Geração Automática de Casos de Teste para Sistemas Baseados em Agentes Móveis

André Luiz Lima de Figueiredo

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciências da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
Dalton Dario Serey Guerrero
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

F475g Figueiredo, André Luiz Lima de
2005 Geração automática de casos de teste para sistemas baseados em agentes móveis/ André Luiz Lima de Figueiredo. — Campina Grande, 2005.
177f. il.

Referencias.
Dissertação (Mestrado em Informática) – Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia.
Orientadores: Patrícia Duarte de Lima Machado e Dalton Dario Serey Guerreiro.

1— Engenharia de Software 2— Teste Formal 3- Agentes Móveis 4_ Padrões de Teste I— Título

CDU 004.41

"GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE PARA SISTEMAS BASEADOS EM AGENTES MÓVEIS"

ANDRÉ LUIZ LIMA DE FIGUEIREDO

DISSERTAÇÃO APROVADA EM 30.05.2005

PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Orientadora

PROF. DALTON DARIO SEREY GUERRERO, D.Sc
Orientador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador

PROF. ALEXANDRE CABRAL MOTA, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

Na busca por mais confiança a respeito da correção de seus sistemas, desenvolvedores têm, cada vez mais, utilizado teste de *software*, onde este pode ser definido como um conjunto de experimentos realizados sobre uma implementação, cujos resultados são observados e analisados. Dentre os diversos tipos de teste, este trabalho se concentra em um tipo especial de teste funcional chamado de Teste Formal. A partir de uma especificação formal e por meio de procedimentos formais, um método de teste formal é capaz de gerar, automaticamente, casos de teste que visam verificar a conformidade entre a especificação e a implementação. Um tipo de sistema em que este tipo de teste é útil é o baseado em Agentes Móveis. Um Agente Móvel pode ser definido como um programa autônomo capaz de migrar pelos diferentes pontos do sistema durante sua execução preservando seu estado. Este tipo de sistema distribuído ainda possui alto grau de complexidade inerente ao seu desenvolvimento devido, principalmente, à imaturidade dos processos de desenvolvimento em relação ao tratamento do conceito de mobilidade. Visando amenizar tal problema, este trabalho apresenta uma proposta de geração automática de casos de teste para sistemas baseados em Agentes Móveis. Tal proposta consiste em apresentar um formalismo de especificação de sistemas a ser usado com Agentes Móveis e uma estratégia de geração de casos de teste através de ferramentas. Além disto, visando tomar proveito dos modelos UML existentes para este tipo de sistema, uma proposta de transformação informal de modelos escritos em tal linguagem para modelos escritos no formalismo utilizado pelo método é apresentada. Após o método ter sido proposto, um estudo de caso foi realizado visando mostrar a aplicabilidade do método. Os casos de teste gerados foram analisados em relação ao seu potencial em serem implementados e em revelarem faltas nos sistemas. Motivados pelos estudos realizados sobre os casos de teste, bem como pelo crescente interesse no uso de padrões de projetos, uma metodologia para a identificação de padrões de teste a partir de especificações de padrões de projeto é proposta, em conjunto com um formato de definição para tais padrões e com um conjunto de padrões de teste para sistemas baseados em Agentes Móveis.

Abstract

Looking for reliability regarding to their systems correction, developers have, increasingly, used software testing. This activity can be defined as a set of experiments performed over an implementation, which the results are observed and analyzed. Among the existing kinds of test, this work takes special attention to a specific kind of functional test, called Formal Test. From a formal specification and through formal procedures, a formal test method is able to, automatically, generate test cases that aim to verify conformance between specification and implementation. Mobile Agent-based systems can benefit from formal testing. Mobile Agent can be defined as an autonomous program that can migrate among different points of a distributed system during its execution preserving its state. Develop this kind of system is still a complex activity, due to, mainly, development processes immaturity regarding to mobility issues handling. Aiming to soften this problem, this work presents an automatic test case generation proposal to Mobile Agent-based systems. This proposal comprises presenting a system specification formalism to be used with Mobile Agents and a strategy for test case generation via tools. Moreover, aiming to take advantage of the existing UML modelos to this kind of system, a informal transformation from models written such language, to the formalism language used by the presented method is proposed. After the method has been proposed, a case study was carried out addressing to show the method's applicability. The generated test cases were analyzed with relation to its potential to be implemented and to reveal system faults. Motivated by the test cases study, and by the increasing interest at the use of design patterns as well, a methodology for identification of test patterns from Mobile Agent design patterns is proposed, in addition to a test pattern template for this kind of pattern, and a set of test patterns to Mobile Agent-based systems.

Agradecimentos

Em primeiro lugar a Deus, por ter planejado este maravilhoso caminho em minha vida e por me permitir segui-lo. À toda a minha família, mãe, irmãos e, principalmente, a meu pai, maior incentivador e responsável pelo meu sucesso, onde mesmo não estando presente, sempre esteve junto através de seus ensinamentos, exemplos de vida e amor. À minha maravilhosa e amada Renata, por sua paciência e seu ombro consolador, ajudando-me imensamente nos momentos mais difíceis. Aos amigos e companheiros de pesquisa do GMF, pela mão amiga e inefável companhia nesta infinita luta pelo conhecimento. A meus orientadores, Patrícia Machado e Dalton Guerrero, pelo apreço e dedicação a este trabalho. Aos professores Jorge Figueiredo e Alexandre Mota, pela imensa contribuição. E, por fim, a todos que, direta ou indiretamente, através de palavras, gestos ou orações contribuíram para o sucesso deste trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização e Motivação

Teste de *software* tem sido cada vez mais utilizado por desenvolvedores de sistemas que objetivam produtos de qualidade. Podemos definir teste como sendo um conjunto de experimentos realizados sobre uma implementação, cujos resultados são observados e analisados [Bei90]. Através desses experimentos, é possível validar uma implementação de acordo com um determinado critério de aceitação. Existem diferentes tipos de teste com os quais estão relacionados diferentes aspectos do sistema que se deseja observar [MS01]. Testes de desempenho são realizados com a finalidade de verificar se o sistema tem o desempenho desejável. Já os testes de robustez são realizados com o intuito de observar como o sistema reage quando o ambiente apresenta um comportamento inesperado. A fim de verificar como o sistema reage sobre forte demanda, testes de carga (ou de estresse) são realizados.

Teste funcional (*black-box* [Bei99]) é aquele que objetiva observar se um dado sistema contempla de forma correta as funcionalidades descritas para ele. Com base na sua respectiva especificação, um sistema é exercitado e suas funcionalidades são observadas através de seu comportamento, onde as possíveis falhas são reveladas. A propósito, utilizaremos os termos **falta**, **falha** e **erro** como definidos por Binder [Bin99]. **Falha** é a manifestação do *software* em não executar de forma correta uma determinada funcionalidade. **Falta** é definida como a ausência de código ou a presença de código incorreto no sistema, e que pode ocasionar uma falha no mesmo. E **erro** é a ação humana que ocasiona uma falta no sistema.

Dentre os diversos tipos de teste, um tipo de teste funcional se destaca - teste formal. Este tipo de teste permite verificar, de forma automática, se um determinado *software* está de acordo com sua especificação formal [Tre99; Gau95]. Neste sentido, é possível observar de forma automática se um *software* realiza corretamente as funcionalidades contidas em sua especificação. Um método de teste formal é aquele em que, dada uma especificação formal de um sistema, ele é capaz de gerar, a partir desta, casos de teste que visam verificar conformidade entre tal especificação e uma implementação. Esta atividade só pode ser automática devido ao uso de especificações formais e de um procedimento de geração também formal.

Durante algum tempo pensou-se que o mundo intuitivo e heurístico dos experimentos não poderia receber conceitos formais e com forte embasamento matemático como os presentes nos métodos formais. Contudo, este pensamento começou a ser mudado quando trabalhos como os apresentados por Tretmans [Tre99] e Gaudel [Gau95] demonstraram que tal união era possível e necessária, e que teste formal poderia ser usado em conjunto com os tradicionais métodos formais (por exemplo, verificação de modelos) no intuito de contribuir para o aumento da qualidade dos sistemas produzidos.

Um tipo de sistema no qual métodos de teste formal são úteis é o baseado em Agentes Móveis. Um Agente Móvel pode ser definido como um programa autônomo capaz de migrar pelos diferentes pontos do sistema durante sua execução preservando seu estado [FPV98; CHK97]. As principais vantagens deste paradigma são: redução do tráfego de rede, diminuição da dependência da latência da rede, execução assíncrona e autônoma, adaptação dinâmica, heterogeneidade, robustez e tolerância a falhas [Lim04]. O paradigma de Agentes Móveis surgiu como uma alternativa aos sistemas distribuídos convencionais revelando-se interessante em aplicações de comércio eletrônico, recuperação de informação, serviços de comunicação de redes, assistentes pessoais e outros [Lim04], onde execução desconectada de um processo em diferentes nós de uma rede seja um requisito essencial da aplicação, custos de conexão sejam altos e a confiabilidade da rede seja baixa.

Apesar das vantagens e da abrangência das áreas de aplicação do paradigma, ainda existe uma certa resistência ao uso do mesmo devido a alguns fatores. Dentre estes, destacamos a complexidade relacionada ao desenvolvimento de Sistemas Baseados em Agentes Móveis (SBAM). Parte dessa complexidade pode ser atribuída à característica distribuída de tais sistemas (concorrência, comunicação assíncrona, etc.), mas em sua maior parte, ela advém da imaturidade dos processos de desenvolvimento em relação ao tratamento do conceito de mobilidade. Tal conceito precisa ser tratado de forma adequada durante diversas etapas do desenvolvimento, e não somente durante a implementação, como pode ser visto no conexto atual de desenvolvimento.

É fato que diversos métodos, técnicas e ferramentas de teste específicos para sistemas distribuídos podem ser encontrados na literatura [FJJV96; BFdV$^+$99; Tre99; RM00; PJLT$^+$02; Jar02; HLU03], porém nada pode ser afirmado com relação à sua eficácia no tratamento de conceitos de mobilidade. Podemos usar tais técnicas para testar Sistemas Baseados em Agentes Móveis, porém não podemos afirmar nada acerca de como as características de mobilidade estão sendo tratadas. Por exemplo, no caso de geração automática de casos de teste, temos diversas ferramentas que podem ser usadas para gerar casos de teste para SBAM. Contudo, dentre as diversas características que serão tratadas, as características de mobilidade não necessariamente serão contempladas.

Especificamente com relação a testes, pouquíssimas propostas para SBAM podem ser encontradas. Os trabalhos apresentados pela comunidade, em geral apresentam propostas focadas em testes estruturais (*white-box*) [DV03]. Isto decorre do fato de que o conceito de mobilidade ainda tem seu tratamento exclusivamente em etapas de implementação e, desta forma, trabalhos sobre teste funcional (i.e., teste baseado em especificação) não existem, já que o conceito de mobilidade não é tratado nas etapas de especificação.

Algumas propostas que visam amenizar a complexidade inerente ao processo de desenvolvimeto de SBAM estão sendo desenvolvidas pela comunidade. Além dos já citados para sistemas distribuídos, outros trabalhos têm se concentrado em propostas para especificação, modelagem e implementação de SBAM, como por exemplo, propostas de metodologias de desenvolvimento [GMM03; Mar03], uso de padrões de projetos específicos para Agentes Móveis [LMSF04; AL98; SSJ02; KKPS98; TOH99], propostas de linguagens de modelagem e especificação [KRSW01], desenvolvimento de plataformas de suporte à implementação de tais sistemas [BMG+04].

Dentre os trabalhos citados acima, é importante destacarmos as propostas que utilizam métodos formais, em que há um crescente interesse neste tipo de abordagem. É possível encontrar, na comunidade, diversos trabalhos já consolidados sobre modelagem e especificação formal, verificação de modelos, e testes formais, porém todos para sistemas distribuídos. Especificamente para SBAM, os trabalhos têm se concentrado sobre modelagem e especificação formal de tais sistemas [LFG03; LMSF04; LFG04; Mik98; AMM01; XD00], deixando forte carência em questões diretamente ligadas à validação e à verificação destes sistemas.

Diante do contexto apresentado, podemos ver que desenvolvedores de SBAM podem se utilizar de trabalhos para especificação e modelagem (inclusive formais) de SBAM, porém técnicas, ferramentas e métodos de teste funcional podem ser encontradas apenas para sistemas distribuídos de forma geral. Como foi dito, este último arcabouço específico para sistemas distribuídos também pode ser útil para SBAM, porém há a necessidade de que haja uma avaliação sobre sua eficácia dentro do contexto de Agentes Móveis, averiguando o tratamento de características específicas ao conceito de mobilidade.

1.2 Objetivos

Tendo em vista a contextualização e a problemática apresentadas na seção anterior, o objetivo do trabalho contido neste documento é o de definir um método de geração automática de casos de teste a partir de especificações para sistemas baseados em Agentes Móveis. A definição do método consistiu em analisar e escolher ferramentas e técnicas mais adequadas ao contexto de sistemas baseados em Agentes Móveis. As linguagens de especificação e modelagem utilizadas durante as diversas etapas do método precisaram ser escolhidas, levando em conta sua adequabilidade junto a tais sistemas.

1.3 Resultados

Decorrente do objetivo do nosso trabalho, temos como resultado principal um método para a geração automática de casos de teste para sistemas baseados em Agentes Móveis. Outros dois resultados adicionais foram obtidos durante a execução deste. Uma transformação informal de modelos em UML-RT (UML Real-Time) [SR98] para RPOO e um trabalho sobre a identificação de padrões de teste a partir de especificações de padrões de projeto. A seguir temos uma breve descrição destes resultados que são apresentados em maiores detalhes no restante deste documento.

Proposta de um método de geração automática de casos de teste para SBAMs. O método a ser apresentado é constituído de um conjunto de ferramentas e linguagens a serem utilizadas durante um processo de geração de casos de teste a partir de especificações. A proposta define que SBAMs serão especificados e modelados com o uso das Redes de Petri Orientadas a Objetos (RPOO) [Gue02b], linguagem formal a partir da qual os casos de teste serão gerados. Após isto, as ferramentas para geração de casos de teste são apresentadas, as quais constituem um gerador de espaço de estados [Sil05] e um gerador de casos de teste [FJJV96].

Transformação de modelos UML-RT em RPOO. No intuito de tomar proveito dos modelos UML existentes para SBAMs para a construção dos modelos RPOO, e visando aproximar o método proposto ao contexto atual dos desenvolvedores de sistemas distribuídos, uma proposta inicial de transformação informal de modelos em UML-RT para RPOO é apresentada. Esta proposta objetiva motivar trabalhos sobre transformação automática entre estes modelos, tendo como conseqüência uma melhor adequação do método proposto por parte dos desenvolvedores de *software*, principalmente os desenvolvedores de sistemas distribuídos.

Identificação de padrões de teste a partir de padrões de projeto. Baseado nos resultados obtidos de um estudo sobre os casos de teste para Agentes Móveis, e na aplicação do estudo de caso do método proposto, uma forma de identificação de padrões de teste a partir de padrões de projetos para Agentes Móveis é apresentada, juntamente com um conjunto de padrões de teste e com um formato de definição para tais padrões.

1.4 Estrutura da Dissertação

As demais partes deste documento foram estruturadas da forma que segue.

Capítulo 2: Fundamentação Teórica Este capítulo traz os principais conceitos abordados pelo trabalho desenvolvido. O paradigma de Agentes Móveis é apresentado, mostrando suas definições, vantagens, desvantagens e questões relacionadas ao desenvolvimento de SBAM. Conceitos de Teste Formal também são apresentados, com ênfase em um tipo especial chamado de teste baseado em propriedades, onde procedimentos de geração automática de casos de teste e os modelos envolvidos são mostrados.

Capítulo 3: Método de Geração Automática de Casos de Teste Este capítulo apresenta o principal produto deste trabalho que consiste em um método de geração automática de casos de teste para SBAM. Uma visão geral do método é apresentada e, em seguida, as diversas etapas que o compõem são apresentadas em detalhes, mostrando e justificando os artefatos e as ferramentas utilizados pelo mesmo.

Capítulo 4: Transformação de Modelos UML-RT para RPOO Com o intuito de facilitar a adaptação, por parte dos engenheiros de *software*, ao método proposto, neste capítulo, é apresentada uma proposta de transformação informal de modelos UML-RT para modelos RPOO. Este capítulo constitui de uma apresentação de UML-RT e de um exemplo apresentando a modelagem de Agentes Móveis com esta linguagem. Após isto, a transformação propriamente dita é descrita e um exemplo simples é utilizado para ilustrar tal transformação.

Capítulo 5: Estudo de Caso - Parte 1 Este capítulo tem como objetivos introduzir o estudo de caso realizado e apresentar o sistema utilizado em tal estudo de caso através de seu modelo UML-RT, bem como a transformação deste modelo em um modelo RPOO, seguindo o processo apresentado no Capítulo 4.

Capítulo 6: Estudo de Caso - Parte 2 Este capítulo apresenta o restante do estudo de caso, que se refere ao método proposto no Capítulos 3. Com o modelo RPOO da aplicação selecionada, as diversas etapas do método foram aplicadas e estão neste capítulo descritas. Os casos de teste gerados pelo método estão analisados tanto em relação às suas habilidades para revelar falhas como em relação ao potencial destes casos de teste de serem implementados.

Capítulo 7: Padrões de Teste para Agentes Móveis Este capítulo traz um resultado adicional ao trabalho. Mostraremos um procedimento sistemático para a identificação de padrões de teste a partir de especificações de padrões de projetos para Agentes Móveis. É mostrado, neste capítulo, um padrão de teste identificado a partir dos casos de teste obtidos no Capítulo 5. É descrito o procedimento de identificação de padrões de teste a partir de padrões de projeto, e é apresentada uma proposta de formato de definição de tais padrões.

Capítulo 8: Conclusão Neste capítulo final, com base em todo o trabalho apresentado, algumas conclusões são mostradas bem como perspectivas para trabalhos futuros.

Apêndice A: Modelos do Sistema de Conferência Este apêndice contém todos os artefatos gerados durante a aplicação do estudo de caso para o sistema de conferências.

Apêndice B: Casos de Teste para o Sistema de Conferências Este apêndice contém o conteúdo dos casos de teste gerados e trabalhados no estudo de caso.

Apêndice C: Padrões de Teste para Sistemas Baseados em Agentes Móveis Este apêndice traz um conjunto de padrões de teste para Agentes Móveis obtidos através da aplicação do procedimento apresentado no Capítulo 7.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como principal objetivo dar suporte técnico aos leitores acerca dos principais conceitos abordados no decorrer do trabalho. Nele, apresentamos o paradigma de Agentes Móveis, onde seus principais conceitos, vantagens e desvantagens são descritos, em conjunto com suas características tecnológicas. Destacamos o desenvolvimento destes sistemas, apresentando as dificuldades inerentes a esta atividade e as principais abordagens utilizadas. Em um segundo momento, o conceito de teste formal é apresentado. Mostramos não só a teoria relativa a tal técnica de teste como também demais conceitos encontrados nas técnicas e ferramentas utilizadas nos métodos de geração automática de casos de teste baseada em propriedades.

2.1 Agentes Móveis

Nesta seção apresentamos o paradigma de Agentes Móveis, mostrando as principais características dos sistemas desenvolvidos neste paradigma, suas vantagens e desvantagens, e questões relacionadas ao seu desenvolvimento.

2.1.1 Sistemas Baseados em Agentes Móveis

Agentes Móveis é um paradigma para o desenvolvimento de sistemas distribuídos que surgiu com o intuito de unir os conceitos oriundos dos agentes de software com os presentes em mobilidade de código [FPV98; CHK97]. Dado que os sistemas de tal paradigma são formados por agentes, podemos dizer que teremos entidades atuando sobre o interesse de uma outra entidade, com características como autonomia, inteligência e cooperação. Já com relação ao termo "móvel", temos que os agentes são capazes de alterar, dinamicamente, suas ligações com seu lugar de execução. Este último aspecto, o de mobilidade, é o alvo principal dos estudos deste trabalho.

A Figura 2.1 apresenta uma visão geral de um SBAM. Nela podemos ver que este tipo de sistema é executado em diferentes computadores interligados por uma rede. Em cada computador estará

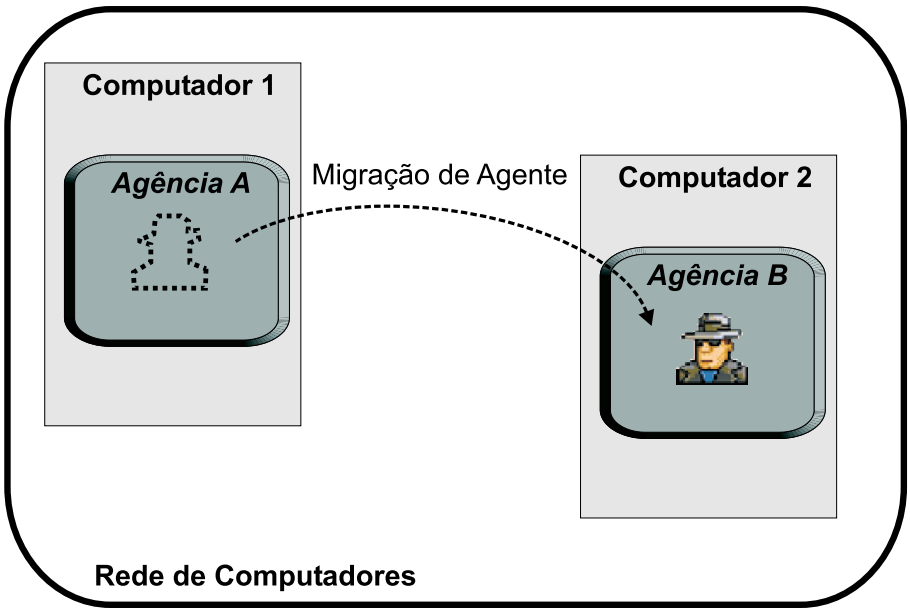

Figura 2.1: Visão Geral de um Sistema Baseado em Agentes Móveis

executando uma agência[1] que abrigarão os agentes do sistema, dando suporte às suas execuções. Estando em uma agência, um agente executará as suas tarefas como, por exemplo, comunicação com outros agentes e acesso a recursos locais. Em um dado instante, um agente poderá se mover para uma outra agência, caracterizando a sua migração. Como pode ser visto na Figura 2.1, um determinado agente estava executando na ***Agência A*** (note que a figura do agente está tracejada) e migrou para a ***Agência B***, para executar suas tarefas nesta segunda agência.

Existem na comunidade diversas tecnologias que visam dar suporte à implementação e execução destes sistemas [BCTR03; Gmb98; Agl]. Tais tecnologias, geralmente, provêem uma API de desenvolvimento e uma plataforma em que os agentes irão executar (agência), resultando no que é conhecido por *sistema de Agentes Móveis* ou *plataforma de execução para Agentes Móveis*. As plataformas constituem uma espécie de máquina virtual para a execução dos agentes, representada pelas agências e cujos serviços são disponibilizados através de sua API (*Application Programming Interface*). Além de prover diversos serviços aos agentes, tais como, criação de agentes, destruição de agentes e comunicação entre agentes, as plataformas de execução podem prover dois tipos de mobilidade. No primeiro tipo, chamado de mobilidade forte, tanto os dados quanto o estado de execução são preservados durante o processo de migração. Já com a mobilidade fraca, apenas os dados são preservados, terminando a execução e a reiniciando após a migração.

2.1.2 Vantagens e Desvantagens

A utilização do paradigma de Agentes Móveis pode trazer uma conjunto de vantagens para quem o utiliza [Lim04], e dentre elas destacamos as apresentadas a seguir. Apesar destas vantagens serem

[1] É possível que em um computador esteja executando mais de uma agência.

encontradas em outros paradigmas, uma das maiores contribuições de Agentes Móveis está no fato de que consegue reuni-las em um único paradigma.

- **Diminuição do tráfego na rede:** Uma vez que os agentes têm a habilidade de migrar pelas diversas máquinas do sistema, os dados poderão ser processados localmente, e apenas parte destes dados (os resultados) precisarão migrar pela rede.
- **Execução assíncrona e autônoma:** Os agentes permitem que as tarefas sejam executadas de forma assíncrona e autônoma, uma vez que eles executam independentemente de quem os criou.
- **Heterogeneidade:** Uma vez que os agentes estão ligados apenas aos seus ambientes de execução, estes podem executar sobre diferentes plataforma de *hardware* e de *software*, configuração comum em sistemas distribuídos.
- **Robustez e tolerância a falhas:** Os agentes são sensíveis ao ambiente, podendo perceber quando este está instável. Com isto, a construção de sistemas robustos e tolerantes a falhas é facilitado, pois este recurso pode ser utilizado no intuito de evitar que a execução dos agentes seja finalizada de forma errada.

Apesar das vantagens apresentadas, tal paradigma também traz algumas desvantagens a quem o utiliza [Pei02], como:

- falta de mecanismos de segurança tanto em relação ao agente quanto em relação ao ambiente em que eles estarão executando;
- necessidade de se instalar um sistema extra (plataforma de execução) em cada computador que poderá receber um agente;
- a construção de agentes muito grandes pode acarretar em um aumento do tráfego na rede, eliminando a vantagem de redução do tráfego na rede;
- complexidade relacionada ao desenvolvimento de Sistemas Baseados em Agentes Móveis (vide Seção 2.1.3).

2.1.3 Desenvolvimento de Sistemas Baseados em Agentes Móveis

O desenvolvimento de sistemas baseados em Agentes Móveis pode trazer uma série de vantagens bem como abranger um conjunto interessante de áreas de aplicação. No entanto, o desenvolvimento deste tipo de sistema ainda é uma tarefa complexa. Parte desta complexidade pode ser atribuída à característica distribuída de tais sistemas, mas em sua maior parte, esta complexidade advém da imaturidade dos processos de desenvolvimento em relação ao tratamento do conceito de

mobilidade. Este é um paradigma cujo arcabouço de métodos, técnicas e ferramentas para seu desenvolvimento ainda se encontra em uma fase prematura de construção. Algumas propostas podem ser encontradas que visam amenizar tal carência. Alguns trabalhos propõem processos que possuem o conceito de mobilidade inserido em diversas fases do desenvolvimento [Mar03], enquanto que outros propõem formatos de artefatos para o paradigma em questão [Gue02a]. Na área de métodos formais (área de atuação deste trabalho), vários formalismos vêm sendo propostos para a modelagem e especificação de Agentes Móveis. Alguns são específicos para a modelagem de mobilidade de código [Mil99], outros apresentam extensões de linguagens [ASB02] para Agentes Móveis, e outros apresentam soluções que usam formalismos já conhecidos [dF03; LFG03; Mil99]. Apesar desses esforços, pouco tem sido feito na direção de propor métodos e técnicas para melhorar os processos de desenvolvimento destes sistemas. Este fato se agrava ainda mais quando nos referimos especificamente a testes. Poucas propostas existem e isso faz com que, em muitos casos, os sistemas sejam testados ainda de maneira *ad-hoc*.

No entanto, dentre estes, o trabalho de Guedes [Gue02a] é um dos que mais contribui para o processo de desenvolvimento de SBAM como um todo, pois não se concentra apenas em alguma fase do desenvolvimento ou em algum tipo de artefato. Em seu trabalho, Guedes faz uma adaptação do trabalho de Larman [Lar99] adicionando aspectos importantes para a representação de mobilidade nas fases de análise e projeto de sistemas. Segundo o modelo proposto, a fase de projeto do sistema gera três artefatos: projeto arquitetural independente de plataforma; projeto detalhado independente de plataforma; e projeto detalhado dependente de plataforma. Os projetos independente de plataforma não consideram quaisquer características inerentes às plataformas de execução de Agentes Móveis, já o projeto detalhado dependente de plataforma é desenvolvido levando-se em consideração a plataforma de execução (ou as possíveis plataformas de execução, gerando mais de um artefato). Esta abordagem é motivada pelo potencial de reusabilidade dos modelos independentes de plataforma, além do poder de compreensão do sistema provida pela abstração dos modelos independentes de plataforma.

Visando esta mesma reusabilidade (de soluções de projeto), diversos trabalho sobre padrões de projeto para Agentes Móveis têm sido propostos pela comunidade [AL98; DW99; SSJ02; KKPS98; TOH99; TOH01; TTOH01; LMFR03; Lim04; LMSF04; LFG04]. Em conjunto com metodologias como a proposta por Guedes, o uso de padrões de projeto tendem a minimizar a complexidade inerente ao desenvolvimento de SBAM, uma vez que diversas soluções poderão ser reutilizadas.

2.2 Teste Formal

Durante algum tempo e ainda em diversos casos hoje em dia, teste de software tem se restringido a uma atividade isolada no fim do processo de desenvolvimento de sistemas, ou seja, que só é executada após a atividade de implementação. Para alguns autores (e.g. [MS01]), teste é um tipo de atividade que deve ser aplicada em vários pontos durante o desenvolvimento do software, e não unicamente no seu fim. Em geral, este processo de teste é definido em separado do processo de desenvolvimento, pois

eles possuem objetivos distintos. Apesar de separados na definição, estes processos são intimamente ligados [MS01]. Além da atividade de teste não ser uma única atividade no fim do processo de desenvolvimento, e sim um conjunto destas, tais atividades têm sido aplicadas cada vez mais cedo durante o desenvolvimento dos softwares. Esta abordagem visa, principalmente, fazer com que as faltas sejam encontradas cada vez mais cedo, evitando sua propagação para as etapas seguintes.

Neste contexto (de testes sendo planejados e executados cada vez mais cedo), surgem de forma interessante os testes funcionais, pois tão cedo existam especificações tão cedo testes poderão ser obtidos. Este tipo de teste (também chamado *black-box*, teste baseado em especificação ou teste de conformidade) visa verificar se uma determinada implementação está em conformidade com sua respectiva especificação. Neste trabalho, nós iremos nos concentrar em um tipo especial de teste funcional chamado de teste formal. Este tipo de teste é caracterizado pela presença de especificações formais, baseadas em algum formalismo, e procedimentos também formais para a geração de casos de teste, oráculos e dados.

Um processo de teste formal possui as fases de geração e execução dos testes, ambas possivelmente automáticas. Durante a geração, a especificação do sistema a ser testado é analisada com o intuito de se obter informações sobre tal sistema que permite, ao testador, predicar sobre sua correção. Neste sentido, são gerados os seguintes artefatos.

- **Casos de Teste:** Descreve um comportamento esperado do sistema.
- **Dados de Teste:** São as entradas do sistema que propiciam a execução dos casos de teste.
- **Oráculos:** Especificação de procedimentos capazes de verificar se uma dada execução do sistema (estimulada pelos dados de entrada) está de acordo com o comportamento definido pelos casos de teste.

Na fase de execução, o que foi gerado anteriormente será implementado e executado sobre o sistema que está sendo testado. Ao final, os resultados da execução dos testes são analisados. O trabalho que será apresentado nos próximos capítulos se concentrará na geração de casos de teste a partir de especificações formais. Desta forma, questões relacionadas com dados de teste e oráculos não estão no escopo do trabalho. Além disso, apesar de apresentarmos uma proposta de implementação e execução dos casos de teste no Capítulo 6, esta proposta servirá apenas para avaliar os casos de teste gerados, principal contribuição deste trabalho.

2.2.1 *Framework* para Teste Formal

Nesta seção, mostraremos o *framework* proposto por Tretmans [Tre99] para o uso de métodos formais em teste de conformidade (teste formal). Este *framework* abstrai os conceitos presentes no processo de teste de conformidade, além de definir uma estrutura que nos permite tratar o processo de teste de uma maneira formal. Veremos, portanto, que este *framework* irá unir o mundo informal de implementações, testes e experimentos com o mundo formal das especificações e dos modelos, nos dando

uma abstração de um processo formal para teste de conformidade. Esperamos que este *framework* facilite o entendimento do leitor em relação a teste formal e seus conceitos.

O primeiro passo para o entendimento de teste de conformidade seria a definição do termo *Conformidade*. Porém, antes de defini-lo, é preciso definir especificações e implementações. Especificação é um conceito formal e, ao universo de todas as especificações é dado o nome de *SPECS*. Implementações são chamadas de IUT (Implementation Under Test), e o seu universo é chamado de *IMPS*. Tendo isto, o conceito de *Conformidade* será expresso através da relação:

IUT **conforms-to** s

significando que IUT é a implementação correta de 's', e onde **conforms-to** $\subseteq$ *IMPS* $\times$ *SPECS*.

Ua vez que IUT não pode ser definido formalmente (o que dificulta a definição formal de **conforms-to**), assume-se que para cada IUT $\in$ *IMPS* existe um modelo correspondente $i_{IUT} \in$ *MODS* (universo de todos os modelos); relação determinada de hipótese de teste. Não necessariamente i_{IUT} é conhecido, apenas assume-se sua existência. Desta forma, podemos tratar implementações de maneira formal através de seus modelos. Assim, conformidade pode ser definida formalmente através da relação chamada de *Relação de Implementação*, definida como **imp** $\subseteq$ *MODS* $\times$ *SPECS*. A implementação estará correta em relação à sua especificação (IUT **conforms-to** s) se, e somente se, IUT é **imp**-relacionado com s (i_{IUT} **imp** s).

O comportamento de uma IUT é verificado através de experimentos executados sobre a mesma (estamos tratando de teste). À especificação desses experimentos é dado o nome de *Caso de Teste* e, à execução dos experimentos é dado o nome de *Execução de Teste*. O *framework* define como *TESTS* o conjunto de todos os casos de teste e como EXEC(*t*, IUT) o procedimento operacional não formal que aplica o caso de teste (*t*) à implementação (IUT). O procedimento EXEC produz uma série de observações que pertencem ao universo de observações definido formalmente por *OBS*. Com o intuito de formalizar EXEC, uma função de observação *obs* : *TESTS* $\times$ *MODS* $\rightarrow \mathbb{P}$(*OBS*) é introduzida, onde $\mathbb{P}$(*OBS*) é o conjunto potência de *OBS*. Assim, *obs* modela formalmente EXEC, pois, pela hipótese de teste temos:

$$\forall \text{IUT} \in \textit{IMPS}\ \exists i_{IUT} \in \textit{MODS}\ \forall \text{t} \in \textit{TESTS} : \text{EXEC}(t, \text{IUT}) = \text{obs}(t, i_{IUT}) \quad (1)$$

Com a finalidade de classificar execuções de teste como certas ou erradas, é adicionada ao *framework* uma função chamada *Função de Veredito*, denotada por $v_t : \mathbb{P}(\textit{OBS}) \rightarrow \{fail,\ pass\}$, que, a partir de observações resultantes dos teste, classifica a execução dos testes como correta (*pass*) ou falha (*fail*). Com isso, a seguinte abreviação é utilizada: IUT **passes** $t \overset{\text{def}}{\Longleftrightarrow} v_t(\text{EXEC}(t, \text{IUT})) = pass$ (2), que pode ser estendida para *Conjunto de Teste T* $\subseteq$ *TESTS* como IUT **passes** $T \overset{\text{def}}{\Longleftrightarrow} \forall t \in T$: IUT **passes** *t*.

Como teste de conformidade consiste em verificar por meio de teste se uma dada implementação está de acordo com sua especificação (de acordo com a relação **imp**), é desejado que exista um conjunto de testes T_s tal que IUT **conforms-to** s $\Longleftrightarrow$ IUT **passes** T_s (3). Tal conjunto de testes é

chamado de completo, porém, na prática, ou não conseguimos obtê-lo ou isto é uma prática inviável. Assim, conjuntos de testes chamados de *sound*, que apenas garantem IUT **conforms-to** s $\Longrightarrow$ IUT **passes** T_s, são utilizados. A seguir temos a formalização para um conjunto de testes específico:

$$\forall i \in MODS : i \textbf{ imp } s \Longleftrightarrow \forall t \in T : v_t(obs(t, i)) = pass \tag{4}$$

Assumindo que a completeza do conjunto de testes foi provada no nível de modelo (*MODS*) e que a hipótese de teste existe, podemos dizer que a conformidade de uma implementação com relação à sua especificação pode ser verificada por meio de teste. A seguir temos a derivação que justifica.

IUT **passes** T

iff (Definição de IUT **passes** T)

$\forall t \in T : IUT$ **passes** t

iff (Definição de IUT **passes** t)

$\forall t \in T : v_t(EXEC(t, IUT)) = pass$

iff (Expressão (1))

$\forall t \in T : v_t(obs(t, i_{IUT})) = pass$

iff (Expressão (4))

i_{IUT} **imp** s

iff ($Definição\ de\ Conformidade$)

IUT **conforms-to** s

Por fim, o *framework* é composto por uma função que tem a finalidade de derivar conjuntos de testes. Esta função, denotada por $der_{\textbf{imp}} : SPECS \rightarrow \mathbb{P}(TESTS)$, tem a finalidade de gerar conjuntos de teste que sejam completos, ou que sejam *sound*.

Um conceito bastante utilizado na comunidade de teste formal é a relação de conformidade chamada **ioco** [Tre99], que se baseia nas entradas e saídas (input/output) dos modelos a serem verificados. Esta relação define que um modelo de uma implementação está de acordo com sua especificação se, a partir de qualquer instante, para uma mesma seqüência de entradas, tanto o modelo quanto a especificação produzirão a mesma saída. Na verdade, não é necessário que as saídas sejam iguais, é preciso que o conjunto de saídas produzido pelo modelo esteja contido no conjunto produzido pela especificação, uma vez que poderemos estar tratando de sistemas não-deterministas.

Este *framework* pode ser de grande contribuição para quem deseja criar ou utilizar um método de teste formal (teste de conformidade), pois apresenta os principais conceitos necessários a tal método em um nível de abstração alto. A proposta deste *framework* tem como principal objetivo unir os conceitos não formais de teste, implementação e experimento com os formalismos de especificações e modelos.

2.2.2 Teste Formal Baseado em Propriedades

Uma das técnicas mais utilizadas pela comunidade de teste formal para a geração dos casos de teste é a baseada em conceitos de verificação de modelos [CGP99] (*Model Checking*). Esta técnica tem o nome de teste baseado em propriedades pois, assim como na verificação de modelos, as propriedades que se deseja verificar são especificadas formalmente e servem de entrada para a técnica. Por essa razão, esta técnica também recebe o nome de teste formal com propriedades explícitas, uma vez que as propriedades são elaboradas pelo testador. Se por um lado a verificação de modelos visa percorrer uma especificação em busca de falta de conformidade entre esta e uma determinada propriedade, a geração de casos de teste baseada em propriedades visa percorrer esta mesma especificação com o intuito de selecionar partes desta especificação que satisfazem a propriedade. Com isto, teste formal pode se utilizar de várias técnicas bem consolidadas de verificação de modelos, como por exemplo, as utilizadas para percorrer, comparar e reduzir grafos [FJJV96]. Esta seção irá explicar este tipo de técnica uma vez que é esta a técnica que a ferramenta a ser utilizada pelo método proposto neste trabalho utiliza para a geração dos casos de teste.

As propriedades utilizadas para a geração dos casos de teste são chamadas de objetivos de teste (em Inglês o termo utilizado é *test purpose*). Os objetivos de teste constituem partes do sistema modelado as quais se deseja testar. Assim, dado o modelo do sistema e um objetivo de teste, a técnica gera casos de teste que visam testar, especificamente, as partes descritas pelo objetivo de teste. Em geral, o processo acontece seguindo os passos ilustrados na Figura 2.2.

Como pode ser visto na Figura 2.2, os sistemas são especificados utilizando-se uma determinada linguagem, muito possivelmente uma linguagem gráfica e já habitual no desenvolvimento de tais sistemas como, por exemplo, UML [IJB99]. Caso esta linguagem não seja baseada em algum formalismo, há a necessidade de uma formalização (pois estamos tratando de geração automática) destes modelos, caso contrário, geralmente é feita uma representação (também chamada de simulação ou transformação) em algum formalismo base. Esta representação ocorre devido ao fato de que a maioria das técnicas e ferramentas para a geração de casos de teste utilizam, como entrada, especificações em formatos bases, por exemplo grafos de transições rotuladas (Labelled Transition System - LTS). Os objetivos de teste são especificados, em geral, também neste formalismo base. No entanto, podem sofrer um processo de transformação semelhante à especificação do sistema, caso sejam descritos com algum outro tipo de linguagem. Com as duas especificações (sistema e objetivo de teste), o sintetizador de teste irá gerar uma especificação do sistema que contém apenas as partes consideradas pelo objetivo de teste, ou seja, irá selecionar na especificação do sistema apenas as partes descritas pelo objetivo de teste. Após isto, com esta especificação sintetizada, a ferramenta de geração de casos de teste se encarregará de gerar os diversos casos de teste, geralmente na mesma linguagem utilizada na especificação sintetizada, porém diagramas de seqüencia, *state charts* e outros também são utilizados.

Os modelos utilizados pelas ferramentas de geração os casos de teste são os LTS's. Este tipo de modelo descreve os sistemas por seus estados e pelas ações responsáveis pelas mudanças de estados. Um LTS é um grafo definido por $M = (Q^M, A, T^M, q^M_{init})$, onde Q^M é o conjunto de estados, A

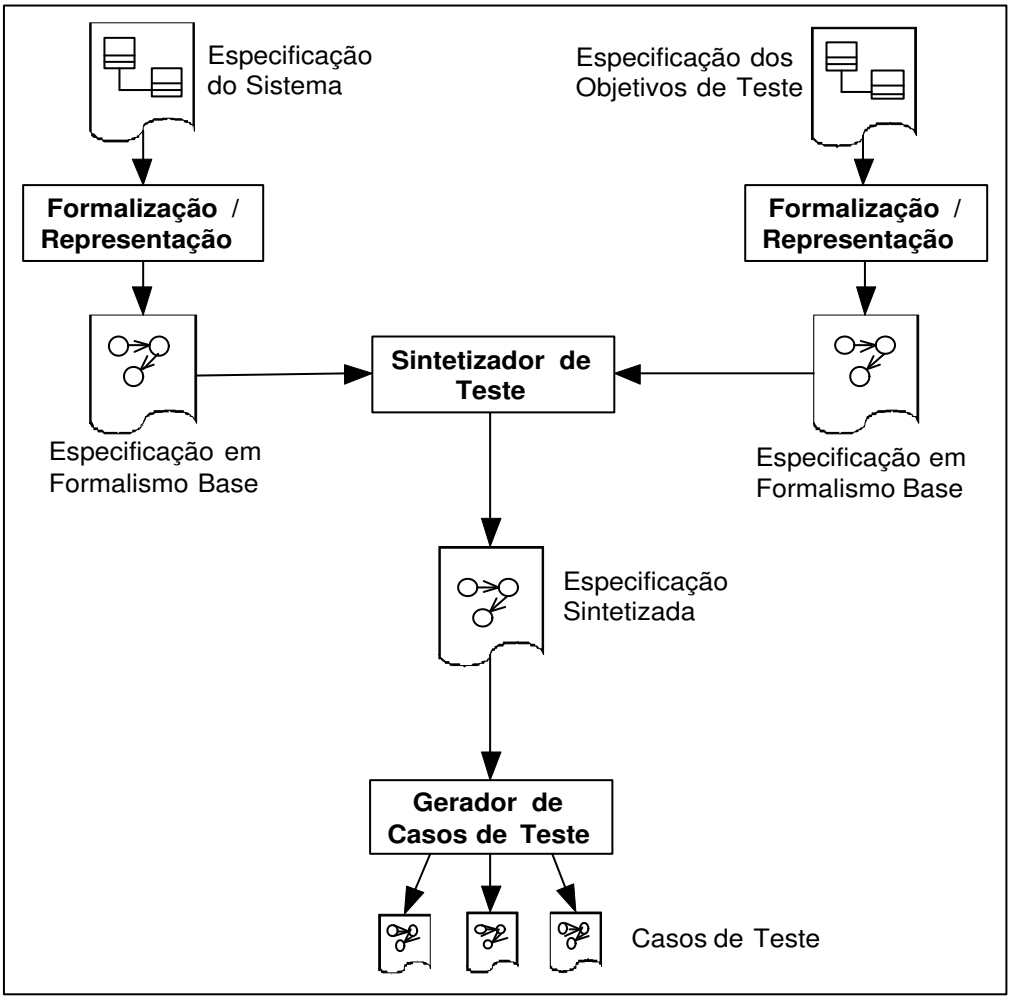

Figura 2.2: Modelo Geral de um Método de Teste Formal Baseado em Propriedades

é o alfabeto de ações, q^M_{init} é o estado inicial, e $T^M \subseteq Q^M \times A \times Q^M$ é a relação de transição. Uma implementação a ser testada será posta em um ambiente de teste que irá se comunicar com ela através de seus pontos de controle e de observação. Desta forma, o ambiente de teste (chamado simplesmente de testador) possuirá, apenas, uma visão externa da implementação, caracterizando o teste funcional. Contudo, as especificações dos sistemas, em geral, descrevem não somente entradas e saídas, mas também ações internas do sistema. Neste sentido, com a finalidade de teste, é preciso que tenhamos uma visão dos sistemas baseado nos seus pontos de controle e de observação, e para isso as ferramentas utilizam IOLTS (Input Output Labelled Transition System), que é um tipo de LTS baseado nas entradas e saídas dos sistemas. Um IOLTS é um LTS em que $A = A_i \cup A_o \cup \{\tau\}$, para A_i o conjunto de ações de entrada, A_o o conjunto de ações de saída e τ um rótulo que identifica uma ação interna ao sistema.

Os objetivos de teste também são especificados com IOLTS ($TP = (Q^{TP}, A, T^{TP}, q^{TP}_{init})$), com o acréscimo de dois conjuntos de estados especiais, $Accept^{TP} \subseteq Q^{TP}$ e $Refuse^{TP} \subseteq Q^{TP}$, em que $Accept^{TP} \cap Refuse^{TP} = \emptyset$. Todas as linhas de execução que levam o modelo a um estado

pertencente a $Accept^{TP}$ sem passar por estados pertencentes a $Refuse^{TP}$ pertencerão a algum caso de teste selecionado pelo respectivo objetivo de teste.

Os casos de teste gerados pelas ferramentas, em geral, também são especificados com IOLTS ($TC = (Q^{TC}, A, T^{TC}, q^{TC}_{init})$). Neste caso, três conjuntos de estados especiais são definidos, $Pass \subseteq Q^{TC}$, $Fail \subseteq Q^{TC}$ e $Inconclusive \subseteq Q^{TC}$, em que $Pass \cap Fail = \emptyset$, $Fail \cap Inconclusive = \emptyset$ e $Pass \cap Inconclusive = \emptyset$. Em $Pass$ estão os estados de aceitação, em $Fail$ os estados de falha e em $Inconclusive$ os estados cujo veredito não pôde ser definido.

A seguir apresentaremos um pequeno exemplo ilustrando o processo de geração de casos de teste baseado em propriedades. A Figura 2.3 apresenta a especificação de um sistema que contém um Agente Móvel encarregado de coletar uma revisão para um artigo e aprová-la. Este agente pode solicitar ao ambiente que seja movido (*!mover*), solicitar a revisão (*!obterRevisao*), solicitar aprovação (*!obterAprovacao*) ou enviar uma mensagem de erro (*!erro*). Já o ambiente interage com o sistema solicitando que o agente busque uma revisão (*?revise*), informando o agente que ele chegou em um destino (*?chegou*), informando o agente que houve falha na migração (*?falhou*), informando uma revisão (*?revisao*), aprovando (*?aprovar*) e não aprovando a revisão (*?naoAprovar*). O estado inicial do sistema é o estado **'0'**.

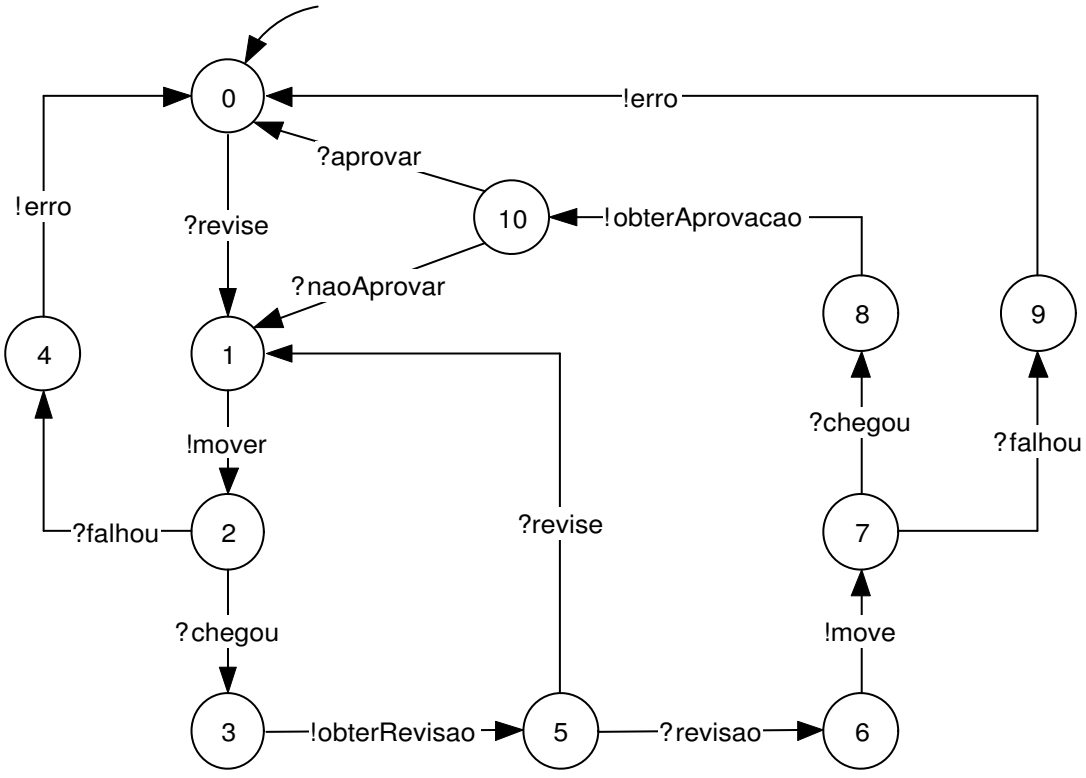

Figura 2.3: Exemplo de Especificação de um Sistema em IOLTS

Um exemplo de objetivo de teste para esta especificação é o contido na Figura 2.4. Este objetivo de teste visa gerar casos de teste que exercitem as linhas de execução do sistema onde o agente solicita

sua migração, esta migração ocorre sem problemas e o agente solicita uma revisão ou uma aprovação. A transição *"!obter*"* é disparada tanto para *!obterRevisao* como para *!obterAprovacao* devido ao uso do *. Além disso, as linhas de execução onde ocorrem falhas na migração (transição *"?falhou"*) não serão exercitadas pelos casos de teste gerados para este objetivo de teste. Desta forma, após serem submetidos a uma ferramenta de geração de casos de teste, essa especificação e esse objetivo de teste produzirão um conjunto de casos de teste que conterá, por exemplo, o caso de teste apresentado na Figura 2.5.

