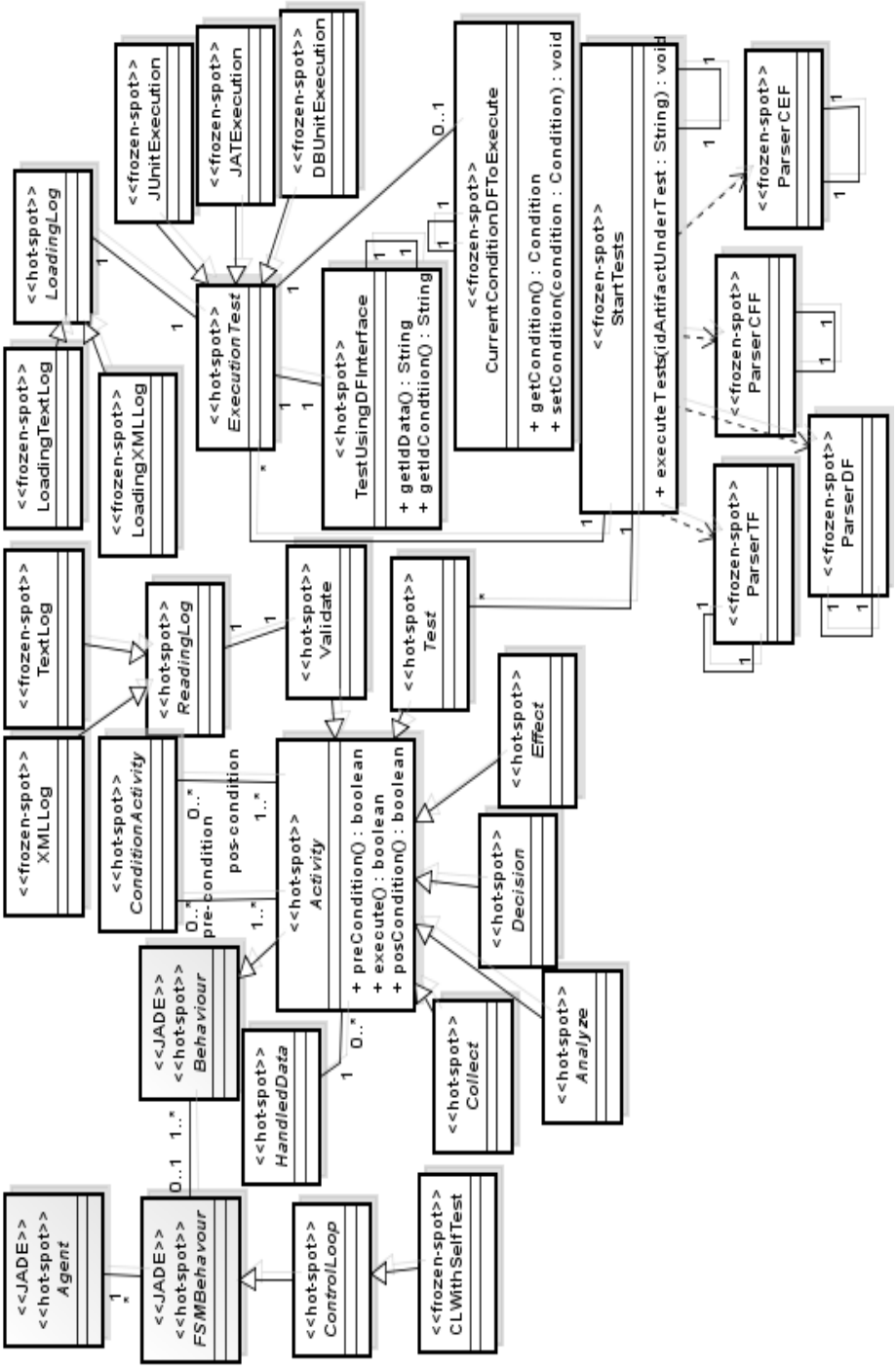

Figura 6. Diagrama de classe do JAAF+T.

