4
Teste de Sistemas Multiagentes Desenvolvidos com o Middleware M-Law

Neste capítulo será apresentado como é possível testar sistemas multiagentes desenvolvidos para o middleware M-Law utilizando agentes stubs. Veremos então, que o custo para o desenvolvimento de testes pode ser muito elevado, já que um agente stub é na verdade um agente como qualquer outro. No entanto, seu comportamento é implementado como a execução de um script.

Seguindo essa linha para a implementação de testes, observamos que para o teste unitário de um agente necessitamos de diversos outros agentes stubs. Além do custo para o desenvolvimento dos agentes stubs, um espalhamento em toda a lógica do teste é causado. Dessa forma, os desenvolvedores são levados a implementarem coordenadores para execução e controle de todos esses agentes.

4.1
Agentes XMLaw

Especialmente, quando lidamos com agentes para XMLaw, temos a seguinte estrutura para o desenvolvimento:

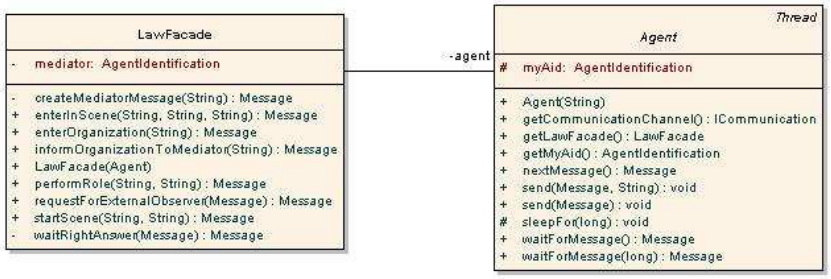

Figura 4.1: Estrutura de Classes de um Agente XMLaw

Podemos identificar pelo diagrama que a classe *Agent* é responsável pelo envio e recebimento de mensagens. Na verdade, ela é livre para o envio de qualquer mensagem. Caso o agente necessite se comunicar, existe uma série de operações de configuração para que uma mensagem seja criada, configurada e finalmente enviada. Considere o seguinte exemplo onde o agente foo envia uma mensagem para o agente bar. A mensagem deve identificar a organização, a cena, a performativa, o

destinatário, o papel do destinatário, o remetente, o papel do remetente e, eventualmente, alguma mensagem para o agente bar, nesse caso, “hello bar”.

```
1 ...
2 Message fooMsg = new Message(Message.INFORM);
3 fooMsg.setContentValue(MessageContentConstants.KEY_ORGANIZATION_EXECUTION_ID,43);
4 fooMsg.setContentValue(MessageContentConstants.KEY_SCENE_EXECUTION_ID,92);
5 fooMsg.setContentValue("info","hello bar");
6 fooMsg.setReceiver(bar,"barRole");
7 send(fooMsg,"fooRole");
8 ...
```

Listagem 4.1: Exemplo de envio mensagem

Devemos também considerar que alguns parâmetros da mensagem são argumentados com identificadores gerados pelo middleware. Forçando portanto que o desenvolvimento de um agente armazene e gerencie esses identificadores em tempo de execução. É o caso dos identificadores de organização e cena.

Uma classe de fachada para o sistema aberto, *LawFacade*, permite o envio de mensagens específicas para o sistema. Essas mensagens incluem a publicação de organizações, a criação de cenas, a entrada em cenas, enfim, todas as operações relativas ao sistema aberto. Dessa forma, pretende-se tornar as instruções no agente mais enxutas e menos suscetíveis a erros.

Ao implementar um agente, pode-se fazê-lo de três maneiras diferentes: delegação, direta ou herança. Conforme vemos na figura 4.2. O uso de herança é o mais recomendado [25].