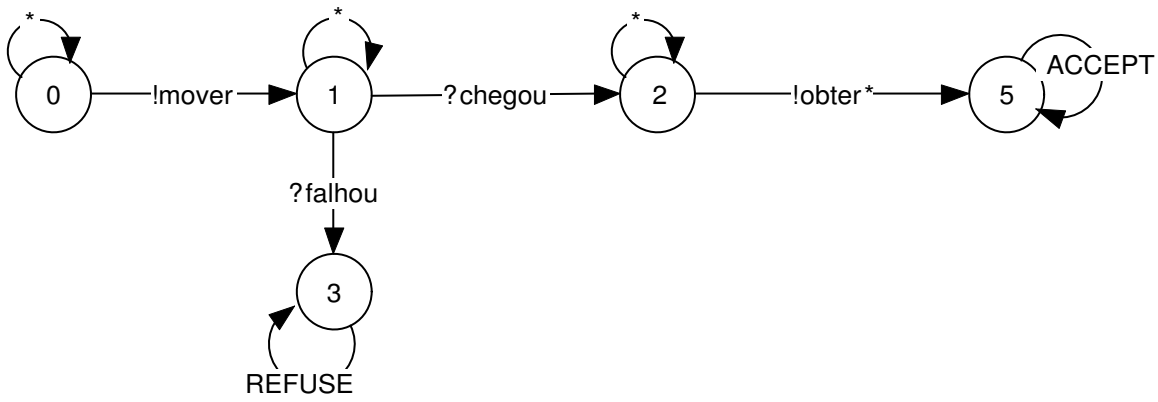

Figura 2.4: Exemplo de Objetivo de Teste em IOLTS

Na Figura 2.5, note que o estado **'11'** é um estado de aceitação e que os estados de falha e as transições que levam o sistema até eles foram omitidos por simplicidade. Estes estados de falha são alcançados por transições que partem de todos os estados do grafo (exceto do estado **'11'**) cujo rótulo é *, significando que qualquer transição a ser disparada em um estado que não seja a prevista pelo caso de teste levará o teste a um estado de falha. Um caso de teste especificado como um IOLTS, como o apresentado na Figura 2.5, na verdade, modela um outro sistema que tem a tarefa de exercitar o sistema ser testado. Neste sentido, as ações que no sistema a ser testado são de saída (iniciadas com *!*) serão de entrada no caso de teste, e as ações que são de entrada (iniciadas com *?*) serão de saída do caso de teste. Desta forma, onde na especificação temos *!* (resp. *?*), nos casos de teste teremos *?* (resp. *!*).

2.3 Conclusão

Este capítulo apresentou os principais conceitos de Agentes Móveis e Teste Formal necessários para o entendimento do trabalho apresentado. Foi mostrado que Agentes Móveis é um paradigma bastante interessante para alguns tipos de aplicações, com suas vantagens e desvantagens. Mostrou-se, também, que o desenvolvimento deste tipo de sistema ainda é uma atividade complexa devido, principalmente, à carência de técnicas, métodos e metodologias específicas para tal paradigma, apesar de algumas propostas terem sido apresentadas à comunidade.

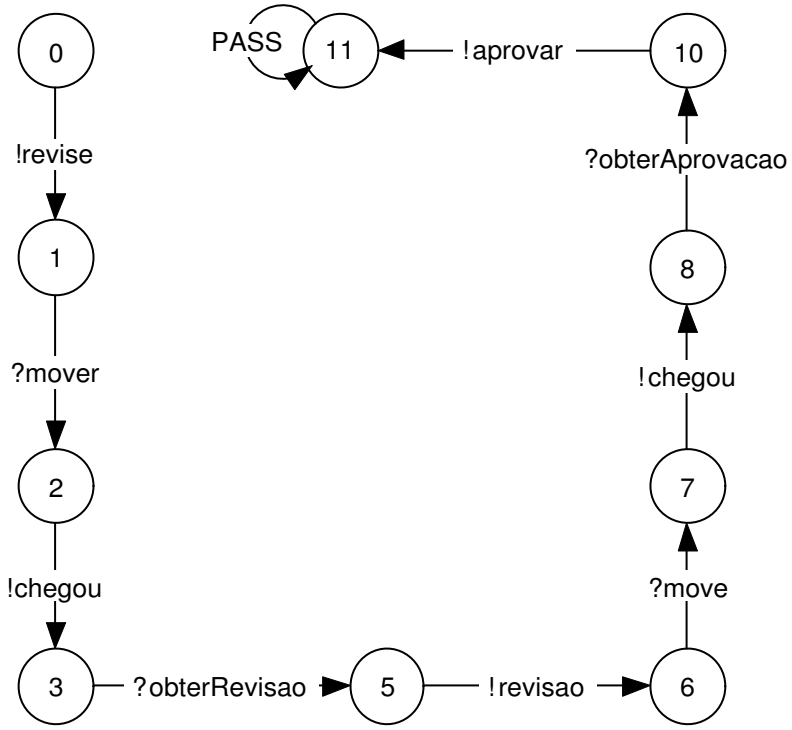

Figura 2.5: Exemplo de Caso de Teste em IOLTS

Com relação a testes formais, seus conceitos foram apresentados formalmente através de um *framework* formal para testes formais e informalmente através de uma modelo básico (Figura 2.2) de teste formal para sistemas distribuídos. O tipo de teste formal utilizado pelo trabalho apresentado nos próximos capítulos foi descrito: teste baseado em propriedades. Os modelos para especificação de sistemas, objetivos de teste e casos de teste apresentados acima, bem como o formato geral de geração automática de casos de teste apresentado na Figura 2.2 foram mostrados neste capítulo devido ao fato de serem amplamente adotados pela comunidade [BFdV$^+$99; FJJV96; JJ02; CJRZ02; Tre99; PJLT$^+$02; Jar02; HLU03; LG02] e por serem do tipo de abordagem contemplada pelas ferramentas utilizadas neste trabalho. No entanto, outras técnicas e ferramentas também podem ser encontradas na comunidade [WLPS00; KGHS98; RM00], porém não estão no escopo deste trabalho.

Capítulo 3

Método de Geração Automática de Casos de Teste

3.1 Introdução

Neste capítulo, apresentamos o principal produto deste trabalho: um método de geração automática de casos de teste para SBAM. Após prover uma visão geral do método, descreveremos as diversas etapas que o compõem, mostrando e justificando os artefatos e as ferramentas utilizados pelo mesmo.

O foco deste trabalho (vide Capítulo 2) é a geração de casos de teste, portanto, devido a limitações de tempo, a geração de dados de teste e de oráculos não estão contempladas neste trabalho. Contudo, no Capítulo 7, mostramos uma proposta de geração semi-automática e de implementação destes oráculos e dados, o que viabilizará a aplicação dos casos de teste gerados pelo método.

Dada a carência de trabalhos nesta área e com o objetivo de propor uma solução inicial para a problemática de geração automática de casos de teste para SBAM, a definição do método consistiu em analisar e escolher ferramentas e técnicas disponíveis na comunidade (principalmente a de sistemas distribuídos) mais adequadas ao contexto de sistemas baseados em Agentes Móveis, integrando-as em um método de geração automática de casos de teste. As linguagens de especificação e modelagem a serem utilizadas durante as diversas etapas do método foram analisadas e selecionadas, levando em conta sua adeqüabilidade junto a tais sistemas. Além da seleção destas técnicas, ferramentas e linguagens, o trabalho apresenta uma forma de se modelar SBAMs de maneira que esta combinação de técnicas e ferramentas seja usada para gerar casos de teste significativos no tocante a mobilidade.

Neste sentido, não estamos propondo novas linguagens de especificação e modelagem, nem novas técnicas ou ferramentas de geração de casos de teste, e sim, usufruindo das já consolidadas pela indústria e comunidade acadêmica em um novo e imaturo contexto - teste formal de SBAM.

Dado que o método que apresentamos objetiva gerar, de forma automática, casos de teste que têm o intuito de testar sistemas baseados em Agentes Móveis, a escolha das técnicas, ferramentas e linguagens que compõe o método foi principalmente baseada em seu potencial em tratar conceitos que envolvam mobilidade de código, bem como as demais características de sistemas distribuídos.

3.2 Visão Geral do Método

Apresentaremos nesta seção uma breve descrição do método com o intuito de facilitar o entendimento de suas etapas. A Figura 3.1 mostra uma visão geral das etapas do método proposto.

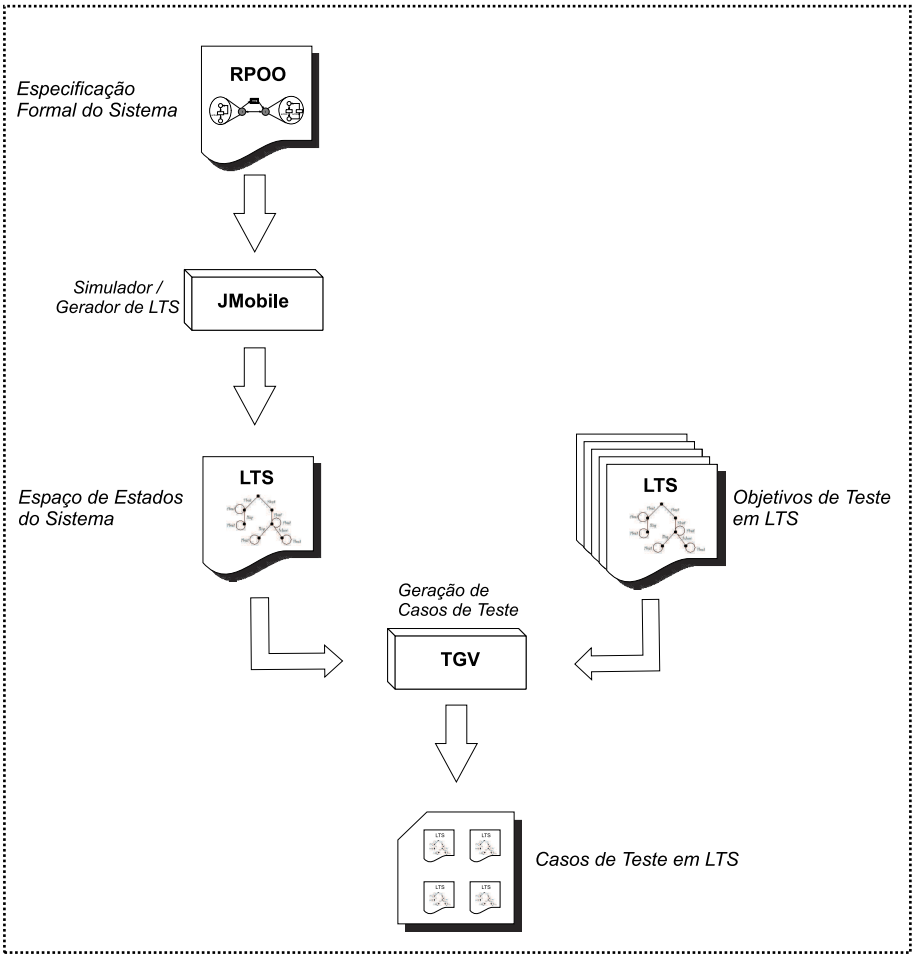

Figura 3.1: Visão Geral do Método de Geração Automática de Casos de Teste

Como pode ser visto na Figura 3.1, o método adota como linguagem de especificação e modelagem o formalismo de Redes de Petri Orientadas a Objetos (RPOO) [Gue02b]. Desta forma, os sistemas a serem testados precisam estar modelados nesta linguagem. Tendo isto, a ferramenta *JMobile* [Sil05] receberá tal especificação como entrada e gerará o seu espaço de estados, que estará no formato de sistema de transições rotuladas (LTS). Os objetivos de teste (parte direita da figura) são especificados, também, em LTS - cada objetivo de teste possui um LTS respectivo. Grande parte das ferramentas de seleção de casos de teste recebe como entrada o sistema especificado em LTS e aplica sua técnica através de seus algoritmos para a geração dos casos de teste. Este é o caso de TGV (Test Generation based on Verification techniques) [FJJV96; JJ02], ferramenta utilizada para a geração dos casos de teste propriamente dita. TGV recebe como entrada o modelo do sistema e a especificação de um objetivo de teste, e seleciona os casos de teste a

partir da especificação com base em tal objetivo de teste. Os casos de teste produzidos são apresentados, também, em LTS e poderão ser implementados ou convertidos para outra linguagem, como por exemplo *message sequence chart* e diagrama de seqüencia de UML.

A seguir, mostraremos em detalhes as etapas do método proposto bem como os artefatos utilizados durante o processo. Falaremos sobre a linguagem de especificação formal RPOO, a ferramenta de geração de LTS JMobile e a ferramenta de seleção de casos de teste TGV. Adicionalmente, ao final, faremos considerações sobre a utilização do método dentro de um processo de desenvolvimento.

3.3 Especificação Formal de Sistemas Baseados em Agentes Móveis

A fim de que a geração dos casos de teste seja automática, é preciso que os modelos a partir de onde os casos de teste serão extraídos sejam formais, escritos utilizando-se de alguma linguagem formal. Um dos passos do nosso trabalho foi o de selecionar tal linguagem, o que resultou na escolha de RPOO.

A seguir, apresentamos RPOO através de um exemplo simples, porém contextualizado a SBAM, já que se trata do modelo de um padrão de projeto para Agentes Móveis. Este exemplo também será usado no decorrer deste capítulo para explicar as demais etapas do método proposto. Um estudo de caso completo pode ser encontrado nos Capítulos 5 e 6.

3.3.1 RPOO

RPOO é um formalismo que foi criado com o intuito de tomar proveito dos modelos estruturais providos pela orientação a objetos (OO) em conjunto com os modelos comportamentais das redes de Petri [Jen92]. Neste sentido, RPOO propõe a modelagem de sistemas sobre duas perspectivas ortogonais: uma provida pela semântica de sistemas de objetos da orientação a objetos; e outra provida pela semântica das redes de Petri coloridas. Através da perspectiva orientada a objetos, os modelos RPOO descrevem as classes do sistema e os relacionamentos entre seus objetos, já as redes de Petri coloridas são utilizadas para descrever o comportamento interno destes objetos. Desta forma, para cada classe do sistema de objetos existe uma rede de Petri que descreve seu comportamento interno.

Para explicar melhor este formalismo, utilizaremos o modelo de um padrão de projetos para Agentes Móveis - o *Itinerary*. Este padrão provê uma forma na qual um agente pode migrar por diferentes agências de um sistema executando uma tarefa específica em cada uma delas. A solução apresentada pelo padrão descreve a infra-estrutura necessária para que este agente (chamado *ItineraryAgent*) migre por tais agências executando esta tarefa em cada uma delas e retorne, ao final, à agência de origem com o resultado da execução.

Na Figura 3.2, temos um diagrama de classes para o padrão *Itinerary*. A classe *Agency* modela a agência que hospeda o *ItineraryAgent* e por onde este mesmo terá que migrar executando sua tarefa. Note que *ItineraryAgent* possui uma lista de agências (*Agencies*) que representa o itinerário a ser percorrido por ele.

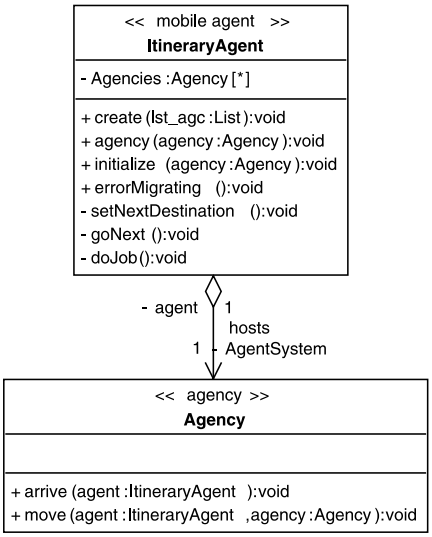

Figura 3.2: Diagrama de Classes do *Itinerary*

A classe *Agency* possui o método *move()* que é invocado pelo agente quando este deseja migrar para uma outra agência e o método *arrive()* que é invocado quando um novo agente chega à agência. Já a classe *ItineraryAgent* possui os seguintes métodos:

create(): Chamado quando o agente é criado;

agency(): Utilizado para informar ao agente a agência em que ele se encontra;

initialize(): Chamado quando se desejar configurar o agente quando ele chega em uma agência;

errorMigrating(): Chamado para informar ao agente que não foi possível migrar para uma determinada agência;

setNextDestination(): Seleciona a próxima agência para onde o agente deve migrar;

goNext(): Chamado quando o agente deseja migrar para a próxima agência do itinerário;

doJob(): Método encarregado de executar determinada tarefa em cada agência;

Como foi dito acima, para cada classe do modelo orientado a objetos, RPOO provê uma rede de Petri colorida responsável por modelar o seu comportamento. Desta forma, nosso exemplo terá duas redes, uma para a classe **ItineraryAgent** (Figura 3.3) e outra para a classe **Agency** (Figura 3.4). Utilizaremos a rede de Petri da classe **ItineraryAgent** (Figura 3.3) para ilustrarmos as definições de redes de Petri que serão apresentadas a seguir.

Como vemos na Figura 3.3, uma rede de Petri é composta por lugares (elípses), transições (retângulos) e arcos (setas direcionadas). Por exemplo, na figura temos um lugar chamado **NextDestination**, uma transição chamada **goNext** e um arco com o rótulo **nextAgency** ligando estes dois elementos. Cada arco pode ligar uma transição a um lugar ou vice-versa, mas nunca duas transições

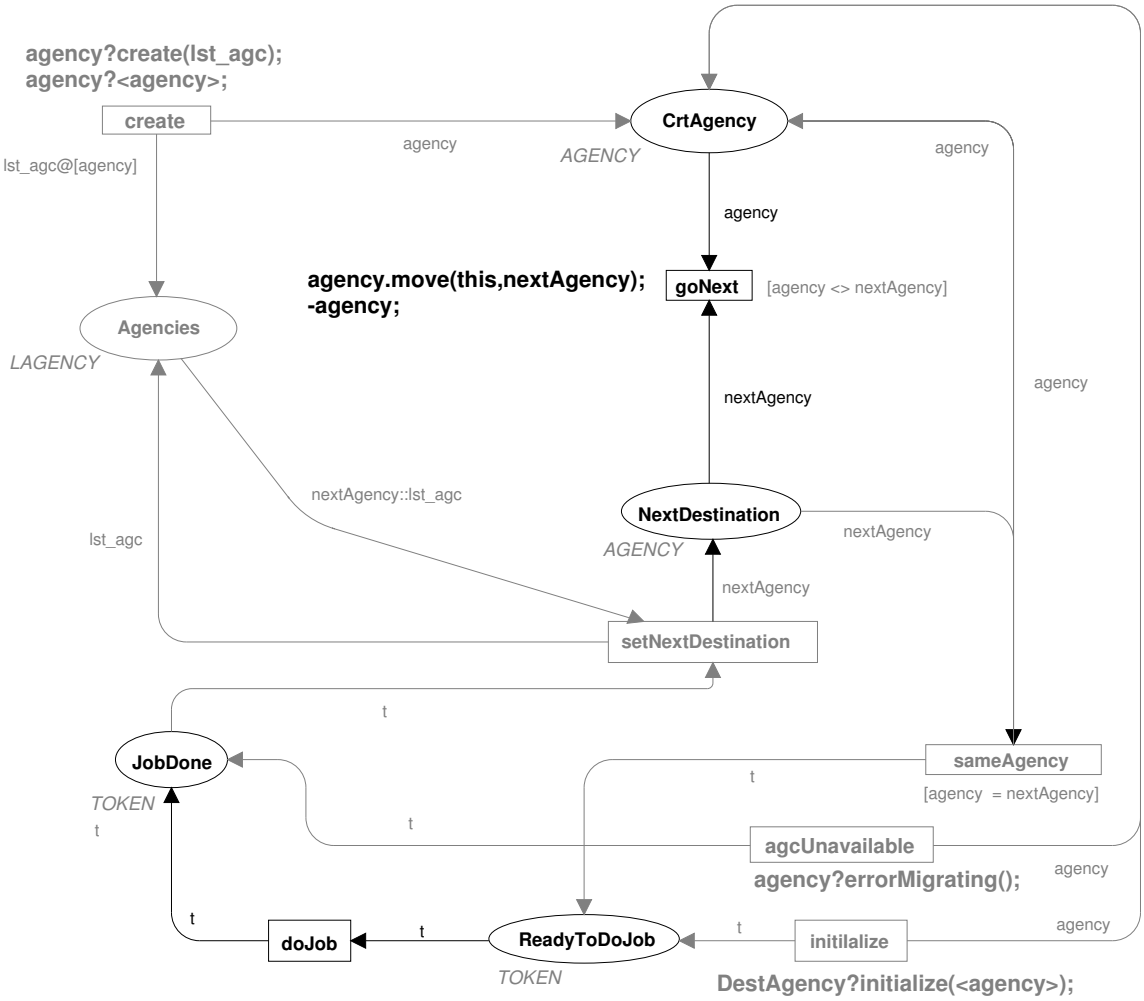

Figura 3.3: Rede de Petri da Classe ItineraryAgent

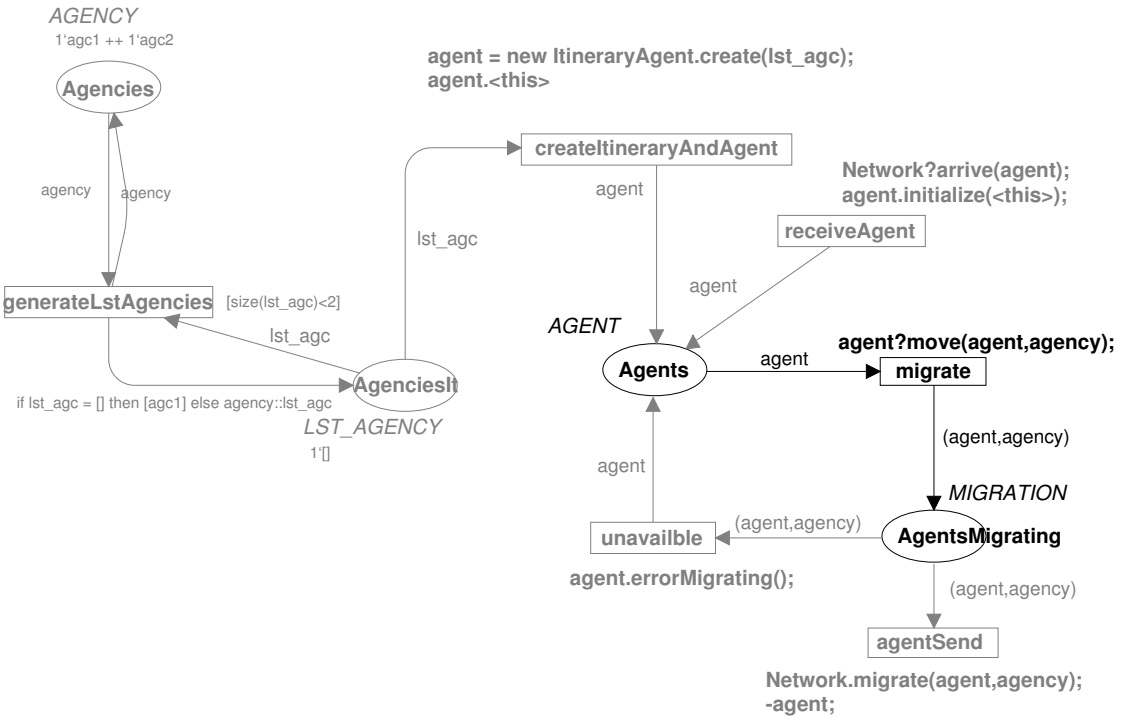

Figura 3.4: Rede de Petri da Classe Agency

ou dois lugares. Os lugares armazenam fichas (representações de dados) que devem ser da mesma cor (tipo) dos lugares. Na figura, o lugar **NextDestination** possui a cor *AGENCY* significando que este lugar poderá armazenar fichas deste tipo. As fichas são adicionadas e retiradas dos lugares de acordo com os arcos e em decorrência do disparo das transições, que representam as ações do sistema modelado. Como pode ser visto na figura, com o disaro da transição **goNext**, uma ficha será retirada do lugar **NextDestination** e outra de **CrtAgency**. Cada lugar possui, além de um nome e da cor, uma marcação inicial informando as fichas contidas no lugar inicialmente (e.g., o lugar **JobDone** possui uma ficha "t" como marcação inicial). As transições possuem um nome e uma guarda que condicionará o disparo da mesma (e.g., a transição **goNext** possui a guarda *"[agency <> nextAgency]"*). Os arcos são rotulados com uma expressão que define as fichas a serem retiradas ou adicionadas a determinados locais.

A transição **goNext** possui apenas arcos de entrada, que são arcos que retiram fichas de lugares. Já a transição **doJob** posui um arco de entrada, que retira fichas do lugar **ReadyToDoJob**, e um arco de saída, que adiciona fichas ao lugar **JobDone**. Os arcos de entrada (resp. de saída) de uma transição são aqueles que têm origem em um lugar (resp. transição) e destino na transição (resp. lugar). Os arcos de entrada, em conjunto com as guardas, compõem as pré-condições para os disparos das transições, pois condicionam os mesmos, bem como as pós-condições de tais disparos, pois, ao retirarem fichas dos lugares, alteram o estado da rede. Já os arcos de saída compõem apenas as pós-condições para os disparos, já que alteram o estado da rede ao adicionarem fichas aos lugares.

Mais de uma transição pode estar habilitada para o disparo em um mesmo instante, de forma que

Ação	**Descrição**
objeto = new Classe()	Criação de Objeto
objeto = clone objeto()	Clone de Objeto
objeto?mensagem()	Consumo de mensagem
objeto.mensagem	Envio de mensagem assíncrona
objeto!mensagem	Envio de mensagem síncrona
-objeto	Desligamento de objetos
end	Auto-destruição

Tabela 3.1: Lista de Ações RPOO

qualquer uma possa ser escolhida (não determinismo) para disparar. Quando uma transição dispara, as fichas que satisfazem as expressões contidas nos arcos de entrada serão retiradas dos respectivos lugares de entrada e outras fichas (é possível que sejam as mesmas) que satisfazem as expressões contidas nos arcos de saídas serão adicionadas aos respectivos lugares de saída.

Como exemplo, podemos ver a transição *doJob* da Figura 3.3. Ao ser disparada, seu arco de entrada retira uma ficha *t* do lugar *ReadyToDoJob* e seu arco de saída coloca uma outra ficha *t* no lugar *JobDone*. Essa transição abstrai a execução da tarefa para qual o agente itinerário é responsável por fazer. Como pode ser visto, quando há uma ficha no lugar *ReadyToDoJob*, isto significa que o agente está pronto para executar sua tarefa, ou seja, ele chegou a uma agência onde a tarefa deverá ser executada. Isto habilita o disparo (execução da tarefa) da transição *doJob*, que ao ser disparada, coloca uma ficha no lugar *JobDone*, informando que a tarefa foi executada.

Como foi dito, os sistemas modelados em RPOO são compostos por objetos que possuem seu comportamento descrito por redes de Petri. A forma com que tal comportamento altera a estrutura dos modelos se dá através de inscrições nas transições, que determinam ações que devem ser executadas quando do disparo de uma determinada transição. Tais ações RPOO (vide Tabela 3.1) alteram o sistema de objetos de um determinado modelo criando ou destruindo objetos, enviando ou consumindo mensagens, etc. Na rede de Petri apresentada na Figura 3.3 temos a transição *goNext*. Tal transição possui duas ações RPOO (encontradas à esquerda do desenho da transição) que, quando for disparada, deverão ser executadas. Uma é o envio de uma mensagem *move* para o objeto armazenado na variável *agency* com os parâmetros *this* (referência do objeto) e *nextAgency* (referência para outra agência), indicada na expressão "*agency.move(this,nextAgency);*". E outra é o desligamento deste objeto com o que está referenciado em *agency* (*-agency;*).

Desta forma, as ações RPOO são responsáveis pelas interações entre os objetos na perspectiva orientada a objetos do modelo. Na Figura 3.5, temos uma possível configuração dos objetos descritos pelo padrão *Itinerary*, onde cada objeto é representado por um círculo cinza e as ligações entre os objetos por setas. Nela, temos dois objetos da classe *Agency* (*sourceAgency* e *agency1*) e um da classe *ItineraryAgent* (*agent*). A ligação entre os objetos *agent* e *sourceAgency* significa que o agente

está localizado em sua agência de origem, ou seja, o agente conhece a agência onde está e a agência conhece o agente que está nela (indicado pela seta bi-direcional). Após o disparo da transição *goNext* localizada na rede da classe *ItineraryAgent* (Figura 3.3), a configuração do sistema passará a ser a apresentada na Figura 3.6. Nela, podemos ver que há uma mensagem pendente originada do objeto *agent* e destinada ao objeto *sourceAgency* chamada *move(agent,nextAgency)* que foi originada da ação RPOO *agency.move(this,nextAgency)* (veja que, na perspectiva OO, os parâmetros não mais são variáveis e sim valores de referências). Além dessa mudança, a Figura 3.6 nos mostra que não há mais uma ligação do objeto *agent* para o objeto *sourceAgency* que foi ocasionada pela ação RPOO -*agency* (o inverso permanece, pois o objeto *sourceAgency* ainda possui referência para o objeto *agent*).

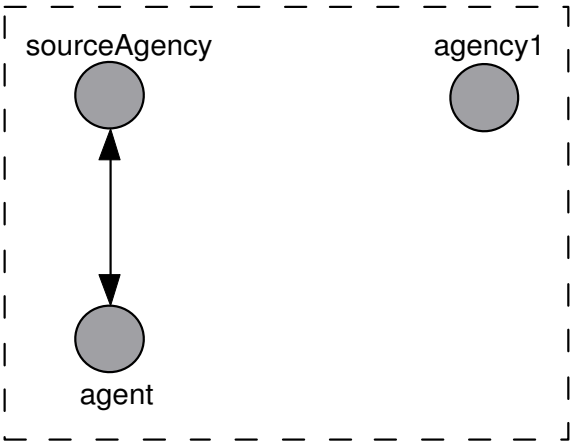

Figura 3.5: Exemplo de Configuração para o modelo do *Itinerary*

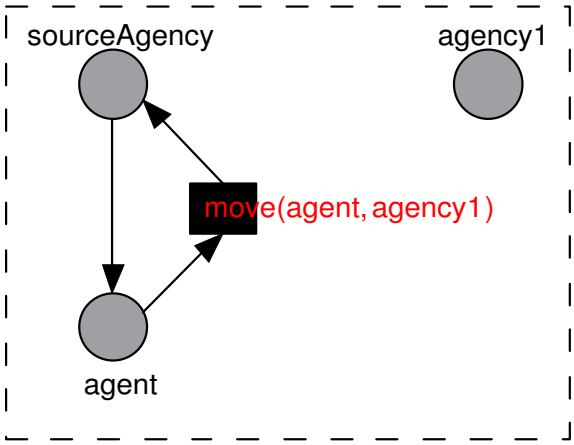

Figura 3.6: Configuração para o modelo do *Itinerary* após o disparo da transição *goNext*

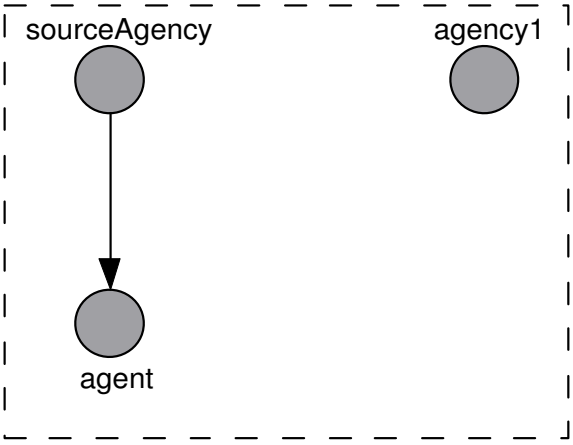

Figura 3.7: Configuração para o modelo do padrão *Itinerary* após o disparo da transição *migrate*

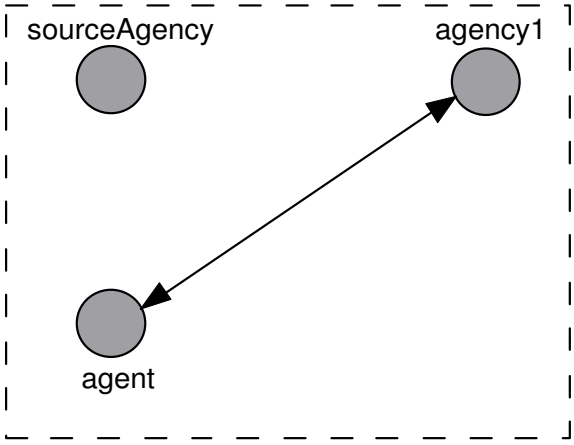

Figura 3.8: Configuração para o modelo do *Itinerary* após a migração do agente para *agency1*

Após o disparo da transição *goNext*, a transição *migrate* da rede de Petri da classe *Agency* (Figura 3.4) poderá disparar. E quando isto ocorrer, além de retirar uma ficha do lugar *Agents* e colocar uma outra ficha no lugar *AgentsMigrating*, devido à sua inscrição, a ação RPOO

agent?move(agent,agency) resultará no consumo da mensagem que estava pendente resultando na configuração mostrada na Figura 3.7. Após o disparo de outras transições e da execução de outras ações RPOO, a configuração do sistema será a apresentada na Figura 3.8 que, como será mostrado a seguir, caracterizará a migração total do agente entre as agências ***sourceAgency*** e ***agency1***.

3.3.2 Modelagem de Agentes Móveis com RPOO

Na Seção 3.3.1, apresentamos a linguagem de especificação RPOO através de um exemplo de um padrão de projeto para Agentes Móveis. Nesta seção, mostraremos como a mobilidade dos agentes é modelada em RPOO. Para tal, utilizaremos o mesmo exemplo contido na Seção 3.3.1, mostrando apenas os detalhes da modelagem da mobilidade.

Os agentes móveis são processos que estão localizados em agências e que podem se mover entre estas agências com o intuito de prover interações locais com demais agentes. Desta forma, é preciso que os modelos apresentem os conceitos de localização dos agentes (nas agências) e que esta localização possa ser alterada dinamicamente, o que será chamado de mobilidade dos agentes.

Haja visto que os modelos a serem trabalhados pelo método de geração de casos de teste estão bem mais próximos da implementação dos sistemas do que de um modelo conceitual, ou seja, possuem certas características de implementação, algumas entidades presentes na implementação estarão presentes no modelo. Sendo assim, as entidades agência e agente estarão presentes e serão modelados como objetos do sistema. O conceito de localidade é modelado por ligações entre os objetos agência e agente, ou seja, caso haja uma ligação entre um objeto agente e um objeto agência isso significará que tal agente está localizado (executando) em tal agência. Neste sentido, a mobilidade é obtida através da criação e remoção de ligações entre estes objetos.

Para ilustrar o que acabamos de dizer, na Figura 3.5 temos um sistema com duas agências (*sourceAgency* e *agency1*) e um agente (*agent*). Devido à ligação existente entre o objeto *agent* e o objeto *sourceAgency*, podemos dizer que o agente modelado por *agent* está localizado na agência modelada por *sourceAgency*. Desta forma, a mobilidade está modelada através dos procedimentos que fazem com que o objeto *agent* não esteja mais ligado ao objeto *sourceAgency* e passe a estar ligado ao objeto *agency1*. As Figuras 3.6 e 3.7 nos mostra parte deste procedimento, que resultará na configuração apresentada na Figura 3.8.

Como foi explicado na Seção 3.3.1, o comportamento dos objetos do modelo está definido pelas redes de Petri. Desta forma, o procedimento necessário para a migração dos agentes também estará. Uma vez que estamos tratando com modelos próximos da implementação, o processo de migração se dará através do envio de uma mensagem de solicitação de migração por parte do agente para a agência em que está localizado, e esta última será responsável por realizar a migração. Esta é a forma implementada pela maioria das plataformas de suporte à execução de Agentes Móveis.

As transições *goNext*, *agcUnavailable* e *initialize* da rede de Petri que descreve a classe *ItineraryAgent* (Figura 3.3) são responsáveis por realizar a solicitação de migração por parte do agente. Caso a agência para onde o agente deseja migrar seja diferente da agência onde ele se encontra

(guarda da transição *goNext*), a transição *goNext* estará habilitada e, ao ser disparada, tal solicitação será enviada (mensagem *move*). A partir deste ponto, o agente ficará aguardando a confirmação de que chegou à agência almejada, através do recebimento da mensagem *initialize*. No entanto, é possível que a agência de destino não esteja disponível naquele momento e, ao invés de uma mensagem *initialize*, o agente receberá uma mensagem *errorMigrating*. Com o recebimento da mensagem *initialize*, a transição *initialize* estará habilitada e, por outro lado, com o recebimento da mensagem *errorMigrating*, a transição *agcUnavailable* estará habilitada.

Com relação à agência onde os agentes estarão executando, sua rede de Petri deve possuir todo procedimento necessário para enviá-lo a uma nova agência, assim como ocorre com as plataformas de execução. De acordo com a rede de Petri da classe *Agency* (Figura 3.4), as transições *migrate*, *agentSend*, *unavailable* e *receiveAgent* são responsáveis pelo envio e recebimento dos agentes entre as agências. Ao receber uma mensagem *move* de um agente, a transição *migrate* é disparada, e isto coloca o agente que deseja migrar em uma lista de agentes a migrar. O disparo das transições *agentSend* e *unavailable* modelam o envio de um agente a uma agência e a falha deste mesmo envio respectivamente. Ao ser disparada a transição *agentSend* a mensagem *migrate* é enviada à rede (objeto *Network*) contendo o agente que está migrando e a agência para onde o agente deseja migrar. Por outro lado, ao ser disparada, a transição *unavailable* envia uma mensagem *errorMigrating* ao agente informando que a agência não está disponível. Por fim, a agência recebe o agente através da mensagem *arrive*, o que habilita a transição *receiveAgent*. Ao disparar a transição *receiveAgent* uma mensagem *initialize* é enviada ao agente informando-o de sua chegada à agência.

Outros serviços como clonagem e comunicação entre agentes podem ser providos pelas agências, dependendo de cada plataforma de execução. No entanto, por se tratar de um exemplo simples e como o principal foco do nosso trabalho é a mobilidade dos agentes, não apresentamos a modelagem destes serviços. Contudo, o Capítulo 5 apresenta um exemplo mais completo onde a modelagem de clonagem e comunicação entre agentes pode ser encontrada.

3.3.3 Justificativa

Alguns formalismos foram analisados e, devido aos fatores que iremos apresentar a seguir, RPOO foi selecionado em detrimento a tais linguagens de especificação e modelagem. Além de RPOO, as seguintes linguagens foram analisadas: Redes de Petri Coloridas [Jen92]; OhCircus [CSW03]; IF [BFG$^+$99] e π-Calculus [Mil99]. As linguagens foram analisadas comparativamente de acordo com os fatores que seguem.

1. **Conhecimento prévio dos autores com o formalismo:** Ter conhecimento prévio com o formalismo permitiria que o autor concentrasse esforços em questões específicas a Agentes Móveis, e não precisasse se ater a questões de linguagens;

2. **Ferramenta de geração de LTS:** As principais ferramentas de geração de casos de teste recebem, como entrada, o sistema especificado em LTS, gerando a necessidade de se existir uma

	1	**2**	**3**	**4**
RPOO	χ	χ	χ	χ
CPN	χ	χ		χ
OhCircus				
IF		χ	χ	
π-Calculus				χ

Tabela 3.2: Tabela Comparativa entre as Linguagens Analisadas

ferramenta que forneça este artefato automaticamente a partir do modelo.

3. **Orientação a Objetos:** Uma vez que os trabalhos de modelagem de SBAM em geral utilizam-se da orientação a objetos (artefatos UML), seria interessante que a linguagem utilizada pelo método possuisse suporte à orientação a objetos, haja visto que isto facilitaria a adaptação do método por parte dos desenvolvedores de SBAM.

4. **Histórico com modelagem de SBAM:** É interessante que a linguagem selecionada já tenha sido utilizada na modelagem de SBAM, uma vez que isto facilitaria na análise sobre seu potêncial em modelar Agentes Móveis e na análise sobre as consequências dessa mesma utilização no processo de geração de casos de teste como um todo.

A Tabela 3.2 apresenta os resultados da análise comparativa entre as linguagens onde as linguagens estão dispostas nas linhas e os fatores analisados nas colunas.

Além de RPOO ter se destacado na análise feita, outros pontos que serão mostrados a seguir nos levaram a escolhe-la como linguagem a ser utilizada pelo método.

- **Suporte do grupo de pesquisa local:** Técnicas e teorias sobre RPOO foram e estão sendo desenvolvidas dentro do grupo de pesquisa local, o que possibilitou um maior suporte técnico e teórico, o que dificilmente haveria com outras linguagens;

- **Experiência do autor em modelar Agentes Móveis com tal formalismo:** Alguns trabalhos com RPOO e redes de Petri colorida com Agentes Móveis foram desenvolvidos no grupo de pesquisa local [LFG04];

- **Consolidação do formalismo de RPOO:** O trabalho sendo desenvolvido com a utilização de RPOO é interessante para o grupo de pesquisa local, já que um de seus principais objetivos é a consolidação deste formalismo, sendo, inclusive, meta de um projeto maior (MOBILE[1]) do grupo de pesquisa local;

- **Viável Conversão de Modelos a partir de UML-RT:** A conversão de modelos em UML-RT para RPOO é mais viável que com outros formalismos (ambas têm perspectiva orientada

[1]Projeto financiado pelo CNPq - processo 552190/2002-0.

a objetos e possuem definições comportamentais baseadas em transições de estados) e, com o suporte do grupo de pesquisa local, foi possível apresentar uma transformação, mesmo que ainda informal, de modelos em UML-RT para RPOO (vide Capítulo 4).

Apesar das vantagens apresentadas acima, é importante citar algumas limitações oriundas da escolha de tal formalismo. Por ser uma linguagem ainda recente, RPOO não possui um suporte ferramental bem consolidado, principalmente no que diz respeito a ferramentas de suporte à modelagem de sistemas. A modelagem de sistemas com RPOO é feita através da utilização de ferramentas específicas para sistemas orientados a objetos (e.g. Gentleware Poseidon for UML[2]) e outras específicas para redes de Petri (e.g. DesignCPN[3]). Desta forma, a estrutura dos sistemas é modelada através de diagramas UML (diagrama de classes, de seqüência, etc.) com uma ferramenta e, utilizando-se de uma segunda ferramenta, as redes de Petri que descrevem o comportamento das classes são modeladas.

Um primeiro problema que vemos com esta abordagem está relacionado com a manutenção dos modelos. Durante o processo de elaboração do modelo, é possível que a integridade do modelo como um todo (diagramas UML e redes de Petri) não seja mantida dada as diversas alterações que são realizadas no mesmo. Por exemplo, durante a construção de uma rede de Petri, é possível que seja identificada a necessidade de se adicionar o envio de uma nova mensagem de um determinado objeto para outro objeto. Esta alteração deverá resultar em uma alteração tanto na rede de Petri como no modelo estrutural do sistema através, por exemplo, da adição de um novo método a uma determinada classe. Caso apenas um dos modelos seja alterado, o modelo como um todo estará inconsistente.

Com o advento de uma ferramenta de modelagem específica para RPOO este problema seria amenizado pois poderíamos ter verificadores capazes de identificar tais inconsistências. Além disso, tal advento amenizaria um segundo problema que está relacionado à utilização da ferramenta de geração de espaços de estado (vide Seção 3.4.1). Tal ferramenta recebe o modelo RPOO através de classes Java desenvolvidas com base em sua API e que representam o modelo do sistema. Atualmente isto precisa ser gerado manualmente, mas poderia ser gerado automaticamente pela ferramenta de modelagem, eliminando a possibilidade de erros durante esta geração.

Além do problema relacionado ao suporte ferramental, por ser uma linguagem recente, RPOO ainda não possui um amplo uso na comunidade, e a adaptação ao uso do método por parte dos desenvolvedores de sistemas poderia ser dificultada. No entanto, os resultados que serão apresentados no Capítulo 4 visam atenuar tal problema.

3.4 Geração de Sistema de Transição Rotulado (Espaço de Estados)

A maioria das ferramentas de geração e seleção automática de casos de teste utilizam LTS (Labelled Transition System) como formalismo de entrada, e como foi dito, a ferramenta TGV também assim

[2]http://www.gentleware.com/

[3]http://www.daimi.au.dk/designCPN/man/

o faz. Portanto, faz-se necessário que haja uma técnica/ferramenta para a geração de LTS a partir de modelos RPOO.

A técnica para tal transformação pode ser encontrada no trabalho de Guerrero [Gue02b] e a única ferramenta que a implementa é a JMobile. A seguir, daremos uma breve descrição desta ferramenta.

3.4.1 JMobile Tools

JMobile Tools [Sil05] tem como objetivo principal dar suporte a ferramentas de simulação de modelos RPOO e de geração de espaços de estados, disponibilizando em sua API classes e métodos responsáveis por prover acessos às informações dos modelos e pela simulação destes modelos. Junto com esta API, os autores disponibilizam um protótipo de ferramenta que é capaz de realizar a simulação de modelos RPOO e a geração de seu respectivo espaço de estados.

O protótipo do gerador do espaço de estados, ferramenta escolhida para compor o método, recebe o modelo RPOO como entrada e gera o seu respectivo espaço de estados em dois tipos de formatos. Um utilizado pelo verificador de modelos Veritas [Rod04; RBGF04] e outro utilizado pela ferramenta Aldebaran [Fer89]. Este último formato será utilizado no nosso método, já que a ferramenta de geração de casos de teste TGV recebe suas entradas em tal formato. O algoritmo utilizado pela ferramenta para a geração do espaço de estados é o de busca por profundidade (*depth first search* - DFS). Ou seja, o algoritmo gera as configurações a partir da execução de eventos sobre as configurações geradas mais recentemente.

Para JMobile, um estado de um modelo RPOO, que a partir de agora será chamado de configuração, é formado pelos estados de cada rede de Petri[4] relativa a cada objeto em conjunto com as ligações existentes entre cada objeto e as mensagens pendentes no sistema de objetos. Por exemplo, como foi mostrado na Seção 3.3.1, a Figura 3.5, em conjunto com os estados das redes de Petri dos objetos *agent*, *sourceAgency* e *agency1*, representa uma configuração do sistema. Já a Figura 3.6 representa uma outra configuração pois, agora, podemos encontrar uma mensagem pendente no sistema e não há mais uma referência do objeto *agent* para o objeto *sourceAgency*.

As transições que fazem os sistemas mudarem de estado são rotuladas por eventos do sistema. Estes eventos são formados por ações RPOO que são executadas em decorrência do disparo de uma ou mais transições das redes de Petri. Considerando o mesmo exemplo apresentado anteriormente, o evento que fez com que o sistema passasse da configuração da Figura 3.5 para a da Figura 3.6 foi ocasionado pelo disparo da transição *goNext* contida na rede de Petri da classe *ItineraryAgent* (vide Figura 3.3). Desta forma, o rótulo deste evento será o nome da transição que o gerou, no caso *goNext*.

Modelos de Entrada e de Saída

Os modelos em RPOO podem ser carregados na ferramenta de duas maneiras. Na primeira delas, uma aplicação é disponibilizada e esta recebe os modelos descritos em arquivos XML. A segunda maneira

[4]Um estado de uma rede de Petri é formado pelas quantidades e tipos de fichas contidas em cada lugar da rede.

de realizar a geração de espaço de estados é através do uso da API que é fornecida. Nesta maneira, os modelos são instanciados em memória através de classes da API e passados como parâmetro para um conjunto de classes que são responsáveis por realizar a geração. Maiores detalhes a respeito deste modelos de entrada podem ser encontrados em [Sil05].