As classes *Test* e *Validate* representam as principais atividades que permitem a aplicação do conceito de autoteste em um *control-loop*. A primeira classe executa um conjunto de testes em algum artefato do sistema a partir da classe *StartTests*. Já a segunda classe avalia os resultados dos testes executados a partir da leitura de logs (uso da classe ReadingLog). Assim, torna-se possível avaliar se a ação testada pode realmente ser executada e considerada pelo agente.

A descrição detalhada de quais testes poderão ser executados, qual é ordem de execução a ser seguida pelos testes, quais dados serão usados como entrada e/ou saída por cada teste, e a descrição dos possíveis critérios de execução dos testes a serem respeitados pelos agentes, são definidos, respectivamente, pelos arquivos TF.xml, CFF.xml, DF.xml e CEF.xml. Portanto, tais arquivos servem como dados de entrada para que os agentes do JAAF+T consigam coordenar os testes. Detalhes de cada arquivo são apresentados a seguir.

4.2. Arquivos XML

Como mencionado, o JAAF+T provê quatro arquivos XML: TF.xml, CFF.xml, DF.xml e CEF.xml. O arquivo TF.xml (*Test File*) descreve os testes que podem ser executados em um sistema. O arquivo CFF.xml (*Control-Flow File*) descreve os fluxos de execução (suítes) e a rastreabilidade dos artefatos testados com os testes. O DF.xml (*Data File*) representa os dados de entrada e saída de cada teste. Já o CEF.xml (*Criterion of Execution File*) descreve critérios de execução que podem ser adotados pelos agentes para decidir quais testes executar.

Para definir quais testes podem ser executados em um sistema autoadaptativo, o arquivo TF.xml provê o elemento *setTest*, cujo objetivo é a representação de conjuntos de testes, assim como ilustrado na Figura 7. A modelagem ilustrada nessa figura é baseada na ferramenta AltovaXML (AltovaXML, 2012).

Cada elemento setTest do TF.xml possui um identificador (uso do atributo id), e pode ser composto por um ou mais testes. Um teste (uso do elemento *test*) é descrito nesse arquivo a partir das seguintes informações:

- Identificador único representado pelo atributo id.

- Informação booleana responsável por indicar se o teste é executado de forma automática ou manual. Uso do atributo *isAutomatic.*
- Nível de teste (unitário, integração, sistema ou aceitação) representado pelo atributo *testLevel*.
- Tipo de teste (ex: regressão, usabilidade, desempenho, etc.) informado a partir do atributo *testType*.
- Obrigatoriedade de execução de cada teste (Costa et al., 2010b), isto é, se o teste é obrigatório ou opcional. Testes obrigatórios são aqueles que são imprescindíveis e tem que ser executados, já os opcionais são aqueles que podem ser liberados da execução, caso haja pressa em adquirir respostas dos testes. Essa informação é representada pelo atributo booleano *isMandatory*.
- Caminho do script de teste (ex: classe Java de um teste) representado pelo elemento *classname*. Tal elemento é obrigatório no TF.xml para todos os testes automatizados. Atualmente o JAAF+T está integrado com as seguintes APIs de teste: JUnit (JUnit, 2012), DBUnit (DBUnit, 2012) e JAT(Coelho ET AL., 2007a; Coelho ET AL., 2007b).
- Prioridade de execução do teste representada pelo elemento *priority*. Essa informação permite o maior detalhamento da prioridade dos testes do que no atributo *isMandatory*. Apesar disso, essas informações podem ser usadas em conjunto para definir, por exemplo, a prioridade dos testes obrigatórios.
- Risco que alguma falha encontrada por um teste pode ocasionar no produto. O teste pode, por exemplo, corromper a base de dados caso não consiga concluir sua execução com êxito, pode enviar mensagens sem nexo, etc. Para representar qual risco está relacionado a cada teste, é utilizado o elemento *risk*.
- Ferramenta usada para executar os testes automatizados. Representado pelo elemento *tool*, considerado obrigatório no TF.xml para testes automatizados.
- Descrição geral do teste representado a partir do elemento *description*. Elemento obrigatório quando o teste for manual, já que o testador precisa reproduzir manualmente um conjunto de passos para finalizar a execução do teste. Portanto, esse elemento pode

fornecer diversas informações, como, por exemplo, a descrição do passo a passo do teste, indicar um documento que possua as informações desse teste, etc.