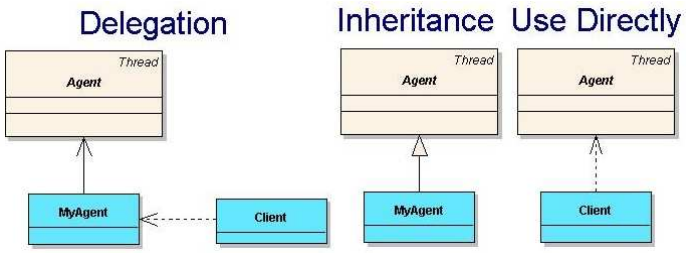

Figura 4.2: Formas de utilização de um agente XMLaw

4.2 Teste de Sistemas Multiagentes

Supomos que um desenvolvedor não está interessado em desenvolver apenas um agente, mas sim, vários deles, cada um com um comportamento diferente. Uma primeira abordagem poderia ser simplesmente desenvolver todos os agentes e colocá-los executando ao mesmo tempo. Com muita sorte será possível encontrar os erros produzidos em tempo de desenvolvimento. No entanto, com um pouco de azar, o erros passarão desapercebidos e os agentes serão enviados para um ambiente

de produção. Segundo a lei de murphy[1], os erros aparecerão nos momentos mais inconvenientes.

Ditados populares à parte, os desenvolvedores querem (ou deveriam querer) desenvolver software com qualidade. Sabemos que existem domínios onde uma falha de software pode ter conseqüências não recompensáveis , como, por exemplo, a perda de vida humana.

Podemos, então, tentar trabalhar de forma mais organizada e desenvolver cenários de execução para verificar a correta execução dos nossos agentes. Imaginemos o desenvolvimento de dois agentes reais: foo e bar. Eles devem trocar mensagens entre si através do ambiente XMLaw. Portanto, antes de colocar os dois agentes executando, podemos desenvolver outros dois agentes stubs: fooStub e barStub, que podem auxiliar na verificação de erros no desenvolvimento dos agentes reais, possibilitando, inclusive, o uso de teste automatizado.

No nosso contexto, um agente real é a implementação de um agente que integra parte de um sistema aberto. Um agente stub será também um agente, assim como o agente real. No entanto, ele possui um comportamento previsível, servindo apenas para simulação de cenários.

Agora, com uma certa organização para execução dos agentes, podemos tentar fazer o agente real foo conversar com o agente stub barStub. Mais especificamente, podemos desenvolver um terceiro agente de teste que executa foo juntamente com barStub e verifica se o comportamento de foo é o esperado. Um exemplo para o comportamento de barStub pode ser visto na listagem 4.2.

```
1  ...
2  Message addLawReply = getMediatorFacade().informOrganizationToMediator(lawURL.
       toString());
3  String orgExec =  addLawReply.getContentValue(MessageContentConstants.
       KEY_ORGANIZATION_EXECUTION_ID);
4  Message enterOrgReply = getMediatorFacade().enterOrganization(orgExec);
5  Message performRoleReply = getMediatorFacade().performRole(orgExec,"bar");
6  Message startSceneReply = getMediatorFacade().startScene("game",orgExec);
7  String sceneExec = startSceneReply.getContentValue(MessageContentConstants.
       KEY_SCENE_EXECUTION_ID);
8  Message enterSceneReply = getMediatorFacade().enterInScene(sceneExec,orgExec,"bar
       ");
9
10 Message barStubMsg = new Message(Message.INFORM);
11 barStubMsg.setContentValue(MessageContentConstants.KEY_ORGANIZATION_EXECUTION_ID,
       orgExec);
12 barStubMsg.setContentValue(MessageContentConstants.KEY_SCENE_EXECUTION_ID,
       sceneExec);
13 barStubMsg.setContentValue("hello","Hello␣foo,␣how␣bar␣you?");
14 barStubMsg.setReceiver(fooId,"bar");
15 send(barStubMsg,"bar");
16 Message answer = waitForMessage();
17 ...
```

[1]Se alguma coisa pode dar errado, dará. E mais, dará errado da pior maneira, no pior momento e de modo que cause o maior dano possível.