Como dissemos, a ferramenta de geração de casos de teste irá utilizar a saída do gerador de espaço de estados no formato Aldebaran. O arquivo com o formato Aldebaran para espaço de estados é um arquivo de texto com extensão **.aut**. Neste formato, os estados são abstraídos e representados apenas por identificadores (números inteiros). A primeira linha do arquivo deve ser uma linha contendo uma descrição do arquivo, no formato que segue:

des (<primeiro-estado>, <número-de-transições>, <número-de-estados>)

Cada linha restante do arquivo representa um arco no espaço de estados. Estas linhas têm a seguinte estrutura:

(<estado-origem>, <evento>, <estado-destino>)

Onde <estado-origem> e <estado-destino> são números representando o estado de origem e o estado de destino, respectivamente, e <evento> é o rótulo do arco referente. A Figura 3.9 apresenta um exemplo deste arquivo.

```
des (0, 5, 5)
(0, "agentForm.fina    lizarRevisao()", 1)
(1, "gui.showTelaAp    rovar()", 2)
(2, "agentForm.apro    varRevisao()", 3)
(3, "agc.move('agcC    oord')", 4)
(3, "gui.showTelaAp    rovar()", 2)
```

Figura 3.9: Exemplo de um Arquivo de Espaço de Estados em Formato Aldebaran

3.4.2 Exemplo

A Figura 3.10 apresenta um trecho do espaço de estados gerado para o modelo do padrão *Itinerary* com o uso de JMobile. As transições tracejadas representam transições que partem de um estado, ou se destinam a um estado, que não está presente na figura, e o estado inicial é o estado **'0'**. Os estados **'5'** e **'22'** são estados finais, em que, estando em um desses estados, o sistema não evolui mais. Para facilitar a visualização do espaço de estados, além de apresentarmos apenas um trecho do mesmo, os rótulos das transições são formados pelo nome da transição da rede de Petri que gerou a transição no grafo, ao invés de usar as ações RPOO como rótulos.

O detalhamento dos estados, que informa a configuração do sistema em um determinado estado, não está descrito pois, para o método, não é necessário. Como está apresentado na Seção 3.5, apenas

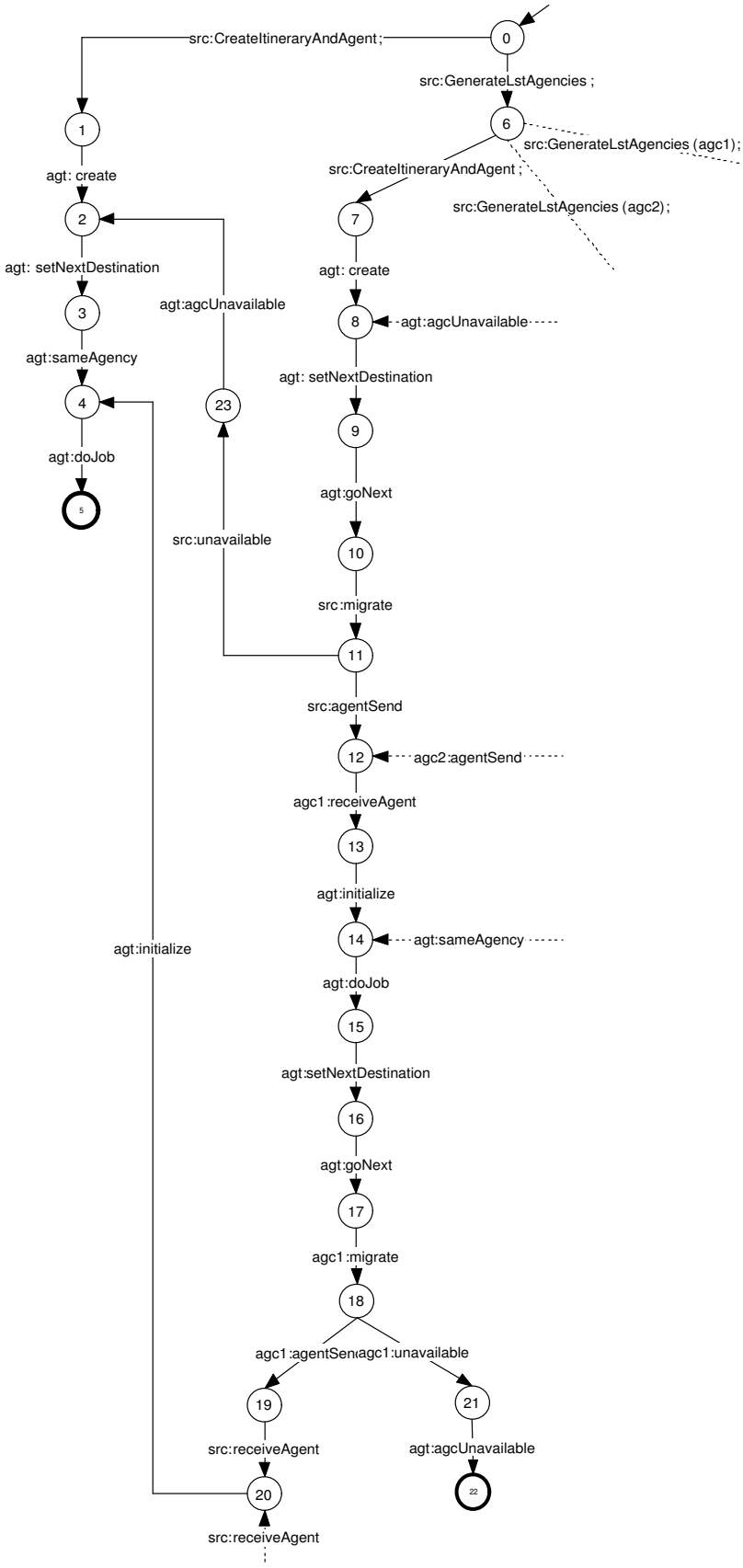

Figura 3.10: Trecho do Espaço de Estados para o Padrão *Itinerary*

o sequenciamento de ações é levado em conta pela ferramenta de geração de casos de teste. No entanto, é importante dizer que a ferramenta JMobile gera também as informações dos estados, que são interessantes para outras atividades de teste, como por exemplo, a geração dos oráculos.

3.4.3 Justificativa

Como foi dito anteriormente, JMobile Tools é a única ferramenta existente para a geração de espaço de estados a partir de modelos RPOO. A única alternativa à sua utilização seria a realização de conversão de modelos RPOO para modelos equivalentes em redes de Petri coloridas e, a partir destas, seria feita a geração do LTS com a utilização de ferramentas como Design-CPN [KCJ98]. Guerrero apresenta em seu trabalho [Gue02b] um algoritmo para esta conversão, porém não há ferramentas que o implemente.

Mesmo sendo a única ferramenta existente, é importante analisarmos como questões de mobilidade são tratadas pela ferramenta. Como mobilidade é modelada como troca de mensagens entre objetos bem como suas ligações, a mobilidade será refletida no respectivo LTS (espaço de estados) já que troca de mensagens e ligações entre objetos refletem eventos e conseqüentemente mudança de configurações no LTS. Desta forma, podemos dizer que a mobilidade dos agentes é tratada pelo gerador de espaço de estados e será passível de análise pela ferramenta de seleção de casos de teste.

3.5 Seleção dos Casos de Teste (TGV)

Uma vez de posse do LTS referente ao sistema a ser testado, este deverá servir de entrada para a ferramenta de seleção de casos de teste TGV, junto com um objetivo de teste, que será responsável por guiar a seleção dos casos de teste.

TGV é uma ferramenta de suporte a teste funcional (*caixa-preta*), baseada em técnicas de verificação de modelos, que provê a geração automática de casos de teste funcionais para sistemas reativos e não determinísticos. Os algoritmos de geração implementados por TGV são baseados nos utilizados pela verificação de modelos, onde o objetivo é selecionar execuções do sistema que satisfaçam uma determinada propriedade (objetivo de teste). Ao final, os casos de teste selecionados são apresentados em LTS e, com o uso de outras ferramentas, poderão ser convertidos para outros formatos como *Message Sequence Charts* (MSC), por exemplo.

3.5.1 Arquitetura Funcional de TGV

A Figura 3.11 nos mostra a arquitetura funcional de TGV, apresentando suas etapas para a geração de casos de teste. Como pode ser visto na figura, o primeiro passo é a execução de um produto síncrono entre a especificação do sistema e o objetivo de teste, o que irá resultar em um terceiro IOLTS. Este produto é realizado com o intuito de extrair da especificação apenas as linhas de execução descritas no objetivo de teste.

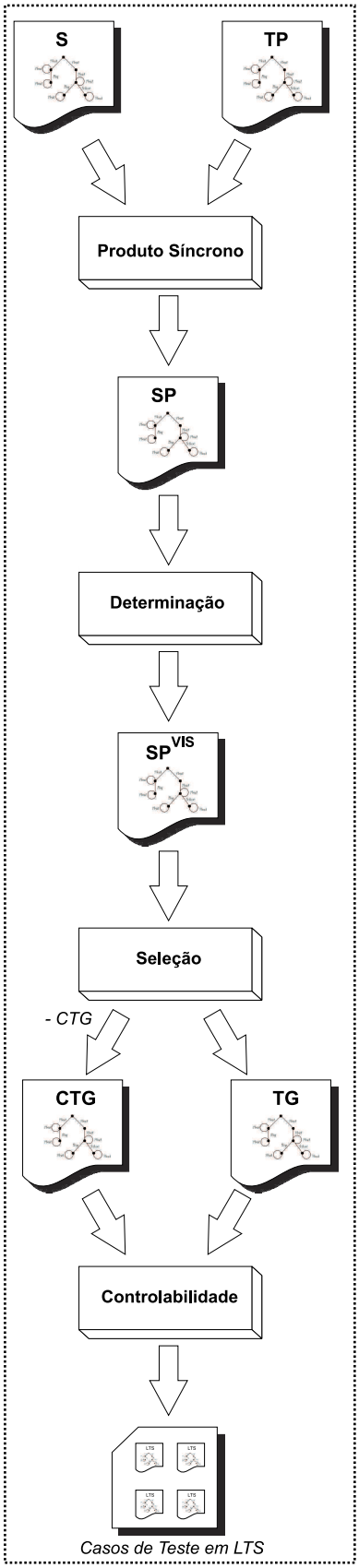

Figura 3.11: Arquitetura Funcional de TGV

O próximo passo consiste em tornar determinístico o IOLTS gerado na etapa anterior. Isto irá gerar um outro IOLTS contendo apenas os comportamentos visíveis da especificação, ou seja, sem as ações internas. Para tal, a ferramenta identifica os pontos do grafo onde o sistema pode entrar em estado de quiescência, decorrente de uma linha de execução que contenha apenas ações internas. E, ao identificar, uma nova transição de saída rotulada δ é colocada como origem e destino neste estado quiescente.

Após isto ser feito, a ferramenta constrói um grafo chamado CTG (complete test graph) que representa o grafo de teste completo especificado pelo objetivo de teste. Além de possuir apenas os caminhos aceitos pelo objetivo de teste, o CTG possui as entradas e saídas invertidas com relação à especificação do sistema. Esta inversão decorre do fato de que este CTG modela um possível programa que será responsável por testar o sistema modelado com a especificação inicial. Desta forma, as saídas do CTG serão entradas para a especificação e suas entradas serão saídas da especificação.

Ao final do processo, os conflitos de controlabilidade são solucionados e os casos de teste são gerados. Conflitos de controlabilidade ocorrem quando, de um determinado estado, partem várias transições de saída ou uma de saída e várias de entrada. Para isto, a ferramenta ou escolhe uma das transições de saída ou retira a transição de saída e deixa as de entrada. Isto faz parte do processo de seleção de casos de teste implementado pela ferramenta. Isto é feito de forma não-determinísta, ou seja, a cada seleção dos casos de teste, um caso de teste diferente pode ser selecionado.

3.5.2 Exemplo

Seguindo o exemplo que estamos usando neste capítulo, a paritr do LTS gerado por JMobile (Figura 3.10) para o padrão *Itinerary*, construímos o objetivo de teste apresentado na Figura 3.12 e fornecemos ambos como entrada para TGV. Após o processo de geração de casos de teste realizado por TGV, que envolve a criação do CTG, um conjunto de casos de teste é gerado. Dentre estes, ilustramos o caso de teste apresentado na Figura 3.13.

O objetivo de teste que apresentamos visa gerar casos de teste para as linhas de execução em que o agente itinerário é recebido em uma agência e, logo após, ele realiza a sua tarefa nesta agência. Atente para o fato de que as linhas de execução em que o agente não consegue migrar para uma determinada agência de seu itinerário não serão selecionadas pelo objetivo de teste. Isto ocorre devido à existência da transição *"*:unavailable"*.

Ao final da execução de TGV com os modelos apresentados, o caso de teste da Figura 3.13 é gerado. Ele apresenta uma linha de execução que, partindo do estado inicial do sistema (estado **'0'**), alcança um estado final e de aceitação **'7'** passando por uma seqüência de ações que condiz com o objetivo de teste. Note que o caso de teste, assim como o objetivo de teste contém apenas ações de entrada e de saída do sistema. Para que isto seja possível, além de entrar com a especificação do sistema e com o objetivo de teste, TGV requer que seja entrado um arquivo contendo a descrição das ações de entrada, saída e ações internas. A Figura 3.14 apresenta o conteúdo deste arquivo para o exemplo que estamos apresentando. As entradas são descritas na primeira parte do arquivo e, na

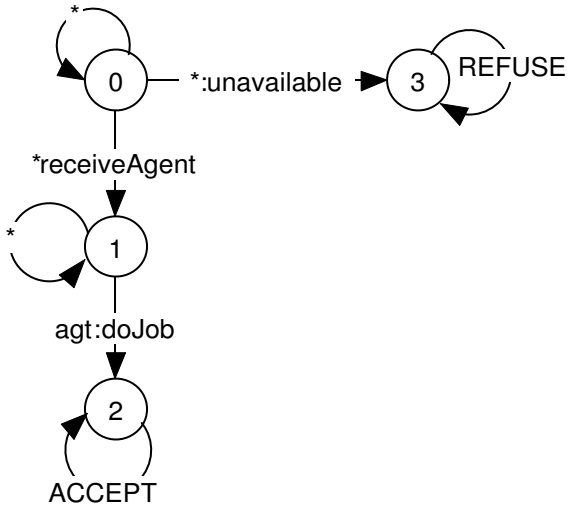

Figura 3.12: Objetivo de Teste para o Padrão *Itinerary*

segunda parte, as saídas. É assumido que as ações que não estão descritas no arquivo são ações internas.

3.5.3 Justificativa

A escolha de TGV se deu com base em um estudo comparativo com outras ferramentas: TORX [BFdV+99]; STG [CJRZ02]; AutoFocus [WLPS00]; e AutoLink [KGHS98]. A seguir mostramos os principais fatores que nos levaram a selecionar TGV como ferramenta para a geração dos casos de teste.

- TGV têm sido amplamente utilizada para validação de sistemas distribuídos devido ao seu potencial em tratar características de concorrência e não-determinismo de tais sistemas;
- TGV recebe como entrada a especificação e o objetivo de teste em formato LTS, não sendo uma ferramenta específica a nenhuma linguagem mais abstrata;
- TGV permite a geração de casos de teste de maneira *on-the-fly*, ou seja, os casos de teste podem ser gerados à medida em que os grafos são tratados, o que ameniza um possível problema de explosão do espaço de estados.
- Por realizar uma geração de casos de teste baseada em propriedades (objetivos de teste), TGV permite que os testadores de sistemas possam especificar propriedades específicas de mobilidade de código e, conseqüentemente, gerar casos de teste específicos para tal. Isto seria muito difícil de ser conseguido com outro tipo de ferramenta que realiza a geração de casos de teste para propriedades pré-definidas ou baseadas em critérios de cobertura;
- TGV tem sido utilizada com sucesso sobre sistemas com espaço de estados de 500.000 estados e 900.000 transições, e não apenas aplicações simples [JJ02];

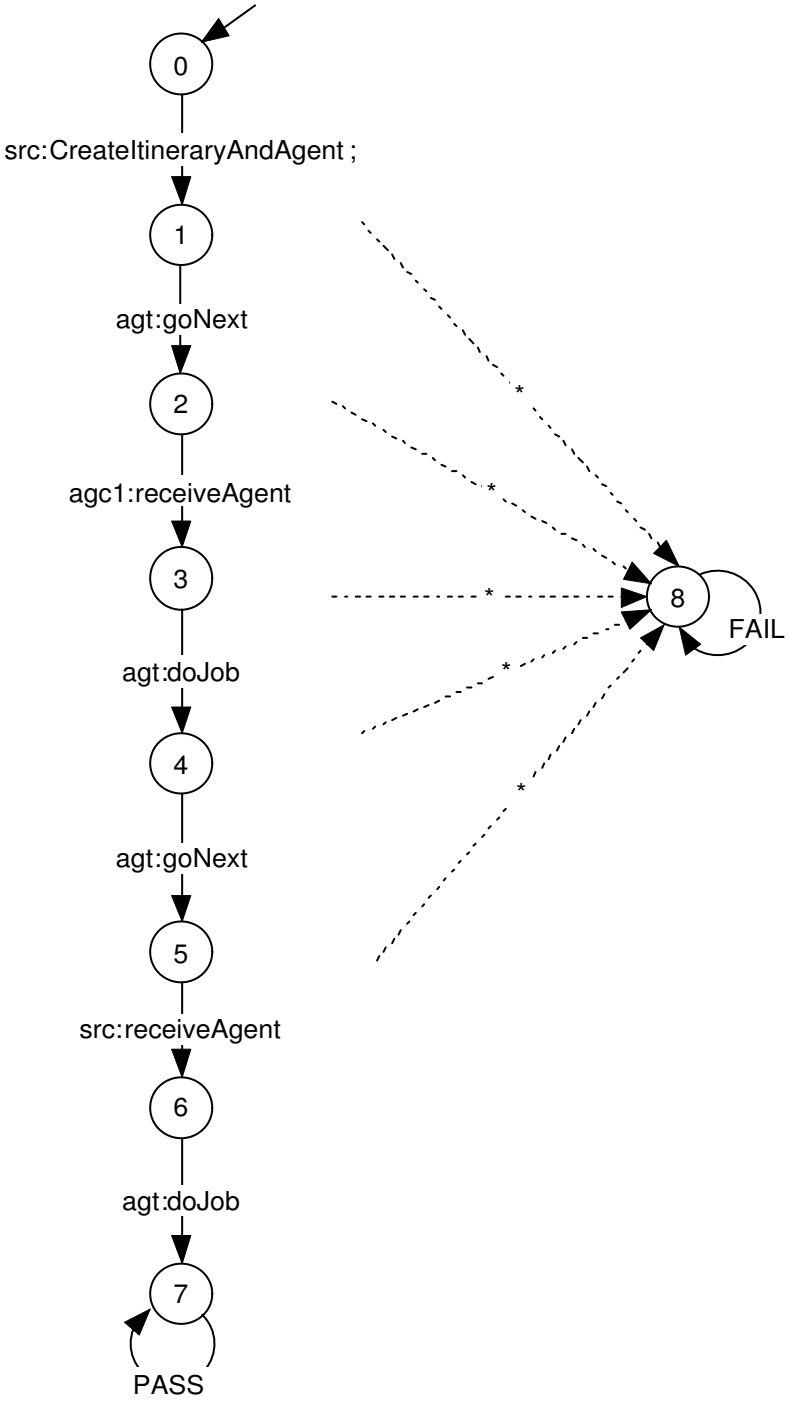

Figura 3.13: Caso de Teste para o Padrão *Itinerary*

```
input
*createItinerary    AndAgent
*receiveAgent
*unavailable
output
*goNext
*doJob
```

Figura 3.14: Conteúdo do Arquivo que Contém Entradas e Saídas para o Padrão *Itinerary*

- Em comparação com outras ferramentas de geração de casos de teste baseadas em técnicas de verificação de modelos, TGV se destaca pois é uma ferramenta construída sobre a teoria de teste [Tre99] e não utiliza uma outra ferramenta de verificação de modelos para a geração de casos de teste, o que torna seus algoritmos mais eficientes.

3.6 Conclusão

Como foi dito, a definição de um método de geração automática de casos de teste consiste, principalmente, na escolha de ferramentas, técnicas e linguagens que melhor se adequam para um tipo de sistema específico - no caso, sistemas baseados em Agentes Móveis. No entanto, podemos dizer que o principal resultado deste trabalho esta na identificação de um método para tal, considerando a eminente carência de tais métodos na área. Ou seja, o tamanho deste trabalho, em conjunto com imaturidade dos processos de teste para SBAM, não nos permite dizer que estamos propondo o melhor método e sim um método de geração automática de casos de teste para SBAM.

O método consiste de uma estratégia para a geração automática de casos de teste a partir de modelos RPOO de um SBAM. Desta forma, a aplicação do método está condicionada à existência de modelos RPOO do sistema a ser testado. Além disso, é necessário que o testador de *software* tenha conhecimento do sistema e de sua modelagem em RPOO, uma vez que a obtenção dos objetivos de teste está condicionada a tal conhecimento.

A seguir, temos alguns pontos fortes do método que devem ser ressaltados:

- Para todas as etapas do método, podemos encontrar uma ferramenta que a realize, o que viabiliza a utilização do método;
- O incentivo ao uso de padrões de projeto e padrões de teste pode trazer conseqüências interessantes ao projeto do software, como por exemplo reusabilidade;
- Cada componente do método (ferramentas e linguagens) pode ser substituída por outra bastando que, para isso, as interfaces sejam preservadas.

Porém, alguns pontos negativos também precisam ser citados:

- Apesar da proposta contida no Capítulo 4, a utilização de RPOO como linguagem de especificação e modelagem pode dificultar um pouco a utilização do método em um primeiro instante, já que RPOO não é muito conhecida pela comunidade de desenvolvimento de software;
- A ferramenta utilizada para a geração do LTS respectivo ao sistema a ser testado ainda é um protótipo e, portanto, é passível de falhas;
- A utilização de objetivos de teste pode não ser simples para sistemas complexos e questões de critérios de cobertura ainda deixam a desejar.

3.6.1 Contextualização em Processos de Desenvolvimento

Como pode ser visto no Capítulo 5, o método proposto neste trabalho foi aplicado a um estudo de caso de forma contextualizada em um processo de desenvolvimento [GMM03]. No entanto, essa escolha se deu apenas para que nosso estudo de caso fosse aplicado em um processo de desenvolvimento e para que desse uma demonstração de como isto poderia acontecer. Na verdade, esperamos que o método possa ser aplicado a quaisquer processos de desenvolvimento, bastando que, para isso, alguns requisitos sejam contemplados.

- É necessário que haja especificações completas e corretas do sistema, ou pelo menos, das partes que se deseja gerar os casos de teste;
- Tais modelos do sistema precisam estar escritos em RPOO ou UML-RT (vide Capítulo 4);

Apesar de não caracterizar um requisito à utilização do método, o uso de padrões de projeto para Agentes Móveis é fortemente recomendado, já que podemos a partir deles: (1) extrair padrões de teste que poderão ser reaproveitados; (2) e utilizar os padrões de projeto como objetivos de teste. Maiores informações sobre este assunto pode ser encontrado no Capítulo 7.

Capítulo 4

Transformação de Modelos UML-RT para RPOO

4.1 Introdução

No Capítulo 3, apresentamos um método para a geração automática de casos de teste para SBAM a partir de modelos RPOO. Ao final do capítulo, mencionamos, como ponto negativo do método, o fato de RPOO não ser uma linguagem comumente utilizada pelos desenvolvedores de sistemas distribuídos. Visando amenizar este problema, apresentamos neste capítulo um resultado adicional ao nossa trabalho. Nesta proposta, os desenvolvedores de sistemas poderíam se aproveitar de modelos em uma linguagem mais próxima de sua realidade já existentes na construção dos modelos RPOO, facilitando a aplicação do método por parte dos desenvolvedores que não são experientes com RPOO. Os modelos já existentes nessa linguagem seriam, portanto, utilizados como fonte de informação na construção dos modelos RPOO, para então o método apresentado poder ser aplicado aos sistemas.

Além de ser comum aos desenvolvedores de sistemas distribuídos, esta linguagem precisa ser apropriada para a modelagem e especificação de Agentes Móveis. Neste contexto, destacamos UML-RT [SR98] pois: (1) é uma linguagem completamente baseada em UML; (2) apesar do nome *Real Time*, é bastante utilizada para a modelagem de sistemas distribuídos e concorrentes; (3) e que, após termos feito uma análise sobre nossas experiências com tal linguagem, se mostrou interessante para a modelagem de Agentes Móveis, como será mostrado mais adiante.

Uma vez que UML-RT não é uma linguagem com semântica formal, uma conversão completa só seria possível através de uma formalização de modelos desta linguagem para o formalismo de RPOO. Trabalhos neste sentido podem ser encontrados na comunidade, como por exemplo o apresentado por Sampaio et al [SMR03; RSM05]. Contudo, precisamos de uma formalização de modelos UML-RT sobre RPOO e, devido ao fato de não existirem técnicas nem ferramentas para tal, e dada a complexidade em realizar esta formalização, este passo não está no escopo do trabalho. O que será apresentado aqui, além do estudo acerca de modelagem de Agentes Móveis em UML-RT, será um mapeamento informal de modelos em UML-RT para modelos RPOO, que visa auxiliar a construção

dos modelos RPOO dada a existência de modelos UML-RT.

Com esta conversão, a Figura 3.1 que representa o método de geração de casos de teste proposto neste trabalho pode ser estendida e vista como na Figura 4.1. Nela, podemos observar que os modelos construídos utilizando-se a linguagem UML-RT são utilizados e, com o auxílio da transformação a ser apresentada a seguir, o engenheiro de *software* constrói os modelos RPOO, com o intuito de utilizá-los na aplicação do metodo.

A Seção 4.2 apresenta UML-RT, mostrando como mobilidade pode ser modelada com tal linguagem. Já a Seção 4.3 apresenta como estes modelos poderão ser utilizados para a construção dos modelos RPOO. E por fim, a Seção 4.4 traz nossas conclusões sobre o capítulo, apresentando algumas considerações adicionais.

4.2 UML para Sistemas de Tempo Real

UML-RT foi criada a partir da linguagem ROOM [SGW94] com o intuito de ser utilizada para modelar sistemas complexos e de tempo real [SR98]. Esta linguagem tem sido bastante utilizada para a modelagem de sistemas distribuídos devido, principalmente, à sua capacidade em modelar componentes autônomos e distribuídos. A proposta dos autores é a de utilizar apenas os dispositivos providos por UML, sem a adição de quaisquer outros mecanismos de modelagem. Para isso, foram utilizados os mecanismos de adaptação (*UML tailoring mechanisms*) providos por UML que são estereótipos, valores rotulados (*tagged values*) e restrições.

4.2.1 Modelagem com UML-RT

Os artefatos geralmente utilizados por UML-RT são diagramas de classe, que modelam a estrutura geral do sistema, e os diagramas de colaboração, que apresentam possíveis configurações do sistema. Além disso, para cada classe (mais adiante também chamada de Cápsula) um diagrama de estados é construído, o que descreverá seu comportamento e conseqüentemente o comportamento do sistema em geral.

UML-RT apresenta quatro novos construtores para modelagem estrutural: Cápsulas, Portas, Protocolos e Conectores. Cápsulas são entidades complexas, independentes, autônomas e possivelmente distribuídas de um sistema. As cápsulas são modeladas por classes com o estereótipo *«capsule»* e que se comunicam com o ambiente unicamente através de sinais de entrada e de saída. Uma hierarquia de cápsulas pode ser construída. É possível definir que uma cápsula é constituída de outras cápsulas, bem como estas últimas por outras cápsulas. Com uma hierarquia de cápsulas, podemos dizer que o comportamento de uma cápsula é definido pelo comportamento de outras cápsulas.

As cápsulas se comunicam entre si através de sinais. Os protocolos («protocol») definem como as cápsulas devem interagir entre si, definindo os sinais que poderão ser enviados e recebidos pelas cápsulas participantes de tais protocolos, bem como a seqüência permitida destes sinais. Os conectores (*«connector»*) são construtores definidos sobre associações de diagramas de colaboração que

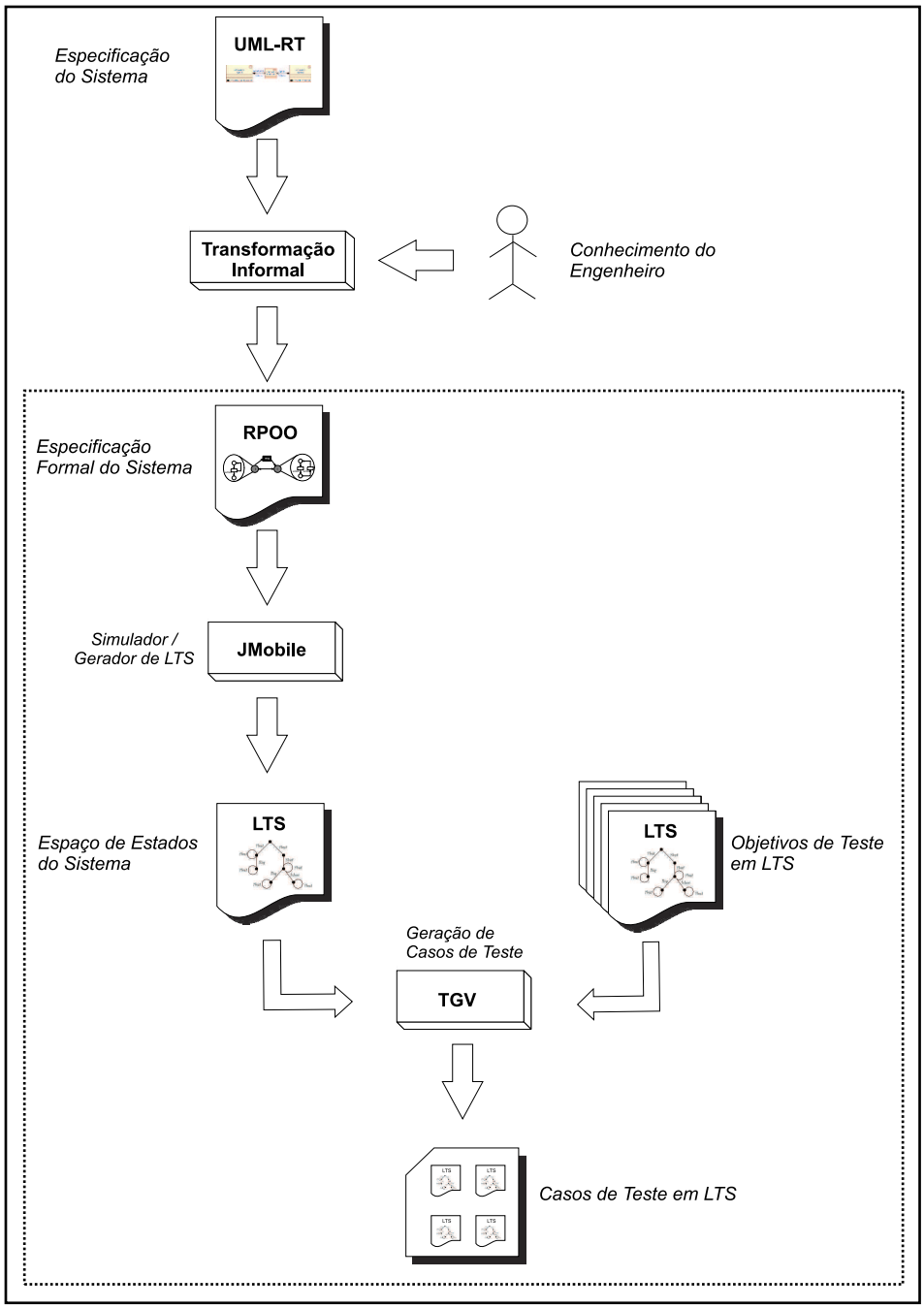

Figura 4.1: Visão Geral do Método de Geração Automática de Casos de Teste a partir de Modelos UML-RT

identificam quais cápsulas participam de quais protocolos. A comunicação entre uma cápsula e o ambiente (demais cápsulas) é feita através de portas (*«port»*). As portas de uma determinada cápsula compõem uma espécie de interface desta cápsula e são chamadas de objetos de fronteira (*boundary object*), pois podem ser vistas na fronteira das cápsulas e são responsáveis por realizar interações com as demais cápsulas. Os objetos que representarão as portas serão instâncias de classes cujo estereótipo é *«protocol»*.

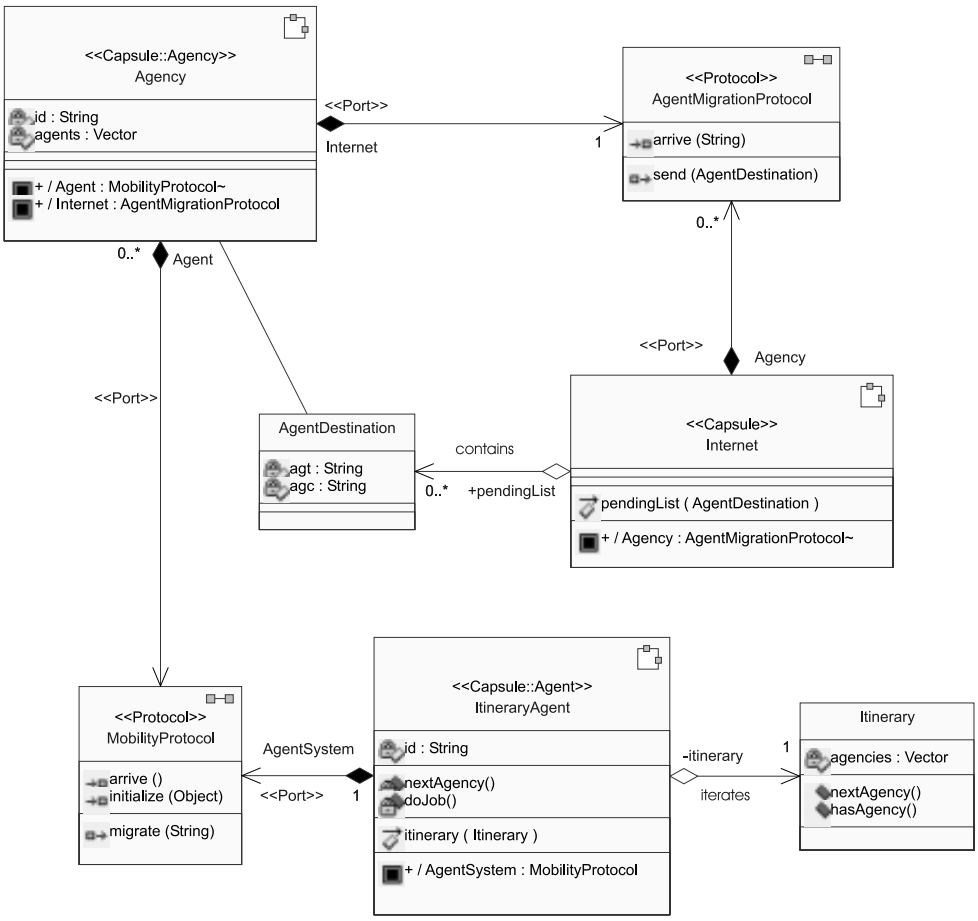

Figura 4.2: Diagrama de Classes para um Modelo UML-RT

A Figura 4.2 nos mostra um diagrama de classes de um modelo UML-RT para o padrão Itinerary (exemplo apresentado no Capítulo 3). Nela podemos encontrar as cápsulas *ItineraryAgent*, *Agency* e *Internet*; os protocolos *MobilityProtocol* e *AgentMigrationProtocol*; e as classes *AgentDestination* e *Itinerary*.

A cápsula *ItineraryAgent* modela o agente itinerário que irá migrar por uma lista de agências (objetos da cápsula *Agency*), a ser gerenciada pela classe *Itinerary*. A migração se dá através da cápsula *Internet*, que tem a função de abstrair a rede por onde os agentes migrarão para alcançar as agências. As entidades *ItineraryAgent*, *Agency* e *Internet* são cápsulas pois possuem execução independente e distribuída. A comunicação entre os objetos da cápsula *ItineraryAgent* e os da cápsula *Agency* será regida pelo protocolo *MobilityProtocol*, já a comunicação entre esses mesmos objetos da

cápsula *Agency* e a rede (objeto de *Internet*) será regida pelo protocolo *AgentMigrationProtocol*.

Assim como pode ser visto nos modelos em UML padrão, as classes (cápsulas) possuem atributos e métodos. Para a cápsula *ItineraryAgent*, um único atributo pode ser visto, *id:String*, e dois métodos podem ser encontrados, *nextAgency()* e *doJob()*. Além disso, um terceiro compartimento localizado logo abaixo dos atributos e dos métodos pode ser encontrado nas cápsulas. Este compartimento apresenta as portas desta referida cápsula, listando os nomes das portas e os seus respectivos protocolos. Na cápsula *ItineraryAgent*, uma porta pode ser encontrada: *AgentSystem* do tipo *MobilityProtocol*.

Os protocolos possuem dois compartimentos: um para os sinais de entrada e outro para os sinais de saída. Por exemplo, o protocolo *MobilityProtocol* possui como sinais de entrada os sinais *arrive()* e *initialize(Object)*, e como de saída o sinal *migrate()*. A definição de entrada e saída é relativa à porta do protocolo chamada base. Ou seja, para cada protocolo binário (duas cápsulas envolvidas), uma das portas será chamada de base e a outra de conjugada (identificada com um "∼"). Os sinais ditos de saída (resp. entrada) têm esse nome pois partem da porta base (resp. conjugada) em direção à porta conjugada (resp. base). Ainda no exemplo, podemos ver que para o protocolo *MobilityProtocol*, a porta *AgentSystem* da cápsula *ItineraryAgent* é a porta base e a porta *Agent* da cápsula *Agency* é a respectiva porta conjugada. Para os protocolos que não sejam binários, para cada sinal, deveremos claramente especificar de onde eles partem bem como para onde eles são destinados. Contudo, os protocolos mais utilizados pelos desenvolvedores que utilizam UML-RT são os binários.

A fim de definirmos o comportamento das cápsulas, um diagramas de estados é construído para cada uma delas. Além disso, os protocolos também podem ter seus comportamentos definidos através de diagramas de estados. A Figura 4.3 apresenta o diagramas de estados para a cápsula *ItineraryAgent*. A semântica dos diagramas de estado é o mesmo provido por UML padrão. Como vemos na referida figura, após o objeto da cápsula ser criado, ele entra no estado *WaitingInitialize*. Neste estado, quando o sinal *initialize* chega pela porta *AgentSystem*, a transição *Initialization* será disparada executando a ação *itinerary = (Itinerary) getMsgData();*. Este disparo fará com que o objeto passe do estado *WaitingInitialize* para o estado *Running*. Com o objeto no estado *Running*, o ponto de escolha *ThereIsAgc* verificará se ainda existem agências para serem visitadas e, caso seja verdade, a transição *Migrating* será disparada, caso contrário, o objeto irá para o estado *STOP*, finalizando a execução, como pode ser visto na Figura 4.4.

4.2.2 Mobilidade com UML-RT

A mobilidade dos agentes é representada em UML-RT de forma semelhante a RPOO, através de ligações e desligamentos entre objetos. Da mesma forma, o conceito de localidade também é o mesmo, onde é dito que um agente está localizado em uma determinada agência se a sua porta referente à comunicação com a agência está conectada à agência. Neste sentido, a mobilidade dos agentes se dá quando um agente se desconecta de uma agência e se conecta a uma outra agência, significando que este agente migrou da primeira para a segunda agência.

Usaremos, a seguir, diagramas de colaboração do exemplo trabalhado anteriormente para ilustrar

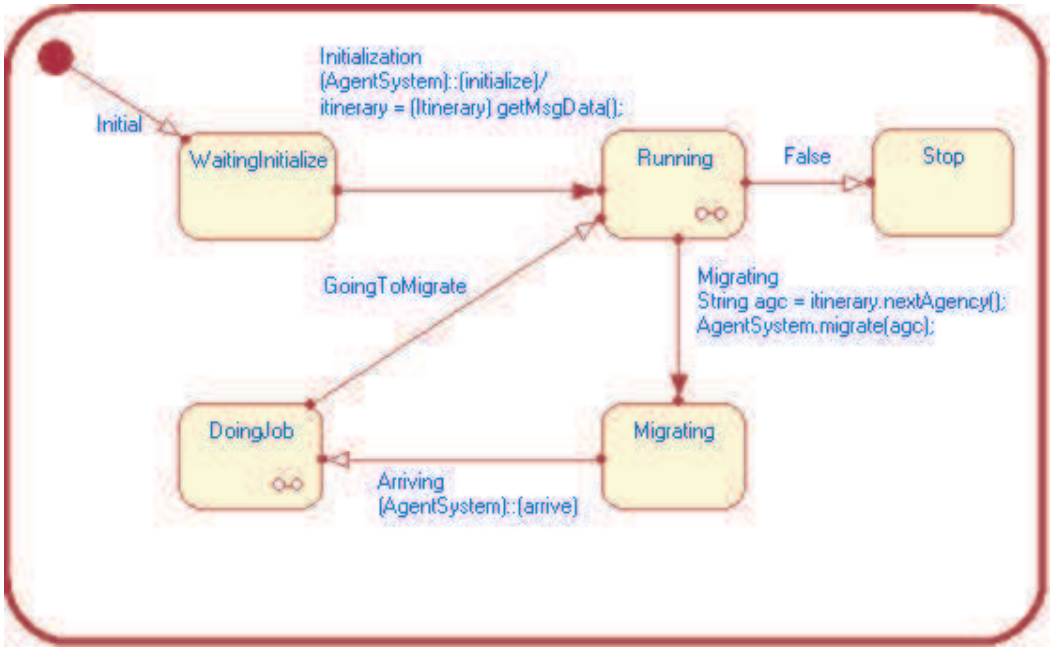

Figura 4.3: Diagrama de Estados para a Classe ItineraryAgent

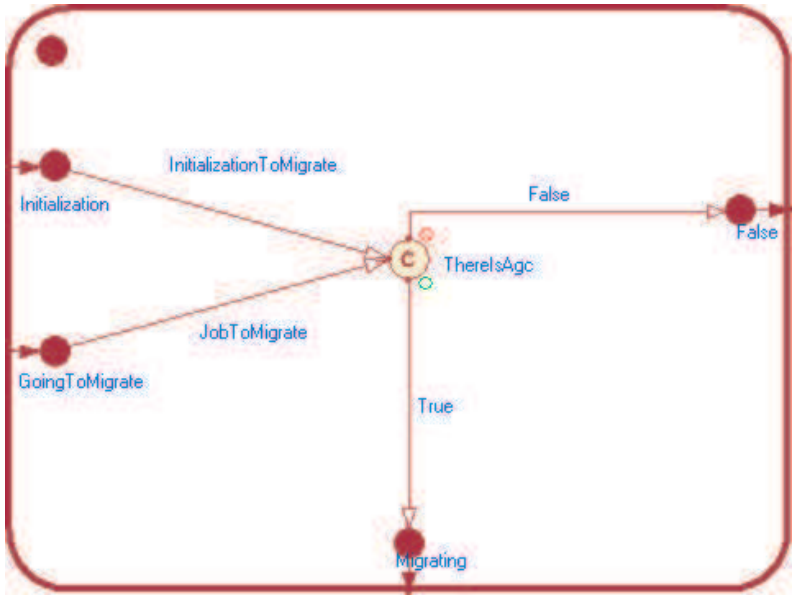

Figura 4.4: Detalhamento do Estado *Running*

o que dissemos. O diagrama apresentado na Figura 4.6 nos mostra uma possível configuração inicial do sistema modelado. Nela, podemos encontrar um objeto chamado *ItinAgt* que representa o agente itinerário, duas agências chamadas *source* e *agc1*, e um objeto que representa a rede, chamado *internet*. Como pode ser visto, as agências estão ligadas à rede e o agente, inicialmente, está localizado na agência *source*.

Em um determinado momento, o agente envia um sinal *migrate()* à agência contendo o identifi-

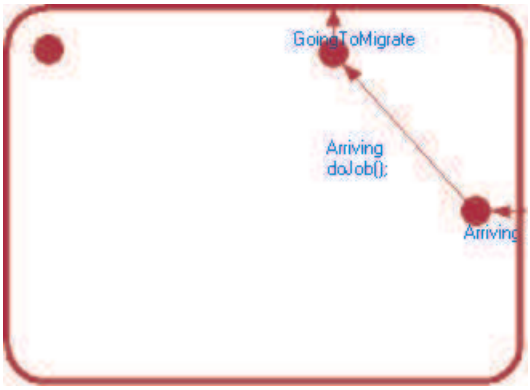

Figura 4.5: Detalhamento do Estado *DoingJob*

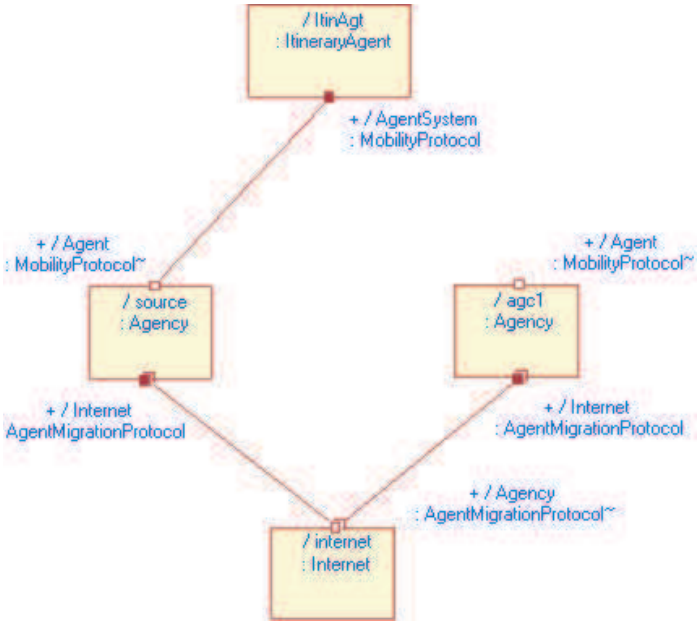

Figura 4.6: Diagrama de Colaboração - Configuração Inicial

cador da agência *agc1*. Sendo assim, a agência se desliga deste agente e envia a responsabilidade de enviá-lo para a agência *agc1* à rede (objeto *internet*), deixando o sistema com a configuração apresentada na Figura 4.7. Como pode ser visto, o agente não está localizado em agência alguma e, por questões de simplicidade (não há comunicação entre a rede e os agentes durante a migração), também não está ligado à rede.

Após isto, a rede envia o agente à agência *agc1* que cria uma ligação com o mesmo, gerando a configuração apresentada na Figura 4.8. Com esta configuração, o agente não mais está localizado na agência *source* e poderá realizar interações locais na agência *agc1*, estando apto a realizar sua tarefa.

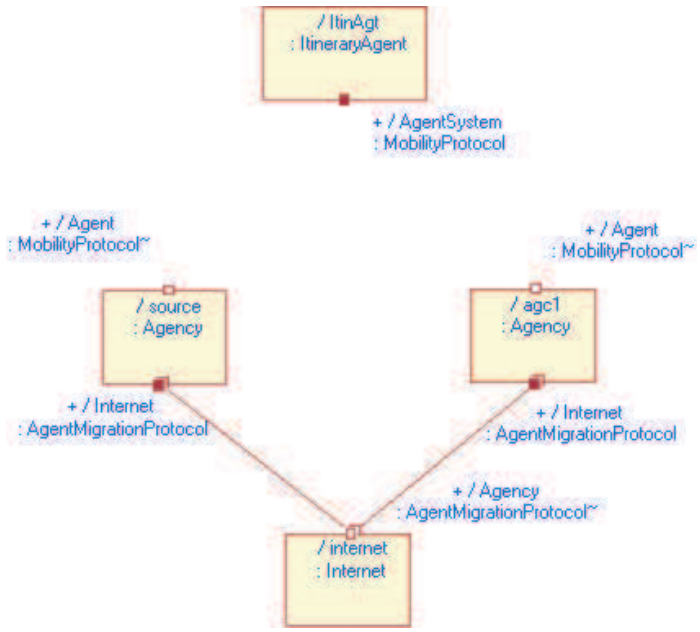

Figura 4.7: Diagrama de Colaboração - ItineraryAgent Migrando

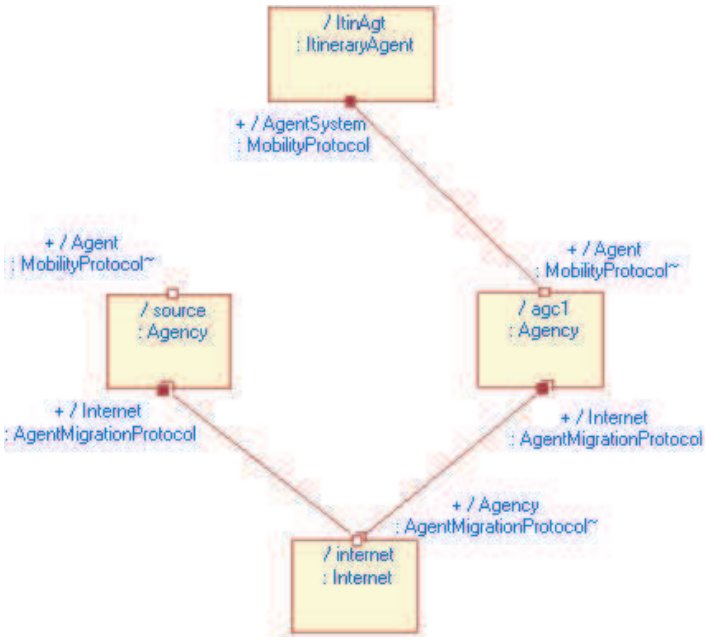

Figura 4.8: Diagrama de Colaboração - ItineraryAgent após Migração

4.3 Construindo Modelos RPOO a partir de Modelos em UML-RT

4.3.1 Mapeamento Informal

Tanto UML-RT como RPOO possuem duas visões para seus modelos: a estrutura destes modelos é apresentada sobre uma perspectiva orientada a objetos (OO), através de classes, objetos, associações e etc.; e o seu comportamento é dado pela descrição específica do comportamento de cada objeto,

através de diagramas de estados para UML-RT e redes de Petri coloridas para RPOO (linguagens com semântica baseada em grafos de transição).