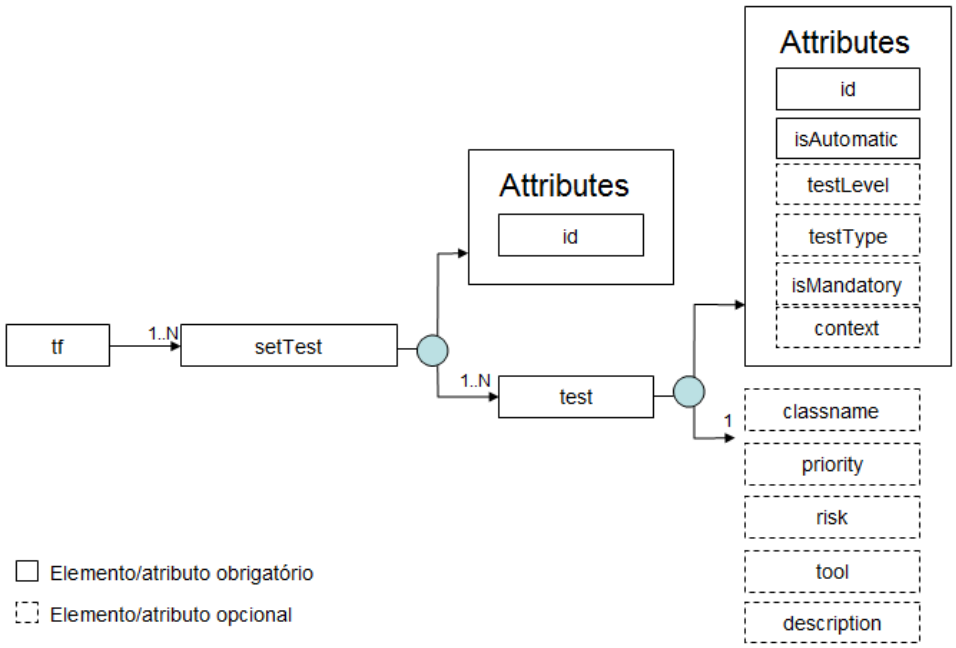

Figura 7. Esquema do arquivo TF.xml.

Os dados de entrada e saída de cada teste são definidos no arquivo DF.*xml* (ver Figura 8) a partir do elemento data. Esse elemento tem um identificador único representado pelo atributo *id* e pode ser composto por um ou mais elementos *condition*. Assim, se um elemento *data* tem duas condições de teste (dois elementos *condition*), um mesmo teste pode ser executado duas vezes considerando diferentes dados de entrada (elemento *input*) e saída (elemento *output*).