Listagem 4.2: Comportamento do agente barStub

Observe que as linhas 2 até 8 nada mais são do que mensagens enviadas ao sistema aberto para a criação dos componentes que regulam as leis, tais como organizações e cenas. O comportamento do agente barStub é simplesmente o envio da mensagem barStubMsg. O comportamento do agente de teste será a recuperação da mensagem de resposta (linha 16) e sua verificação com algum valor esperado.

Dessa forma, podemos desenvolver agentes reais um pouco mais confiáveis, pois temos como simular, de forma automatizada, seu comportamento. Obviamente, agentes que possuam comportamentos mais elaborados e complexos demandarão um esforço bem maior de simulação.

Nesse contexto, podemos então definir o conceito de cenários de teste. Um cenário é a simulação de uma possível interação em um sistema multiagente. Por exemplo, o teste dos agentes foo e bar descritos anteriormente, podem ser descritos em cenários de teste. Cenários hipotéticos seriam:

- Cenário 1 - Agente foo envia mensagem m1 para o agente bar e espera mensagem m2 como resposta.
- Cenário 2 - Agente bar envia mensagem m3 para o agente foo e espera mensagem m4 como resposta.
- Cenário 3 - Agente bar envia mensagem m5 como convite para foo entrar na cena descrita na mensagem. Foo entra na cena e envia mensagem m6 informando sua entrada. Agente bar verifica se agente foo realmente entrou na cena.

Eventualmente um desenvolvedor pode estar interessado em testar não apenas o comportamento de um único agente, mas sim de um conjunto de agentes. Seguindo essa estratégia de testes, podemos considerar a execução conjunta de agentes reais e stubs em um cenário de testes que considere as combinação dos agentes envolvidos. Logo, um único cenário de teste pode possuir diversas execuções distintas.

4.2.1 Esforço de Desenvolvimento

Vimos como um agente de software pode ser testado. No entanto, o esforço necessário para que isso seja possível pode ser bem maior do que o desenvolvimento do agente em si. Essa seção não visa estabelecer uma forma capaz de medir o esforço para o desenvolvimento do teste de um agente; ela apenas analisa o número necessário de agentes stubs que deverão ser desenvolvidos para que se tenha uma cobertura completa dos testes. Dessa forma, um desenvolvedor pode ter uma indicação de quanto foi a cobertura dos testes desenvolvidos.

Vamos supor um sistema hipotético que possui uma lei para regular a interação entre os agentes. Esse sistema permite que n agentes diferentes coexistam. Podemos então definir um cenário para cada tipo de interação que desejamos simular, digamos que s cenários foram definidos para nosso sistema. Cada cenário pode, no pior caso, incluir todos os agentes, portanto n.

Podemos então analisar de perto um único cenário. Suponha que para cada um dos n agentes envolvidos exista um agente stub associado que simula o comportamento de seu agente real para o cenário em questão.

A execução mais simples desse cenário seria aquela que envolvesse apenas os agentes *stubs*. Nesse caso, podemos dizer que essa execução exercita a lei desenvolvida para o sistema. Para cada cenário existe apenas uma execução desse tipo.

O segundo tipo de execução teria apenas um agente real e $n-1$ agentes stubs. Para esse caso, podemos dizer que estamos desempenhando um teste unitário para o agente real em questão. Como existem n agentes diferentes no cenário, podemos considerar, então, que existem n execuções diferentes para este cenário.

Uma outra forma de execução, seria a combinação entre um conjunto de agentes reais com agentes stubs. Esse tipo de execução estaria preocupada com o teste de integração entre os agentes, pois verifica o comportamento de um grupo de agentes reais. O número de execuções possíveis que seguem esse conceito de integração é a soma de combinações de todos os conjuntos de agentes reais de tamanho 2 até $n-1$. Portanto $\sum_{k=2}^{n-1} \frac{n!}{k!(n-k)!}$.

Finalmente, a execução do cenário pode ser feita apenas com agentes reais, o que caracteriza uma verificação do sistema. Esse tipo de execução é o mais próximo do que seria a execução do sistema, já que não envolve testes automatizados, sendo desconsiderada nessa análise. Existe apenas uma execução desse tipo para cada cenário.