Desta forma, a transformação também será feita sobre duas perspectivas. Primeiro na visão estrutural e depois na visão comportamental. A estrutura dos modelos permanecerá a mesma, ou seja, será descrita pelas mesmas classes e seus relacionamentos. Já a transformação da visão comportamental basicamente mostrará a conversão dos diagramas de estados que descrevem os objetos de UML-RT para as redes de Petri coloridas que descreverão estes mesmos objetos em RPOO.

Visão Estrutural - OO

Haja visto que os modelos estruturados, de ambas as linguagens, são vistos sobre uma perspectiva orientada a objetos, o mesmo modelo OO de UML-RT será o de RPOO. Os objetos em RPOO também são independentes, autônomos e possivelmente distribuídos [Gue02b], portanto, as cápsulas e os protocolos de UML-RT poderão ser classes RPOO sem que seus objetos deixem de ter semântica de entidades independentes, autônomas e possivelmente distribuídas.

Os estereótipos *«capsule»*, *«connector»*, *«protocol»* e *«port»* podem continuar existindo, pois facilitariam o entendimento do modelo em RPOO. No entanto, é importante notar que esses construtores servem para facilitar o entendimento do modelo com relação à transformação feita, portanto não têm relação com o domínio do problema, como por exemplo *«Agent»* e *«Agency»* (vide [KRSW01]).

Seguindo o que foi apresentado acima, o diagrama de classes UML-RT apresentado na Figura 4.2 poderá ser transformado no diagrama de classes para RPOO apresentado na Figura 4.9.

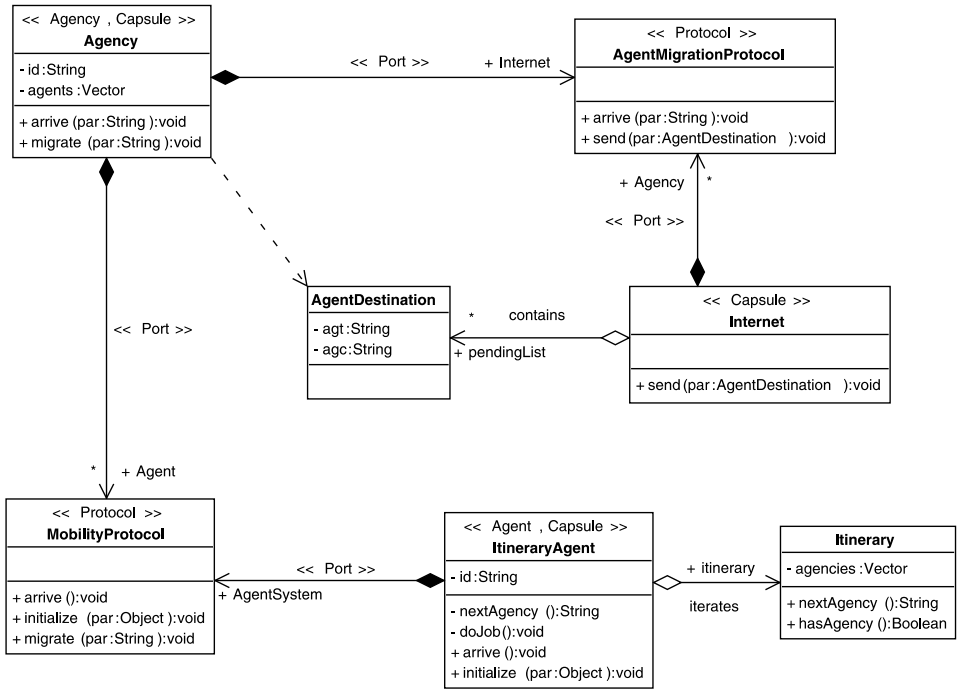

Figura 4.9: Diagrama de Classes para RPOO obtido de UML-RT

Note que um método referente a cada sinal de entrada foi acrescentado às classes com estereótipo

«capsule». Isto é feito como forma de os objetos referentes a estas classes poderem receber os sinais de entrada através de suas portas, o que em UML-RT é implícito.

A seguir temos os passos para a obtenção do modelo OO de RPOO a partir do mesmo tipo de modelo em UML-RT.

1. As cápsulas, protocolos e demais classes de UML-RT serão classes no modelo RPOO.

2. Os relacionamentos entre as cápsulas, protocolos e demais classes de UML-RT serão os mesmos no modelo RPOO, inclusive o relacionamento de composição existente entre cápsulas e protocolos, cujo estereótipo é *«port»*.

3. Para cada sinal de entrada (resp. de saída) de uma porta base (resp. conjugada), um método com mesmo nome e parâmetros serão criados na respectiva cápsula.

Em UML-RT, qualquer comunicação entre as cápsulas e o ambiente deve ser feita unicamente através de suas portas. Quando os modelos são traduzidos para RPOO, esta restrição não mais pode ser vista nos modelos, ou seja, é possível que um objeto de uma cápsula consiga uma referência para outro e este envie mensagens diretamente sem que as suas portas sejam utilizadas. No entanto, como esses modelos em RPOO advêm de modelos em UML-RT, este tipo de comportamento não esperado não estará presente nos modelos, porém, é importante atentar para isto pois poderá alterar a característica independente das cápsulas.

Uma outra consideração importante é a relação entre as classes estáticas de UML-RT com classes de RPOO, que são dinâmicas por definição. Apesar de todo objeto RPOO ser independente e possuir seu comportamento dinâmico descrito por uma rede de Petri, classes estáticas de UML-RT também podem ser modeladas em RPOO como classes, bastando que para isso a rede de Petri que a descreva contenha apenas o comportamento de seus métodos de recuperação e atualização dos atributos (métodos *getters* e *setters*).

Visão Comportamental (Diagramas de Estados -> Rede de Petri Colorida)

Tendo um diagrama de classes que descreve a estrutura do modelo, o próximo passo é construir as redes de Petri que irão descrever o comportamento dos objetos do sistema. O comportamento de cada uma dessas redes será completamente baseado no comportamento do respectivo diagrama de estados no modelo em UML-RT. Desta forma, a seguir apresentaremos os passos necessários para a conversão dos diagramas de estado em CPN. O diagrama de estados apresentado nas Figuras 4.3, 4.4 e 4.5 serão utilizados para ilustrar esses passos onde, ao final da explanação, obteremos a rede de Petri colorida que descreve o comportamento da classe *ItineraryAgent*.

A seguir, temos os passos para a transformação dos diagramas de estado de modelos UML-RT para redes de Petri coloridas de RPOO. Esta transformação é baseada nos estudos de Peterson [Pet92], onde ele apresenta a equivalência entre diagramas de estados e redes de Petri através da conversão

de modelos em diagramas de estados para redes de Petri. Os passos estão enumerados, porém essa ordem possui apenas um caráter didático e não tem relação com a correção da transformação.

1. Para cada estado do diagrama de estados, existirá um lugar com o mesmo nome e cuja cor será "*E*", significando que, se há uma ficha "*e*" no lugar, o objeto referente se encontra no respectivo estado.

2. Para cada atributo da classe teremos um lugar cuja cor será a classe do atributo.

3. Para cada transição no diagrama de estados, existirá uma transição na rede respectiva, com um arco que retira uma ficha ("*e*") do lugar que representa o estado anterior e coloca uma ficha ("*e*") no lugar que representa o estado posterior da respectiva transição no diagrama de estados.

4. Os gatilhos das transições no diagrama de estados serão convertidos para inscrições RPOO de recebimento de mensagens (*obj?mensagem(par)*) nas respectivas transições da rede de Petri, onde: o objeto da porta será o objeto do qual se espera a mensagem (***obj****?mensagem(par)*); o sinal será o nome da mensagem (*obj?****mensagem****(par)*); e os dados dos sinais serão parâmetros da mensagem (*obj?mensagem(****par****)*).

5. As ações decorrentes das transições nos diagramas de estados são tratadas da forma que segue.

   5.1 Para cada ação de envio de sinal por uma determinada porta, uma inscrição de envio de mensagem assíncrona para o objeto relativo à porta será criada. Além disso, a seqüência de mensagens necessária para a mensagem chegar até o objeto deverá ser adicionada. É importante lembrar que em UML-RT quando se envia um sinal a uma porta, a forma com que este sinal chegará ao objeto final é implícita. Já em RPOO, é preciso, além de enviar um sinal à porta, fazer com que esta porta envie também um sinal à porta do objeto destinatário, e que este último envie um sinal ao objeto final.

   5.2 Ações de ligação e de desligamento serão transformadas em ações deste tipo entre as respectivas portas dos objetos em questão.

   5.3 Métodos privados que possuem seu comportamento abstraído, ou seja, existe uma chamada à sua execução, porém não existe a descrição do seu comportamento, devem ser transformados em transições nas redes de Petri, onde o disparo representará a execução. Em UML-RT, as chamadas a esses métodos geralmente são colocados como ações de transições, porém em RPOO, para uma melhor análise dos modelos, é melhor que as chamadas a esses métodos sejam abstraídos como disparo de transições.

   5.4 Sempre que um atributo tiver sua referência modificada, o seu respectivo lugar deverá ser atualizado.

   5.5 Os demais tipos de ação devem ser traduzidos de forma específica com a sua semântica, por exemplo:

(a) Chamada a métodos de classes será transformada em envio e recebimento de mensagens síncronas. E caso haja retorno de método, o procedimento necessário deverá ser criado.

6. Os seguintes passos descreverão como as guardas das transições dos diagramas de estados deverão ser tratadas nas redes de Petri:

   6.1 Para guardas que não envolvam chamadas a métodos de outras classes, ou envio de mensagens de forma geral, é possível que para cada guarda de uma transição do diagrama de estados, uma expressão seja adicionada à guarda da transição respectiva na rede de Petri.

   6.2 Contudo, o passo anterior não é possível para guardas mais complexas. A semântica de guardas em UML define que, caso um evento ocorra (chegada de um sinal), se a guarda não for satisfeita, além da transição não ocorrer, o sinal será perdido. Desta forma, uma maneira de modelar guardas mais complexas com RPOO é colocando os procedimentos (transições, lugares e inscrições) necessários para a averiguação de cada guarda antes da transição ocorrer.

7. Os pontos de escolha (*Choice Points* em Inglês) são modelados de forma semelhante às guardas:

   7.1 Caso a condição seja simples, basta acrescentar tal condição à guarda da transição e colocar um arco de saída referente ao valor *TRUE* do ponto de escolha. Além disso, acrescenta-se uma cópia da transição que difere desta unicamente por ter sua guarda invertida (e.g. *not(guarda)*) e ter seu arco de saída referente ao valor *FALSE* do ponto de escolha.

   7.2 Caso a condição envolva o envio de mensagens, o procedimento necessário à averiguação deverá ser colocado de modo que, ao final, a transição referente a *TRUE* seja executada caso o teste resulte em verdadeiro e refrente a *FALSE* caso contrário.

8. Caso tenhamos hierarquia de estados, teremos ainda sim um lugar para cada estado (inclusive os estados mais abstratos) e as transições deverão remover e adicionar fichas nesses estados. Note que, desta forma, é possível termos mais de um lugar com uma ficha "*e*", contanto que estes estados representem uma hierarquia de abstração.

A seguir, ilustraremos os passos apresentados através de um exemplo. Mostraremos como o diagrama de estados referente à cápsula *ItineraryAgent* (vide Figuras 4.3, 4.4 e 4.5) foi convertido em uma rede de Petri colorida seguindo os passos apresentados.

A Figura 4.10 mostra a aplicação dos passos 1, 2 e 3. Nela podemos ver que para cada estado e para cada atributo temos um lugar na rede de Petri que o representa. Além disso, vemos também que para cada transição do diagrama de estados temos uma transição na rede de Petri.

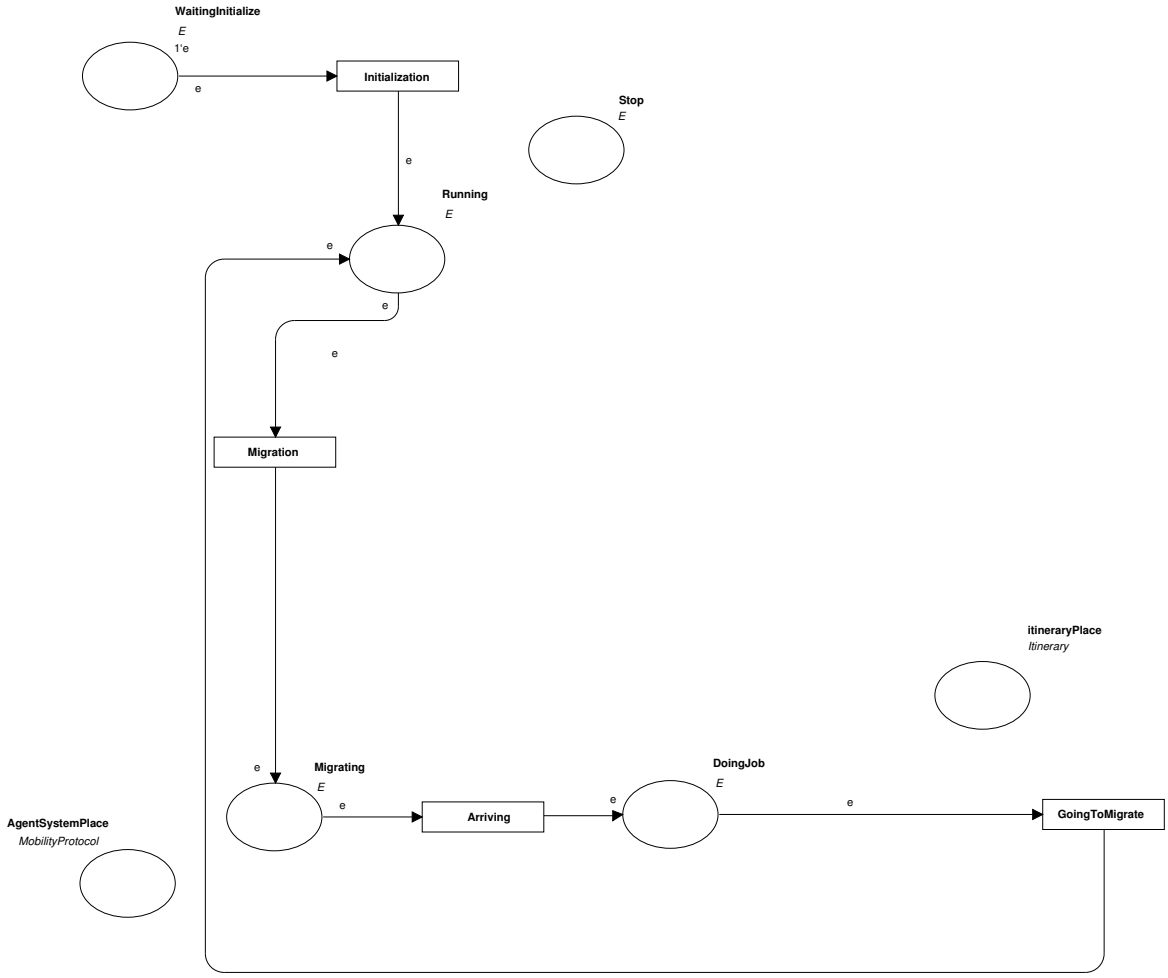

Figura 4.10: Transformação do Diagrama de Estados que Descreve a Classe *ItineraryAgent* para sua Respectiva Rede - Passo 1

O passo 4 está ilustrado na Figura 4.11. Cada gatilho de cada transição foi convertido para uma inscrição RPOO de consumo de mensagem. Por exemplo, a transição *Initialization* possui o gatilho "*(AgentSystem)::(Initialize)/*", portanto sua respectiva transição na rede de Petri possuirá a inscrição "*AgentSystem?Initialize(<itinerary>)*". O parâmetro <*itinerary*> (chegada da referência ao objeto *itinerary*) precisou ser adicionado devido ao fato de que, na especificação do sinal, tal parâmetro é esperado.

Na Figura 4.12, o passo 5 e seus sub-passos foram aplicados. O que pode ser visto na figura é a conversão das ações de cada transição para a rede de Petri. As ações contidas na transição *Migrating* do diagrama de estados foram modeladas em dois passos:

1. A conversão da expressão "*String agc = itinerary.nextAgency();*" foi realizada de acordo com o sub-passo 5.5(a). A transição *Migrating* na rede de Petri foi desdobrada em duas, onde a original é responsável por enviar uma mensagem síncrona ao objeto *itinerary* solicitando a próxima agência, e uma outra foi adicionada para receber o valor desejado (*agc*).

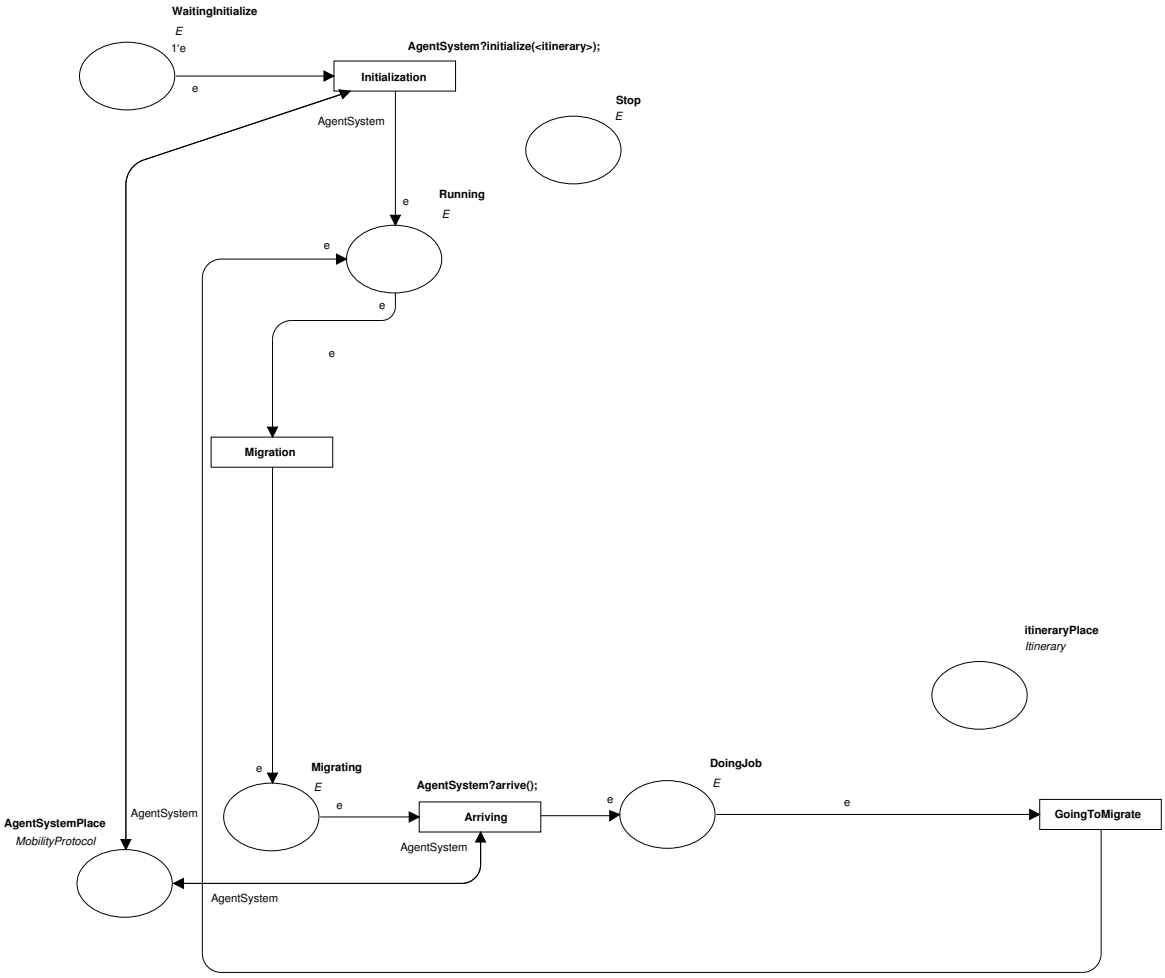

Figura 4.11: Transformação do Diagrama de Estados que Descreve a Classe *ItineraryAgent* para sua Respectiva Rede - Passo 2

2. A expressão "*AgentSystem.migrate(agc);*" foi traduzida no envio de uma mensagem assíncrona para a porta *AgentSystem*, como dito pelo sub-passo 5.1. é importante lembrar que as redes de Petri que modelam as classes *MobilityProtocol* e *Agency* devem possuir o comportamento necessário para enviar e receber a mensagem, respectivamente.

Ainda na Figura 4.12, podemos observar a aplicação do sub-passo 5.3, onde uma transição foi adicionada (*doJob*) para abstrair a execução do método privado *doJob()* (ação da transição que pode ser vista na Figura 4.5), em conjunto com um novo lugar chamado *DoingJob_1*. E, para ilustrar a execução do sub-passo 5.4, na transição *Initialization*, é possível ver um novo arco de saída para o lugar *itineraryPlace*, que é decorrente da ação "*itinerary = (Itinerary) getMsgdata();*" contida na transição *Initialization*. Este arco atualiza a referência contida no atributo *itinerary* com o valor do parâmetro <*itinerary*>, seguindo a semântica da respectiva ação.

Por fim, iremos mostrar como o ponto de escolha visto na Figura 4.4 foi modelado no exemplo, finalizando a rede de Petri que descreve o comportamento da classe *ItineraryAgent* (Figura 4.13).

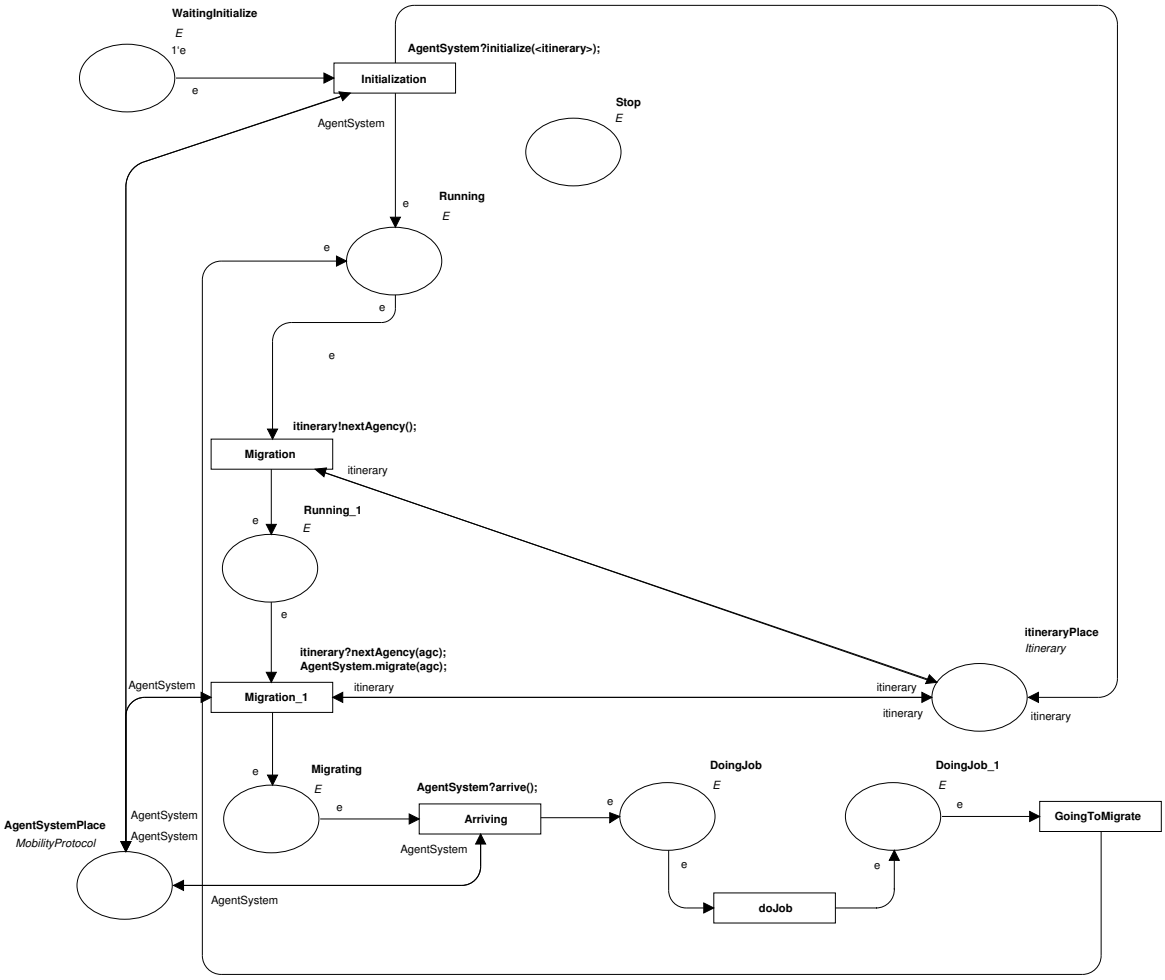

Figura 4.12: Transformação do Diagrama de Estados que Descreve a Classe *ItineraryAgent* para sua Respectiva Rede - Passo 3

A transformação é feita de acordo com o Passo 7 apresentado acima, mais precisamente o subpasso (b), já que se trata de uma condição complexa. Um lugar chamado *ThereIsAgc* foi adicionado para representar que o ponto de escolha deverá ser executado. Note que uma ficha é colocada neste lugar no mesmo instante em que uma ficha é colocada no lugar que representa o estado *Running*. A referida condição é composta pela expressão "*itinerary.hasElement();*", e está modelada pelas transições *ThereIsAgc_T*, *ThereIsAgc_Yes* e *ThereIsAgc_No*, onde a primeira é responsável por enviar a mensagem de solicitação, e as duas restantes são para o caso da resposta ser *Yes* ou *No*, respectivamente. Caso a transição *ThereIsAgc_Yes* dispare, ou seja, ainda haja agências no itinerário, a transição *Migration* estará habilitada a ser disparada. Caso contrário (*ThereIsAgc_No* dispare), a transição *ThereIsAgc_stay* estará habilitada a disparar, levando o objeto ao estado *Stop*.

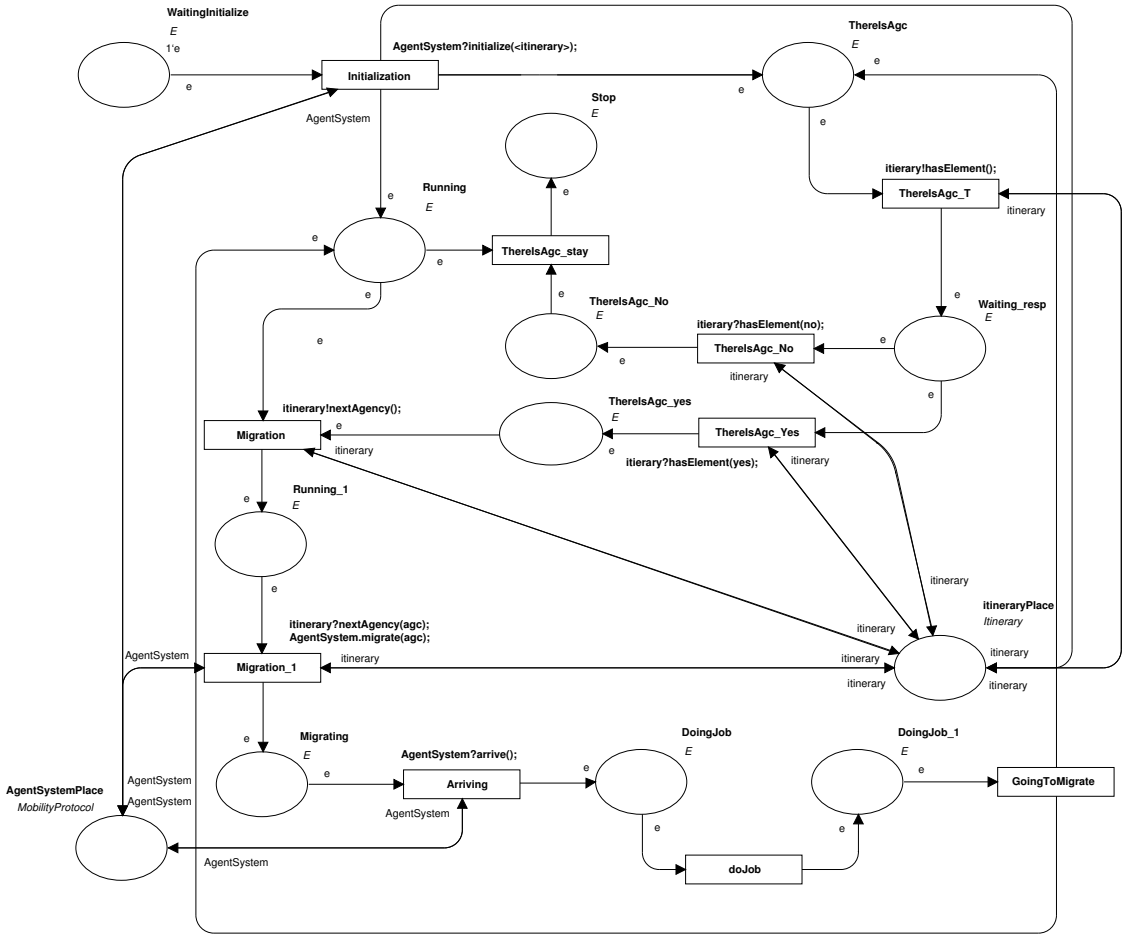

Figura 4.13: Transformação do Diagrama de Estados que Descreve a Classe *ItineraryAgent* para sua Respectiva Rede - Passo Final

Estado Inicial

O estado inicial do modelo RPOO (sistema de objetos e redes de Petri) será formado da maneira que segue.

1. As instâncias e ligações necessárias deverão ser criadas no sistema de objetos de acordo com um diagrama de colaboração do modelo UML-RT que contém a configuração inicial do sistema. Caso não haja tal diagrama, o engenheiro poderá criar uma ou mais configurações iniciais com o intuito de simular e validar o sistema, por exemplo.

2. Uma ficha "*e*" deverá ser a marcação inicial do lugar respectivo ao estado inicial do diagrama de estados que representa aquele objeto. Note na Figura 4.13 que o lugar *WaitingInitialize* possui tal marcação inicial (*1'e*).

4.3.2 Considerações sobre os Modelos

Apesar de termos passos de um mapeamento informal, capazes de auxiliar na transformação dos modelos em UML-RT para RPOO, algumas considerações sobre tais modelos são importantes no intuito de facilitar esta transformação e, principalmente, diminuir a possibilidade de erros. Além disto, no final desta seção, terceremos alguns comentários acerca dos modelos obtidos em RPOO, em especial sua utilização pelo método apresentado no Capítulo 3.

O primeiro comentário está relacionado com a complexidade das guardas, pontos de escolha e ações de forma geral. É importante salientar que quanto menos complexos estes forem, mais fácil (conseqüentemente, menos passível de erro) será a sua transformação. Isto se dá pelo fato de que a transformação destes elementos não é direta, precisando que cada caso seja analisado em separado.

Em UML-RT, podemos associar uma lista de eventos para uma única transição de um diagrama de estados com a semântica de que aquela transição será disparada caso pelo menos um destes eventos ocorra (uma lista do tipo "OU"), além da guarda ser satisfeita, é claro. Além disso, como foi apresentado nesta seção, cada um destes eventos é traduzido para RPOO como uma inscrição do tipo "*porta?mensagem(dado);*" da respectiva transição RPOO. No entanto, é preciso salientar que uma lista de inscrições deste tipo possui a semântica de uma lista do tipo "E", ou seja, a transição só estará habilitada para o disparo caso haja uma mensagem pendente que satisfaça cada inscrição (todos os respectivos eventos tenham ocorrido). Desta forma, a tradução de uma lista de eventos de uma transição UML-RT precisa ser traduzida de forma diferente: para cada evento da lista, uma nova transição RPOO clone da respectiva transição UML-RT deverá ser criada, inclusive com os mesmos arcos de entrada e de saída. Desta forma, a quantidade de transições RPOO referentes a cada transição UML-RT será o número de eventos que esta última possui.

Quanto ao modelo RPOO obtido ao final da transformação, é importante notar que este não teria tanta diferença para um modelo construído por um modelador RPOO sem conhecimento prévio deste modelo UML-RT. Basicamente, as diferença seriam duas:

- *Visão estrutural:* Provavelmente, as portas não estariam presentes no modelo RPOO criado independente do modelo UML-RT. Inclusive, no estudo de caso apresentado no Capítulo 5, mostramos que, para protocolos cujas portas são apenas repassadoras de sinais, não há a necessidade de se criar classes RPOO para estes protocolos, o que facilita a transformação e torna o modelo final mais simples.

- *Visão comportamental:* Para um modelador RPOO que fosse fazer o modelo independente do modelo UML-RT, as redes de Petri que descrevem as classes conteríam menos lugares significando o estado do objeto. Isto ocorre porque os estados dos objetos RPOO são dados pela marcação de todos os lugares, e não apenas pelo conteúdo de um único lugar como ocorre após a transformação.

Para o efeito desejado, o impacto causado por estas diferenças na aplicação destes modelos no método apresentado no Capítulo 3 pode ser considerado nulo, pois para o método, o que interessa é

apenas a troca de mensagens entre os objetos, mais precisamente, a troca de mensagens entre a parte do sistema que se deseja testar e o ambiente - no caso, agências, agentes e interfaces com o usuário.

4.4 Conclusão

Neste capítulo, apresentamos um procedimento informal de transformação de modelos em UML-RT para modelos em RPOO. Apesar de não ser automática, a transformação é simples, devido ao fato de que estamos tratando linguagens semelhantes[1] de um ponto de vista do modelador.

É importante ressaltar que o objetivo dos resultados obtidos neste capítulo é o de estimular uma futura transformação automática entre as duas linguagens, fazendo, assim, com que os desenvolvedores precisem conhecer apenas uma linguagem, no caso UML-RT. De posse do procedimento aqui apresentado, a construção de modelos RPOO com base em modelos UML-RT já desenvolvidos pelos engenheiros será facilitada, estimulando a aplicação do método de geração de casos de teste a estes sistemas modelados em UML-RT.

Mesmo com uma futura automação desta transformação, ainda assim precisaríamos utilizar uma linguagem formal (no nosso caso RPOO) para a geração dos casos de teste, já que, para tal, é preciso que a linguagem a partir de onde os casos de teste serão gerados tenha semântica formal.

Assumindo o processo desde o modelo UML-RT, passando pelo modelo RPOO, até os casos de teste, nada pode ser afirmado sobre estes casos de teste em relação ao modelo UML-RT inicial. Uma vez que o mapeamento apresentado é informal e que serve apenas como base para a construção do modelo RPOO, os casos de teste gerados pelo processo podem ser relacionados apenas com o modelo RPOO. Como conseqüência, temos que os modelos RPOO precisam estar corretos em relação aos modelos UML-RT.

[1] Ambas as linguagens possuem uma perspectiva orientada a objetos para a visão estrutural dos modelos e uma perspectiva comportamental descrita por uma linguagem definida sobre LTS.

Capítulo 5

Estudo de Caso - Parte 1

5.1 Introdução

Os Capítulos 5 e 6 têm como objetivo mostrar uma aplicação do método proposto nos Capítulos 3 e 4 sobre um determinado sistema. Durante tal aplicação, uma análise e um estudo sobre os resultados são apresentados. O objetivo deste estudo de caso é o de apresentar um exemplo prático do uso do método, mostrando sua aplicabilidade e apresentando indícios reais de sua viabilidade. Este capítulo, em especial, apresenta a modelagem do sistema em estudo com a linguagem UML-RT, e a construção de um modelo RPOO a partir deste, com o auxílio da transformação apresentada no Capítulo 4.

A aplicação selecionada para realizar o estudo de caso foi desenvolvida por Guedes e pode ser encontrada em sua dissertação de mestrado [Gue02a]. Trata-se de um sistema de apoio às atividades de comitês de programa em conferências, que será chamado simplesmente de sistema de conferências. A aplicação gerencia atividades de comitês de programas de conferências, tais como submissão de artigos, processo de avaliação e notificação de aceitação ou rejeição de artigos aos autores. Esta aplicação foi selecionada com base nos seguintes requisitos:

- **Disponibilidade de código-fonte:** Necessário para viabilizar a implementação dos casos de teste.
- **Disponibilidade da especificação (de preferência UML):** Necessário para viabilizar a construção dos modelos de entrada em UML-RT para o métodos.

Além desses requisitos, o conhecimento prévio dos autores com a plataforma de execução, linguagem de programação, código fonte e especificação da aplicação vieram a facilitar a aplicação do método e também foram utilizados como requisitos para a escolha da aplicação.

Não só a especificação como também o código fonte podem ser encontrados em [Gue02a]. A aplicação foi desenvolvida em Java sobre a plataforma Grasshopper[1] [Gmb98]. Diagramas de classe, de seqüência e de colaboração de UML foram utilizados por Guedes para a especificação do sistema.

[1]Esta plataforma é fornecida pela IKV++, possui distribuição gratuita e é baseada na linguagem Java.

A partir destes diagramas, construímos os diagramas UML-RT referentes à aplicação, que serviram de entrada para o nosso método. Desta forma, assumindo que a especificação do sistema apresentada por Guedes está correta, conseqüentemente assim também os nossos modelos em UML-RT, a aplicação do método de geração de casos de teste no estudo de caso tem como objetivo encontrar falhas na implementação do sistema de conferências tendo como base sua respectiva especificação.

Na Seção 5.2, iremos apresentar o sistema de conferência através de seus modelos em UML-RT. Em seguida (Seção 5.3), mostramos como tais modelos foram convertidos para RPOO. Por fim, as conclusões relacionadas a este passo do processo são apresentadas (Seção 5.4).

5.2 Sistema de Conferência em UML-RT

Com base nos diagramas de classe e de seqüência do sistema apresentados por Guedes [Gue02a], construímos os diagramas em UML-RT. Como apresentado no Capítulo 4, tais diagramas são: diagrama de classe contendo as classes do sistema e seus relacionamentos em um contexto universal; diagramas de colaboração decrevendo o relacionamento entre os objetos em determinados contextos; e um diagrama de estados para cada classe com os estereótipos *«Capsule»* e, opcionalmente, *«Protocol»*.

O processo de conversão dos modelos em UML padrão, apresentados originalmente em [Gue02a], para os modelos UML-RT, que serão apresentados em seguida, não será descrito, uma vez que estamos especialmente interessados no processo de teste partindo dos modelos já em UML-RT. Contudo, podemos dizer que o diagrama de classes foi fortemente baseado no diagrama de classes original e que os diagramas de estados foram retirados de diagramas de seqüência e de colaboração, preservando a lógica do sistema.

Com o intuito de facilitar o entendimento do sistema e principalmente o entendimento do comportamento dos agentes envolvidos no sistema, a seguir apresentaremos os agentes envolvidos no sistema e as agências que irão hospedar tais agentes. Mais adiante, apresentaremos a descrição do sistema através de um diagrama de comportamento que descreve a forma com que estes agentes irão migrar entre as agências e se comunicar com os atores externos e entre si.

- ***Agente Coordenador***: Agente estacionário responsável por prover interface gráfica com o coordenador do membro de comitê e, ao receber a solicitação de revisão de um artigo, cria um agente ***Agente Formulário Revisão***, delegando a responsabilidade de obter as revisões para os artigos para tal.

- ***Agente Formulário Revisão***: Responsável por obter uma revisão para um determinado artigo. É um agente móvel que migra por agências de revisores em busca de revisões e, ao final, registra essa revisão junto ao ***Agente Coordenador***.

- ***Agência Coordenador***: Agência onde estará executando o ***AgenteCoordenador*** e onde o coordenador do comitê de programa estará localizado.

- ***Agência Membro Comitê***: Agência por onde o ***Agente Formulário Revisão*** irá passar e onde estará situado o membro do comitê responsável por um determinado artigo.
- ***Agência Revisor***: Agência por onde o ***Agente Formulário Revisão*** irá passar e onde estará situado um possível revisor de um determinado artigo.

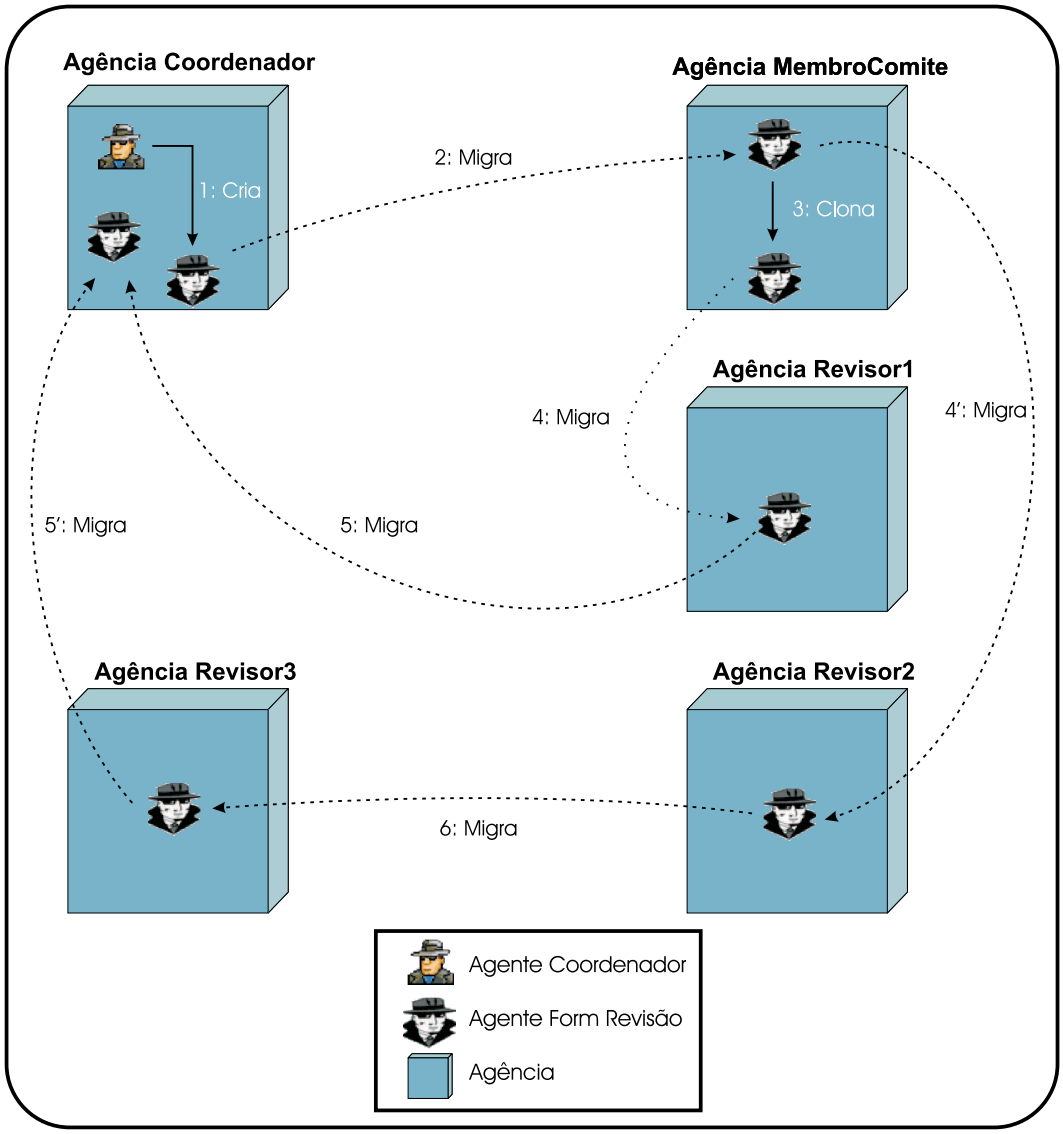

Figura 5.1: Diagrama de Comportamento dos Agentes

A Figura 5.1 mostra o comportamento dos principais agentes do sistema para um cenário básico. Nela, vemos a presença de um ***Agente Coordenador*** na ***Agência Coordenador***. Por meio de uma interface gráfica, provida pelo ***Agente Coordenador***, o coordenador do comitê de programa seleciona um artigo a ser revisado, o membro do comitê responsável por encontrar revisores para este artigo e a quantidade de cópias deste formulário desejada (quantidade de revisões). O ***Agente Coordenador***, então, cria um ***Agente Formulário Revisão*** contendo o artigo a ser revisado (1). O ***Agente Formu-***

lário Revisão migra para a máquina do membro do comitê (***Agência MembroComite***) (2). O ***Agente Formulário Revisão*** gera um clone para cada formulário solicitado pelo coordenador menos um, já que ele mesmo irá conter uma revisão (3). O membro do comitê, através de suas interfaces gráficas, redireciona os agentes ***Agente Formulário Revisão*** existentes em sua máquina para outros revisores, informando se deseja que eles retornem quando tiverem sido revisados ou não. Cada ***Agente Formulário Revisão*** migra para a máquina de um revisor (4 e 4'). O revisor recebe o ***Agente Formulário Revisão*** e pode optar por revisar o artigo diretamente ou redirecioná-lo para um outro revisor (6), informando se deseja que ele retorne quando tiver sido revisado para aprovação. Quando o processo de revisão é finalizado o ***Agente Formulário Revisão*** retorna diretamente para a máquina do coordenador do comitê de programa caso nenhum dos remetentes tenha solicitado o seu retorno (5 e 5'). Do contrário, o ***Agente Formulário Revisão*** volta para o último remetente que solicitou o seu retorno. Caso ainda exista um remetente anterior a este, que também tenha solicitado o seu retorno, o ***Agente Formulário Revisão*** retornará para ele. Após ter sido aprovado por todos os remetentes que solicitaram o seu retorno, o ***Agente Formulário Revisão*** volta para a máquina do coordenador de programa.

A partir deste ponto, mostraremos a modelagem UML-RT. É possível notar que a modelagem inclui termos em Inglês e outros em Português. Isto decorre do fato de que o sistema de conferências foi modelado utilizando-se de termos em Português (e.g. AgenteCoordenador), no entanto, as partes relativas às plataformas de execução foram feitas utilizando-se de termos em Inglês (e.g. Agency), no intuito de se aproximar dos termos utilizados pelas plataformas. Nas Figuras 5.2, 5.3, 5.4, 5.5 e 5.6, apresentamos partes do diagrama de classes UML-RT para o sistema de conferências (o diagrama completo pode ser encontrado no Apêndice A). Como visto na Figura 5.2, cada agente do sistema (**AgenteCoordenador** e **AgenteFormRevisao**) está modelado como uma cápsula, bem como as agências (**Agency**), a rede (**Internet**) e a entidade **Conferência**, que representa o restante das classes do sistema. Tais entidades são modeladas como cápsulas pois possuem comportamento independente e são passíveis de distribuição, podendo estar executando em diferentes máquinas. Essas entidades se comunicam através de protocolos específicos. Os agentes se comunicam com as agências através do protocolo **AgencyProtocol**, já o agente **AgenteCoordenador** se comunica com **Conferência** através do protocolo **ConferenciaProtocol**. Os agentes se comunicam entre si através do protocolo **ResultadosProtocol** e as agências se utilizam dos serviços da **Internet** através do protocolo **AgentMigration**.