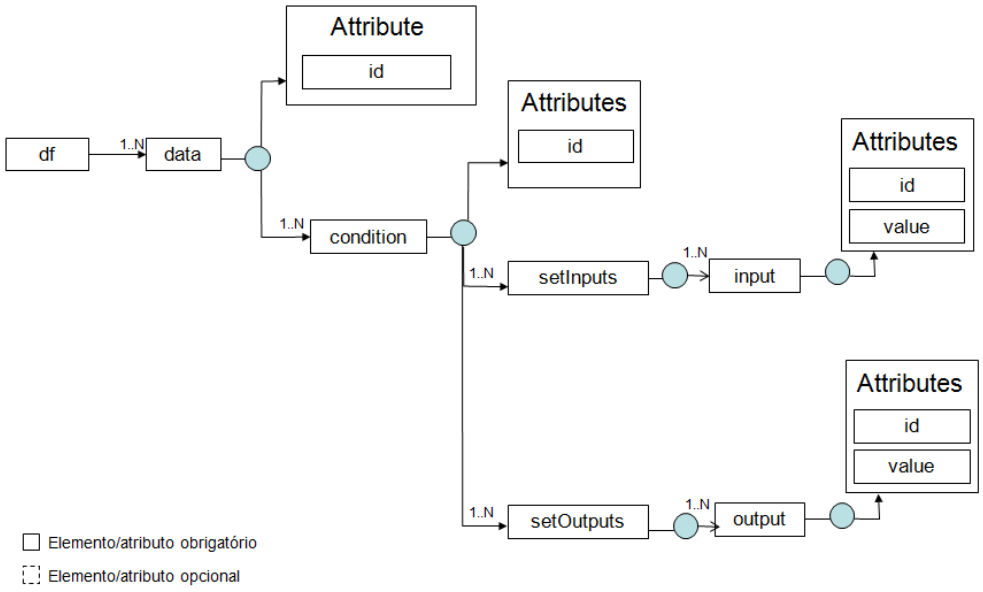

Figura 8. Esquema do arquivo DF.xml.

O elemento *condition* possui um identificador (representado pelo atributo *id*), além dos elementos *setInputs* e *setOutputs*, responsáveis por definir, respectivamente, os dados de entrada e saída do teste. Portanto, para que cada teste possa usar os dados definidos no DF.xml, eles devem implementar a interface *TestUsingDFInterface* (ver Figura 6). Nela são definidos dois métodos: *getIdData() e getIdCondition()*. O primeiro método informa o id do elemento *data*, enquanto que o segundo informa o id da condição de teste a ser utilizada. Caso seja necessário o agente executar de forma automática todas as condições de teste de um mesmo elemento data, basta retornar o valor “null” no método *getIdCondition*. No Código 1 é apresentado um exemplo de teste responsável por validar um método soma (*sum*) da classe *Sum*, enquanto que o Código 2 apresenta a definição dos dados usados pelo teste no DF.xml da aplicação. Perceba que a classe *CurrentConditionDFToExecute*, que aplica o padrão Singleton, é utilizada pelo teste exemplo, já que permite a recuperação da condição de teste corrente a ser usada. Dessa maneira o teste pode manipular suas informações da forma que julgar necessário.

```java
public class TestSum extends TestCase implements TestUsingDFInterface{

    @Override
    public String getIdData()
    {
        // TODO Auto-generated method stub
        return "dataSum1";
    }

    @Override
    public String getIdCondition()
    {
        // TODO Auto-generated method stub
        return "sum";
    }

    public void testSum()
    {
        Condition condition = CurrentConditionDFToExecute.getInstance().getCondition();

        int value1 = Integer.parseInt(condition.getValueByInputId("value1"));
        int value2 = Integer.parseInt(condition.getValueByInputId("value2"));
        int expectedResult = Integer.parseInt(condition.getValueByOutputId("result"));

        Sum instanceSum = new Sum();
        int result = instanceSum.sum(value1, value2);
        assertEquals(expectedResult,result);
    }
}
```

Código 1. Exemplo de teste que usa dados representados no DF.xml.

```xml
<df>
      <data id="dataSum1">
            <condition id="sum">
                  <setInputs>
                        <input id="value1" value="1"/>
                        <input id="value2" value="2"/>
                  </setInputs>
                  <setOutputs>
                        <output id="result" value="3"/>
                  </setOutputs>

            </condition>
      </data>
    ...
</df>
```

Código 2. Exemplo de DF para ilustrar relação dos testes com dados representados.