De uma forma geral, podemos dizer que o total de execuções possíveis para um cenário com n agentes é a soma entre o número de execuções para o teste unitário (n), o número de execuções para os teste de integração ($\sum_{k=2}^{n-1} \frac{n!}{k!(n-k)!}$) e 1, que representa a simulação da lei do sistema para o cenário em questão. Em particular podemos ver os testes unitários e a simulação da lei do sistema como casos particulares para $k = 1$ e $k = 0$ do teste de integração. Temos, portanto, a equação 4-1 que indica o número de execuções ε_υ para um único cenário υ com n agentes.

$$\varepsilon_\upsilon = \sum_{k=0}^{n-1} \frac{n!}{k!(n-k)!} \tag{4-1}$$

Finalmente podemos definir que o total de execuções ε para um sistema como a soma das execuções de todos os cenários. Como estamos supondo o pior caso,

todos os cenários possuem um número igual de agentes, e um número igual também de quantidade de execuções possíveis, representadas por ε_1. Logo o número de execuções distintas, no pior caso, para um sistema pode ser obtido pela equação 4-2.

$$\varepsilon = s * \varepsilon_1 \tag{4-2}$$

Podemos ver, portanto, que o número de execuções para testar um sistema pode ser demasiado. Devido à análise quantitativa temos o número máximo de execuções necessárias para o teste de todos os cenários de um sistema e, a partir desta informação, podemos saber o quão próximo encontram-se os teste desenvolvidos de uma cobertura completa.

Um ponto importante é a velocidade de crescimento do número de execuções. Na figura 4.3 podemos ver um gráfico ilustrando a quantidade de execuções para um sistema variando-se a quantidade de agentes e o número de cenários de teste. Os pontos em destaque marcam a quantidade de execuções para quantidade de cenários. As linhas de baixo para cima representam, respectivamente, sistemas com 2, 3, 4 e 5 agentes. Observamos que para um sistema simples, com cinco cenários e três agentes, por exemplo, existe um total de 40 execuções possíveis para uma cobertura completa no pior caso.

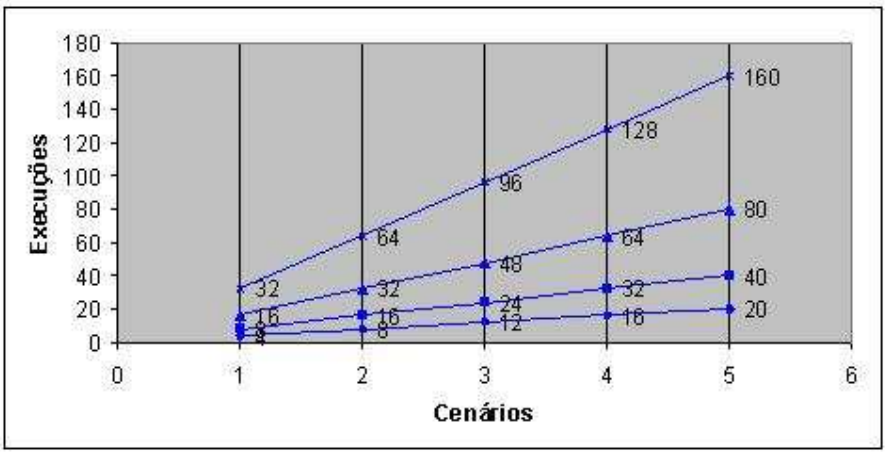

Figura 4.3: Quantidade de Execuções por Cenário de Teste

Vemos, portanto, que existe um crescimento linear referente ao número de cenários em questão. No entanto, o número de agentes causa um impacto exponencial no número de execuções possíveis. A cada novo agente incluído o número de execuções possíveis dobra. O crescimento linear sugere que devemos ter ferramentas que apóiem a criação de cenários, enquanto que o crescimento exponencial sugere que devemos ter ferramentas para facilitar a criação dos agentes e também combinar as possíveis execuções automaticamente.