Cada um dos agentes possui uma interface gráfica (vide Figura 5.3) pela qual os usuários do sistema (revisores, membros de comitê, etc.) se comunicam com o mesmo. As interfaces propriamente ditas não estão modeladas no sistema, contudo, a forma com que os agentes irão interagir com as mesmas está modelada através dos protocolos **GuiAgenteCoordenador** e **GuiAgenteFormRevisao**. O primeiro descreve as interações entre o **AgenteCoordenador** e o coordenador do comitê de programa, já o segundo as interações entre o **AgenteFormRevisao** e os membros de comitê e revisores de artigos. Esta forma de modelagem se justifica pelo fato de que nós não estamos interessados em

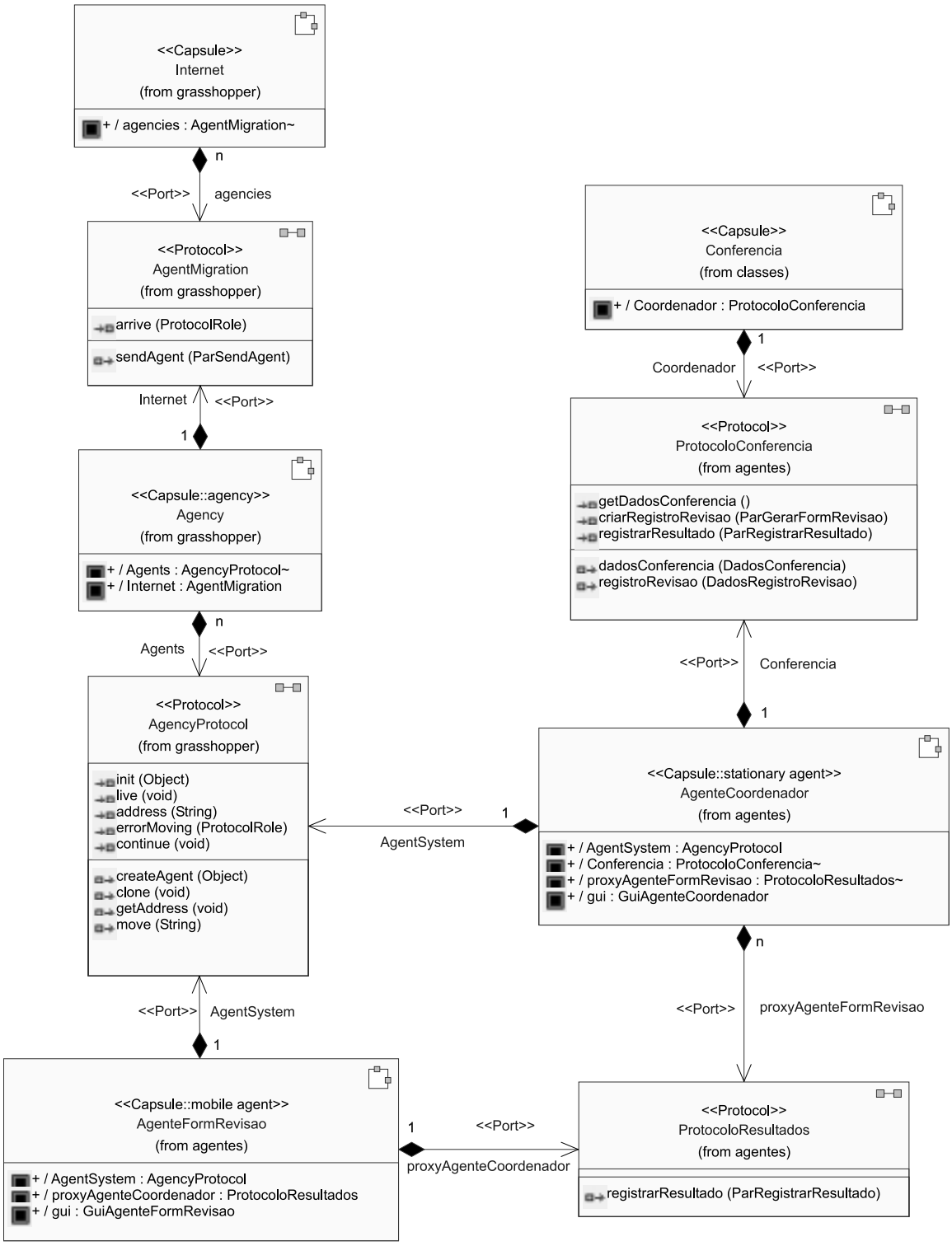

Figura 5.2: Diagrama de Classes UML-RT do Sistema de Conferências - Cápsulas e Protocolos

testar o comportamento das interfaces, e sim o comportamento das demais entidades em decorrência e através das interações acima citadas.

Como pode ser visto no diagrama da Figura 5.4, há uma classe no sistema chamada de **Itine-**

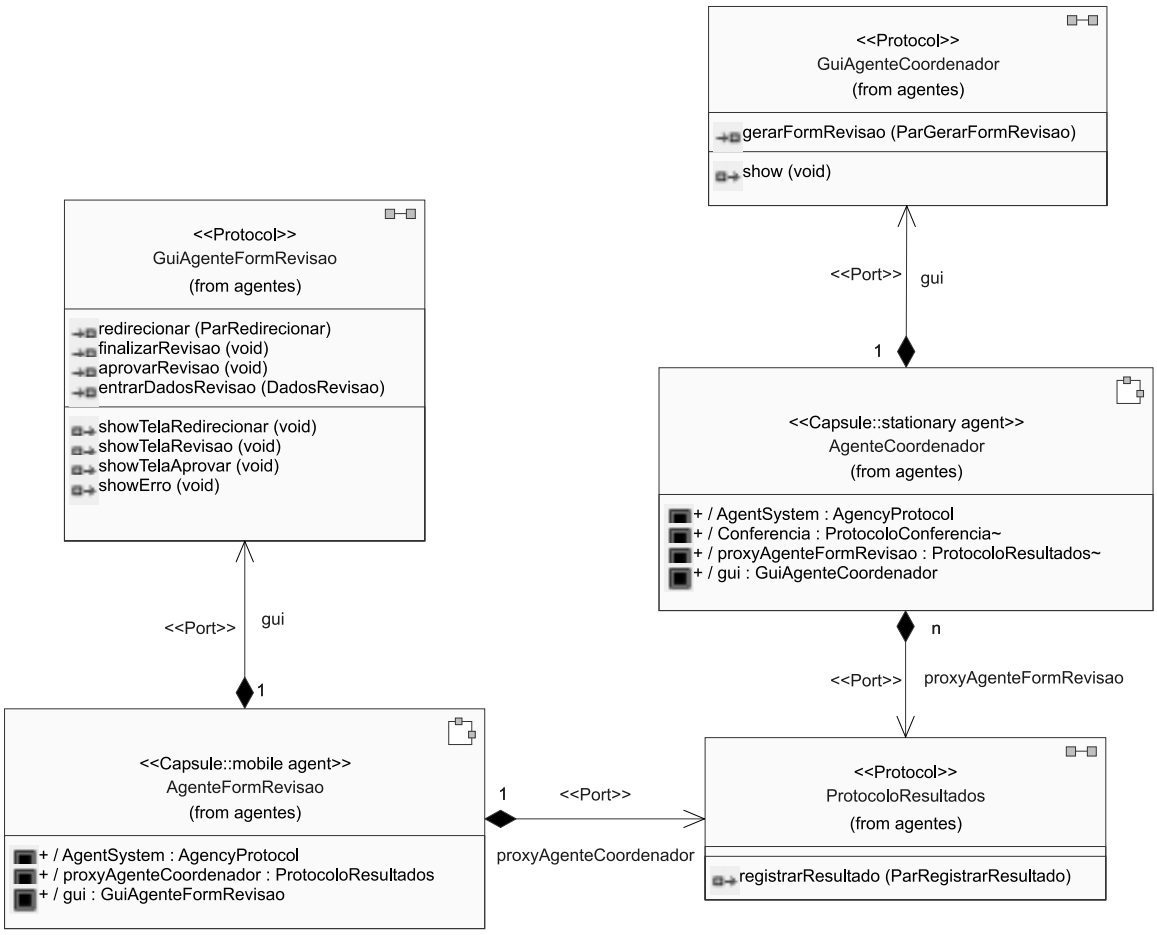

Figura 5.3: Diagrama de Classes UML-RT do Sistema de Conferências - Agentes e Interfaces

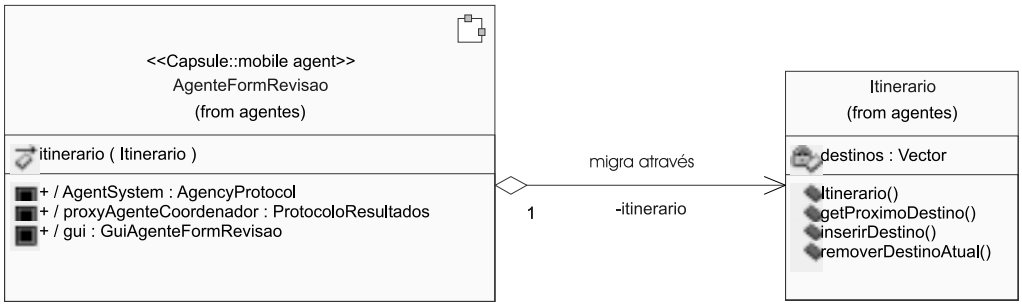

Figura 5.4: Diagrama de Classes UML-RT do Sistema de Conferências - **AgenteFormRevisao** e **Itinerario**

rario, que não apresenta estereótipos. Esta classe modela o itinerário a ser seguido pelo agente **AgenteFormRevisao** quando este necessita aprovar uma determinada revisão do artigo o qual é responsável. Tanto o membro do comitê de programa como os revisores que redirecionaram o agente para outros revisores podem solicitar que ele retorne à sua agência para aprovar a revisão que lá estiver contida. Desta forma, o itinerário a ser percorrido pelo agente é composto pelas agências cujos revisores solicitaram tal retorno.

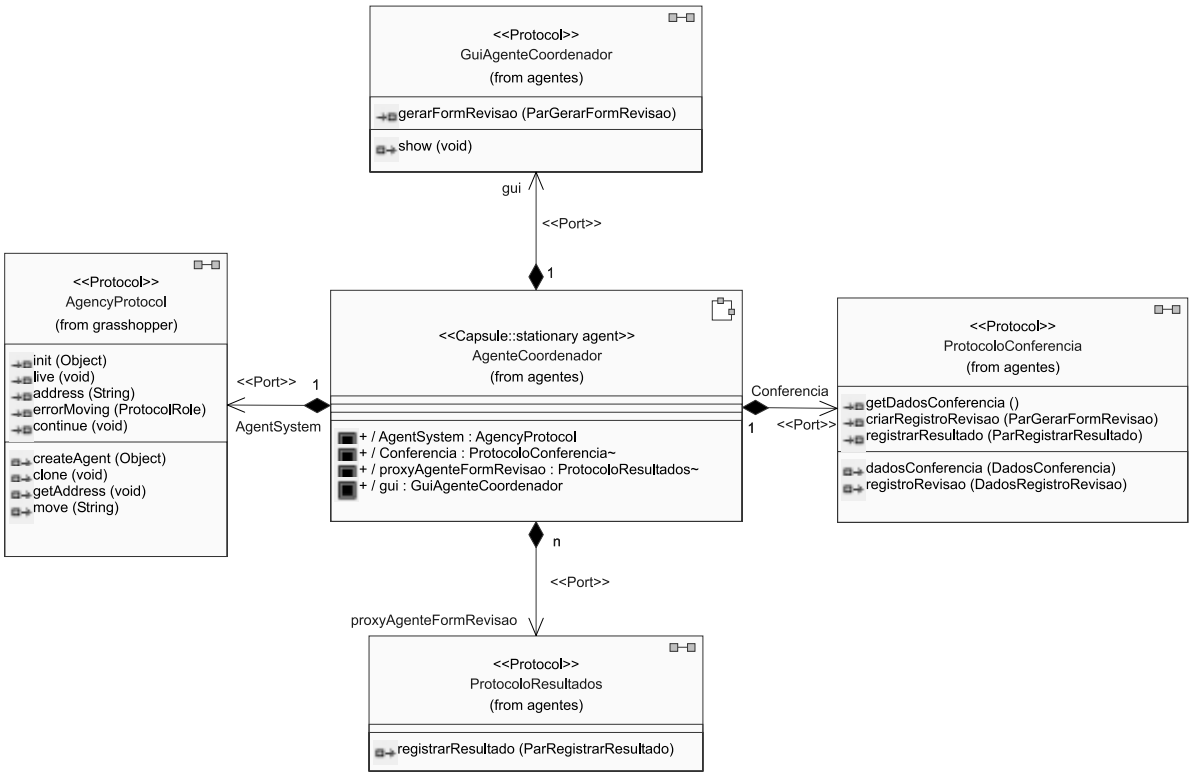

Figura 5.5: Diagrama de Classes UML-RT do Sistema de Conferências - **AgenteCoordenador**

A Figura 5.5 detalha a cápsula **AgenteCoordenador** e seus relacionamentos. O **AgenteCoordenador** é responsável por receber as solicitações para as revisões de artigos através de sua interface gráfica (**GuiAgenteCoordenador**) e criar o **AgenteFormRevisao** que tem a função de obter as revisões destes artigo. O **AgenteCoordenador** se comunica com a **Conferencia** através do protocolo **ConferenciaProtocol** para obter informações sobre a conferência, registrar as revisões a serem feitas e registrar os resultados das revisões. O **AgenteCoordenador** está localizado em uma determinada agência e se utiliza desta através do protocolo **AgencyProtocol** para criar o **AgenteFormRevisao**.

A Figura 5.6 detalha a cápsula **AgenteFormRevisao**. Ao ser criado, o **AgenteFormRevisao** poderá clonar-se, caso seja solicitado mais de uma revisão para o artigo, e migrar por entre as agências dos revisores utilizando-se dos serviços providos pela **Agency** através do protocolo **AgencyProtocol**. Ao término de seu trabalho, cada **AgenteFormRevisao** envia ao **AgenteCoordenador** por meio de **ResultadosProtocol** os resultados das revisões para que sejam registrados por este último junto à **Conferencia**.

À seguir, apresentaremos como o comportamento do sistema foi modelado através dos diagramas de estados de cada cápsula. Da mesma forma que com cápsulas, é possível descrever o comportamento dos protocolos através de um diagrama de estados em UML-RT. Este comportamento reflete uma descrição da forma na qual as mensagens serão trocadas pelos participantes do protocolo. Porém, como os nossos protocolos são apenas transmissores de mensagens, não há a necessidade de definirmos diagramas de estados para eles.

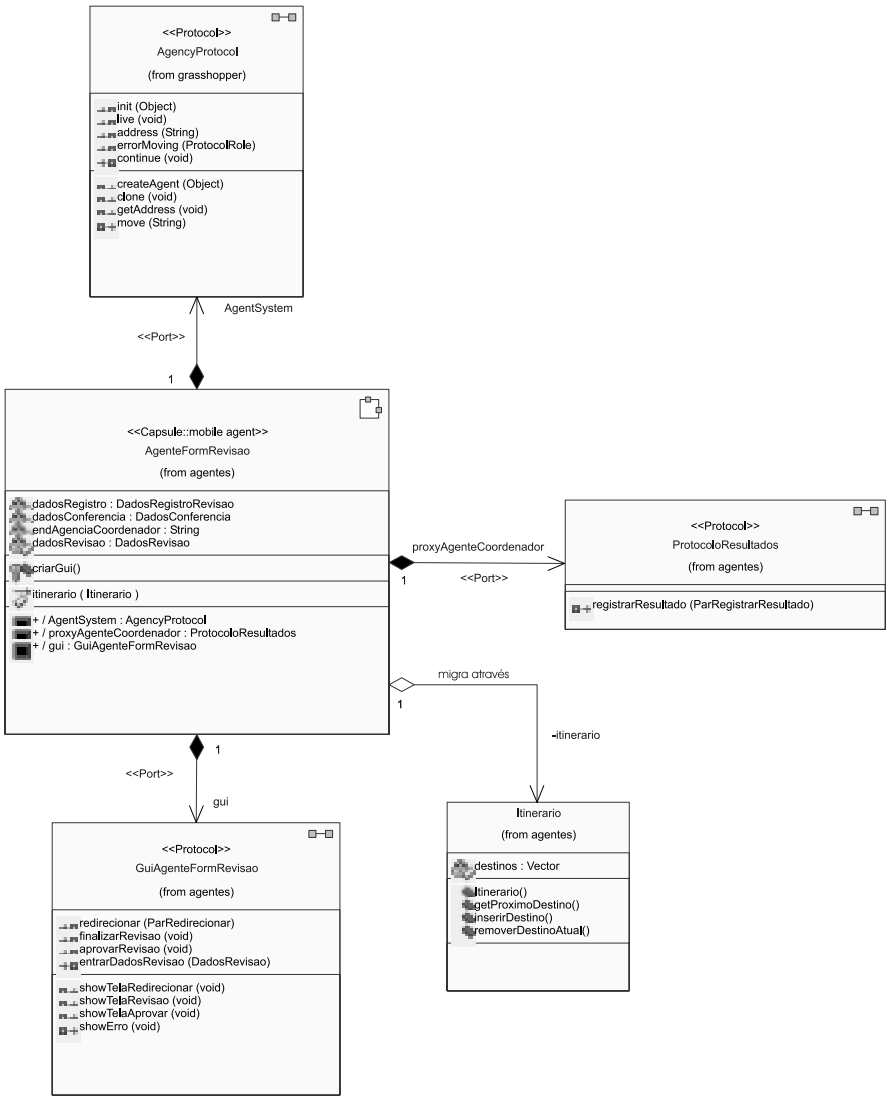

Figura 5.6: Diagrama de Classes UML-RT do Sistema de Conferências - **AgenteFormRevisao**

As Figuras 5.7, 5.8 e 5.9 apresentam os diagramas de estados para as cápsulas **Conferencia**, **Agency** e **Internet** respectivamente. Estas cápsulas possuem um comportamento simples, onde as entidades possuem apenas um estado. Este comportamento é definido como respostas a estímulos (recebimento de mensagens) produzidos pelo ambiente. Esta modelagem é decorrente do fato de que estamos especialmente interessados no comportamento dos agentes e, desta forma, estas cápsulas tiveram seu comportamento abstraido desta maneira.

Por outro lado, as cápsulas **AgenteCoordenador** e **AgenteFormRevisao** possuem diagramas de estados (Figuras 5.10 e 5.12 respectivamente) mais complexos. Como pode ser visto na Figura 5.10, quando o **AgenteCoordenador** é criado (estado **Criado**), e em decorrência do disparo da transição **Executar**, ele envia uma mensagem *"show()"* através da porta **gui**, o levando ao estado **Executando**. Isto significa que o agente solicita à sua interface gráfica que uma tela seja mostrada ao usuário. Estando neste estado, o agente, ao receber uma mensagem *"gerarFormRevisao()"* de sua interface

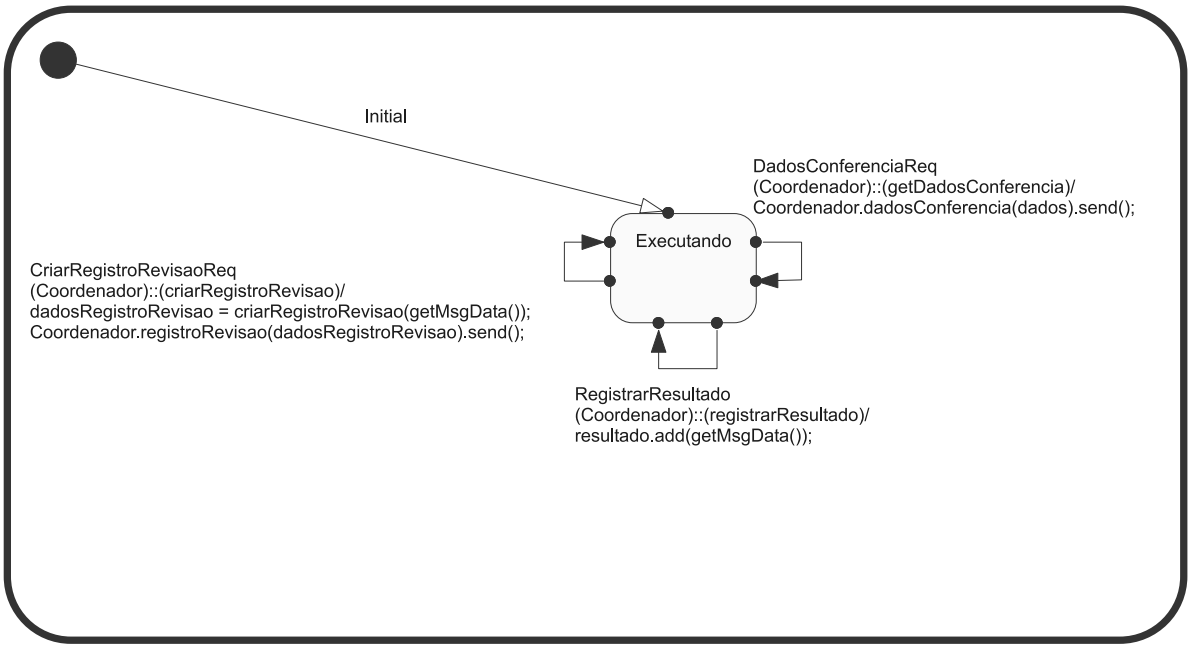

Figura 5.7: Diagrama de Estados UML-RT da Cápsula **Conferencia**

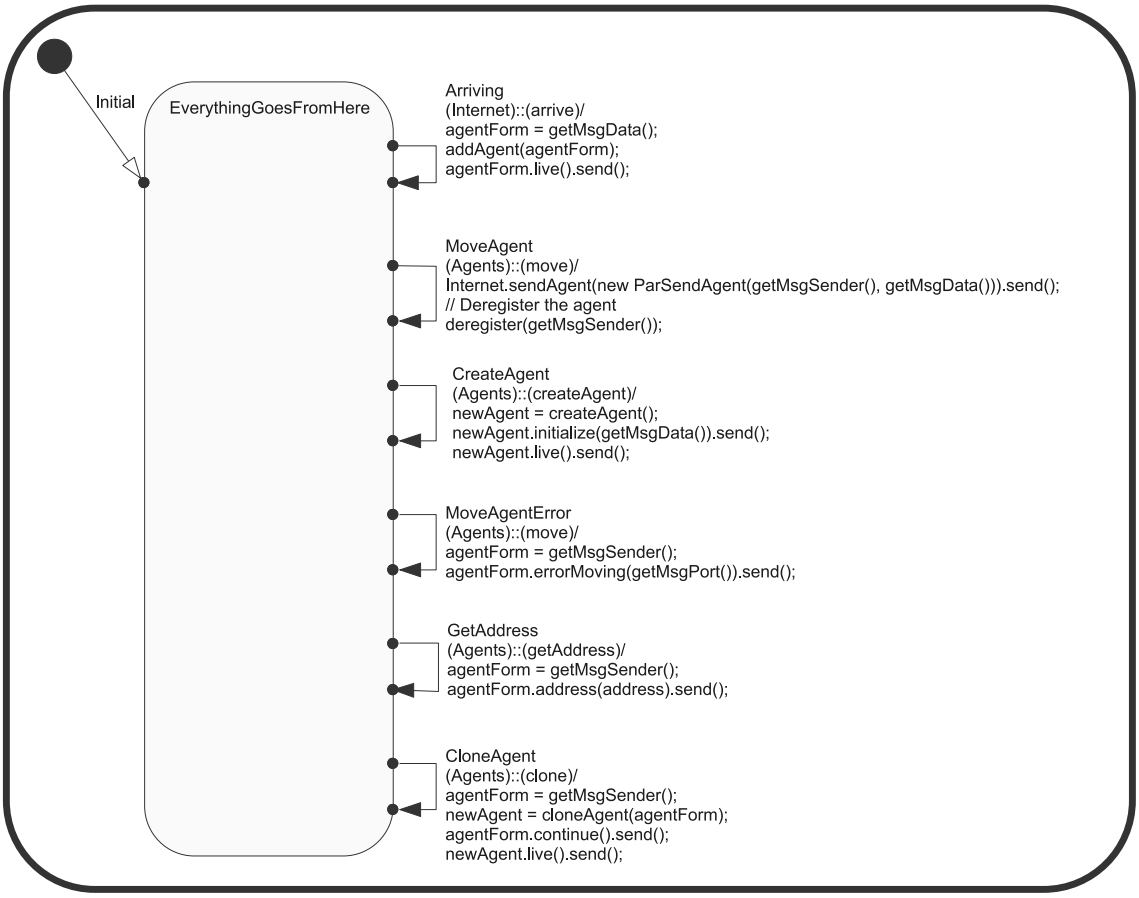

Figura 5.8: Diagrama de Estados UML-RT da Cápsula **Agency**

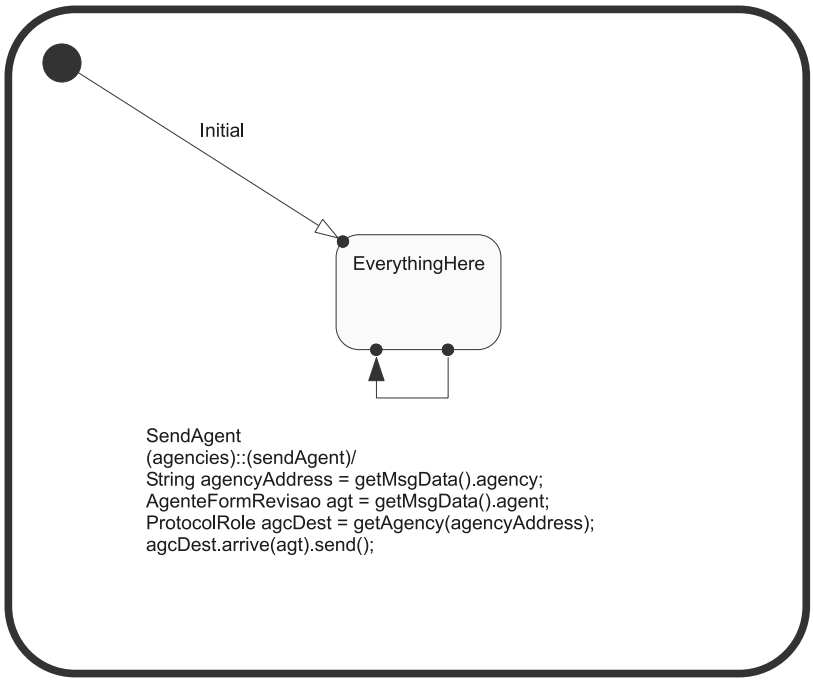

Figura 5.9: Diagrama de Estados UML-RT da Cápsula **Internet**

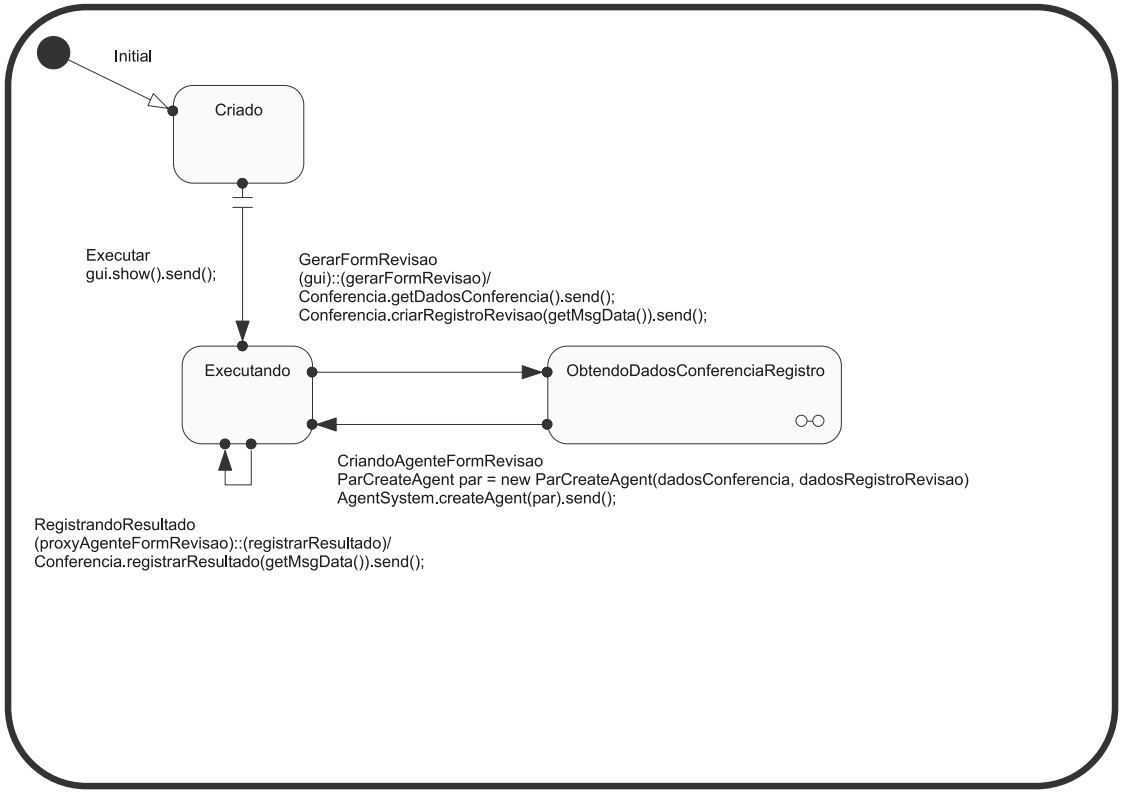

Figura 5.10: Diagrama de Estados Principal UML-RT da Cápsula **AgenteCoordenador**

gráfica, envia as mensagens *"getDadosConferencia()"* e *"criarRegistroRevisao()"*, e passa para o estado **ObtendoDadosConferenciaRegistro**, conforme descrito pela transição **GerarFormRevisao**.

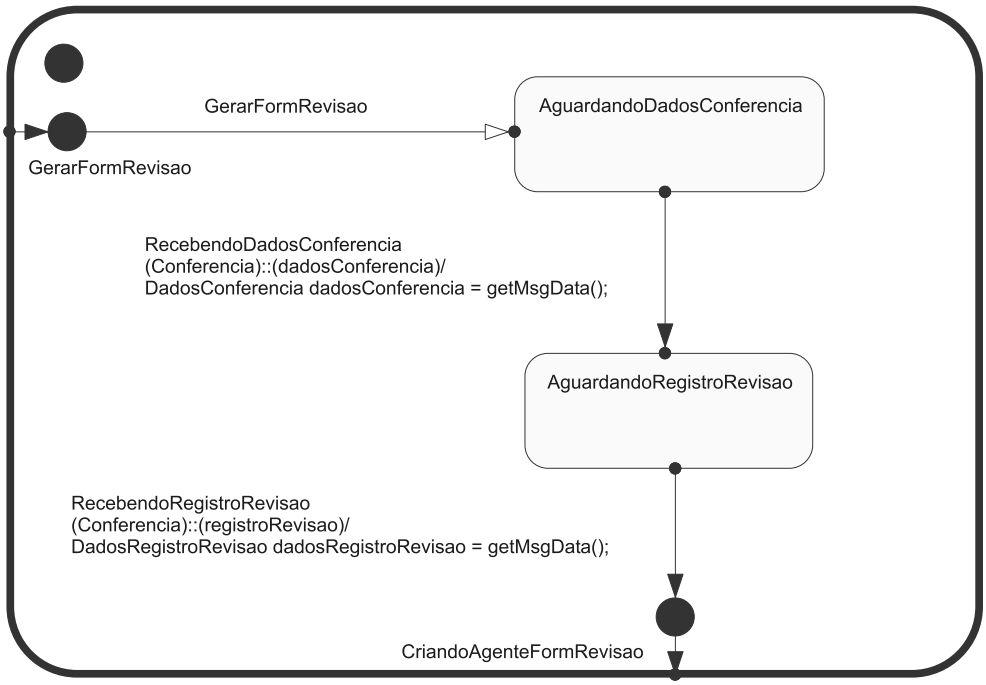

Figura 5.11: Detalhamento do Estado **ObtendoDadosConferenciaRegistro** do Diagrama de Estados UML-RT da Cápsula **AgenteCoordenador**

Este estado é detalhado por um outro diagrama de estados contido na Figura 5.11. Ainda no estado **Executando**, o agente pode receber a mensagem *"registrarResultado()"* proveniente de um **AgenteFormRevisao** pela porta **proxyAgenteFormRevisao** e, em isto acontecendo, uma mensagem *"registrarResultado()"* será enviada à porta **Conferencia**, como visto na transição **RegistrandoResultado**.

Quando o agente **AgenteCoordenador** atinge o estado **ObtendoDadosConferenciaRegistro** através da transição **GerarFormRevisao**, ele atinge, também, o sub-estado **AguardandoDadosConferencia** (vide Figura 5.11). Neste estado, ao receber a mensagem *"dadosConferencia()"*, ele passa ao estado **AguardandoRegistroRevisao**, guardando o conteúdo da mensagem em *dadosConferencia*, como visto na ação *"DadosConferencia dadosConferencia = getMsgData();"*. Em **AguardandoRegistroRevisao**, o agente fica aguardando a chegada da confirmação do registro da revisão através da mensagem *"registroRevisao()"*. Com a sua chegada, a transição **RecebendoRegistroRevisao** dispara gerando a ação *"DadosRegistroRevisao dadosRegistroRevisao = getMsgData();"* que acarreta na colocação do dado da mensagem em *dadosRegistroRevisao*. Além disso, a transição **RecebendoRegistroRevisao** também serve como gatilho para a transição **CriandoAgenteFormRevisao** que, ao ser disparada, solicita à agência (porta **AgentSystem**) a criação de um **AgenteFormRevisao** com os parâmetros *dadosConferencia* e *dadosRegistroRevisao* e coloca o agente no estado **Executando** novamente.

A Figura 5.12 apresenta o diagrama de estados para a cápsula **AgenteFormRevisao**. Como pode ser visto, quando criado (estado **Criado**), o agente recebe a mensagem *"init()"* da agência em que está localizada, solicita o endereço da agência e passa para o estado **Inicializando**. Quando receber

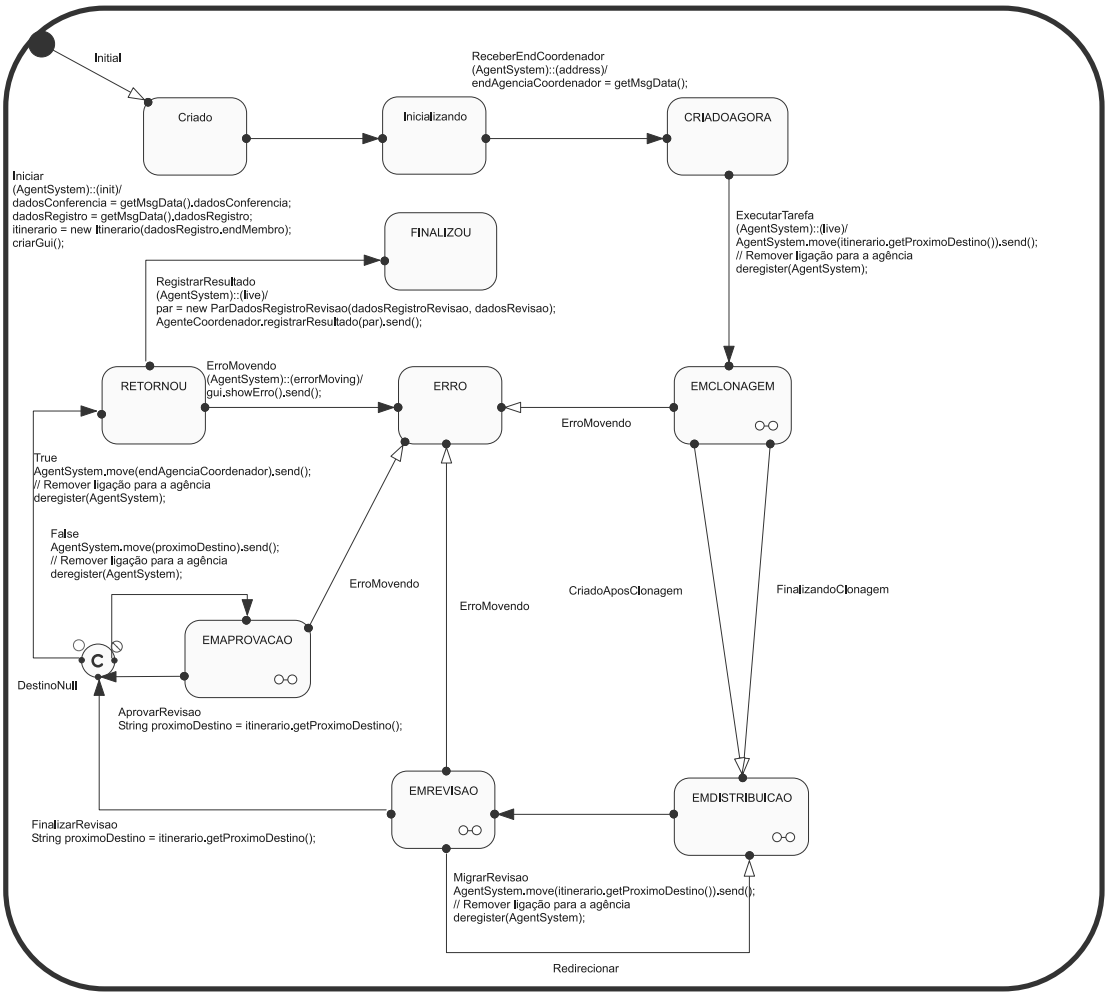

Figura 5.12: Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

o endereço da agência (recebimento da mensagem *"address()"*), o agente passa para o estado **CRIADOAGORA**. A partir deste ponto, o agente irá executar sua tarefa propriamente dita que é a de se clonar o número de vezes necessária e ir para as agências dos revisores coletando a revisão para o seu artigo, aprovando tal revisão e retornando para a agência onde o **AgenteCoordenador** se encontra para registrar o resultado.

Seguindo estes objetivos, em **CRIADOAGORA**, quando o agente recebe a mensagem *"live()"*, ele migra para a agência do membro de comitê e entra no estado de **EMCLONAGEM**. Neste estado, o agente irá clonar-se "N" menos uma vezes, onde "N" é o número de revisões solicitadas pelo coordenador, e irá para o estado de **EMDISTRIBUICAO**. Os agentes que foram clonados também estarão no estado **EMCLONAGEM**, pois quando foram clonados o agente original estava em tal estado, e também irão para o estado **EMDISTRIBUICAO**. Neste momento, cada agente solicita ao membro de comitê o endereço da agência para a qual ele deve ir obter a revisão e se é necessário que ele retorna para o membro de comitê aprovar tal revisão. De posse destas informações, o agente migra para a agência do revisor e entra no estado **EMREVISAO**. Neste ponto, o revisor poderá redirecionar

o agente para um outro revisor, informando o agente se precisa voltar para receber a aprovação ou não, assim como acontece com o membro de comitê, colocando o agente, novamente, no estado **EMDISTRIBUICAO**. Caso o revisor entre com uma revisão e finalize a revisão, o agente irá para o estado de **EMAPROVACAO** que fará com que o mesmo migre para as agências de todos os que solicitaram o seu retorno solicitando sua aprovação. Quando as aprovações estiverem concluídas, o agente retorna para a agência do coordenador e entra no estado **RETORNOU**. Neste estado, o agente se comunica com o ***AgenteCoordenador*** para registrar o resultado da revisão e entra no estado **FINALIZOU**, encerrando a sua execução. Em qualquer estado em que o agente solicita sua migração para uma determinada agência, uma transição **ErroMovendo** pode ocorrer fazendo com que o agente entre no estado de **ERRO** e finalize a sua execução. Este comportamento abstrai os casos em que as agências ou a rede está indisponível.

Os estados **EMCLONAGEM**, **EMDISTRIBUICAO**, **EMREVISAO** e **EMAPROVACAO** possuem seus comportamentos detalhados por outras máquinas de estados que estão mostradas, respectivamente, nas Figuras 5.13, 5.14, 5.15 e 5.16.

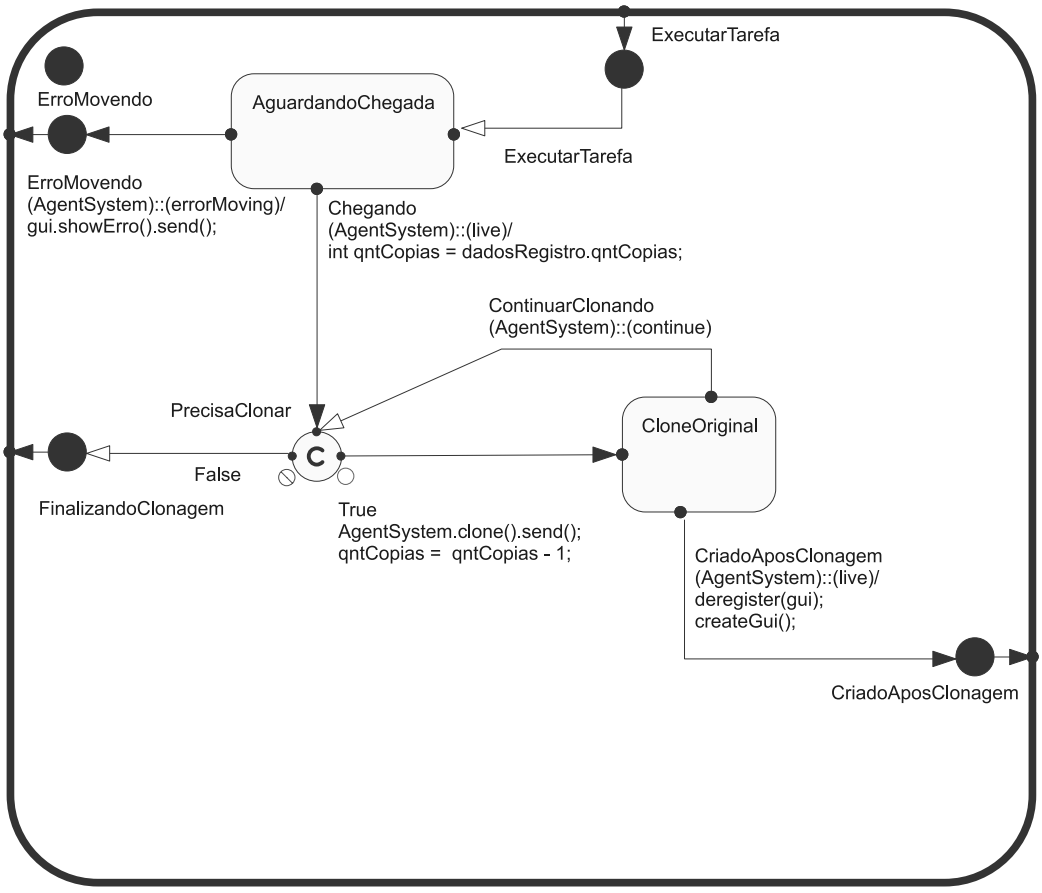

Figura 5.13: Detalhamento do Estado **EMCLONAGEM** do Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

Como visto na Figura 5.13, quando o agente chega no estado **EMCLONAGEM**, ele atinge o

sub-estado **AguardandoChegada**. Neste estado ele pode ir para o estado **ERRO**, caso receba uma mensagem de erro da agência, ou disparar o ponto de escolha **PrecisaClonar**, que irá testar se a quantidade de cópias necessárias é maior que um. Caso não haja a necessidade de clonar (transição **False**), o agente sai do estado **EMCLONAGEM** e vai para o estado **EMDISTRIBUICAO**, através do ponto de junção **FinalizandoClonagem**. Se a clonagem for necessária, o agente envia uma mensagem *"clone()"* à agência solicitando a sua clonagem e entra no estado **CloneOriginal**. Caso este agente seja o agente que solicitou a clonagem, ele irá receber uma mensagem *"continue()"* da agência e a transição **ContinuarClonando** irá disparar. Caso ele seja um clone, ele irá receber a mensagem *"live()"* e irá para o estado **EMDISTRIBUICAO** através do ponto de junção **CriadoAposClonagem**.

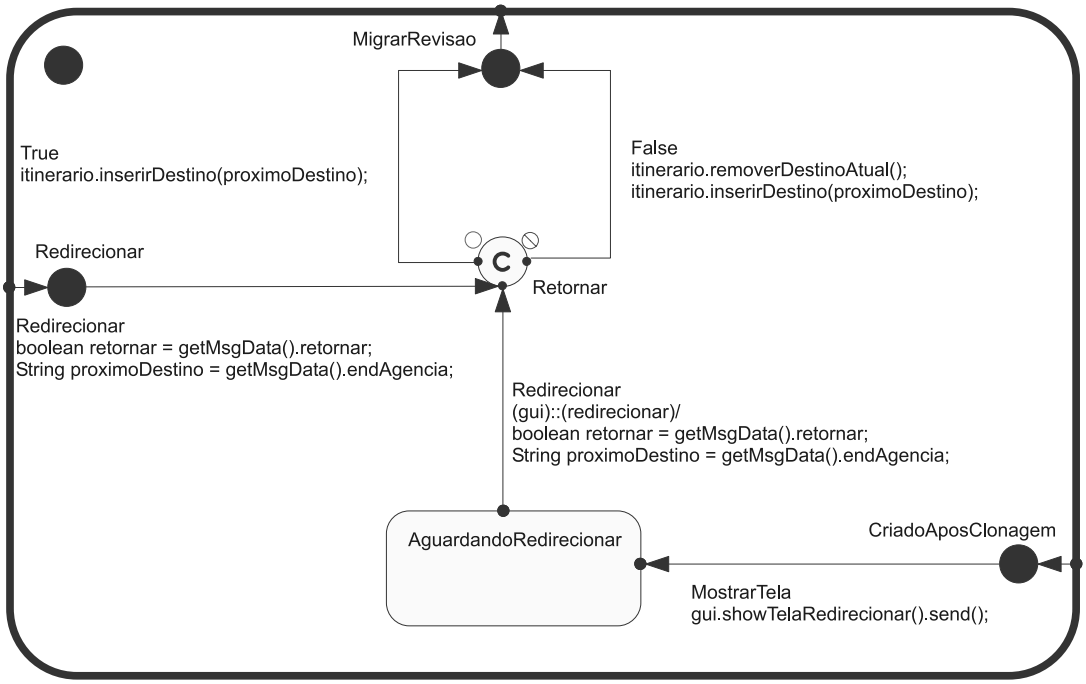

Figura 5.14: Detalhamento do Estado **EMDISTRIBUICAO** do Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

Chegando no estado **EMDISTRIBUICAO** (Figura 5.14), o agente solicita à sua interface gráfica que ela mostre a tela de redirecionamento ao usuário e fica aguardando (estado **AguardandoRedirecionar**) que este entre com os dados. Ao entrar com os dados de redirecionamento (transição **Redirecionar**), o ponto de escolha **Retornar** será avaliado de acordo com o valor do dado *retornar*. Caso seja para retornar a esta agência (transição **True**), o endereço da agência atual permanece no itinerário e o endereço da próxima agência é adicionado. Caso contrário, o endereço da agência atual é retirado e a próxima é adicionada ao itinerário.

Estando no estado **EMREVISAO** (Figura 5.15) e, ao chegar na agência do revisor, o agente passa do estado **AguardandoChegada** para **AguardandoRevisao** através da transição **MostrarTela** que ainda solicita à interface gráfica que esta mostre a tela de revisão ao usuário. Neste estado, o revisor poderá redirecionar o agente para um outro revisor (transição **Redirecionar**) ou entrar com

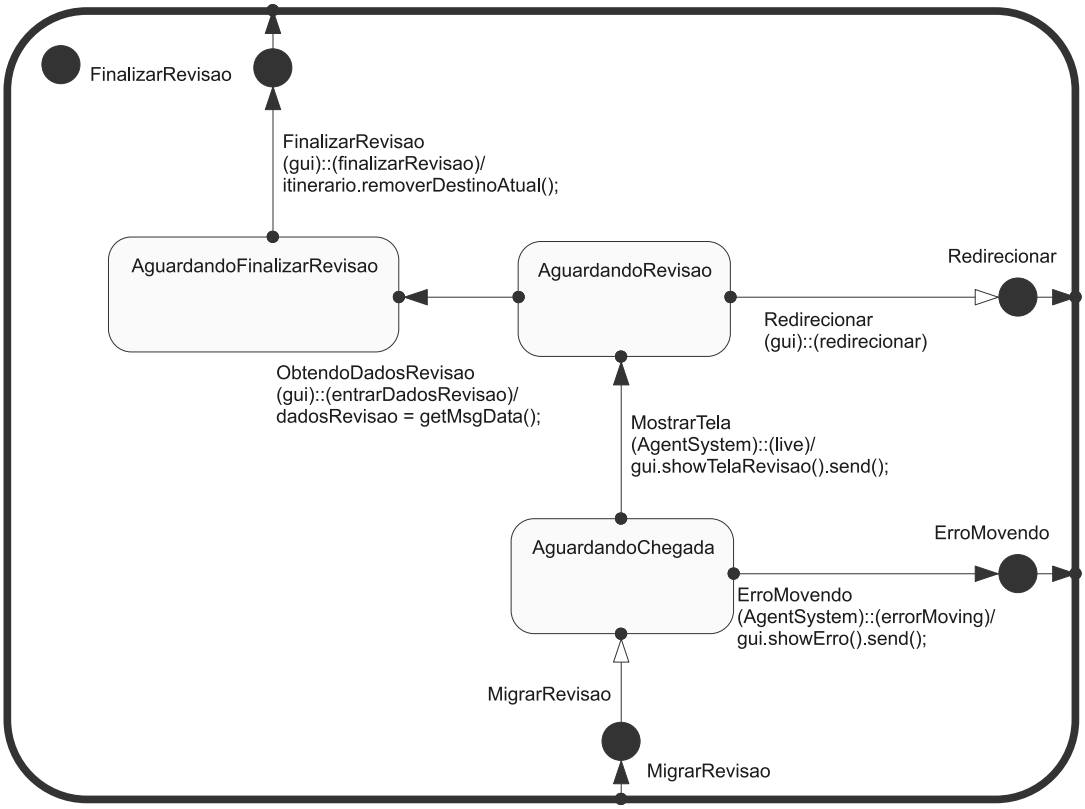

Figura 5.15: Detalhamento do Estado **EMREVISAO** do Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

dados de revisão e, posteriormente, finalizar a revisão com a mensagem *"finalizarRevisao()"*, o que levará o agente ao estado **EMAPROVACAO** através do ponto de junção **FinalizarRevisao**.