Já o arquivo CFF.xml (ver Figura 9) representa quais testes (elemento *test*) devem ser executados para validar algum artefato (representado pelo elemento *artifact*) do SUT. Um artefato pode ser, por exemplo, um comportamento JADE, assim como um web-service, classe ou método. Cada artefato possui as seguintes informações: um identificador (uso do atributo *id*), um tipo (uso do atributo *type*), um log que armazena os resultados dos testes executados (uso do atributo *logPath*), um critério de execução (representado

pelo atributo *criterionid*) que é descrito no arquivo CEF.xml (explicado mais a frente), além de um ou mais testes (uso do elemento *test*).

Cada teste responsável por validar algum artefato possui dois atributos: *type* e *id*. O primeiro informa se o elemento *test* está relacionado a um teste (*test*) ou a um conjunto de testes (*setTest*), definido no arquivo TF.xml. Dois tipos podem ser fornecidos: *test* ou *setTest*. Já o atributo *id* informa o identificador do teste ou do conjunto de teste correspondente. Portanto, CFF.xml depende dos testes descritos no arquivo TF.xml.

Vale ressaltar que a ordem em que os elementos *test* estão definidos no arquivo CFF.xml, informa a ordem de execução considerada pelo JAAF+T. Quando um elemento *test* do CFF.xml estiver referenciando um *setTest*, a prioridade dos testes desse conjunto (definido no TFF.xml) é considerada na execução. JAAF+T adota que os menores números são aqueles com maior prioridade, isto é, 0 (zero) é o teste com maior prioridade, assim como testes com prioridade 2 (dois) possuem maior prioridade do que os testes com prioridade 3 (três).

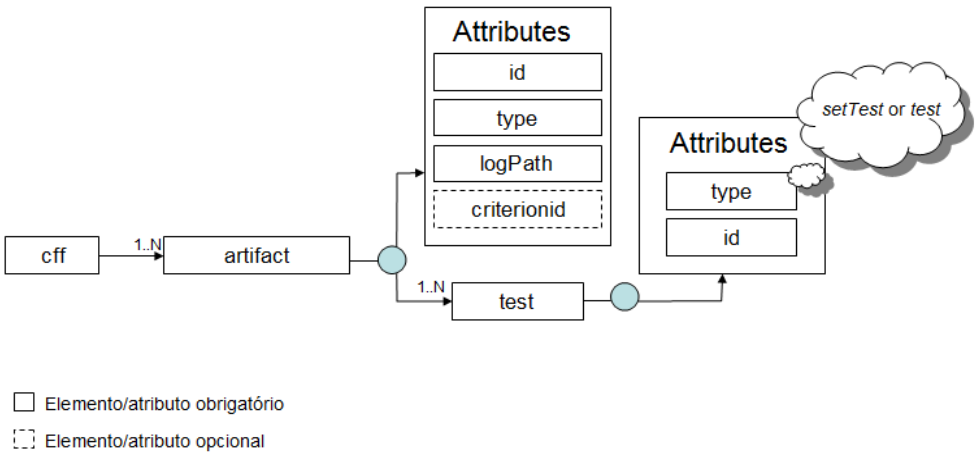

Figura 9. Esquema do arquivo CFF.xml.

Por fim, o arquivo CEF.xml (ver Figura 10) é responsável por descrever os possíveis critérios que os agentes devem respeitar quando estiverem coordenando a execução dos testes. Cada critério (uso do elemento *criterion*) possui um identificador único representado pelo atributo *id*, além de sete elementos: (i) priority, (ii) *testLevel*, (iii) *testType*, (iv) *risk*, (v) *isMandatory*, (vi) *isAutomatic* e (vii) *context*.