Ao concluir a revisão, o ponto de escolha **DestinoNull** (Figura 5.12) verificará se algum usuário solicitou a aprovação da revisão. Caso não haja agências no itinerário (transição **False**), o agente migrará para a agência do usuário que solicitou o seu retorno e entrará no estado **EMAPROVACAO**. Caso contrário ele migrará para a agência do coordenador para registrar o resultado. No estado **EMAPROVACAO** (Figura 5.16), o agente aguardará sua chegada na agência de destino e, quando isto ocorrer (chegada da mensagem *"live()"*), solicitará a apresentação da tela de aprovação (mensagem *"showTelaAprovacar()"*). Após isto, aguardará a aprovação da revisão (mensagem *"aprovarRevisao()"*) que o levará novamente ao ponto de escolha **DestinoNull**.

Tendo a estrutura do modelo e o seu comportamento descrito, o último diagrama a ser mostrado é um diagrama de colaboração apresentando uma possível configuração inicial do sistema e que pode ser visto na Figura 5.17. Nela encontramos duas agências de revisores (*agcRev1* e *agcRev2*), a agência do coordenador (*agcCoord*) e a agência do membro de comitê (*agcMemb*), todas ligadas, por suas portas **Internet**, ao objeto *net*. Além disso, vemos o agente *agentCoord* localizado na agência *agcCoord* que está ligado à conferência (*conf*). Note que não há algum *AgenteFormRevisao* já que estes serão criados durante a execução do sistema.

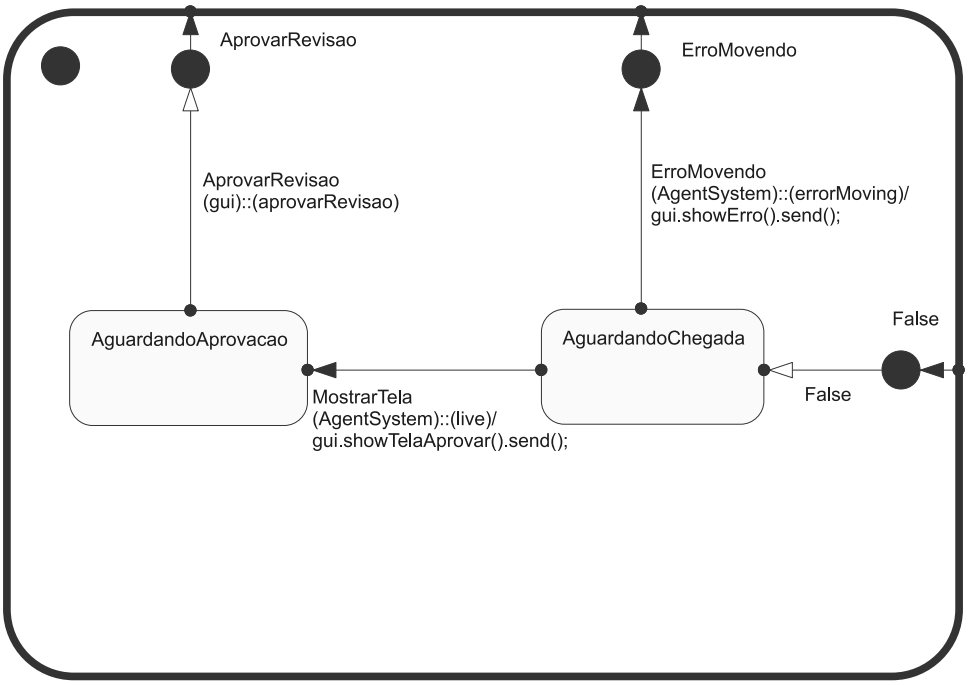

Figura 5.16: Detalhamento do Estado **EMAPROVACAO** do Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

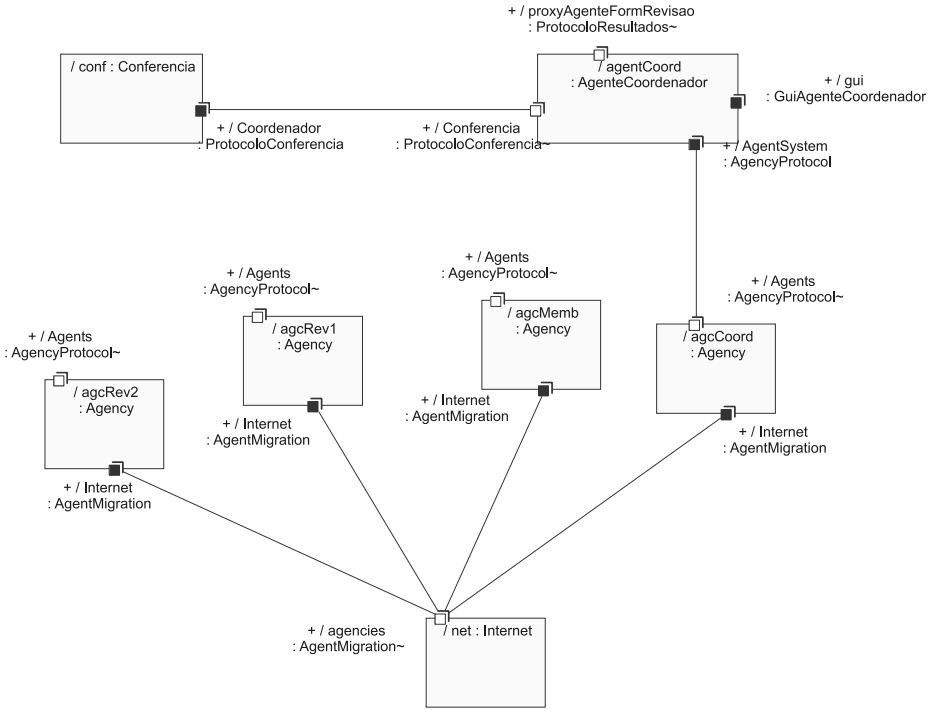

Figura 5.17: Diagrama de Colaboração UML-RT para uma Possível Configuração Inicial do Sistema

Em geral, os modeladores de SBAMs não costumam colocar em seus modelos o tratamento de erros como, por exemplo, indisponibilidade de agências, endereços de localidades incorretos, etc. No

entanto, tal prática se torna interessante pois, além de enriquecer o modelo, irá propiciar a geração de casos de teste específicos para testar partes conhecidamente críticas dos sistemas, que são os tratamentos de erros. Como dissemos no começo desta seção, a construção dos modelos UML-RT foi totalmente baseada nos modelos UML contidos em [Gue02a], que por sua vez não apresentavam o tratamento desses erros acima citados. Para manter a fidelidade aos modelos de Guedes, optamos por não acrescentar o tratamento desses erros, pois para isso precisaríamos ou acrescentar um comportamento que acreditaríamos ser o esperado, ou observaríamos o sistema em execução, bem como seu código, para descobrir o comportamento. No entanto, isto não seria interessante para o nosso trabalho pois, poderíamos encontrar falta de conformidade que não significariam faltas no código (e sim no comportamento adicionado por nós aos modelos), ou estaríamos modelando exatamente o código do sistema o que não nos levaria a qualquer detecção de falta de conformidade entre modelo e código. Sendo assim, o comportamento colocado nos nossos modelos corresponde ao mínimo esperado para um agente móvel, que é o de detectar que uma agência está indisponível e entrar em um estado de erro, enviando ao usuário uma mensagem de erro.

É importante salientar que esta modelagem está especialmente interessada em características de mobilidade dos agentes, abstraindo outras questões como persistência, autenticação de usuário, etc. Contudo, esta abstração não se fez necessária pelo fato de estarmos utilizando UML-RT ou por causa do método de teste que estamos propondo, mas sim por restrições de espaço e de tempo do nosso trabalho como um todo.

5.3 Criação de Modelos RPOO a partir de Modelos UML-RT

Uma vez de posse do modelo em UML-RT, o primeiro passo para a aplicação do método de geração automática de casos de teste é o de converter esse modelo em um modelo RPOO através da transformação apresentada no Capítulo 4. Nesta seção, mostramos esta conversão inicialmente pelo modelo estrutural (diagrama de classes) e posteriormente pelo modelo comportamental, através de transformações das máquinas de estados em redes de Petri.

5.3.1 Modelo Estrutural

O diagrama de classes a ser convertido é o que foi apresentado nas Figuras 5.2, 5.3, 5.4, 5.5 e 5.6, e que pode ser visto por completo no Apêndice A. O processo de conversão do diagrama de classes em UML-RT para um em RPOO é simples e consiste basicamente em acrescentar elementos antes omitidos (e.g. métodos para o recebimento de sinais de entrada nas cápsulas) e agrupar sinais de entrada e saída em um único compartimento nos protocolos. Lembre-se que cápsulas e protocolos são classes e que portas são relacionamentos de composição. Neste sentido, a Figura 5.18 apresenta o diagrama de classes em RPOO resultante da transformação.

Note que os estereótipos de UML-RT foram mantidos, como sugerido no Capítulo 4, para facilitar o entendimento do diagrama. As alterações mais importantes em relação ao diagrama em UML-RT

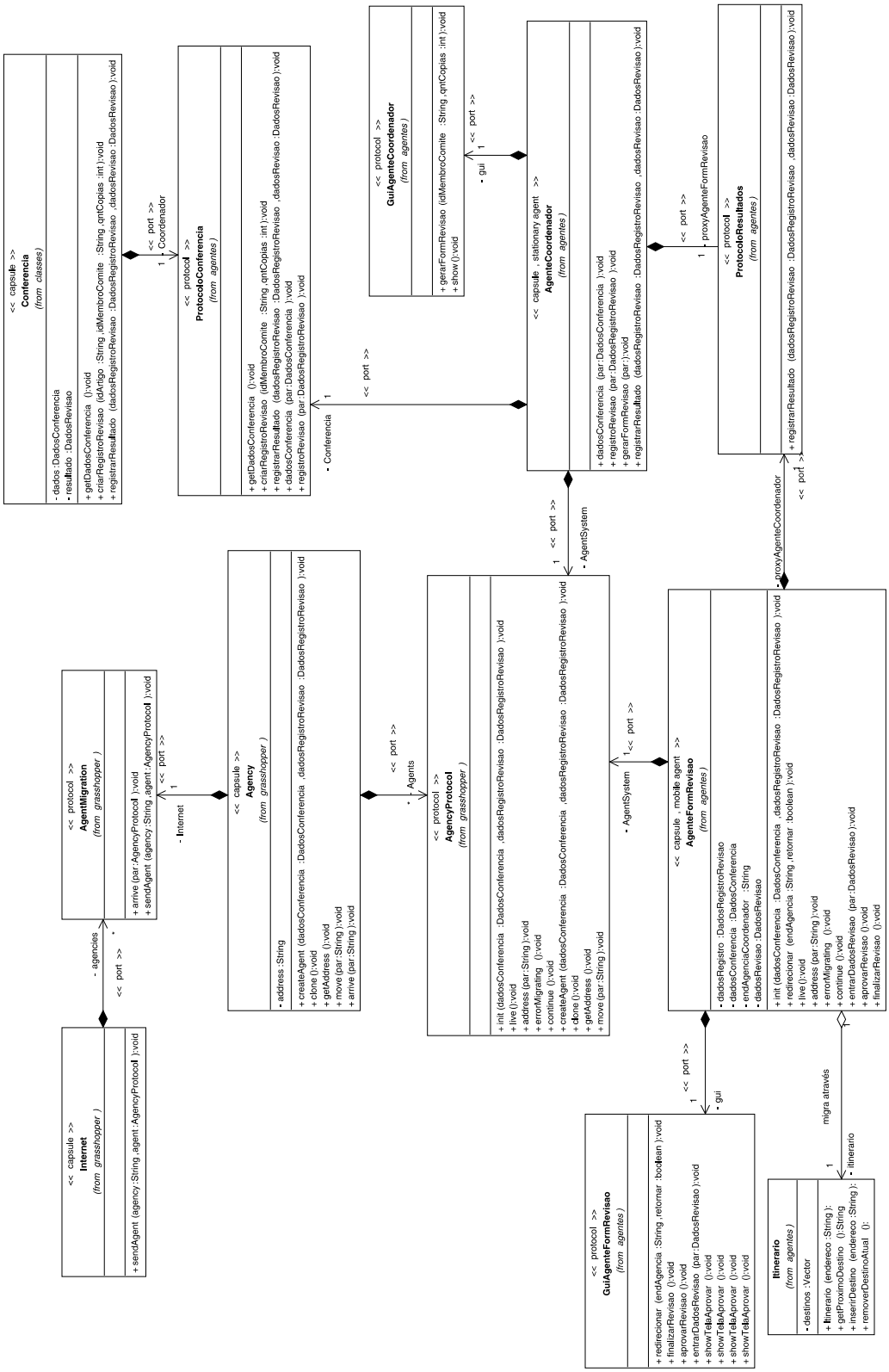

Figura 5.18: Diagrama de Classes RPOO para Sistema de Conferências

são as seguintes.

- Adição de um método para cada sinal de entrada nas cápsulas. Por exemplo, à cápsula **Agente-Coordenador**, foi adicionado o método *"registrarResultado()"*, uma vez que o sinal de mesmo nome no protocolo **ResultadosProtocol** é um sinal de entrada. Isto é feito para que as portas (instâncias dos protocolos) tenham como informar às cápsulas a chegada de um determinado sinal, no caso chamando este seu método;
- Retirada dos métodos privados das cápsulas. Em RPOO, não há a necessidade de se colocar métodos privados em classes, uma vez que um objeto não envia mensagens a ele mesmo e, como esses métodos descrevem comportamentos internos dos objetos, seu comportamento estará descrito na rede de Petri que modela o comportamento da classe como um todo.

É importante dizer que, quando falamos em diagrama de classes RPOO ou diagrama de classes UML-RT, estamos falando em diagrama de classes UML padrões, já que RPOO e UML-RT se utilizam de tal diagrama. No entanto, insistimos com a nomenclatura de "diagrama de classes RPOO" e "diagrama de classes UML-RT" no intuito de facilitar o entendimento do leitor quando desejamos diferenciar os diagramas utilizados nos modelos UML-RT dos utilizados em RPOO. Neste sentido, alguém poderia questionar o fato de que o diagrama de classes apresentado por Guedes em [Gue02a] (UML padrão) não é igual ao diagrama de classes RPOO (UML padrão). Isto é devido ao fato de que este último advém de uma transformação de um modelo UML-RT, que por sua vez está mais focado nas características de distribuição e independência das entidades envolvidas, mais precisamente dos agentes e demais entidades de suporte à sua execução.

5.3.2 Modelo Comportamental

Feita a conversão do modelo estrutural (diagrama de classes), o próximo passo consiste em realizar a transformação dos diagramas de estados de UML-RT para as redes de Petri de RPOO. Dados os diagramas de estados das Figuras 5.7, 5.8, 5.9, 5.10 e 5.12, além dos diagramas de estados que detalham estados, foi construída uma rede de Petri para cada um desses diagramas seguindo os passos apresentados no Capítulo 4. No entanto, nesta seção, apresentaremos apenas a rede de Petri relativa à classe **AgenteFormRevisao**, uma vez que sua máquina de estados possui todos os elementos importantes para a apresentação da transformação. Caso o leitor deseje, toda a transformação pode ser encontrada no Apêndice A.

Segundo o procedimento de transformação contido no Capítulo 4, cada protocolo seria transformado em uma classe e, conseqüentemente, possuiria uma rede de Petri para descrever o seu comportamento. Entretanto, como os protocolos apresentados no sistema de conferências não possuem um diagrama de estados, servindo apenas como retransmissores de mensagens entre as cápsulas envolvidas, e com o intuito de simplificar[2] o modelo, os protocolos não possuem uma rede de Petri que os descrevem e o seu comportamento estará distribuído pelas cápsulas que participam do protocolo.

[2]Simplificar segnifica ter menos classes para dar manutenção e, principalmente, ter um espaço de estados menor

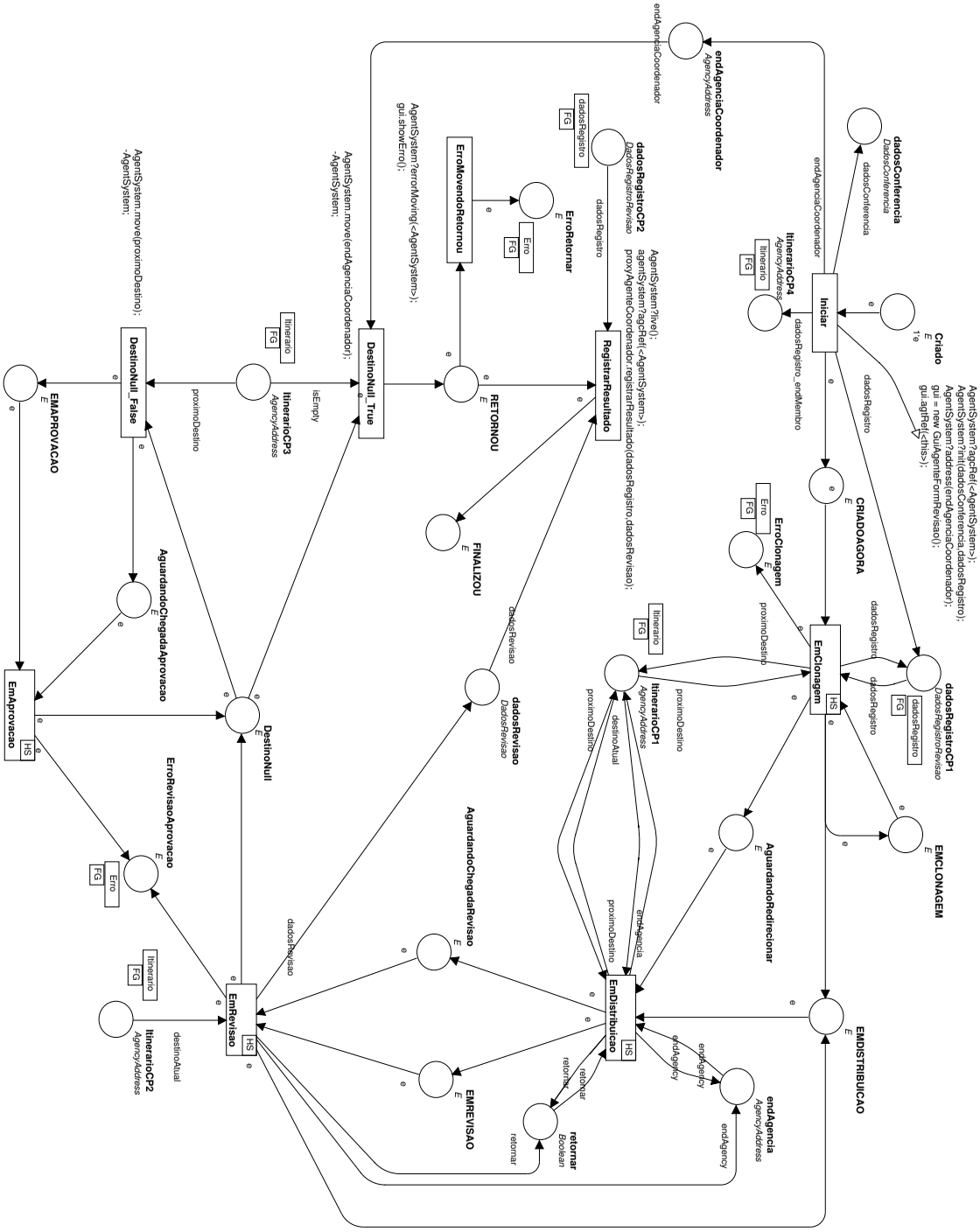

Figura 5.19: Página Principal da Rede de Petri que Descreve o Comportamento da Classe **AgenteFormRevisao**

A Figura 5.19 apresenta a página principal da rede de Petri da classe **AgenteFormRevisao**. Assim como foi feito na modelagem do diagrama de estados da cápsula **AgenteFormRevisao**, a respectiva rede de Petri também foi hierarquizada, acrescentando-se transições de substituição [3]. As transições

[3]Uma transição de substituição é uma transição cujo comportamento abstrai a execução de uma outra rede de Petri, as sim como um estado que possui um outro diagrama de estados que detalhe o comportamento de um objeto.

de substituição são as transições **EmClonagem**, **EmDistribuicao**, **EmRevisao** e **EmAprovacao**, e as suas relativas redes de Petri são as redes apresentadas nas Figuras A.8, A.9, A.10 e A.11 contidas no Apêndice A.

No diagrama de classes apresentado na Figura 5.18, podemos ver que a classe **Itinerario** está agregada à classe **AgenteFormRevisao**. Como foi explicado anteriormente, esta classe abstrai um lista de endereços de agências por onde o agente **AgenteFormRevisao** deverá migrar. Por ser um comportamento simples, inclusive não é uma cápsula e nem possui diagrama de estados para descrevê-la, sua modelagem em RPOO foi feita através de um lugar na rede de Petri da classe **AgenteFormRevisao** chamado **Itinerario**, e cuja cor é *AgencyAddress*. Desta forma, os endereços de agências que estiverem neste lugar deverão ser percorridos pelo agente e os arcos de entrada e de saída originados ou destinados a este lugar representarão as operações da classe **Itinerario**.

Como pode ser visto nas redes de Petri, para cada estado do respectivo diagrama de estados e para cada atributo da cápsula, um lugar com o mesmo nome foi adicionado à rede de Petri, bem como para cada transição da máquina de estados existe uma transição na rede de Petri com mesmo nome. Os pontos de escolha também tiveram sua tradução e estão presentes na rede através de lugares e transições. Iremos explicar a rede de Petri da classe **AgenteFormRevisao** primeiramente apresentando a rede principal e depois mostrando as redes que descrevem as transições de substituição.

Em geral, os modelos em UML-RT são mais abstratos que modelos em RPOO. Consequentemente, em alguns pontos da transformação se faz necessário a adição de alguns elementos por parte de quem está fazendo a conversão, como, por exemplo, a modelagem do comportamento de métodos privados. No modelo em UML-RT, a cápsula **AgenteFormRevisao** possui um método chamado *"criarGui()"* que tem a função de criar a interface gráfica do agente. Em RPOO, este método não mais está presente no modelo e, sempre que há uma chamada a este método, a ação RPOO *"gui = new GuiAgenteFormRevisao();"* é acrescentada, como pode ser visto na transição **Iniciar** da Figura 5.19. Com isto, o comportamento do método *"criarGui()"*, antes abstrato, agora é concreto.

Outro ponto de concretização decorre do fato de não estarmos utilizando portas para fazermos a comunicação entre as cápsulas, pois, como foi dito, não há a necessidade. Caso estivéssemos utilizando-se de portas, estas seriam responsáveis por gerenciar as referências às outras portas nos momentos de criação, ligação e desligação das cápsulas. Por exemplo, quando um agente é criado por uma agência, as portas **AgentSystem** e **Agents** se encarregam de obter a referência uma da outra para que possam trocar mensagens. Isto não é colocado explicitamente nos modelos UML-RT. Já em RPOO, como iremos simular os modelos e gerar o seu espaço de estados, precisamos que isto esteja nos modelos. Sendo assim, quando o agente é criado, o objeto da classe **AgenteFormRevisao** necessita de uma referência para o objeto que o criou (classe **Agency**). Isto é feito com o recebimento da mensagem *"agcRef()"* por meio da inscrição *"AgentSystem?agcRef(<AgentSystem>);"*, que tem esta semântica, na transição **Iniciar**.

No intuito de diminuir o espaço de estados final do modelo e de simplificar a rede de Petri da classe **AgenteFormRevisao**, algumas transições foram suprimidas. Como pode ser visto na rede

de etri da Figura 5.19, a transição **Iniciar** é resultado da conversão de duas transições do diagrama de estados, as transições **Iniciar** e **ReceberEndCoordenador**. Desta forma, abstraimos o estado **Inicializando** da classe **AgenteFormRevisao**.

Ao disparar a transição **Iniciar**, dados serão colocados nos lugares **Itinerario**, **dadosConferencia** e **dadosRegistro**, além de uma ficha no lugar **CRIADOAGORA**, significando que o agente passou para o estado **CRIADOAGORA**. Neste estado, a transição de substituição **EmClonagem** poderá disparar e isto resultará em uma ficha colocada no lugar **Erro** ou em fichas nos lugares **EMDISTRIBUICAO** e **AguardandoRedirecionar**. Com este último modo de disparo, a transição **EmDistribuicao** estará habilitada para o disparo. Após seu disparo, a transição **EmRevisao** disparará e colocará uma ficha no lugar **DestinoNull**. Este lugar, em conunto com as transições **DestinoNull_True** e **DestinoNull_False**, modela o ponto de escolha **DestinoNull** do diagrama de estados da cápsula **AgenteFormRevisao**. O lugar **DestinoNull** significa que, caso uma ficha esteja lá, o ponto de escolha deverá ser avaliado. As transições **DestinoNull_True** e **DestinoNull_False** modelam as transições do diagrama de estados resultante da avaliação do ponto de escolha para verdadeiro e falço, respectivamente. Note que o teste do ponto de escolha (*"proximoDestino == null"*) é modelado com os arcos de entrada das transições. Caso a transição **DestinoNull_False** dispare, em seguida a transição de substituição **EmAprovacao** estará habilitada para disparo, e, caso **DestinoNull_True** dispare, a transição **RegistrarResultado** estará habilitada para disparo, registrando o resultado da execução do agente e o levando ao estado **FINALIZOU**.

Como foi dito quando apresentamos o modelo UML-RT do sistema, as interfaces gráficas dos agentes não estavam modeladas, e sim a forma com que elas irão interagir com os agentes através dos protocolos. No entanto, a geração do espaço de estados para tal modelo resultaria em uma explosão do espaço de estados[4], uma vez que toda e qualque possível interação seria gerada.

Como foi dito no começo desta seção, não apresentamos as redes de Petri pra todas as classes e sim a rede de Petri principal da classe **AgenteFormRevisao**, uma vez que o intuito da seção, bem como o do capítulo, é o de ilustrar uma aplicação do método mostrando diversos elementos utilizados para tal conversão e não o de apresentar, em detalhes, o sistema trabalhado. Uma importante conclusão a respeito da tranformção dos modelos em UML-RT para modelos RPOO é que, assim como foi dito no Capítulo 4 e pode ser visto pelo que foi apresentado, tal transformação é simples, uma vez que o mapemento entre os elementos dos modelos é claro e, na maioria das vezes, direto. Por exmplo, os mapementos entres os estados e atributos dos diagramas de estados em lugares nas redes de petri, transições dos diagramas de estados em transições nas redes de Petri, e pontos de escolha em lugares e transições.

[4]Este termo é utilizado quando o espaço de estados é tão grande a ponto que as ferramentas de geração, verificação de modelos e outras não conseguem tratá-lo ou o tempo e a capacidade de processamento é inviável.

5.4 Conclusão

Neste capítulo, apresentamos a modelagem do sistema utilizado em nosso estudo de caso em UML-RT, e mostramos a criação de um modelo RPOO a partir deste. Dentro do contexto de geração de casos de teste, proposto neste trabalho, este é um passo que visa aproximar os desenvolvedores de SBAMs ao método de geração automática de casos de teste proposto do Capítulo 3. Este é um passo inicial, porém bastante importante para uma futura obtenção de um método automático de geração de casos de teste para este tipo de sistema a partir de modelos UML-RT. A transformação mostrada neste capítulo, além de facilitar uma imediata aplicação do método por parte dos desenvolvedores de sistemas distribuídos, motiva a automação do processo, uma vez que mostra a semelhança entre as linguagens envolvidas de um ponto de vista do modelador de sistemas.

Capítulo 6

Estudo de Caso - Parte 2

6.1 Introdução

Este capítulo complementa o Capítulo 5, mostrando o restante da aplicação do estudo de caso referente ao método de geração de casos de teste proposto neste trabalho. Mais especificamente, as etapas apresentadas no Capítulo 3 serão ilustradas por meio de suas aplicações no sistema apresentado no Capítulo 5. O modelo RPOO obtido neste capítulo será utilizado como entrada para as etapas de simulação e geração de espaço de estados que, por conseqüência, propiciará a geração dos objetivos de teste, geração dos casos de teste, e implementação e execução destes casos de teste.

Na Seção 6.2, mostraremos a geração do de espaço de estados a partir dos modelos RPOO. Com o espaço de estados, na Seção 6.3 apresentamos alguns objetivos de teste, mostrando como eles foram selecionados com o intuito de servirem de entrada para a geração de caso de testes, mostrada na Seção 6.4. Tais casos de teste foram implementados e executados sobre a aplicação, como pode ser visto na Seção 6.5. Por fim, as considerações finais são apresentadas (Seção 6.6).

6.2 Simulação e Geração de Espaço de Estados dos Modelos RPOO

Uma vez de posse dos modelos RPOO, o passo seguinte do estudo de caso foi o de realizar algumas simulações no modelo no intuito de ganhar confiança de que realmente este modelo reflete o sistema especificado por Guedes [Gue02a] e, posteriormente, gerar o seu espaço de estados para que sirva de entrada para a ferramenta de geração de casos de teste TGV.

A ferramenta utilizada para a simulação e geração de espaço de estados é a JMobile Tools. O modelo foi instanciado em memória com o uso de sua API e fornecido como parâmetro para o simulador e gerador de espaço de estados. Desta forma, os modelos foram convertidos do formato gráfico apresentado na seção anterior para classes Java a serem instanciadas em memória e submetidas à ferramenta. Este processo de conversão foi feito manualmente e não iremos apresentá-lo aqui uma vez que, com o advento de uma ferramenta de modelagem de RPOO, é esperado que este processo seja realizado de forma automática por tal ferramenta. Sendo assim, a seguir mostraremos como a

simulação foi realizada sobre os modelos, apresentado seus resultados. Em seguida, mostraremos como foi realizada a geração de espaço de estados, apresentando algumas considerações sobre este espaço de estados.

Tanto as simulações quanto a geração do espaço de estados foram feitas partindo do cenário inicial apresentado pelo diagrama de colaboração da Figura 5.17. Além disso, por questões de limitações do gerador de espaço de estados, a quantidade de revisões para um artigo foi de no máximo dois, ou seja, no máximo dois agentes ***AgenteFormRevisao*** estariam executando ao mesmo tempo.

6.2.1 Simulação dos Modelos RPOO

A simulação dos modelos foi realizada através de uma interface textual que disponibiliza informações sobre o estado atual do sistema e as transições habilitadas para disparo. Com esta interface, é possível ver as transições habilitadas para cada estado e escolher uma para executar. Com a execução da transição, um novo estado (é possível que seja igual ao anterior) é apresentado e as transições habilitadas para aquele estado são disponibilizadas.

Durante a simulação, alguns erros de modelagem foram encontrados. Haja visto que o processo de obtenção dos modelos RPOO a partir dos modelos UML-RT não é feito de forma automática, alguns erros podem ser inseridos nos modelos por parte do engenheiro responsável pela transformação. Além disso, o processo de concretização citado anteriormente também pode acrescentar erros ao modelo.

Uma vez que os modelos em UML podem conter ambigüidades e serem incompletos, a geração dos modelos UML-RT também pode inserir faltas no modelo. Estas faltas também foram encontradas durante o processo de simulação, e suas relativas correções foram feitas tanto no modelo UML-RT como, conseqüentemente, no modelo RPOO.

Como em todo processo de teste baseado em modelos, a existência de faltas no modelo é um problema, uma vez que, caso tenhamos modelos incorretos, os testes retirados a partir destes poderão rejeitar aplicações corretas, já que os testes estarão incorretos. Como no nosso caso o modelo é formal (RPOO), algumas técnicas (verificação de modelos, simulação de modelos, etc.) podem ser utilizadas no intuito de aumentarmos a nossa confiança na correção do modelo. Em relação ao nosso estudo de caso, por se tratar de um modelo relativamente simples, com poucas classes e objetos, apenas a simulação foi o suficiente para nos dar confiança sobre o modelo.

Uma vez que os erros encontrados através da simulação foram simples, onde basicamente refletiam erros inseridos pelo modelo por questões de mal entendimento de especificações, etc., não entraremos em detalhes a respeito dos mesmos. No entanto, a descoberta de erros no modelo UML-RT e suas correções nos mostraram que a manutenção de dois modelos (UML-RT e RPOO) foi simples, com base no procedimento sistemático apresentado no Capítulo 4. Este procedimento se tornou útil também durante a manutenção, onde foi utilizado para aplicar as correções feitas nos modelos UML-RT aos modelos RPOO, preservando a consistência entre os modelos.

6.2.2 Geração de Espaço de Estados dos Modelos RPOO

Para a geração do espaço de estados do modelo RPOO do sistema de nosso estudo de caso, o gerador apresentado na ferramenta JMobile Tools foi utilizado. Como dissemos no Capítulo 3, este gerador recebe os modelos RPOO e gera o espaço de estados através de um processo totalmente automático, que não exige qualquer tipo de intervenção humana. O espaço de estados do modelo contém 85.325 estados e 175.573 transições, e foi gerado em um tempo aproximado de 20 horas e 25 minutos, em um computador com processador de 1.8Ghz e 1Gb de memória RAM. Este espaço de estados foi gerado para um modelo onde:

- um artigo poderia ter uma ou duas revisões, consequentemente um ou dois agentes de formulário de revisão estaria executando ao mesmo tempo;
- além do membro de comitê, um revisor poderia redirecionar o formulário para um outro revisor, solicitando ou não o seu retorno.

Uma vez que o espaço de estados do sistema sem quaisquer restrições, ou seja, para um número indeterminado de formulários de revisão e um número indeterminado de revisores, seria infinito, houve a necessidade de se assumir tais restrições apresentadas acima. Entretanto, tais restrições não comprometem o estudo realizado neste capítulo, pois preservam o comportamento dos agentes envolvidos.

O espaço de estados gerado por JMobile é um LTS, desta forma possuirá transições que representam entradas, saídas e ações internas do sistema de uma forma não distinta. No entanto, estamos tratando de casos de teste funcionais, de forma que os pontos de controle (entradas) precisam ser diferenciados dos pontos de obsevação (saídas), e as ações internas precisam ser ocultadas, já que não aparecerão nos casos de teste gerados.

O processo de ocultar as ações internas, conhecida na comunidade por *hiding*, se dá com o uso de uma ferramenta (contida no pacote de TGV) que recebe o LTS do sistema e um arquivo que descreve os pontos de controle e de observação, e resulta como saída em um outro LTS que se diferencia do primeiro no fato de ter todas as transições referentes a ações internas rotuladas com um rótulo especial: *"i"*. Já o processo de diferenciar os pontos de controle dos pontos de observação é feito no momento da geração dos casos de teste propriamente dito, onde um dos parâmetros para tal geração é um arquivo contendo a descrição, em separado, dos pontos de controle e de observação do sistema de conferências. Desta forma, é preciso definir quais serão os pontos de controle e os pontos de observação, deixando, por conseqüência, as demais como ações internas.

A definição dos pontos de controle e de observação de um sistema está diretamente relacionada com o domínio do problema. Para o caso específico de agentes móveis e como estamos especialmente interessados no comportamento dos agentes, os pontos de controle serão aqueles onde as entidades externas aos agentes (agências, interfaces gráficas e demais entidades do sistema, e.g. objetos da classe **Conferencia**) enviam mensagens a estes, e os pontos de observação serão aqueles onde os

Pontos de Controle	
*.*agentCoord\.dadosConferencia(.*).**	Mensagem enviada para o ***AgenteCoordenador*** com os dados da conferência.
*.*agentCoord\.registroRevisao(.*).**	Mensagem enviada para o ***AgenteCoordenador*** com os dados do registro da revisão.
*.*agentCoord\.gerarFormRevisao(.*).**	Mensagem enviada para o ***AgenteCoordenador*** solicitando a geração do formulário de revisão.
*.*agent.*\.init(.*).**	Mensagem enviada para um agente com os dados de inicialização.
*.*agent.*\.live(.*).**	Mensagem enviada para um agente habilitando a sua execução.
*.*agent.*\.address(.*).**	Mensagem enviada para um agente informando o endereço da agência onde este está executando.
*.*agent.*\.errorMoving(.*).**	Mensagem enviada para um agente informando falha no processo de migração.
*.*agent.*\.continue(.*).**	Mensagem enviada para um agente informando que a clonagem foi efetuada.
*.*agentForm.*\.redirecionar(.*).**	Mensagem enviada para o ***AgenteFormRevisao*** solicitando o seu redirecionamento.
*.*agentForm.*\.entrarDadosRevisao(.*).**	Mensagem enviada para o ***AgenteFormRevisao*** com os dados de revisão.
*.*agentForm.*\.finalizarRevisao(.*).**	Mensagem enviada para o ***AgenteFormRevisao*** solicitando que a revisão seja finalizada.
*.*agentForm.*\.aprovarRevisao(.*).**	Mensagem enviada para o ***AgenteFormRevisao*** aprovando sua revisão.

Tabela 6.1: Tabela Contendo os Pontos de Controle do Sistema de Conferências

agentes enviam mensagens a tais entidades externas. As Tabelas 6.1 e 6.2 mostram, respectivamente, as ações que definem os pontos de controle e de observação para o sistema de conferências trabalhado neste capítulo.

Note que a definição dos pontos de controle e de observação é feita com o uso de expressões regulares. Por exemplo, a expressão *".*agent.*\.live(.*).*"* informa que uma mensagem *live* pode ser enviada a qualquer agente (*agent.**) cujo nome comece com *agent* e com quaisquer parâmetros (*live(.*)*). Além disso, a presença de um *'.*'* no começo e no fim permite que possam existir outras ações na mesma transição, e o caracter *'\'* é utilizado como de escape para o caracter especial *'.'*. Neste sentido, a transição que contém o rótulo apresentado abaixo será um ponto de controle, devido à ação acima citada.

Pontos de Observação	
*.*conf\.getDadosConferencia(.*).**	Mensagem enviada ao objeto da classe **Conferencia** solicitando os dados da conferência.
*.*conf\.criarRegistroRevisao(.*).**	Mensagem enviada ao objeto da classe **Conferencia** solicitando a criação de um registro de revisão.
*.*conf\.registrarResultado(.*).**	Mensagem enviada ao objeto da classe **Conferencia** solicitando qe os resultados da revisão sejam registrados.
*.*gui.*\.show.*(.*).**	Mensagem enviada a uma interface gráfica solicitando que uma determinada tela seja apresentada.
*.*agc.*\.getAddress(.*).**	Mensagem recebida por uma agência solicitando o seu endereço.
*.*agc.*\.move(.*).**	Mensagem recebida por uma agência solicitando que um agente se mova para uma determinada agência.
*.*agc.*\.clone(.*).**	Mensagem recebida por uma agência solicitando que um agente seja clonado.
*.*agc.*\.createAgent(.*).**	Mensagem recebida por uma agência solicitando que um agente seja criado.
*.*agentCoord\.registrarResultado(.*).**	Mensagem enviada pelo ***AgenteFormRevisao*** ao ***AgenteCoordenador*** solicitando que este uma revisão seja registrada.

Tabela 6.2: Tabela Contendo os Pontos de Observação do Sistema de Conferências

agcRev1:net?arrive(agentFormOriginal) & *agcRev1:agentFormOriginal.live()* & agcRev1:agentFormOriginal.agcRef(agcRev1)

O fato de poder existir mais de uma mensagem em uma transição poderia permitir que, em uma mesma transição, encontrássemos uma mensagem que define um ponto de controle e uma outra que define um ponto de observação, impedindo a classificação de tal transição em um ponto de controle ou de observação do sistema. No entanto, isto não acontecerá para o caso de mensagens assíncronas, uma vez que as ações que definem os pontos de controle nunca estarão em uma mesma transição que ações que definem pontos de observação. Isto ocorre devido ao fato de que os pontos de controle são definidos por ações de envio de mensagem de entidades externas aos agentes, e os pontos de observação por ações de envio de mensagens de agentes, desta forma, em cada transição do espaço de estados, ou uma transição RPOO de um agente irá disparar ou uma transição de entidades externas disparará.

Contudo, poderemos ter mensagens de sincronização que poderia fazer com que duas transições, uma de um agente e outra de uma entidade externa, disparasse ao mesmo tempo, resultando em uma transição no espaço de estados com ações de ponto de controle e de observação. Este fato pode ser evitado de forma simples durante a modelagem, colocando mensagens assíncronas no lugar de síncronas. Isto é possível pois, na maioria das plataformas de execução de agentes, a comunicação entre agentes e entidades externas se dá de maneira assíncrona. De toda forma, se isto não for possível e assumindo que as entidades externas possuem um comportamento correto (estamos testando os agentes), tal transição será definida como ponto de observação, pois refletirá um comportamento dos agentes a ser verificado.

6.3 Seleção de Objetivos de Teste

A ferramenta de geração de casos de teste utilizada pelo método (TGV) trabalha com objetivos de teste explícitos, necessitando que os mesmos sejam elaborados pelo testador para que sirvam de entrada para a ferramenta. Os objetivos de teste em TGV são grafos acíclicos que visam selecionar partes específicas do espaço de estados do sistema, em conseqüencia, linhas de execução específicas desejadas para teste. Os objetivos de teste são elaborados tendo como base o IOLTS do sistema, desta forma, eles possuirão apenas transições que representem entradas (pontos de controle) e saídas (pontos de observação) do sistema, e não conterá ações internas, como apresentado no Capítulo 2.

Uma vez que o sistema de nosso estudo de caso se utiliza de padrões de projeto para Agentes Móveis, os nossos objetivos de teste foram elaborados visando testar as partes do sistema que implementam estes padrões de projeto. Ou seja, para cada padrão de projeto utilizado no sistema, um objetivo de teste foi elaborado de forma que casos de teste fossem selecionados para testar a implementação destes padrões pelo sistema. Além disso, um outro objetivo de teste foi elaborado com o objetivo de gerar casos de teste para execuções onde algum problema com a rede ocorre, seja por indisponibilidade de agências, falhas de comunicação na rede ou endereçamento errado.

O processo de obtenção dos objetivos de teste é feito manualmente tendo como base o conhecimento do espaço de estados do sistema e, no caso específico deste estudo de caso, o conhecimento dos padrões de projeto no contexto do sistema. O conhecimento do espaço de estados se resume no conhecimento das ações do sistema (dos rótulos das transições) e em seu seqüenciamento (ordem com que aparecem no espaço de estados). Isto decorre do fato de que os objetivos de teste são grafos que descrevem de forma abstrata um conjunto de linhas de execuções, que são seqüencias de ações. Já o conhecimeto dos padrões de projeto pode ser visto como o entendimento da forma com que o espaço de estados reflete, em uma forma abstrata, o uso destes padrões.

Por exemplo, a Figura 6.1 apresenta um objetivo de teste para o padrão *Itinerary* (apresentado no Capítulo 4). Para construí-lo, foi necessário conhecer que o padrão é utilizado no sistema a partir do momento em que um ***AgenteFormRevisao*** finaliza a sua revisão (ação *".*agenteForm.*\.finalizarRevisao(.*).*"*), seguindo o seqüenciamento de mensagens apresentado, até que o ***AgenteCoordenador*** recebe uma mensagem de registrar resultado de revisão (ação *".*agenteCoord.*\.registrarResultado(.*).*"*). Além disto, foi necessário ter o conhecimento das ações que ocorrem bem como suas ordens. Note que este é um grafo abstrato e que só as ações interessadas estão no grafo, deixando as demais abstraídas nas transições com rótulo *"*"*.

A seguir apresentaremos uma breve descrição dos padrões de projeto para agentes móveis utilizados no sistema e mostraremos os objetivos de teste elaborados para tais padrões: *Itinerary*, *Master-Slave* e *Branching*.

O padrão de projeto *Itinerary* foi apresentado na Seção 3.3.1 por meio de uma aplicação simples. No estudo de caso deste capítulo, este padrão é implementado pelo agente ***AgenteFormRevisao*** quando este percorre uma lista de agências (seu itinerário) cujos revisores requereram a aprovação do formulário de revisão, onde a tarefa a ser executada é a de aprovar as revisões contidas no formulário.

O objetivo de teste para o padrão *Itinerary* pode ser visto nas Figuras 6.1 e 6.2. Pelas figuras, vimos que o objetivo de teste seleciona a parte do sistema que se inicia a partir do momento em que um revisor finaliza a sua revisão (vide transição *".*agenteForm.*\.finalizarRevisao(.*).*"* que parte do estado **'0'** para o estado **'1'**) e então começa o processo de aprovação, que é o trecho do sistema que o padrão *Itinerary* é implementado. Além disso, podemos ver que o objetivo de teste seleciona as linhas de execução em que há pelo menos uma agência no itinerário do agente. Isto é obtido no momento em que pelo menos uma transição *".*gui.*\.showTelaAprovar(.*).*"* precisa ser executada (transição que parte do estado **'2'** para o estado **'3'**), ou seja, há pelo menos uma agência solicitando aprovação.

As Figuras 6.1 e 6.2 apresentam o mesmo objetivo de teste apresentado em formato gráfico e texto, respectivamente. A ferramenta recebe como entrada um arquivo texto (Figura 6.2), no entanto utilizaremos apenas o modo gráfico (Figura 6.1) para ilustrar os objetivos que seguem.

O padrão de projeto *MasterSlave* descreve uma solução onde um agente chamado ***Master*** delega uma determinada tarefa a um agente chamado ***Slave***. No sistema de conferência, este padrão é implementado pelo agente ***AgenteCoordenador*** no papel de ***Master*** e pelo agente ***AgenteFormRevisao*** no

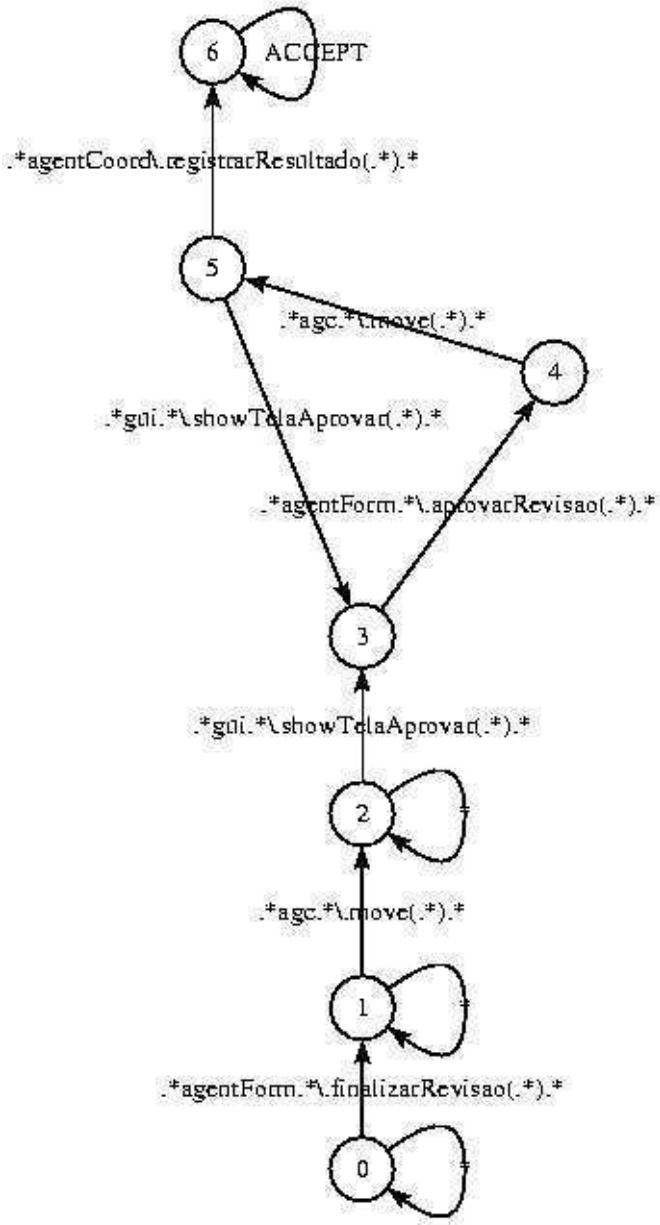

Figura 6.1: LTS do Objetivo de Teste para o Padrão de Projeto *Itinerary* - modo Gráfico

papel de ***Slave***, onde o primeiro delega ao segundo a tarefa de obter as revisões para um determinado artigo. A Figura 6.3 mostra o objetivo de teste para este padrão. Como pode ser visto na figura, com este objetivo de teste selecionamos as execuções em que o agente ***AgenteCoordenador*** cria um ***AgenteFormRevisao*** (vide trasições *".*agcCoord\.createAgent(('dadosconf1',('agcMemb',1),a gentCoord)).*"* e *".*agcCoord\.createAgent(('dadosconf1',('agcMemb',2),a gentCoord)).*"*) e, ao final, os agentes retornam com o resultado de sua tarefa.

O padrão de projeto *Branching* é aplicável a um contexto onde um agente deseja executar uma determinada tarefa em um certo número de agências, tal que esta tarefa pode ser feita de forma paralela. Neste sentido, o padrão apresenta uma solução onde o agente se clona "N - 1" vezes, tal

```
des (0, 10, 5)
(0, *, 0)
(0, ".*agentForm.*\    .finalizarRevisao(.*).*", 1)
(1, *, 1)
(1, ".*gui.*\.showT    elaAprovar(.*).*", 2)
(2, *, 2)
(2, ".*agentForm.*\    .aprovarRevisao(.*).*", 3)
(3, ".*agc.*\.move(    'agcCoord').*", 4)
(3, ".*gui.*\.showT    elaAprovar(.*).*", 2)
(3, *, 3)
(4, ACCEPT,  4)
```

Figura 6.2: LTS do Objetivo de Teste para o Padrão de Projeto *Itinerary* - modo Texto (Aldebaran)

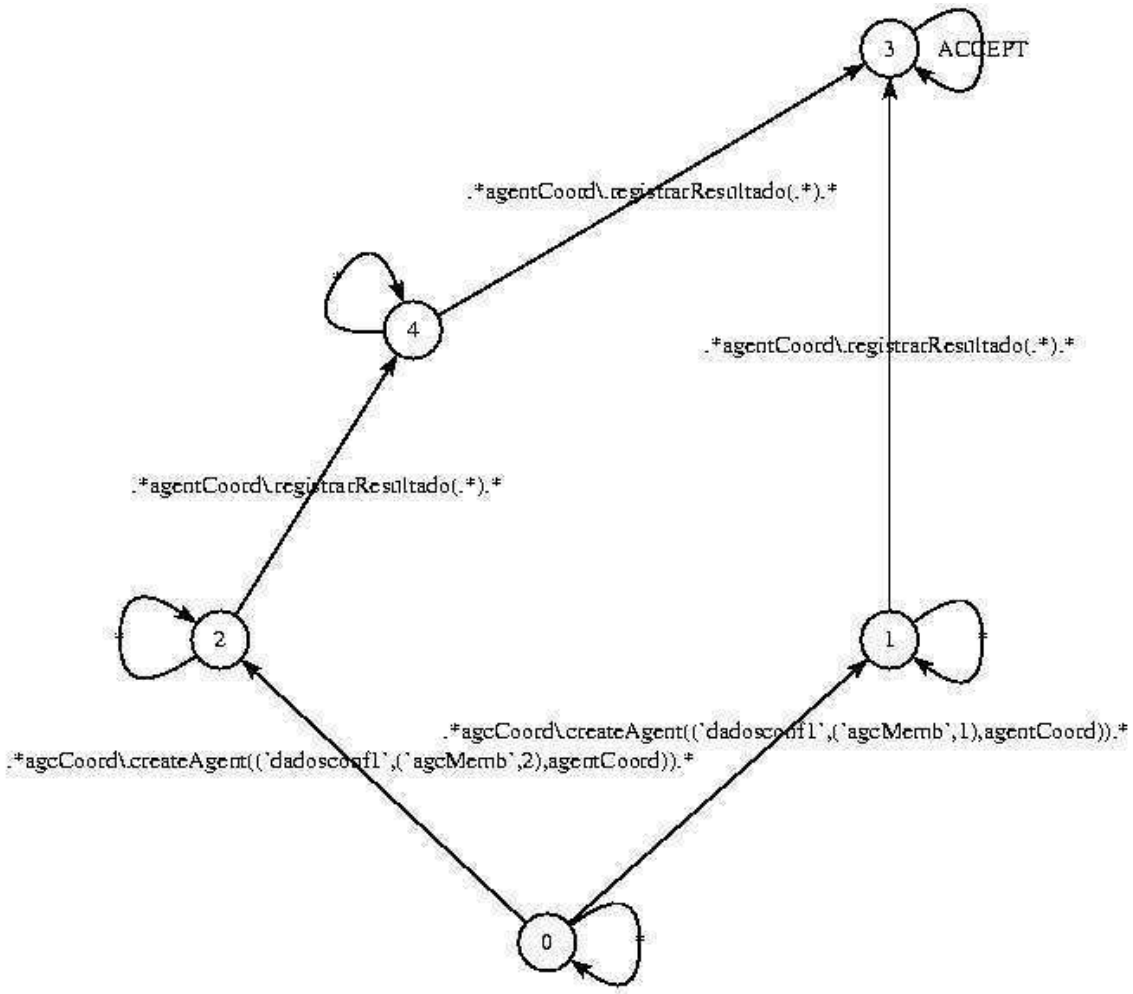

Figura 6.3: LTS do Objetivo de Teste para o Padrão de Projeto *MasterSlave*

que "N" é o número de agências. No sistema de conferências, este padão é utilizado para o caso em que o ***AgenteCoordenador*** solicita ao ***AgenteFormRevisao*** que este obtenha mais de uma revisão para um artigo. Com esta solicitação, o ***AgenteFormRevisao*** irá se clonar o quanto for necessário e cada cópia irá migrar para as agências dos revisores. A Figura 6.4 apresenta o objetivo de teste para este padrão. Como pode ser visto na figura, o objetivo de teste seleciona as execuções em que o agente ***AgenteFormRevisao*** se clona e, tanto este quanto o agente clone retorna à agência do ***AgenteCoordenador***.

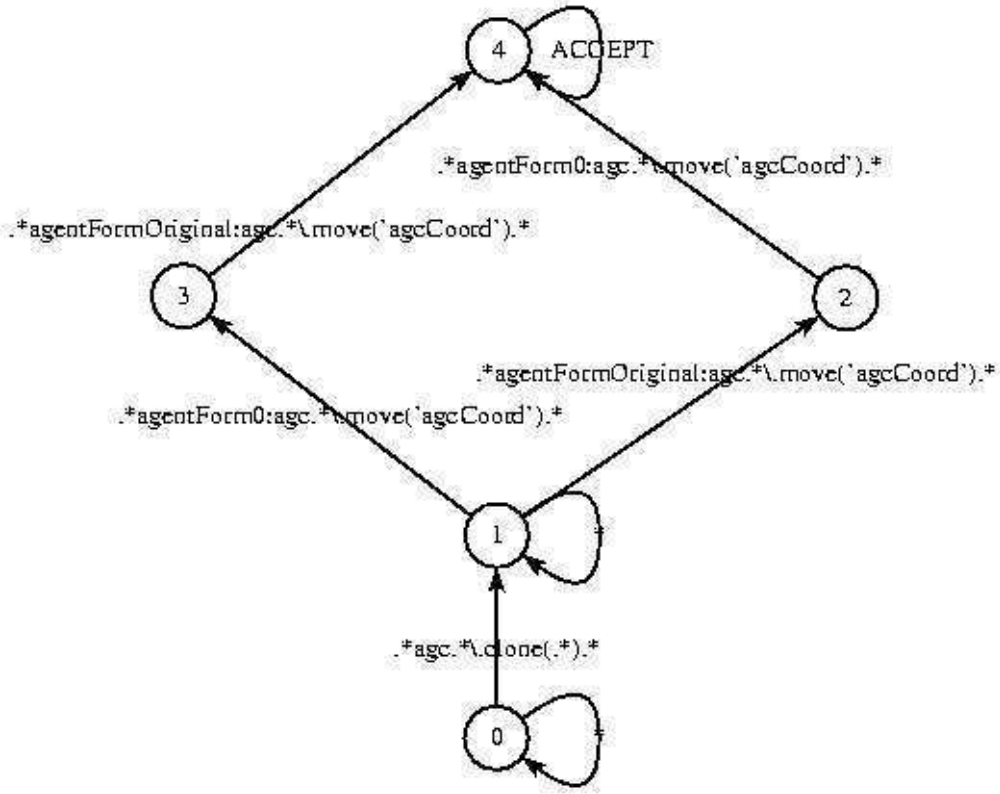

Figura 6.4: LTS do Objetivo de Teste para o Padrão de Projeto *Branching*

Como pode ser visto nos padrões apresentados, nenhuma execução que contenha falha na tentativa de um agente migrar para uma determinada agência será exercitada pelos casos de teste. Os objetivos de teste irão selecionar casos de teste onde os agentes irão conseguir migrar por todas as agências desejadas. Desta forma, decidimos por elaborar um objetivo de teste que visasse gerar casos de teste especificamente para as execuções onde, em algum momento, um agente não consiga migrar para uma determinada agência. A Figura 6.5 apresenta tal objetivo de teste. Segundo o objetivo de teste, os casos de teste em que uma agência envia uma mensagem de erro a um agente e este envia uma mensagem à sua interface gráfica são selecionados, como pode ser visto nas transições *".*agent.*\.errorMoving(.*).*"*, que parte do estado **'1'** em direção ao estado **'3'**, e *"gui.*\.showErro().*"*, que parte do estado **'3'** em direção ao estado de aceitação **'4'**.

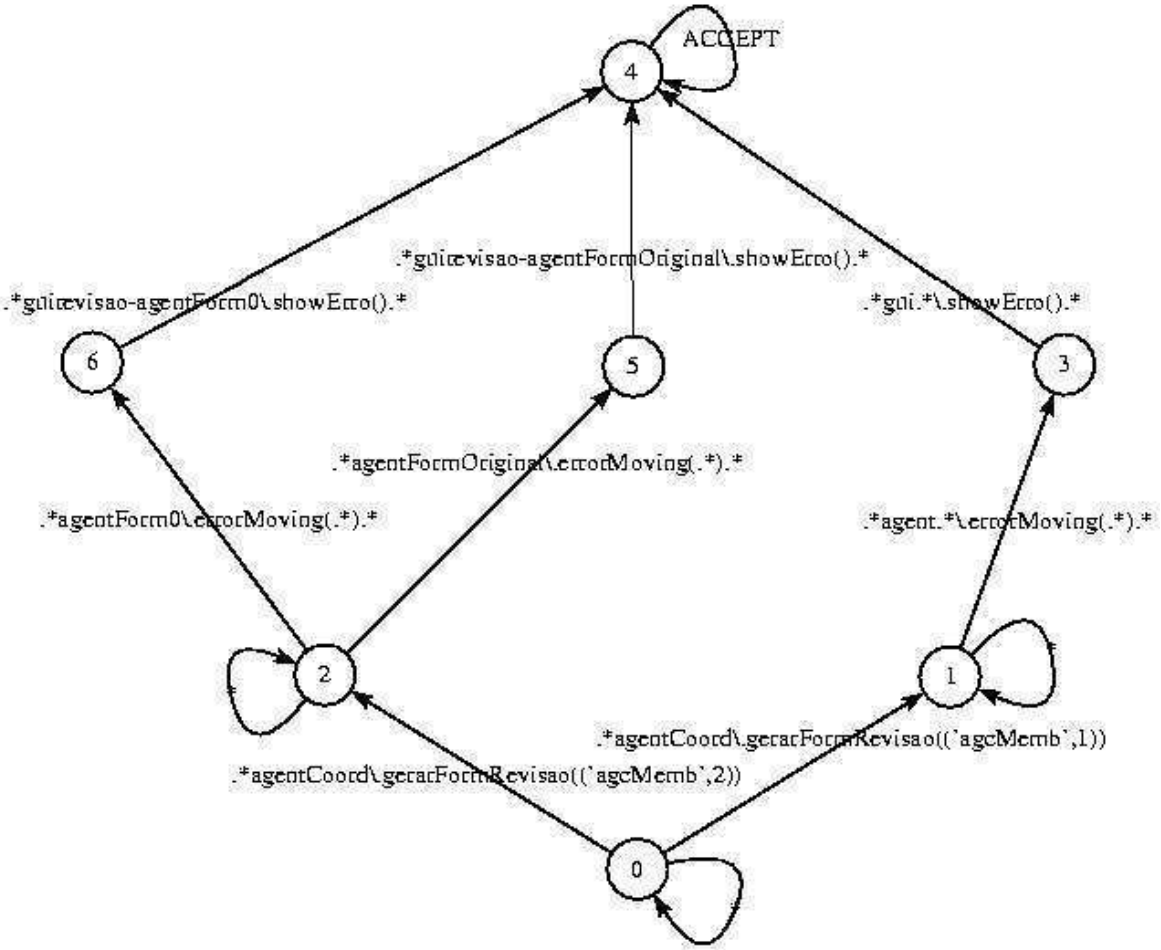

Figura 6.5: LTS do Objetivo de Teste de Erro de Migração

6.4 Geração de Casos de Teste

Nesta seção, é mostrada a geração de casos de teste propriamente dita a partir da especificação do sistema (espaço de estados) e com base nos objetivos de teste elaborados e que foram descritos na seção anterior. Tal geração se deu com o uso da ferramentra TGV, onde o espaço de estados e os objetivos de teste foram entradas do processo, e um grafo foi produzido contendo os casos de teste para cada objetivo de teste. Estes grafos produzidos contêm todos os casos de teste para cada respectivo objetivo de teste, no entanto, devido a restrições de tempo, alguns casos de teste (caminhos de tais grafos) foram selecionados para serem implementados e executados.

O processo de geração dos casos de teste com TGV é simples, uma vez que se trata de entrar um IOLTS do sistema a ser testado e um objetivo de teste, ambos no formato Aldebaran, e a ferramenta irá retornar um outro LTS, chamado de *Complete Test Graph*, contendo os casos de teste gerados. Uma vez que este grafo gerado é relativamente grande, sua visualização através de uma figura fica inviável. Desta forma, a Tabela 6.3 apresenta algumas informações para cada grafo gerado a partir dos objetivos de teste apresentados na seção anterior.

Objetivo de Teste	**Qtd. de Estados**	**Qtd. de Transições**	**Tempo de Geração**
Padrão Itinerary	30.381	69.452	8 min. e 23 seg.
Padrão MasterSlave	17.180	43.608	8 min. e 8 seg.
Padrão Barnching	16.954	43.201	9 min. e 28 seg.
Erro de Migração	29.425	60.378	7 min. e 46 seg.

Tabela 6.3: Dados sobre a Geração dos Casos de Teste por Objetivo de Teste

Como foi dito, de cada grafo de teste gerado, alguns casos de teste foram selecionados para serem implementados e executados sobre a aplicação. Uma vez que o intuito deste capítulo é o de apresentar a viabilidade de implementação dos casos de teste gerados pelo método e de mostrar a eficiência dos mesmos em revelar falhas nas aplicações, selecionamos um conjunto de casos de teste para mostrar sua implementação e execução, mostrando as falhas encontradas, quando for o caso. A seleção desses casos de teste foi feita por meio de uma simples seleção aleatória de caminhos dos grafos de teste gerados. Estes caminhos possuem como estado inicial o estado inicial do grafo de teste e como estado final um estado de aceitação.

Os casos de teste são sequências de entradas e saídas do sistema. O testador proverá as entradas ao sistema a ser testado e é esperado que este retorne as saídas descritas. A cada rótulo de transição do caso de teste a ferramenta acrescentada a palavra *INPUT* ou a palavra *OUTPUT*. A transição que contém a palavra *INPUT* significa que o testador deve esperar aquela ação como uma entrada sua, ou seja, uma saída do sistema a ser testado. Já as transições que possuem a palavra *OUTPUT* significam que o testador deve produzir aquela ação para que sirva de entrada para a aplicação a ser testada. Note que os termos *INPUT* e *OUTPUT* se referem ao programa que irá testar a aplicação. A Figura 6.6 apresenta um trecho de um determinado caso de teste para o sistema de conferências.

Como pode ser visto na figura, a primeira ação (linha 2) do caso de teste é o envio de uma mensagem do ***AgenteCoordenador*** para a sua interface gráfica solicitando que uma tela seja mostrada. A palavra *INPUT* informa que esta é uma mensagem que deve ser observada pelo testador, é uma entrada para quem irá testar o sistema. A sequência de ações procede até que, na linha 25, o *agentFormOriginal* solicita que seja migrado para a agência ***agcCoord***.

Na seção seguinte apresentaremos os casos de teste escolhido para apresentação, mostrando a forma com que eles foram implementados, executados e as falhas encontradas, quando for o caso.

6.5 Implementação e Execução dos Casos de Teste de Teste

O processo de implementação e execução dos casos de teste visa mostrar a viabilidade de os casos de teste gerados pelo método de geração automática serem implementados. Além disso, a execução dos casos de teste irá revelar o seu potencial em revelar falhas nas implementações. Uma implementação de casos de teste consiste em definir como os pontos de controle serão mapeados em entradas do sistema, como os pontos de observação serão mapeados em saídas observáveis do sistema, e definir

```
 1 <initial   state>
 2 "agentCoord:gu    icoord.show(); INPUT"
 3 "guicoord:agen    tCoord.gerarFormRevisao(('agcMemb',1)); OUTPUT"
 4 "agentCoord:co    nf.getDadosConferencia() &
    agentCoord:conf    .criarRegistroRevisao(('agcMemb',1)); INPUT"
 5 "conf:agentCoo    rd.dadosConferencia('dadosconf1'); OUTPUT"
 6 "conf:agentCoo    rd.registroRevisao(('agcMemb',1)); OUTPUT"
 .
 .
 .
23 "agentFormOrig    inal:
    guirevisao-agen    tFormOriginal.showTelaAprovar(); INPUT"
24 "guirevisao-ag    entFormOriginal:
    agentFormOrigin    al.aprovarRevisao(); OUTPUT"
25 "agentFormOrig    inal:agcMemb.move('agcCoord'); INPUT  (PASS)"
26 <goal   state>
```

Figura 6.6: Trecho de um Caso de Teste para o Sistema de Conferências

como o testador poderá predicar sobre as execuções do sistema com base nos casos de teste. A nossa implementação e execução não são feitas automaticamente, uma vez que para isto precisaríamos de um estudo maior e consequentemente um tempo maior. No entanto, os objetivos apresentados acima poderão ser, claramente, atingidos através de uma implementação e de uma execução manual.

6.5.1 Implementação

A abordagem escolhida para a implementação dos casos de teste é simples, onde, basicamente, cada ponto de controle foi mapeado em uma entrada para a implementação e cada ponto de observação para uma saída da implementação. Tanto os pontos de controle como os de observação geraram instrumentações no código que visavam gerar linhas de execução que pudessem ser comparadas com os casos de teste gerados. Desta forma e após a execução de cada caso de teste, um arquivo contendo as ações executadas será criado (arquivo de *log*). Este arquivo é usado para predicar sobre as execuções do sistema, uma vez que as ações ali contidas devem ser as mesmas esperadas pelo caso de teste, para uma execução correta.

As Tabelas 6.4 e 6.5 apresentam, respectivamente, o mapeamento entre cada ponto de controle e uma entrada para o sistema, e cada ponto de observação e a instrumentação necessária para que este seja observável pelo testador.

A instrumentação do código dos agentes foi feita através de mensagens que estes enviam a uma aplicação externa (*Test Driver*). Precisávamos, portanto, de uma aplicação externa que fosse capaz

Implementação dos Pontos de Controle
agentCoord.dadosConferencia(): Os dados da conferência deverão estar no Banco de Dados. Como os agentes não fazem nenhum tipo de processamento especial para diferentes tipos de dados, esses podem ser dados quaisquer.
agentCoord.registroRevisao(): Os dados de registro revisão serão os mesmos solicitados pela entrada *agentCoord.gerarFormRevisao()*.
agentCoord.gerarFormRevisao(): Esses dados serão entrados pelo testador através da interface gráfica do ***AgenteCoordenador***.
agent.init(): Esta entrada é gerada automaticamente pela plataforma de execução, através da chamado ao método *"init()"* do agente.
agent.live(): Esta entrada é gerada automaticamente pela plataforma de execução, através da chamado ao método *"live()"* do agente.
agent.address(): Esta entrada é gerada automaticamente pela plataforma de execução, através do retorno do método *"getLocation()"* chamado pelo agente.
agent.errorMoving(): Esta entrada é gerada automaticamente pela plataforma de execução, através do lançamento de uma exceção. No entanto, o testador deverá gerar a condição de erro de alguma maneira, como por exemplo desligando a agência de destino para qual o agente solicitou a migração, ou informando o endereço de uma agência inválida.
agent.continue(): Esta entrada é implícita, ou seja, é apenas o retorno do método *"copy()"* chamado pelo agente.
agentForm.redirecionar(): Esses dados serão entrados pelo testador através da interface.
agentForm.entrarDadosRevisao(): Esses dados serão entrados pelo testador através da interface.
agentForm.finalizarRevisao(): Esses dados serão entrados pelo testador através da interface.
agentForm.aprovarRevisao(): Esses dados serão entrados pelo testador através da interface.

Tabela 6.4: Tabela Contendo o Mapeamento entre os Pontos de Controle do Sistema de Conferências e sua Implementação

Implementação dos Pontos de Observação
conf.getDadosConferencia(): Instrumentação do código do agente no momento em que este solicita os dados da conferência.
conf.criarRegistroRevisao(): Instrumentação do código do agente no momento em que este solicita a criação do registro revisão.
conf.registrarResultado(): Instrumentação do código do agente no momento em que este solicita que o resultado de uma revisão seja registrado.
gui.show(): Instrumentação do código do agente no momento em que este solicita à interface gráfica a apresentação de alguma tela através dos métodos do tipo *"showTela*()"*.
agc.getAddress(): Instrumentação do código do agente no momento em que este solicita o endereço da agência (método *"getLocation()"*).
agc.move(): Instrumentação do código do agente no momento em que este solicita sua migração para uma agência, através do método *"move()"*.
agc.clone(): Instrumentação do código do agente no momento em que este solicita sua clonagem, através do método *"copy()"*.
agc.createAgent(): Instrumentação do código do agente no momento em que este solicita que outro agente seja criado, através do método *"createAgent()"*.
agentCoord.registrarResultado(): Instrumentação do código do agente no momento em que este solicita ao ***AgenteCoordenador*** a registração dos resultados.

Tabela 6.5: Tabela Contendo o Mapeamento entre os Pontos de Observação do Sistema de Conferências e sua Implementação

de coletar informações enviadas pelos agentes de forma centralizada e colocá-las em um arquivo. A solução adotada advém de uma abordagem sugerida pelos desenvolvedores da plataforma Grasshopper [Gmb01], onde eles mostram como comunicar aplicações externas e os agentes. Esta abordagem pode ser vista na Figura 6.7. Esta solução é baseada no sistema produtor-consumidor, onde os agentes produzem informações e o *Test Driver* as consome. Como pode ser visto na Figura 6.7, um objeto chamado *ServerObject* é colocado de forma que ambos (*Test Driver* e agente) tenham uma referência para tal e possam utilizá-lo para colocar informações ou retirar informações de lá.

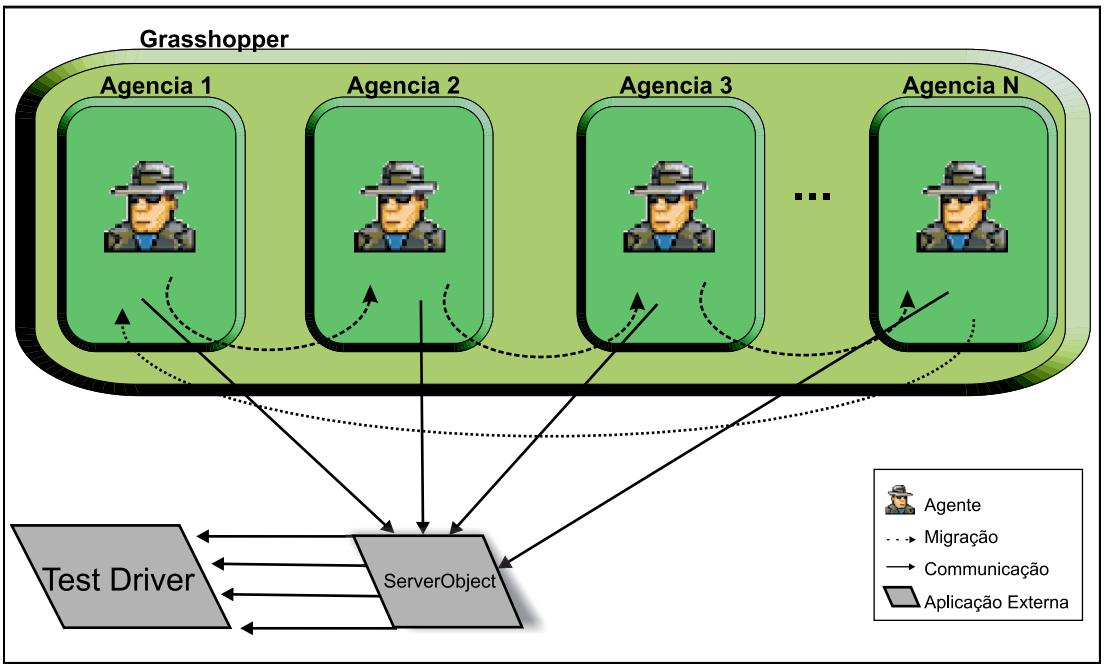

Figura 6.7: Comunicação entre Agentes e o *Test Driver*

6.5.2 Execução

A execução dos casos de teste gerados foi feita de forma manual, como foi dito. As ações descritas pelos mesmos foram entradas no sistema e suas saídas foram observadas. Na pratica, para cada caso de teste, o sistema foi exercitado com as entradas previstas nos casos de teste e, com o arquivo de *log* gerado pelo *Test Driver*, a execução foi avaliada em correta ou incorreta em relação à especificação. A seguir (Tabelas 6.6, 6.7, 6.8 e 6.9), serão apresentados os casos de teste exercitados e os resultados das execuções. Apresentaremos uma breve descrição dos casos de teste e mostraremos os resultados da execução, através das falhas encontradas, caso existam. Os casos de teste podem ser encontrados no Apêndice B.

A execução dos casos de teste, apesar de manual, se mostrou interessante em revelar falhas. Por se tratar de um sistema relativamente simples (poucas entradas e saídas), tal execução foi viável e, principalmente, nos permitiu avaliar o potencial dos casos de teste em revelar falhas. Especificamente em relação a característica de mobilidade, os casos de teste se mostraram úteis em revelar falhas

Objetivo de Teste para o Padrão *Itinerary*	
Caso de Teste 01	
- A quantidade de revisões solicitadas é igual a um (um agente ***AgenteFormRevisao*** será criado). - Membro do comitê o envia à agência ***agcRev1*** e solicita o seu retorno. - Revisor reenvia o formulário à agência ***agcRev2*** sem solicitar o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	O sistema se comportou em conformidade com a especificação.
Caso de Teste 02	
- A quantidade de revisões solicitadas é igual a um (um agente ***AgenteFormRevisao*** será criado). - Membro do comitê o envia à agência ***agcRev1*** e solicita o seu retorno. - Revisor reenvia o formulário à agência ***agcRev1*** (mesma agência em que se encontra) solicitando o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	O agente não consegue migrar para a agência ***agcRev1*** em sua segunda tentativa, dando indícios de que há uma falha no tratamento de quando o agente solicita uma migração para a mesma agência em que está localizado.
Caso de Teste 03	
- A quantidade de revisões solicitadas é igual a dois (dois agentes ***AgenteFormRevisao*** serão criados). - Apenas o agente original é utilizado. - Membro do comitê o envia à agência ***agcRev1*** e solicita o seu retorno. - Revisor reenvia o formulário à agência ***agcRev2*** solicitando o seu retorno. - Revisor entra com dados de revisão e o finaliza. - Agente retorna à agência ***agcRev1*** e o revisor aprova a revisão. - Agente retorna à agência do membro de comitê e o membro aprova a revisão.	
Resultados:	O agente não migra para a agência ***agcRev1*** para a aprovação, apenas para a agência do membro de comitê. Uma vez que a lista de agências que solicitaram a revisão é implemenada pelo padrão *Itinerary*, encontramos fortes indícios de que o padrão não foi implementado de forma correta.

Tabela 6.6: Casos de Teste Extraídos do Objetivo de Teste para o Padrão *Itinerary*

Objetivo de Teste para o Padrão *MasterSlave*	
Caso de Teste 01	
- A quantidade de revisões solicitadas é igual a um (um agente ***AgenteFormRevisao*** será criado). - Membro do comitê o envia à agência ***agcRev1*** e não solicita o seu retorno. - Revisor reenvia o formulário à agência ***agcRev2*** sem solicitar o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	Nenhuma falta de conformidade foi encontrada.
Caso de Teste 02	
- A quantidade de revisões solicitadas é igual a dois (dois agentes ***AgenteFormRevisao*** serão criados). - Um agente é exercitado primeiro (original) até concluir a revisão para depois o outro (clone) ser executado. - Membro do comitê o envia à agência ***agcRev1*** e não solicita o seu retorno. - Revisor entra com dados de revisão e o finaliza. - Membro do comitê envia o outro agente à agência ***agcRev1*** e não solicita o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	Segundo agente não registra resultado junto ao agente coordenador. Com esta inconformidade, averigüamos que o agente ***AgenteFormRevisao*** (agente *escravo*) não realiza sua tarefa corretamente.
Caso de Teste 03	
- A quantidade de revisões solicitadas é igual a dois (dois agentes ***AgenteFormRevisao*** serão criados). - Um agente é exercitado primeiro (clone) até concluir a revisão para depois o outro (original) ser executado. - Membro do comitê o envia à agência ***agcRev1*** e não solicita o seu retorno. - Revisor entra com dados de revisão e o finaliza. - Membro do comitê envia o outro agente à agência ***agcRev1*** e não solicita o seu retorno. - Revisor entra com dados de revisão e o finaliza.	
Resultados:	Primeiro agente não registra resultado junto ao agente coordenador. Com esta inconformidade, averigüamos que o agente ***AgenteFormRevisao*** (agente *escravo*) não realiza sua tarefa corretamente quando este se trata de um clone.

Tabela 6.7: Casos de Teste Extraídos do Objetivo de Teste para o Padrão *MasterSlave*

Objetivo de Teste para o Padrão *Branching*	
Caso de Teste 01	
- A quantidade de revisões solicitadas é igual a dois (dois agentes ***AgenteFormRevisao*** serão criados). - Os agentes são executados alternadamente. Após uma entrada ter sido feita para um agente, aguarda-se uma saída deste mesmo agente (apresentação de uma tela ou finalização da execução) e então a próxima entrada será para o outro agente, até terminarem suas execuções. - Membro do comitê envia os agentes à agência ***agcRev1*** e não solicita os retornos. - Revisor entra com dados de revisão e os finaliza.	
Resultados:	O agente clone não registra o resultado, deixando indícios de que o processo de clonagem não está correto.
Caso de Teste 02	
- A quantidade de revisões solicitadas é igual a dois (dois agentes ***AgenteFormRevisao*** serão criados). - Os agentes são executados alternadamente, no entanto as entradas são colocadas em um mesmo instante. - Membro do comitê envia o agente original à agência ***agcRev1*** e o agente clone até à agência ***agcRev2***, e nenhuma aprovação é solicitada.	
Resultados:	O agente clone não registra o resultado, deixando indícios de que o processo de clonagem não está correto.

Tabela 6.8: Casos de Teste Extraídos do Objetivo de Teste para o Padrão *Branching*

Objetivo de Teste para Erro de Migração	
Caso de Teste 01	
- A quantidade de revisões é igual a um (um agente ***AgenteFormRevisao*** será criado). - Membro do comitê o envia à agência ***agcRev1*** e solicita o seu retorno. - ***agcRev1*** está indisponível. - Agente apresenta mensagem de erro.	
Resultados:	Agente nem solicita a migração e nem informa mensagem de erro.
Caso de Teste 02	
- A quantidade de revisões é igual a um (um agente ***AgenteFormRevisao*** será criado). - Membro do comitê o envia à agência ***agcRev1*** e solicita o seu retorno. - Revisor entra com dados de revisão e os finaliza. - Agência do coordenador está indisponível. - Agente apresenta mensagem de erro.	
Resultados:	Agente não informa mensagem de erro.
Caso de Teste 03	
- A quantidade de revisões é igual a dois (dois agentes ***AgenteFormRevisao*** são criados). - Agência do membro de comitê está indisponível. - Agente apresenta mensagem de erro.	
Resultados:	Agente nem solicita a migração e nem informa mensagem de erro.
Caso de Teste 04	
- A quantidade de revisões é igual a dois (dois agentes ***AgenteFormRevisao*** são criados). - Um agente é exercitado primeiro (original) até concluir a revisão para depois o outro (original) ser executado. - Membro do comitê o envia à agência ***agcRev1*** e não solicita o seu retorno. - Revisor entra com dados de revisão e o finaliza. - Membro do comitê envia o outro agente à agência ***agcRev1*** e não solicita o seu retorno. - Revisor o redireciona para ***agcRev2*** que está indisponível. - Agente apresenta mensagem de erro.	
Resultados:	Agente nem solicita a migração e nem informa mensagem de erro.

Tabela 6.9: Casos de Teste Extraídos do Objetivo de Teste para Erro de Migração

na implementação desta característica, como pode ser visto nos Casos de Teste 02 e 03 do padrão *Itinerary* (Tabela 6.6) e nos quatro casos de teste do objetivo de teste *Error Moving*.

Como pôde ser visto pelos resultados das execuções dos casos de teste, os objetivos de teste se mostraram interessantes quando se deseja exercitar (testar) partes específicas de um sistema, uma vez que seus casos de teste detectaram faltas no comportamento especificado pelos objetivos ou mostraram conformidade de tal comportamento. Uma vez que os objetivos de teste foram extraídos de especificações de padrões de projetos, um estudo realizado sobre os casos de teste gerados e sua utilização como premissas para a identificação de padrões de teste foi realizado, cujo os resultados são apresentado no Capítulo 7.

6.6 Considerações Finais

O estudo de caso apresentado neste capítulo teve como objetivo mostrar uma aplicação real do método de geração automática de casos de teste no intuito de apresentar indícios de sua aplicabilidade e de mostrar o potencial dos casos de teste em revelar falhas no sistema. Além disso, a implementação dos casos de teste permitiu mostrar que, mesmo de forma manual, estes casos de teste são passíveis de implementação garantindo a aplicabilidade do processo como um todo.

Outros estudos de caso com diferentes sistemas de diferentes tamanhos precisarão ser feitos no intuito de obter uma validação completa do método. De forma a auxiliar tal validação, algumas possíveis métricas a serem utilizadas são propostas a seguir.

- **Métricas de Cobertura:** Dado que estamos tratando de teste baseado em especificação, métricas de cobertura dos casos de teste com relação à especificação precisam ser utilizadas para avaliar o método. É preciso avaliar mais rigorosamente se a especificação das propriedades de SBAM, tais como mobilidade e localidade, através dos objetivos de teste está proporcionando uma boa cobertura destas mesmas propriedades na especificação pelos casos de teste gerados.

- **Taxa de Defeitos:** É importante uma avaliação mais detalhada sobre a quantidade de faltas encontradas na implementação por casos de teste gerados e executados pelo processo. Uma vez que os métodos de teste baseados em especificação contribuem para a identificação de faltas também na especificação, a medição da taxa de defeitos também precisa ser feita sobre as faltas encontradas na especificação. Desta forma, o potencial do método proposto em revelar faltas tanto na especificação quanto na implementação poderá ser melhor avaliado.

- **Limitações dos Sistemas diante da Complexidade dos Modelos:** Uma vez que alguns passos do método de geração de casos de teste envolvem processamentos que requerem considerável quantidade de recurso computacional como, por exemplo, a geração de espaço de estados, é preciso avaliar o tamanho e a complexidade dos modelos que limitam a aplicação do método.

- **Tempo e Esforço de Desenvolvimento:** Com relação ao processo de desenvolvimento de *software* como um todo, é preciso avaliar o impacto da utilização do método em um processo,

medindo o tempo e o esforço gastos para tal, avaliando a viabilidade da aplicação do método.

Capítulo 7

Padrões de Teste para Agentes Móveis

7.1 Introdução

Diversos tipos de padrões (de projeto, arquiteturais, de teste, etc.) têm sido cada vez mais utilizado pelos desenvolvedores de sistemas na busca por maior eficiência e eficácia em seus processos de desenvolvimento. Isto é dado, possivelmente, devido ao fato de que o uso destes padrões pode propiciar maior produtividade e maior viabilidade em decorrência, por exemplo, da reusabilidade de soluções que proporciona. Neste contexto, padrões de teste surgem como interessantes aliados na busca por sistemas de qualidade. Um padrão de teste pode ser definido como um tipo de teste que se aplica a um determinado contexto com o intuito de revelar falhas conhecidas e recorrentes [Bin99; Lan01].

Padrões de teste é um assunto ainda imaturo na comunidade de engenharia de software, no entanto, com potencial para o aumento da eficiência e eficácia das atividades de teste. Com a identificação de contextos recorrentes, padrões de teste podem ser utilizados para identificar falhas também recorrentes, ou seja, falhas que comumente são encontradas nestes contextos. Em outras palavras, um padrão de teste proverá uma forma de se obter um determinado conjunto de casos de teste conhecidamente eficazes em revelar certos tipos de falhas.

Pelo fato de ser um tópico ainda imaturo, padrões de teste possuem: carência de formatos de definição mais consolidados e amplamente utilizados pela comunidade; falta de classificação e catalogação, no intuito de facilitar sua busca e a utilização; poucos trabalhos propondo novos padrões ou falando sobre suas utilizações. Estes problemas estão, de certa forma, relacionados. Por exemplo, o fato de não existirem catálogos para este tipo de padrão dificulta a consolidação de formatos de definição que, por conseqüência, não estimula a proposta de novos padrões bem como sua utilização.

Um outro fator importante para a proposta de novos padrões de teste está relacionada com a forma com que estes padrões serão identificados. A maioria dos trabalhos que propõem padrões nada fala a respeito da forma com que estes padrões foram identificados, ou simplesmente, transparecem que a identificação ocorreu de forma empírica. Neste capítulo, além de apresentarmos um exemplo de padrões de teste (Seção 7.3) e um formato de definição para tais padrões (Seção 7.2), mostraremos

também uma forma de como identificá-los baseada no uso de padrões de projeto (Seção 7.4).

O nosso trabalho sobre padrões de teste iniciou-se após uma análise feita sobre os casos de teste apresentados no Capítulo 5, em que pudemos observar que alguns destes poderiam ser utilizados sobre outras aplicações. Sendo assim, a partir dos casos de teste obtidos e executados no Capítulo 5, alguns padrões de teste foram identificados. Além disso, uma forma de se identificar padrões de teste a partir de padrões de projeto foi observada e é apresentada neste capítulo. Além disso, um formato de definição de padrões de teste para agentes móveis identificados a partir de padrões de projeto é mostrado. Estes resultados também podem ser encontrados em [FAM04].

Este capítulo está organizado da seguinte forma. A Seção 7.2 apresenta um formato de definição para padrões de teste que foram identificados a partir de padrões de projetos. A Seção 7.3 ilustra um padrão de teste que foi obtido a partir de casos de teste gerados durante o estudo de caso deste trabalho. Esta seção visa ilustrar padrões de teste e apresentar uma forma empírica de se obter tais padrões. Já na Seção 7.4, uma metodologia para identificação de padrões de teste a partir de padrões de projeto para SBAMs é apresentada. E por fim, a Seção 7.5 apresenta as conclusões do capítulo.

7.2 Formato de Definição de Padrões de Teste para Agentes Móveis

Como foi dito na seção anterior, é necessário uma maior consolidação das propostas de formato de definição de padrões de teste. Mais precisamente, um formato para padrões de teste aplicáveis a contextos de padrões de projetos para agentes móveis precisa ser definido, uma vez que nas seções seguintes iremos apresentar padrões de teste deste tipo. Nesta seção, estaremos apresentando uma proposta de um formato de definição de padrões de teste identificados a partir de padrões de projetos para agentes móveis. Este formato é baseado nos formatos propostos por Lima [Lim04] e, principalmente, por Binder [Bin99]. Em outras palavras, esta proposta conterá os principais elementos propostos por Binder, além de outros oriundos da proposta de Lima, que tem o intuito de contemplar dois conceitos particulares a este trabalho: padrões de projeto e agentes móveis.

Em seu livro, Binder propõe um formato de definição de padrões de teste baseado no formato proposto por Gamma et al [GHJV95] para padrões de projeto. Dos elementos que ele propõe, alguns foram retirados e outros redefinidos com o intuito de contemplar novas características, tais como mobilidade e relacionamento com padrões de projeto. Dentre os elementos redefinidos, destacamos o *Contexto*. Nele, a informação sobre qual padrão de projeto o padrão de teste está relacionado é disponibilizada. A fim de contemplar uma característica específica do paradigma de agentes móveis, que é a de todo sistema necessitar de uma plataforma específica para a sua execução, o elemento *Solução* foi removido e outros dois foram acrescentados em seu lugar - *Estrutura* e *Implementação*. Com isto, os sistemas (bem como os testes) poderão possuir dois tipos de modelos: os modelos independentes e os dependentes de plataforma. Assim, a solução que o padrão de teste propõe, deve ser apresentada de forma independente de quaisquer plataformas bem como, adicionalmente, através de modelos ligados às diferentes plataformas de execução. Esta abordagem de modelagem pode

ser vista na metodologia de desenvolvimento apresentada em [Gue02a] e no trabalho de Lima em [Lim04].

Com isto, podemos apresentar o formato de definição de padrões de teste que propomos, cujos elementos estão descritos na Tabela 7.1. Propomos que os elementos *Nome*, *Contexto*, *Intenção*, *Estrutura*, *Implementação*, *Forças* (caso existam) e *Exemplos* sejam obrigatórios, e que os demais sejam utilizados para enriquecer as definições.

Elemento	***Descrição***
Nome	O padrão deverá possuir um nome descrito por uma palavra significativa ou por uma frase curta. Abstrações conceituais são interessantes.
Contexto	O contexto deverá informar o(s) padrão(ões) de projeto para o qual o padrão de teste é aplicável. Além disso, no contexto estará descrito a aplicabilidade do padrão de teste bem como as pré-condições que fazem com que a solução seja aplicada ao problema e com que esta solução seja vantajosa.
Intenção	Breve descrição do tipo de suíte de teste produzido pelo padrão de teste.
Estrutura	Aqui é apresentada a solução independente de plataforma. O modelo da solução aqui descrita será independente de onde o sistema a ser testado estará sendo executado. Este elemento traz o modelo de teste, que aqui significa a representação das responsabilidades e implementação que são os focos do projeto de teste [Bin99], um procedimento que descreve como o padrão deverá ser aplicado na implementação, e o oráculo, o qual é responsável por informar se a implementação passou ou falhou para cada caso de teste.
Implementação	Este elemento descreve uma solução apresentada no elemento *Estrutura* de forma dependente de plataforma. Ou seja, modelos diretamente ligados às plataformas de execução são apresentados para diversas plataformas.
Forças	Este elemento proverá restrições e *trade-offs* sobre o qual o padrão de teste está submetido.
Exemplos	Provê alguns exemplos de aplicações e utilizações do padrão. É comumente utilizado para facilitar o entendimento dos elementos *Estrutura* e *Implementação*.
Modelo de Faltas	Descreve os tipos de faltas que podem ser alcançadas, disparadas e propagadas. Pode ser formalmente ou heuristicamente (usando experiências ou considerações práticas) descrito.
Usos conhecidos	Provê aplicações onde o padrão já foi usado.

Tabela 7.1: Elementos do Formato de Definição de Padrões de Teste. (continua)

Elemento	***Descrição***
Padrões Relacionados	Apresenta os padrões que possuem alguma relação (similar ou complementar) com este, seja através das forças, contexto ou solução que apresentam.
Referência	Caso o provedor do padrão não seja o criador do mesmo, este elemento informará o(s) seu(s) autor(es), bem como onde encontrar maiores informações sobre o padrão.

Tabela 7.1: Elementos do Formato de Definição de Padrões de Teste.

7.3 Identificação de Padrões de Teste a partir de Casos de Teste

Alguns casos de teste obtidos no Capítulo 5 se apresentaram com grande potencial para serem considerados como padrões de teste. Uma vez que seu potencial em revelar falhas foi averiguado, o próximo passo consiste em mostrar que os casos de teste podem ser aplicados a diferentes aplicações sobre um mesmo contexto. Com isto, é possível descrever os casos de teste de forma mais abstrata de maneira que possam ser utilizados em outras aplicações.

Os casos de teste que mostraram este potencial são os que foram obtidos a partir do objetivo de teste *Error Moving*. Dado que os casos de teste se mostraram eficazes em revelar falhas na aplicação, foi preciso averiguar seus potenciais em serem aplicáveis a outras aplicações baseadas em Agentes Móveis. Uma vez que estamos tratando de SBAM's, questões relativas a serviços indisponíveis, endereçamento errado, falha de comunicação, e outros, muito possivelmente estarão presentes na maioria dos sistemas. Desta forma, o tratamento de erros precisará ser testado. Com isto, estes casos de teste serão úteis a outras aplicações, mostrando que as falhas que eles detectam podem estar presentes em diversos contextos.

Desta forma, os casos de teste apresentados no capítulo anterior, referentes ao objetivo de teste *Error Moving*, foram abstraídos e definidos como padrão de teste, como apresentado na Tabela 7.2. Para a definição, utilizamos elementos do formato de definição apresentado na Seção 7.2. Apesar deste padrão de teste não ter sido identificado a partir de padrão de projeto, este formato se mostra bastante útil.

Elemento	***Descrição***
Nome	*ErrorMoving*.
Contexto	Este padrão pode ser aplicado a qualquer sistema baseado em Agentes Móveis, principalmente àqueles em que a confiabilidade da rede seja baixa.

Tabela 7.2: Definição do Padrão de Teste *ErrorMoving*. (continua)

Elemento	***Descrição***
Intenção	Os casos de teste produzidos por este padrão visam testar o tratamento de erro nos processos de migração do sistema.
Estrutura e Implementação	- Gerar linhas de execução para os trechos em que os agentes se movem por agências. - Para cada ponto onde um agente tenta migrar para uma agência, um caso de teste deve ser gerado colocando a agência alvo indisponível. - A execução é verificada observando-se o comportamento do agente, que deve enviar uma mensagem de erro e não pode parar a execução do sistema.
Forças	O processo de automação deste padrão está diretamente condicionado à automação da geração dos casos de teste e da plataforma, uma vez que colocar agências indisponíveis pode não ser uma atividade automatizável.
Modelo de Faltas	As faltas disparadas pelos casos de teste geralmente estão relacionadas com o tratamento errado dos agentes em relação à forma com que as plataformas os informam sobre a indisponibilidade das agências. Muitas plataformas, por exemplo, fazem isto com o uso de exceções que em muitos casos não são tratadas de forma correta pelos agentes.

Tabela 7.2: Definição do Padrão de Teste *ErrorMoving*

7.4 Identificação de Padrões de Teste a partir de Padrões de Projeto

Como dissemos na introdução deste capítulo, os padrões de testes são geralmente identificados de forma empírica. Ou seja, ao notar que um conjunto de casos de teste é eficaz em revelar certas falhas e que estes podem ser aplicados a diferentes sistemas (com pequenas adaptações), um padrão de teste é definido para que outras pessoas possam obter tais casos de teste. No entanto, nesta seção, iremos propor um procedimento sistemático para a identificação dos padrões de teste baseado em padrões de projeto. Em resumo, mostraremos que, ao extrairmos casos de teste a partir de especificações de padrões de projeto e, ao mostrarmos que estes casos de teste são eficazes em revelar falhas, podemos identificar e definir padrões de teste.

Os casos de teste obtidos no Capítulo 5 foram gerados tendo como base (objetivos de teste) especificações de padrões de projetos. Uma vez que os padrões de projetos são utilizados por diversas aplicações, é possível que estes mesmos casos de teste também possam ser utilizados nestas outras aplicações, contanto que, para isto, as características inerentes a cada aplicação sejam diferenciadas nos casos de teste de cada aplicação. Baseado no que foi dito, intencionamos propor uma forma de identificar padrões de teste a partir de especficações de padrões de projetos.

A metodologia para a identificação de padrões de teste proposta é apresentada na Figura 7.1 e descrita abaixo.

- **Identificação de Casos de Teste:**

  - A partir da especificação de um padrão de projeto ou de um sistema que contenha padrões de projetos, casos de teste são extraídos. Se porventura os casos de teste forem extraídos da especificação de um sistema, estes devem objetivar especificamente os trechos que implementam o padrão de projeto.

  - Sobre os casos de teste extraídos, é feita uma abstração removendo-se características específicas das implementações e agrupando aqueles que são semelhantes. Uma vez que se abstrai, por exemplo, dados de testes, há um conjunto de casos de teste que tendem a ser iguais, uma vez que irão descrever os mesmos procedimentos de teste.

- **Implementação e Execução dos Casos de Teste:** Os casos de teste abstraídos são, então, exercitados em diferentes aplicações. Este passo é feito no intuito de averiguar a eficácia dos casos de teste em revelar falhas em diferentes aplicações, requisito básico para a identificação de um padrão de teste.

- **Identificação dos Padrões de Teste:** Com os resultados da execução, para cada grupo de casos de teste semelhantes cuja eficácia em revelar falhas for comprovada, é definido um padrão de teste. Isto constitui, além da identificação propriamente dita do padrão, na escrita do mesmo utilizando-se o formato proposto na Seção 7.2.

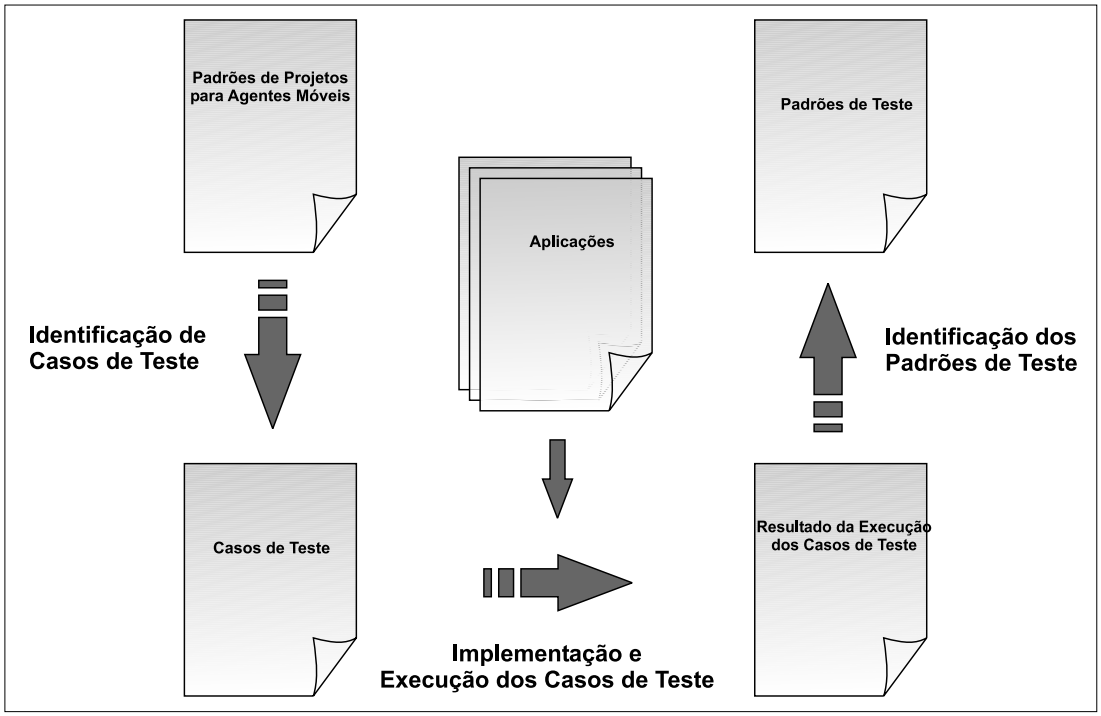

Figura 7.1: Metodologia para Identificação de Padrões de Teste

No intuito de ilustrar o procedimento apresentado, mostraremos um exemplo de um conjunto de casos de teste extraídos para o padrão *Itinerary* que foram utilizados para definir um padrão de teste.