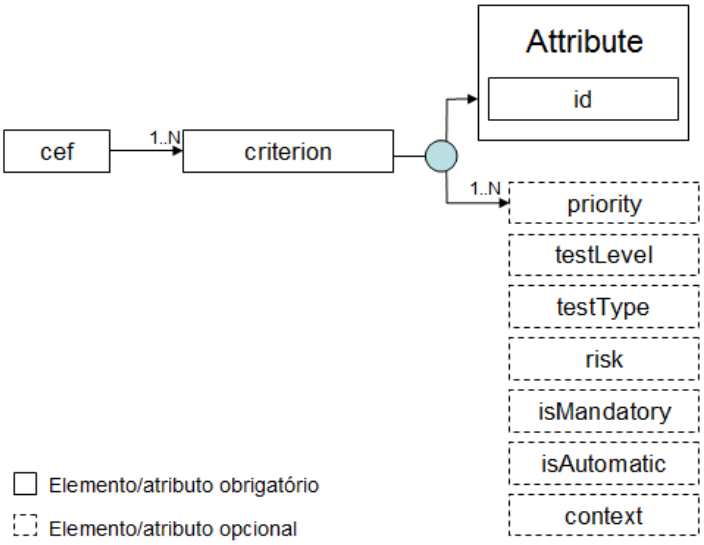

Figura 10. Esquema do arquivo CEF.xml.

O elemento priority descreve a prioridade dos testes a serem executados. Assim, se o testador desejar que os agentes executem somente os testes com prioridade 0 (zero), deve ser criado o elemento “<priority>0</priority>” no critério correspondente. No entanto, caso deseje-se executar testes com prioridade zero ou um, dois elementos priority devem ser criados, assim como apresentado no Código 3. A partir do momento que nenhuma prioridade for informada, o JAAF+T considera que testes com qualquer prioridade podem ser executados. Essa mesma linha de raciocínio é adotada para os outros elementos que podem fazer parte dos critérios de execução.

O elemento *testLevel* informa o nível de teste a ser considerado, como, por exemplo, unitário (unit), integração (integration), sistema (system) e aceitação (acceptance). No critério descrito no Código 3 é definido que somente os testes unitários serão executados.

O elemento *testType* define o tipo de teste que poderá ser executado, como, testes de regressão, usabilidade, segurança, etc. No Código 3 o critério considera que somente testes de regressão poderão ser executados.

Já o elemento *risk* descreve qual o risco dos testes a ser considerado. Enquanto que *isMandatory* informa se testes obrigatórios e/ou opcionais devem ser executados segundo o critério. Como tanto o elemento *risk* como *isMandatory* não foram fornecidos no Código 3, o JAAF+T considera que tanto testes obrigatórios como opcionais com qualquer risco poderão ser executados.

```
<cef>
    <criterion id="crit_example">
        <priority>0</priority>
        <priority>1</priority>
        <testLevel>unit</testLevel>
        <testType>regression</testType>
        <isAutomatic>true</isAutomatic>
        <context>v7_00</context>
    </criterion>
    ...
</cef>
```

Código 3. Exemplo do CEF.xml com um critério com duas prioridades definidas.

O elemento *isAutomatic* informa se os testes automáticos e/ou manuais poderão ser executados. No Código 3 é informado que somente testes automáticos serão executados.

Por fim, o elemento *context* descreve o contexto que os testes devem ser executados. Código 3 informa que somente testes relacionados a "v7_00" do sistema em teste devem ser executados. Portanto, a condição criada a partir do critério representado no Código 3 é a seguinte:

SE (teste possui prioridade igual à zero **OU** igual a um) **E**
(teste com nível de teste unitário) **E**
(teste com tipo de teste de regressão) **E**
(teste ser executado de forma automática) **E**
(teste estar no contexto v7_00) **ENTÃO**
TESTE VÁLIDO PARA EXECUÇÃO

Perceba que o framework permite a criação de diferentes critérios. No entanto, para que algum agente possa respeitar algum critério definido, o testador deve informar tal critério no artefato a ser testado (uso do atributo *criterionid* do elemento *artifact* no CFF.xml). Caso nenhum critério seja informado, o JAAF+T considera que todos os testes definidos no CFF.xml poderão ser executados.

Na Tabela 1 é apresentada a relação de onde cada informação mencionada no capítulo 3 está sendo usada em qual arquivo XML do framework (TF.xml, DF.xml, CFF.xml e/ou CEF.xml). Veja a seguir.

Informações usadas para coordenação dos agentes	Arquivo XML que as informações são usadas
Testes a serem executados	TF.xml e CFF.xml
Ferramentas necessárias para executar testes	TF.xml
Critério de execução	CEF.xml e CFF.xml
Níveis de teste	TF.xml e CEF.xml
Tipos de Teste	TF.xml e CEF.xml
Testes manuais e automatizados	TF.xml e CEF.xml
Prioridade de execução	TF.xml e CEF.xml
Artefatos a serem testados	CFF.xml
Risco do teste	TF.xml e CEF.xml
Dependências entre testes	CFF.xml
Contexto de execução	TF.xml e CEF.xml
Dados de entrada e saída usados pelos testes	DF.xml
Logs com os resultados dos testes executados	CFF.xml

Tabela 1. Mapeamentos de informações úteis para coordenação da execução dos testes nos arquivos XML do JAAF+T.

4.3. Pontos Fixos e Flexíveis

Nesta seção são apresentados em detalhe os pontos flexíveis (*hot-spots*) e fixos (*frozen-spots*) (Fayad e Johnson, 1999) do JAAF+T. Os pontos flexíveis são os seguintes:

- **Criação de diferentes processos de autoadaptação:** A partir da extensão da classe *Control-Loop* diferentes processos de autoadaptação podem ser criados, representando assim diferentes control-loops.
- **Criação de atividades:** Estendendo a classe *Activity* novas atividades que fazem parte de diferentes control-loops podem ser criadas.
- **Definição de pré- e pós-condições de atividades:** A partir da classe *ConditionActivity*, torna-se possível definir pré- e pós-condições de atividades definidas.

- **Informar diferentes testes para execução:** A partir do TF.xml diferentes testes podem ser definidos para serem executados a partir dos agentes JAAF+T.
- **Informar diferentes condições de teste:** A partir do DF.xml diferentes dados de entrada e saída podem ser definidos e usados por um ou mais testes.
- **Definir diferentes suítes de execução:** A partir do CFF.xml diversos fluxos de controle podem ser definidos.
- **Definir critérios de execução:** CEF.xml permite a definição de diferentes critérios de execução que devem ser respeitados pelos agentes de software.
- **Executar diferentes tipos de teste:** Para permitir a execução de testes desenvolvidos a partir de diferentes ferramentas, a classe *ExecutionTest* foi definida. Atualmente, o framework provê suporte para a execução de testes unitários a partir do uso das APIs JUnit, DBUnit e JAT. Caso deseje executar outros tipos de teste usando como base outras APIs, a classe *ExecutionTest* deve ser estendida e todos os testes da aplicação (representados no TF.xml) devem implementar a interface *TestUsingDFInterface*, que permite ao JAAF+T controlar dados de entrada e saída usados (definidos no DF.xml). Para que fosse possível utilizar o JAT, tivemos que alterar sua API para permitir a execução de testes em uma plataforma JADE já em execução. Anteriormente, toda vez que o JAT executasse um teste, a plataforma JADE era iniciada. Consequentemente, quando o teste finalizava sua execução, a plataforma também era parada. Atualmente, o JAT verifica se uma plataforma JADE está em execução antes de executar um teste. Caso ela esteja no ar, assim que um teste finalizar sua execução, a plataforma continua em funcionamento sem afetar a execução do restante do sistema.
- **Ler logs em diversos formatos:** A partir da extensão da classe *ReadingLog* torna-se possível ler diferentes tipos de log.
- **Escrever em diferentes logs:** Estendendo a classe *LoadingLog*, torna-se possível gravar o resultado dos testes executados em diferentes formatos de log.

Já os pontos fixos do JAAF+T são os seguintes:

- **Control-loop provido pelo framework:** O JAAF+T provê o control-loop ilustrado na Figura 5 a partir da classe *CLWithSelfTest*. Tal classe utiliza seis atividades definidas: Coleta, Análise, Decisão, Teste, Validação e Efetuar.
- **Leitura de logs em formatos pré-definidos pelo framework:** O framework provê suporte para a leitura de dois tipos de logs: XML (uso da classe *XMLLog*) e texto (uso da classe *TextLog*).
- **Parser do arquivo TF.xml:** Oferece suporte para leitura do arquivo TF.xml a partir da classe *ParserTF* que implementa o padrão de projeto *Singleton* (Gamma et al., 1994).
- **Parser do arquivo CFF.xml:** Oferece suporte para ler dados do arquivo CFF.xml a partir da classe *ParserCFF*. Tal classe também implementa o padrão *Singleton*.
- **Parser do arquivo DF.xml:** Oferece suporte para ler dados do arquivo DF.xml a partir da classe *ParserDF*. Tal classe implementa o padrão *Singleton*.
- **Parser do arquivo CEF.xml:** Assim como o TF.xml, CFF.xml e DF.xml, o arquivo CEF.xml é lido por uma classe Singleton chamada de *ParserCEF*.
- **Procedimento para realizar a execução dos testes:** Para executar os testes desejados, o framework provê a classe *StartTests*. Tal classe aplica o padrão Façade (Gamma et al., 1994), que visa simplificar a execução dos diferentes testes. Assim, o método *executeTests*, já está implementado, realiza o controle das execuções dos testes. O trabalho da instância do JAAF+T é principalmente preencher os arquivos XML (TF.xml, CFF.xml, DF.xml e CEF.xml) para que o tratamento das execuções seja realizado de forma transparente e automática.

4.4. Como Usar o Framework JAAF+T

Para implementar um agente autoadaptativo que aplique autoteste a partir do JAAF+T, os passos a serem realizados são os seguintes:

1. Definir no arquivo TF.xml os testes que poderão ser executados no sistema autoadaptativo.
2. Implementar diferentes formas de executar testes (ex: unitário, funcional, etc.) a partir da extensão da classe *ExecutionTest*. Assim como mencionado na Seção 4.3, o framework oferece suporte para a execução de testes unitários baseados no JUnit, DBUnit e JAT.
3. Definir no arquivo DF.xml os dados de entrada e saída a serem usados por testes.
4. Definir a partir das classes *LoadingLog* e *ReadingLog*, como os resultados dos testes executados serão escritos e lidos, respectivamente.
5. Definir os critérios de execução dos testes a partir do arquivo CEF.xml.
6. Definir no CFL.xml os fluxos de execução dos testes.
7. Definir qual processo de autoadaptação será utilizado. Pode ser usado o processo padrão oferecido pelo framework (ver Figura 5), ou criar algum outro a partir da classe *ControlLoop*. Se o desenvolvedor/testador quiser criar novas atividades, basta estender a classe *Activity* para defini-las.
8. Criar um agente de software que estenda a classe *Agent*, fornecida pelo framework JADE, e associá-lo com um processo de autoadaptação.

4.5. Considerações Finais

Neste capítulo foi apresentado o Java self-Adaptive Agent Framework for self-Test (JAAF+T), framework responsável por permitir a criação de agentes de software autoadaptáveis capazes de realizar autotestes. Para que esses agentes pudessem ser criados, o framework permite: (i) a criação de diferentes processos de autoadaptação; (ii) a criação de diferentes atividades, que compõem os processos de autoadaptação; (iii) oferece um conjunto de atividades pré-definidas, dentre elas a atividade de Teste e Validação que podem ser reusadas em diferentes processos; e (iii) utiliza um conjunto de arquivos XML pré-configurados (TF.xml, CFF.xml, DF.xml e CEF.xml), responsáveis por representar informações úteis para a coordenação dos testes.

No capítulo a seguir é apresentado um modelo conceitual que ilustra em detalhe como estão relacionadas as informações manipuladas pelo JAAF+T para permitir a coordenação dos testes de software. Além disso, o modelo proposto pretende ajudar na geração dos arquivos XML usados pelo framework, já que boa parte dessas informações estão representadas em tais arquivos.