Outros padrões de teste podem ser encontrados no Apêndice C. Estes casos de teste foram gerados utilizando-se a metodologia apresentada nos Capítulos 3 e 4.

Os casos de teste descrevem passos que visam verificar se os agentes de uma aplicação tratam de forma correta ou não os casos em que há agências indisponíveis (*off-line*) em seus itinerários. Eles exercitam as aplicações de tal forma que os itinerários dos agentes que implementam o padrão *Itinerary* possuam agências *off-line* e verificam se algum erro ocorre quando eles tentam migrar para tais agências.

Seguindo a metodologia apresentada acima para a identificação de padrões de teste, após termos identificado os casos de teste e os abstraído, aplicamos estes casos de teste em três aplicações diferentes no intuito de verificar sua capacidade de revelar falhas. Essas três aplicações foram:

- (1) uma aplicação desenvolvida como parte do trabalho realizado por Lima em [Lim04] que contém uma utilização simples do padrão *Itinerary*;
- (2) uma aplicação de venda de passagens aéreas chamada *Monitoring Changing Conditions* que foi um dos resultados do trabalho de Medeiros em [MMG02];
- (3) o sistema de apoio às atividades de comitês de programa em conferências apresentado no Capítulo 5.

Os resultdos da execução dos casos de teste vieram para comprovar a eficácia deste tipo de caso de teste em revelar certos tipos de falhas. Na aplicação (1), temos um agente que migra por seu itinerário realizando uma tarefa simples, que é a impressão de uma mensagem na saída padrão. Ao executar os casos de teste nesta aplicação, encontrou-se uma falha no momento em que o agente tenta migrar para a agência *off-line*, onde a aplicação é finalizada e o agente não percorre o restante do itinerário realizando a sua tarefa. O problema decorre do não tratamento de uma exceção que caracteriza esta situação de falha das agências.

A aplicação (2) utiliza o padrão quando o cliente do sistema deseja pesquisar preços de passagens em diversas empresas de transporte aéreo. A falha encontrada foi que o agente, ao não encontrar a agência de destino, volta à agência de origem com alguns resultados. Assim, o agente deixa de percorrer o restante das agências do itinerário, fornecendo dados incompletos. Este comportamento fere a especificação do sistema, que diz que o agente deve percorrer todo o seu itinerário e por último retornar à agência de origem.

Na aplicação (3), o padrão *Itinerary* é utilizado quando um artigo possui a revisão finalizada e precisa que seja aprovada. Neste sistema encontramos uma falha significativa. Quando o agente responsável pelo encaminhamento das revisões migra para uma agência que está *off-line*, o agente finaliza a sua execução não retornando para a agência do coordenador com a revisão. Dissemos que esta falha é significativa devido ao fato de que sua persepção não é fácil pois nenhum erro aparente é enviado a usuário algum e os dados que o agente carrega são perdidos.

Uma vez que a eficácia dos casos de teste foi obtida, o passo seguinte consiste em definir o padrão de teste segundo um determinado formato de definição. O formato utilizado para definir este padrão, bem como os apresentados no Apêndice C, é o que está definido na Seção 7.2. A seguir temos a definição do padrão de teste chamado *Black Hole*, cuja origem vem dos casos de teste acima apresentados.

- **Nome:** Black Hole
- **Contexto:** Implementações que se utilizam do padrão de projeto *Itinerary*. Este padrão se torna útil quando aplicado a um contexto onde as agências são instáveis, podem estar indisponíveis a qualquer instante.
- **Intencão:** Testar implementação com o objetivo de verificar se os agentes tratam de forma correta a existência de agências *off-line* em seus itinerários.
- **Estrutura**

  Modelo de Teste: Os modelos independentes de plataforma podem ser encontrados nas Figuras 7.2 e 7.3. Nelas, são apresentados, respectivamente, um diagrama de classes UML para o projeto de teste e um diagrama de seqüencia contendo uma execução básica de um caso de teste.

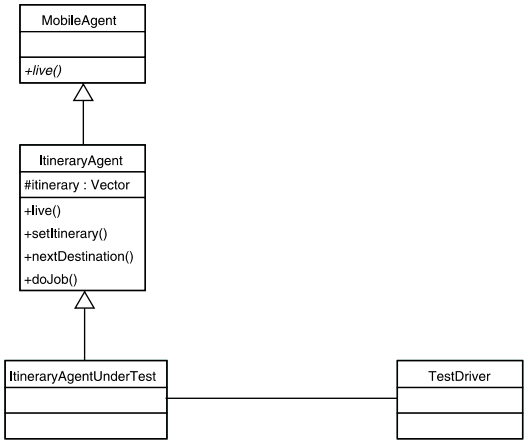

Figura 7.2: Diagrama de Classes para o Padrão de Teste *Black Hole*

Procedimento

- Construir via herança um agente "empacotador" (*wrapper*), chamado ***ItineraryAgentUnder-Test***, para o agente que será testado.

- Sobrescrever os métodos responsáveis por executar a tarefa e por fazer o agente migrar para a próxima agência. Esses métodos irão chamar o método original do agente e enviar uma mensagem ao *Test Driver* informando do ocorrido.

- O *Test Driver* irá instanciar uma lista de agências.

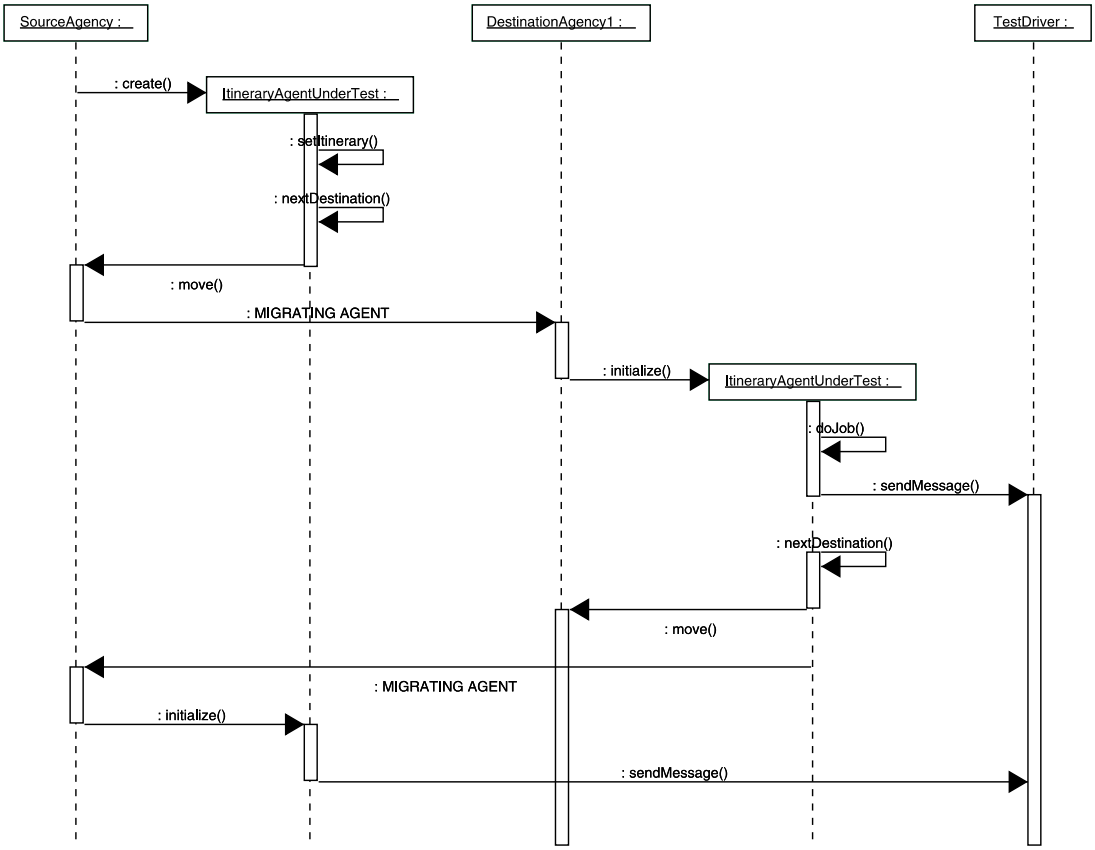

Figura 7.3: Diagrama de Sequência para o Padrão de Teste *Black Hole*

- A aplicação será executada com o agente *wrapper* e com seu itinerário contendo as agências instanciadas pelo *Test Driver* mais o endereço de uma agência *off-line*.

Oráculo: Usando as informações enviadas pelo agente *wrapper*, os seguintes ítens devem ser verificados.

- Deve haver uma mensagem informando que o agente chegou em cada agência do seu itinerário, exceto na agência insdisponível.

- Deve haver uma mensagem informando que o agente executou sua tarefa em cada agência do seu itinerário, exceto na agência insdisponível.

- Deve haver uma mensagem informando que o agente retornou à agência original.

- **Implementação**

  Plataforma Grasshopper

O diagrana de classes e de sequência para o padrão específicos para a plataforma Grasshopper estão presentes, respectivamente, nas Figuras 7.4 e 7.5.

Os diagramas de classe e de sequência do padrão para a plataforma Grasshopper são semelhantes aos diagramas independentes de plataforma, onde a diferença consiste na maneira com

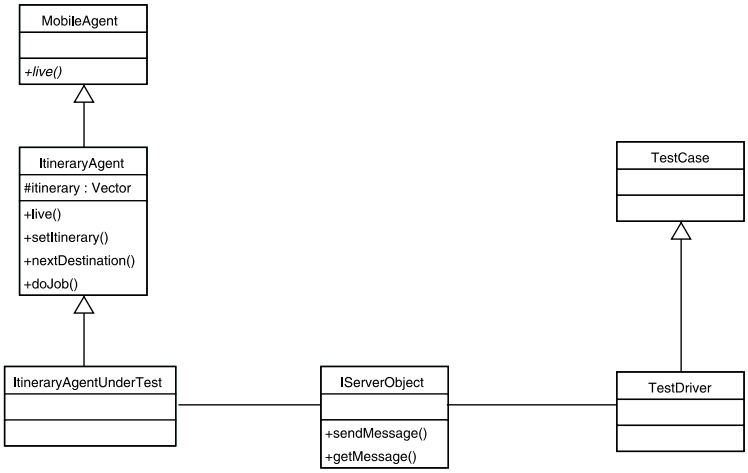

Figura 7.4: Diagrama de Classes do Padrão *Black Hole* para a Pltaforma Grasshopper

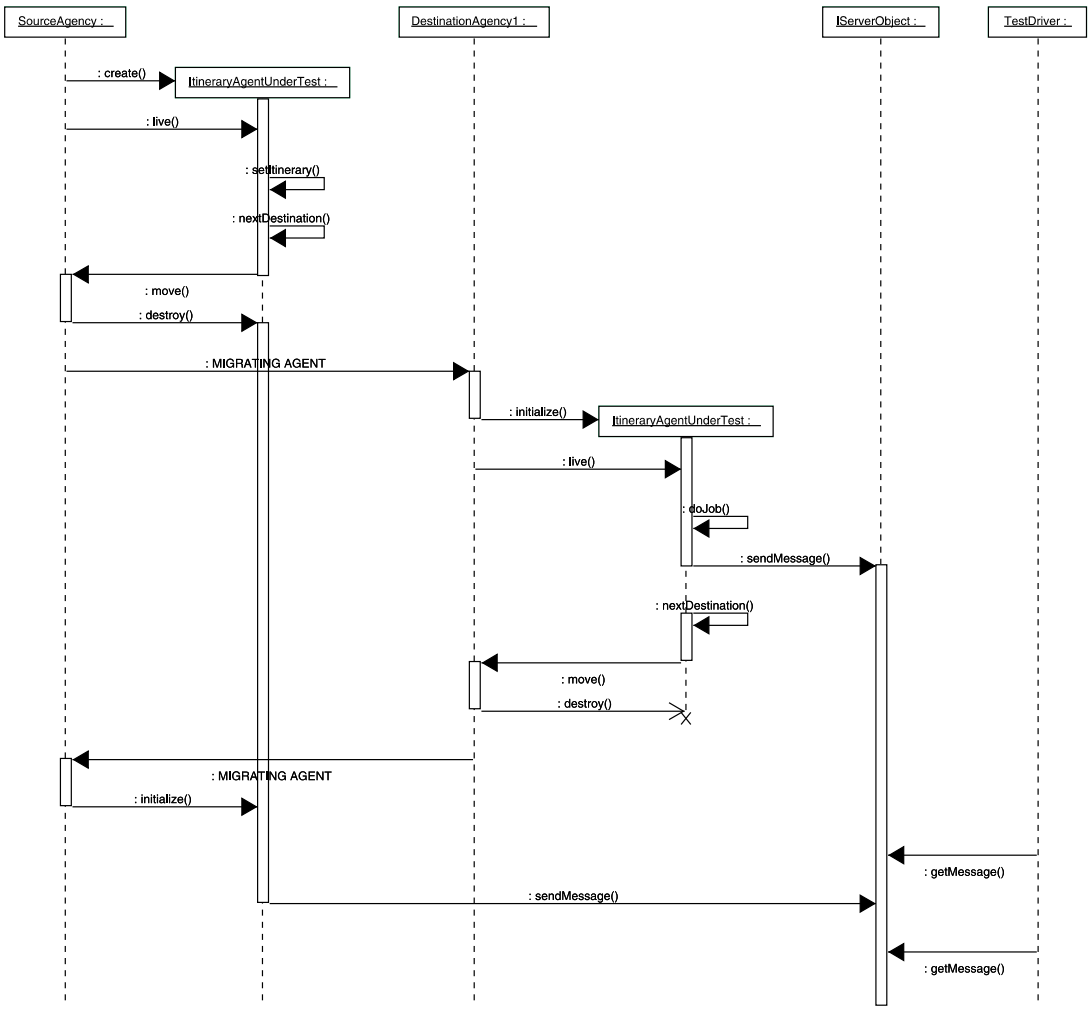

Figura 7.5: Diagrama de Sequência (execução básica) do Padrão *Black Hole* para a Pltaforma Grasshopper

que o agente *wrapper* irá se comunicar com o *Test Driver*. A solução para tal vem da forma com que o manual da plataforma [Gmb01] recomenda para a comunicação entre agentes e aplicações externas. A idéia principal é colocar um objeto chamado *ServerObject* entre o agente *wrapper* e o *Test Driver* o qual será acessado por ambos. A solução é baseada no sisema produtor-consumidor, onde o agente produz informações e o *Test Driver* as cosome.

- **Forças:**

  (1) Colocar agências indisponíveis de forma automática (e.g. via *Test Driver*) pode não ser possível em algumas plataformas, dificultando a automação da execução dos casos de teste.

  (2) Geralmente as plataformas lançam exceções quando os agentes solicitam uma migação para uma agência indisponível, o que pode dificultar a execução dos testes.

- **Modelo Faltas**

  - Não tratamento de exceções lançadas pela plataforma quando uma agência não está disponível, resultando na parada do sistema.

  - Não tratamento de agências que estão após uma agência indisponível no itinerário do agente.

  - Parada de execução de uma agente após este ter tentado migrar para uma agência indisponível.

7.5 Conclusão

Este capítulo apresentou um conceito não utilizado no nosso trabalho até o momento: padrões de teste. Motivados pelos casos de teste extraídos no Capítulo 5, um padrão de teste foi apresentado. Além disso, uma forma de identificar padrões de teste oriundos de especificações de padrões de projetos foi apresentada, em conjunto com um formato de definição para ser utilizado com tais padrões.

Em relação ao formato de definição de padrões proposto neste capítulo, podemos dizer que, provavelmente, será de fácil uso por escritores de padrões pois é baseado em outras propostas já consolidadas pela comunidade [GHJV95; Bin99]. Com a consolidação do uso de padrões de projeto para Agentes Móveis, a metodologia proposta neste capítulo para identificação de padrões de teste poderá ser de grande valia para a comunidade de padrões de teste, pois contribuirá para o aumento do arcabouço de tais padrões.

O conjunto de padrões de teste proposto neste trabalho (Apêndice C), em união com os demais que poderão surgir, vêem para aumentar o arcabouço destes padrões disponíveis tanto para estudo pela comunidade como para uso pelos desenvolvedores que necessitam de formas de validação dos sistemas.

Uma ferramenta para a catalogação de padrões de teste identificados a partir de padrões de projeto para SBAMs foi desenvolvida por Barbosa [Bar05]. Esta ferramenta permite a adição e o acesso aos padrões de teste por parte dos desenvolvedores. Esses padrões de teste são catalogados e apresentados utilizando-se o formato apresentado na Seção 7.2. Esta ferramenta é um interessante incentivo ao uso

deste tipo de padrão de teste, uma vez que facilita o acesso aos mesmos colocando-os em um mesmo repositório e os relacionandos.

Capítulo 8

Conclusão

Como foi apresentado, este trabalho propôs um método para a geração automática de casos de teste para sistemas baseados em Agentes Móveis. Além disso, baseado nos estudos realizados sobre os casos de teste obtidos no estudo de caso e na utilização de padrões de projeto como premissas para a identificação de padrões de teste, um trabalho relacionado com a identificação destes padrões foi apresentado.

O método de geração automática consiste em um processo onde, a partir da especificação do sistema em RPOO e dos objetivos de teste elaborados em LTS, ferramentas dispostas na comunidade são utilizadas para gerar casos de teste. O processo de definição do método contemplou um estudo realizado sobre como as características de mobilidade presentes nos sistemas baseados em Agentes Móveis seriam tratadas pelas ferramentas e linguagens utilizadas. Uma proposta de modelagem de Agentes Móveis com RPOO foi apresentada bem como a forma na qual mobilidade iria se refletir no IOLTS gerado a partir desse modelo. As ferramentas também passaram por um estudo sobre como questões de mobilidade estariam sendo tratadas durante os processos a fim de que pudessem fazer parte do método.

Visando facilitar a aplicabilidade imediata do método por parte dos desenvolvedores de sistemas distribuídos, um procedimento informal para a construção de modelos RPOO a partir de modelos UML-RT foi apresentado. Apesar de não ser uma tranformação automática, esta proposta objetiva tomar proveito de modelos UML-RT já existentes para a construção de modelos RPOO, viabilizando a aplicação do método sobre os sistemas.

No intuito de ganhar confiança acerca da aplicabilidade do método proposto a um sistema real, um estudo de caso foi realizado e seus resultados foram apresentados neste documento. Este estudo foi feito sobre a aplicação do método a uma determinada aplicação, cujos resultados trouxeram fortes indícios a respeito da eficácia dos casos de teste gerados em revelar falhas e no potencial desses casos de teste de serem implementados.

Em um terceiro momento deste documento, o tema padrões de teste para SBAM foi tratado. A partir de um conjunto de casos de teste gerados durante o estudo de caso um padrão de teste foi identificado e documentado. No entanto, a principal contribuição desta parte do trabalho está

na proposta de uma metodologia de identificação de padrões de teste a partir de especificações de padrões de projeto para Agentes Móveis. Além disso, um formato de definição para tais padrões foi proposta no intuito de ser utilizada pela comunidade.

8.1 Contribuições

Como resultado principal do trabalho, temos um método para a geração automática de casos de teste para sistemas baseados em Agentes Móveis. Com tal método, os desenvolvedores poderão agregar às suas metodologias de desenvolvimento o método de teste proposto, aumentando o suporte metodológico e técnico do paradigma. Com uma possível automação (implementação) do método em trabalhos futuros e com trabalhos sobre a geração dos dados de teste e dos oráculos, o desenvolvedor poderá usufruir de uma ferramenta importantíssima para a validação do seu *software*.

Agregada ao método proposto, uma transformação informal de modelos em UML-RT para modelos em RPOO foi apresentada neste trabalho. Com esta proposta, os desenvolvedores de sistemas distribuídos são incentivados a aplicar o método a sistemas modelados em UML-RT, uma vez que eles passam a dispor de uma proposta de transformação informal para a construção de modelos RPOO.

Um outro resultado importante de ser ressaltado neste ponto decorre do fato de RPOO ter sido escolhida como linguagem de especificação dos SBAMs a serem testados. RPOO é um formalismo cujo uso tem crescido com o tempo, inclusive se mostrando interessante para modelagem e especificação de Agentes Móveis [RGG+04; dF03]. Desta forma, propostas de métodos, técnicas e ferramentas que utilizem tal formalismo vêm para contribuir ainda mais para este crescimento.

Algumas outras contribuições que o método de geração automática proporciona podem ser ressaltadas. A seguir as apresentamos.

- A proposta de modelagem de SBAM em UML-RT, apresentada no Capítulo 4, constitui-se de uma importante contribuição para os desenvolvedores de SBAMs.
- A aplicação do estudo de caso propiciou uma importante experimentação de TGV em SBAM. Mesmo que alheio ao método proposto, averiguou-se que a ferramenta TGV pode ser interessante para SBAMs.
- A geração de espaços de estado para sistemas complexos, como SBAMs, foi uma experimentação também importante para as linhas de pesquisa do contexto de RPOO.
- A aplicação do estudo de caso como um todo serviu como importante validação para o sistema utilizado durante esta atividade.

Com o conjunto de padrões de teste identificado, aliado aos já propostos pela comunidade de padrões de teste, o desenvolvedor de sistemas poderá usufruir de um interessante arcabouço de soluções para a validação dos sistemas. Além disso, esperamos contribuir para que novos padrões sejam

propostos, uma vez que a metodologia apresentada para a identificação de padrões, em conjunto com o formato de definição proposto, estimule esta atividade.

8.2 Trabalhos Futuros

Com a conclusão deste trabalho, a oportunidade de outros que, principalmente, complementam este surgem. A seguir temos a descrição de algum desses trabalhos que propomos.

- **Formalização da transformação de UML-RT para RPOO:** Neste trabalho apresentamos uma transformação informal de modelos UML-RT para RPOO, com o intuito de facilitar a adaptação por parte dos desenvolvedores de sistemas distribuídos ao método. O advento da formalização desta transformação poderia propiciar uma automação nesta atividade, tornando o método apto a receber modelos em UML-RT.

- **Trabalhos sobre a identificação dos objetivos de teste:** Assim como ocorre com as propriedades na verificação de modelos, a identificação de objetivos de teste não é uma tarefa simples no processo. A utilização de especificações de padrões de projeto é interessante, porém não é suficiente. Neste sentido, trabalhos sobre a identificação de objetivos de teste, por exemplo, a partir de propriedades especificadas para verificação de modelos, seria de grande valia para o processo de geração de casos de teste.

- **Aplicação do método em outros estudos de caso:** A fim de obter maior confiança acerca da eficácia dos casos de teste gerados pelo método, novos estudos de caso pode ser realizados fazendo uma análise mais rigorosa sobre parâmetros, como cobertura dos casos de teste em relação a especificação.

- **Proposições de novos padrões de teste para Agentes Móveis:** De posse de catálogos de padrões de projeto para Agentes Móveis, como o apresentado por Lima [Lim04], e utilizando-se da metodologia apresentada neste trabalho, novos padrões de teste podem ser propostos para a comunidade, aumentando o arcabouço já existente.

Bibliografia

[Agl] Aglets. http://aglets.sourceforge.net.

[AL98] Yariv Aridor e Danny B. Lange. Agent Design Patterns: Elements of Agent Application Design. Em *Proceedings of the Second International Conference on Autonomous Agents*, páginas 108–115. ACM Press, Maio 1998.

[AMM01] Jean-Paul Arcangeli, Christine Maurel, e Frédéric Migeon. An API for High-Level Software Engineering of Distributed and Mobile Applications. Em 8^{th} *IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS01)*, páginas 155–161. IEEE-CS Press, 2001.

[ASB02] Aline Santos Andrade, Flávio Assis Silva, e Frederico Barboza. Extensão da Linguagem Promela para Especificação de Sistemas baseados em Agentes Móveis. Em *IV Workshop de Comunicação sem Fio (WCSF'2002)*, São Paulo - Brasil, Outubro 2002.

[Bar05] Ana Emília Victor Barbosa. Ferramenta de Catalogação de Padrões de Teste para a Verificação de Aplicações Baseadas em Agentes Móveis. Relatório técnico, Departamento de Sistemas e Computação - CCT - UFCG, Campina Grande, Paraíba, Brasil, Janeiro 2005.

[BCTR03] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, e Giovanni Rimassa. *JADE PROGRAMMER'S GUIDE*. http://jade.cselt.it/docs, Fevereiro 2003.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2^{nd} edition. 1990.

[Bei99] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1999.

[BFdV+99] Axel Belinfante, Jan Feenstra, René de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, e Lex Heerink. Formal Test Automation: A Simple Experiment. Em G. Csopaki, S. Dibuz, e K. Tarnay, editores, 12^{th} *International Workshop on Testing of Communicating Systems*, páginas 179–196. Kluwer Academic Publishers, 1999.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, e Laurent Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. Em *World Congress on Formal Methods (1)*, páginas 307–327, 1999.

[Bin99] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[BMG+04] Peter Braun, Ingo Mller, Sven Geisenhainer, Volkmar Schau, e Wilhelm R. Rossak. Agent Migration as an Optional Service in an Extendable Agent Toolkit Architecture. Em Ahmed Karmouch, Larry Korba, e Edmundo Madeira, editores, *Proceedings of the First International Workshop on Mobility Aware Technologies and Applications (MATA 2004)*, volume 3284 de *Lecture Notes in Computer Science*, páginas 127–136, Florianópolis - Brasil, Outubro 2004. Springer Verlag.

[CGP99] Edmund M. Clarke, Orna Grumberg, e Doron Peled. *Model Checking*. MIT Press, 1999.

[CHK97] David Chess, Colin Harrison, e Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Em Jan Vitek e Christian F. Tschudin, editores, *Mobile Object Systems: Towards the Programmable Internet (MOS'96), Linz (Austria), Julho 1996 (Selected Presentations and Invited Papers)*, volume 1222 de *Lecture Notes in Computer Science*, páginas 25–45. Springer-Verlag, 1997.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, e Elena Zinovieva. STG: a Symbolic Test Generation tool. Em *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 de *LNCS*, Grenoble, França, Abril 2002. Springer-Verlag.

[CSW03] Ana Cavalcanti, Augusto Sampaio, e Jim Woodcock. A Unified Language of Classes and Processes. Em *St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering*, Satellite Workshop at FM'03, 2003.

[dF03] André Luiz Lima de Figueiredo. Padrões de Projetos para Agentes Móveis. Relatório técnico, COPIN, UFCG, Maio 2003.

[DV03] Marcio Delamaro e Auri Vincenzi. Structural Testing of Mobile Agents. Em Nicolas Guelfi, editor, *International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI 2003)*, Lecture Notes on Computer Science. Springer, Novembro 2003.

[DW99] Dwight Deugo e Michael Weiss. A Case for Mobile Agent Patterns. Em *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3)" at Autonomous Agents '99*, páginas 19–22, Maio 1999.

[FAM04] André Figueiredo, Antônio Almeida, e Patrícia Machado. Identifying and documenting test patterns from mobile agent design patterns. Em Ahmed Karmouch, Larry Korba, e Edmundo Madeira, editores, *Proceedings of the First International Workshop on Mobility Aware Technologies and Applications (MATA 2004), Florianopolis (Brazil), October 2004*, volume 3284 de *Lecture Notes in Computer Science*, páginas 359–368, Florianópolis - Brasil, Outubro 2004. Springer Verlag.

[Fer89] Jean-Claude Fernandez. Aldebaran: A Tool for Verification of Communicating Processes. Relatório técnico, Rapport SPECTRE, C14, Laboratoire de Génie Informatique - Institut IMAG, Grenoble - França, Setembro 1989.

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, e Cesar Viho. Using on-the-fly Verification Techniques for the Generation of Test Suites. Em Rajeev Alur e Thomas A. Henzinger, editores, 8^{th} *International Conference on Computer Aided Verification CAV*, volume 1102, páginas 348–359, New Brunswick, NJ, USA, 1996. Springer Verlag.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, e Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, Maio 1998.

[Gau95] Marie-Claude Gaudel. Testing Can Be Formal, Too. Em *TAPSOFT'95: Theory and Practice of Software Development,* 6^{th} *International Joint Conference CAAP/FASE*, volume 915 de *Lecture Notes in Computer Science*, páginas 82–96, Aarhus, Denmark, Maio 1995. Springer.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company Co., Inc., New York, NY, 1995.

[Gmb98] IKv++ GmbH. *Grasshopper - A Platform for Mobile Software Agents*. http://www.grasshopper.de, 1998.

[Gmb01] IKV++ GmbH. *Grasshopper Programmer's Guide - Release 2.2*. http://www.grasshopper.de, Março 2001.

[GMM03] Fabiana Guedes, Patrícia Machado, e Vivianne Medeiros. Developing Mobile Agent-Based Applications. Em 29^{th} *Conferência Latino Americana de Informática - CLEI 2003*, La Paz, 2003.

[Gue02a] Fabiana Guedes. Um Modelo para o Desenvolvimento de Aplicações Baseadas em Agentes Móveis. Dissertação de Mestrado, Universidade Federal de Campina Grande, 2002.

[Gue02b] Dalton Dario Serey Guerrero. *Redes de Petri Orientadas a Objeto*. Tese de Doutorado, COPELE, UFCG, 2002.

[HLU03] Olaf Henniger, Miao Lu, e Hasan Ural. Automatic Generation of Test Purposes for Testing Distributed Systems. Em 3^{rd} *International Workshop on Formal Approaches to Testing of Software, FATES'03*, volume 2931, páginas 78–191, Montreal, Quebec - Canadá, Outubro 2003. Lecture Notes in Computer Science.

[IJB99] James Rumbaugh Ivar Jacobson e Grady Booch. *The Unified Modeling Language*. Addison-Wesley Professional, 1999.

[Jar02] Claude Jard. Principles of Distributed Test Synthesis based on True-concurrency Models. Em Ina Schieferdecker, Hartmut König, e Adam Wolisz, editores, *TestCom*, volume 210 de *IFIP Conference Proceedings*, páginas 301–316. Kluwer, 2002.

[Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 de *Monographs in Theoretical Computer Science*. Springer-Verlag, 1992.

[JJ02] Claude Jard e Thierry Jéron. TGV: theory, principles and algorithms. Em 6^{th} *World Conference on Integrated Design & Process Technology - IDPT'02*, Pasadena, California, USA, Junho 2002.

[KCJ98] Lars M. Kristensen, Soren Christensen, e Kurt Jensen. The practitioners Guide to Coloured Pertri Nets. *International Journal on Software Tools for Technology Transfer STTT*, 2(2):98–132, 1998.

[KGHS98] Beat Koch, Jens Grabowski, Dieter Hogrefe, e Michael Schmitt. Autolink - A Tool for Automatic Test Generation from SDL Specifications. Em *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT98)*, páginas 21–23, Boca Raton, Florida, Outubro 1998.

[KKPS98] Elizabeth A. Kendall, P. V. Murali Krishna, Chirag V. Pathak, e C. B. Suresh. Patterns of Intelligent and Mobile Agents. Em Katia P. Sycara e Michael Wooldridge, editores, *Proceedings of the* 2^{nd} *International Conference on Autonomous Agents (Agents'98)*, páginas 92–99, New York, 1998. ACM Press.

[KRSW01] Cornel Klein, Andreas Rausch, Marc Sihling, e Zhaojun Wen. Extension of the Unified Modeling Language for mobile agents. Em Keng Siau e Terry Halpin, editores, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 8, páginas 116–128. Idea Publishing Group, 2001.

[Lan01] Manfred Lange. It's Testing Time! - Patterns for Testing Software. Em *Proceedings of* 6^{th} *European Conference on Pattern Languages of Programs - EuroPLoP2001*, Bad Irsee, Germany, Julho 2001.

[Lar99] Craig Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, Inc., 1999.

[LFG03] Emerson Lima, Jorge Figueiredo, e Dalton Guerrero. Comparative Study of Mobile Agent Design Patterns: a Coloured Petri Nets Approach. Em *VI Workshop de Métodos Formais*, Campina Grande - Paraíba - Brasil, Outubro 2003.

[LFG04] Emerson Lima, Jorge Figueiredo, e Dalton Guerrero. Using Coloured Petri Nets to Compare Mobile Agent Design Patterns. *Electronic Notes in Theoretical Computer Science - Selected Papers from VI Workshop on Formal Methods*, 95:287–305, Maio 2004.

[LG02] Grégory Lestiennes e Marie-Claude Gaudel. Testing Processes from Formal Specifications with Inputs, Outputs and Data Types. Em 13^{th} *International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 Novembro 2002, Annapolis, MD, USA*, páginas 3–14. IEEE Computer Society, 2002.

[Lim04] Emerson Lima. Formalização e Análise de Padrões de Projeto para Agentes Móveis. Dissertação de Mestrado, Universidade Federal de Campina Grande, 2004.

[LMFR03] Emerson Lima, Patrícia Machado, Jorge Figueiredo, e Flávio Ronison. Implementing Mobile Agent Design Patterns in the JADE Framework. jade.cselt.it, 2003.

[LMSF04] Emerson Lima, Patrícia Machado, Flávio Sampaio, e Jorge Figueiredo. An Approach to Modelling and Applying Mobile Agent Design Patterns. *SIGSOFT Softw. Eng. Notes*, 29(3):1–8, 2004.

[Mar03] Paulo Marques. *Component-based Development of Mobile Agent Systems*. Tese de Doutorado, Universidade de Coimbra, Departamento de Engenharia Informática, Faculdade de Ciências e Tecnologia, Universidade de Coimbra, Julho 2003.

[Mik98] Tommi Mikkonen. Formalizing Design Patterns. Em *Proceedings of the* 20^{th} *international conference on Software engineering*, páginas 115–124. IEEE Computer Society, 1998.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[MMG02] Vivianne Medeiros, Patrícia Machado, e Fabiana Guedes. Desenvolvimento de Aplicações Baseadas em Agentes Móveis. Em *ENIC'2002*, João Pessoa - PB - Brasil, Dezembro 2002. Anais do X Encontro de Iniciação Científica da UFPB.

[MS01] John McGregor e David Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

[Pei02] Holger Peine. Application and Programming Experience with the Ara Mobile Agent System. *Software - Practice & Experience*, 32(6):515–541, 2002.

[Pet92] James Peterson. *Petri Net Theory and the Modeling of Systems*, volume 2. Prentice-Hall, Englewood Cliffs, New Jersey-USA, 1992.

[PJLT+02] Simon Pickin, Claude. Jard, Yves Le Traon, Thierry Jéron, Jean-Marc Jezequel, e Alain Le Guennec. System Test Synthesis from UML Models of Distributed Software. Em *Forte 2002, 22^{nd} IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, Houston, Texas*, LNCS. Springer, Novembro 2002.

[RBGF04] Cássio Rodrigues, Paulo Barbosa, Dalton Guerrero, e Jorge Figueiredo. RPOO Model Checker. Em *Tools Session - SBES'2004*, Brasília - Brasil, Outubro 2004.

[RGG+04] Cássio Rodrigues, Fabrício Guerra, Dalton Guerrero, Jorge Figueiredo, e Taciano Silva. Modeling and Verification of Mobility Issues using Object-oriented Petri Nets. Em 3^{rd} *International Information and Telecommunication Technologies Symposium*, São Carlos, São Paulo - Brasil, Dezembro 2004.

[RM00] Sita Ramakrishnan e John McGregor. Modelling and Testing OO Distributed Systems with Temporal Logic Formalisms. Em 18^{th} *International IASTED Conference Applied Informatics*, 2000.

[Rod04] Cássio Leonardo Rodrigues. Verificação de Modelos em Redes de Petri Orientadas a Objetos. Dissertação de Mestrado, Coordenação de Pós-graduação em Informática - COPIN, UFCG, 2004.

[RSM05] Rodrigo Ramos, Augusto Sampaio, e Alexandre Mota. A Semantics for UML-RT Active Classes via Mapping into Circus. Em Martin Steffen e Gianluigi Zavattaro, editores, 7^{th} *IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 3535 de *Lecture Notes in Computer Science*, páginas 99–114, Atenas, Grécia, Junho 2005. Springer.

[SGW94] Bran Selic, Garth Gullekson, e Paul Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[Sil05] Taciano Silva. Simulação Automática e Geração de Espaço de Estados de Modelos RPOO. Dissertação de Mestrado, Coordenação de Pós-graduação em Informática - COPIN, UFCG, Campina Grande - PB, 2005.

[SMR03] Augusto Sampaio, Alexandre Mota, e Rodrigo Ramos. Class and Capsule Refinement in UML for Real Time. Em *Proceedings of the 6^{th} Brazilian Workshop on Formal Methods*, páginas 16–34, Campina Grande, Brasil, Outubro 2003.

[SR98] Bran Selic e James Rumbaugh. Using UML for Modelling Complex Real-Time Systems. *ObjecTime Limited/Rational Software White Paper*, 1998.

[SSJ02] Ajay Kr. Singh, Ravi Sankar, e Vikram Jamwal. Design Patterns for Mobile Agent Applications. Em *Workshop on Ubiquitous Agents on embedded, wearable, and mobile devices held in conjunction with the Conference on Autonomous Agents and Multiagent Systems*, Bologna - Itália, 2002.

[TOH99] Yasuyuki Tahara, Akihiko Ohsuga, e Shinichi Honiden. Agent System Development Method Based on Agent Patterns. Em *Proceedings of the* 21^{st} *international conference on Software engineering*, páginas 356–367. IEEE Computer Society Press, 1999.

[TOH01] Yasuyuki Tahara, Akihiko Ohsuga, e Shinichi Honiden. Behavior Patterns for Mobile Agent Systems from the Development Process Viewpoint. Em *Fifth International Symposium on Autonomous Decentralized Systems*, páginas 239– 242, 2001.

[Tre99] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. Em J.C.M Baeten e S. Mauw, editores, *CONCUR'99* – 10^{th} *Int. Conference on Concurrency Theory*, volume 1664 de *Lecture Notes in Computer Science*, páginas 46–65. Springer-Verlag, 1999.

[TTOH01] Yasuyuki Tahara, Nobukazu Toshiba, Akihiko Ohsuga, e Shinichi Honiden. Secure and Efficient Mobile Agent Application Reuse Using Patterns. Em *Proceedings of the 2001 Symposium on Software reusability*, páginas 78–85. ACM Press, 2001.

[WLPS00] Guido Wimmel, Heiko Loetzbeyer, Alexander Pretschner, e Oscar Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Software Testing, Verification & Reliability*, 10(4):229–248, 2000.

[XD00] Dianxiang Xu e Yi Deng. Modeling Mobile Agent Systems with High-Level Petri Nets. Em *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, páginas 3177–3182, Outubro 2000.

Apêndice A

Modelos do Sistema de Conferência

Este apêndice apresenta todos os modelos utilizados e gerados durante o estudo de caso. Na Seção A.1 são apresentados os modelos elaborados para o sistema em UML-RT, que são um diagrama de classes e um diagrama de estados para cada cápsula do diagrama. Na Seção A.2 são apresentados os modelos RPOO originados dos modelos UML-RT citados acima. Esta seção traz o diagrama de classes obtido e uma rede de Petri para cada diagrama de estados do modelo UML-RT.

A.1 Modelos em UML-RT

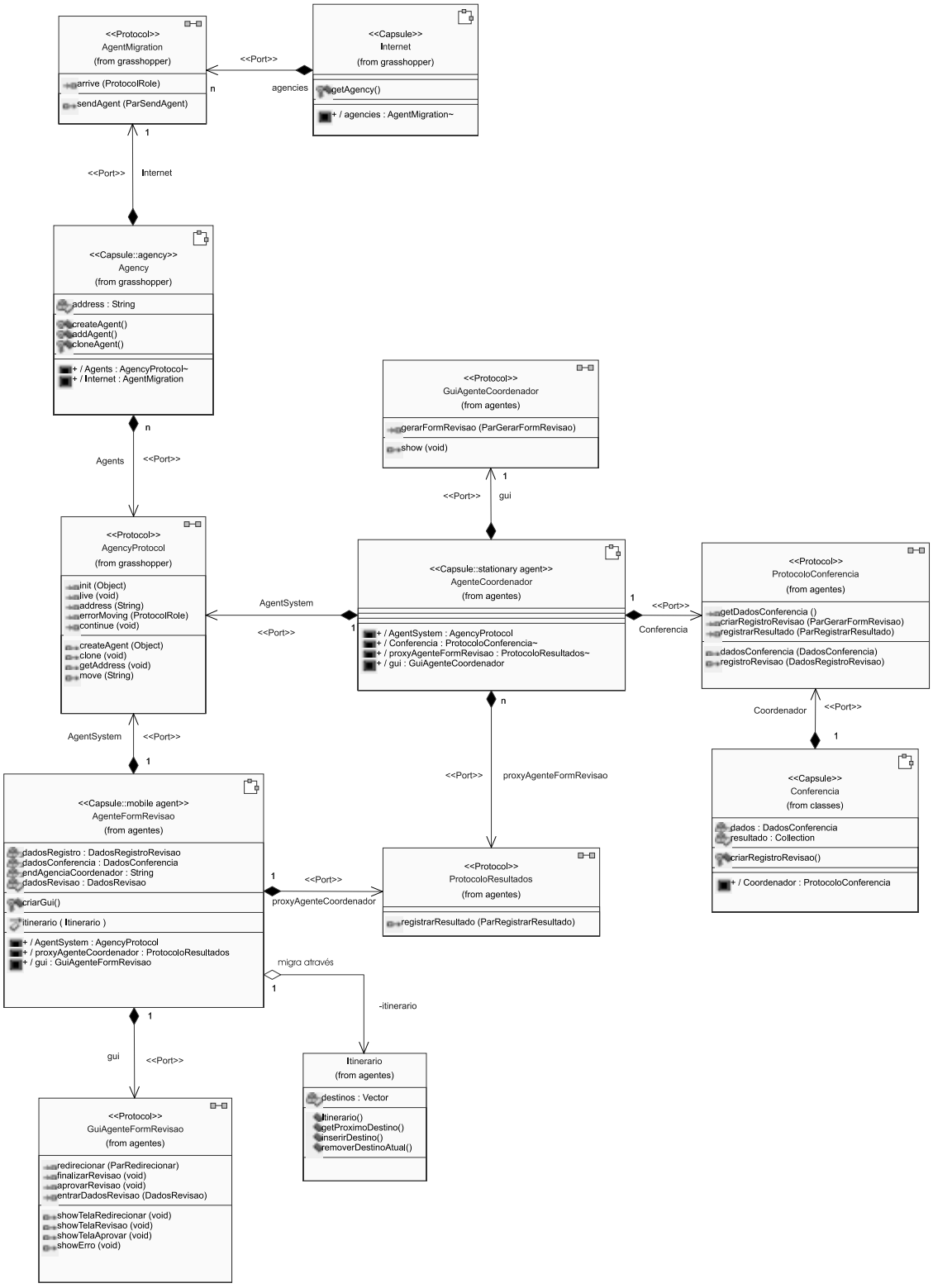

Figura A.1: Diagrama de Classes UML-RT do Sistema de Conferências

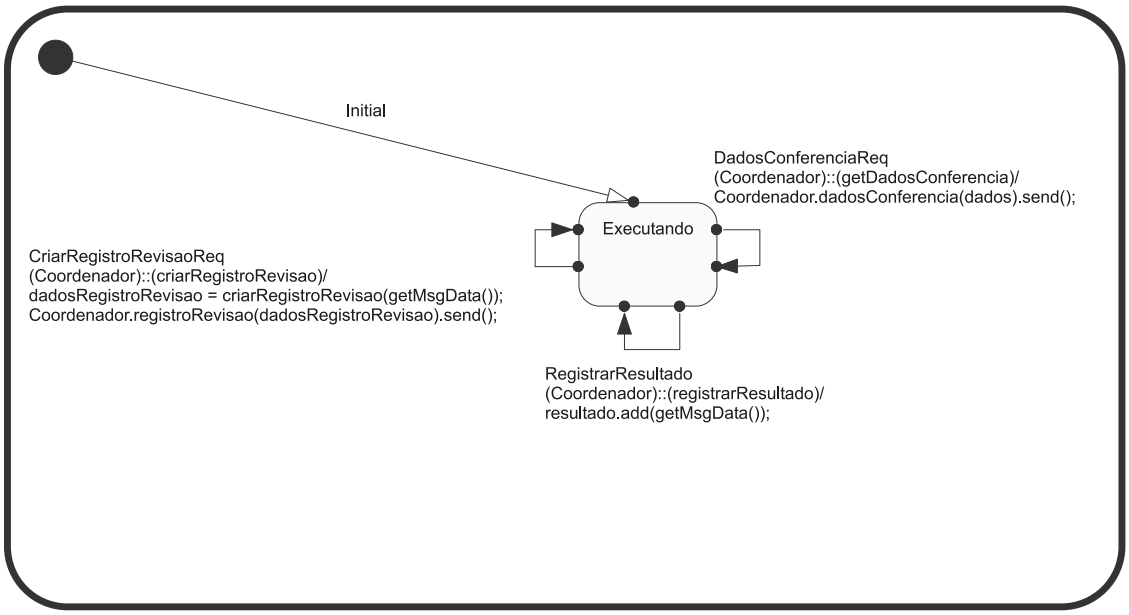

Figura A.2: Diagrama de Estados UML-RT da Cápsula **Conferencia**

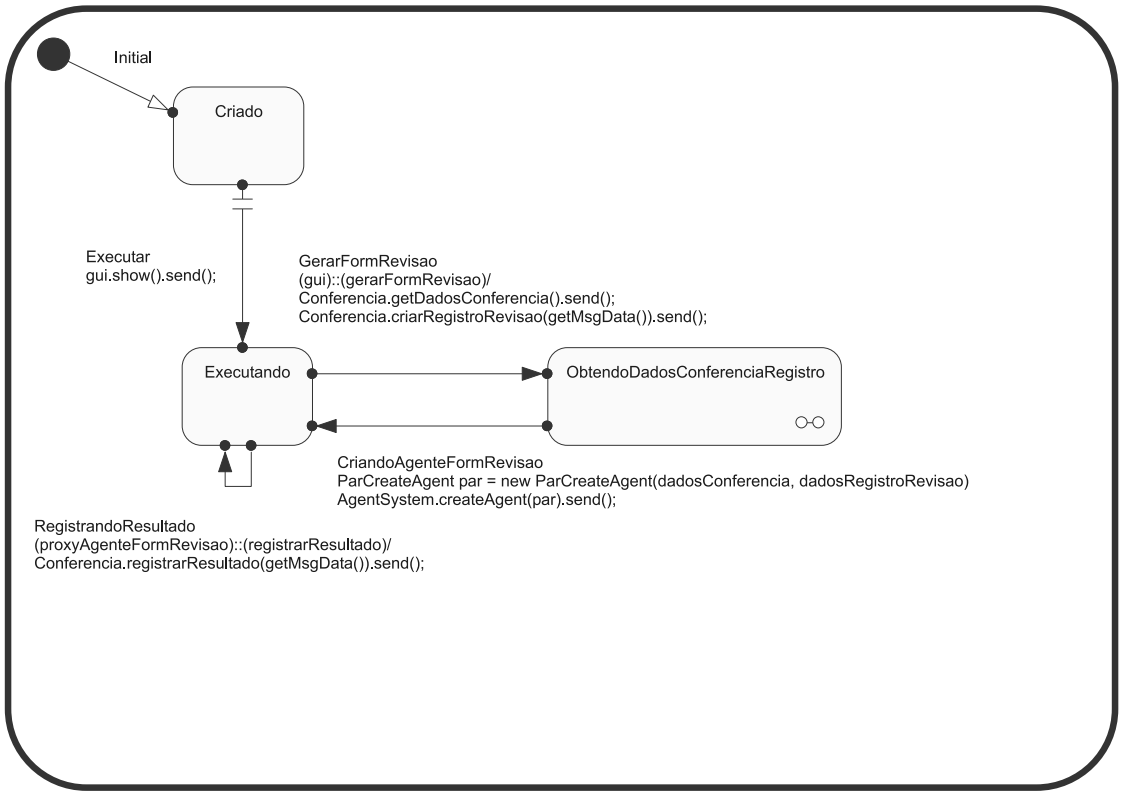

Figura A.3: Diagrama de Estados UML-RT da Cápsula **AgenteCoordenador**

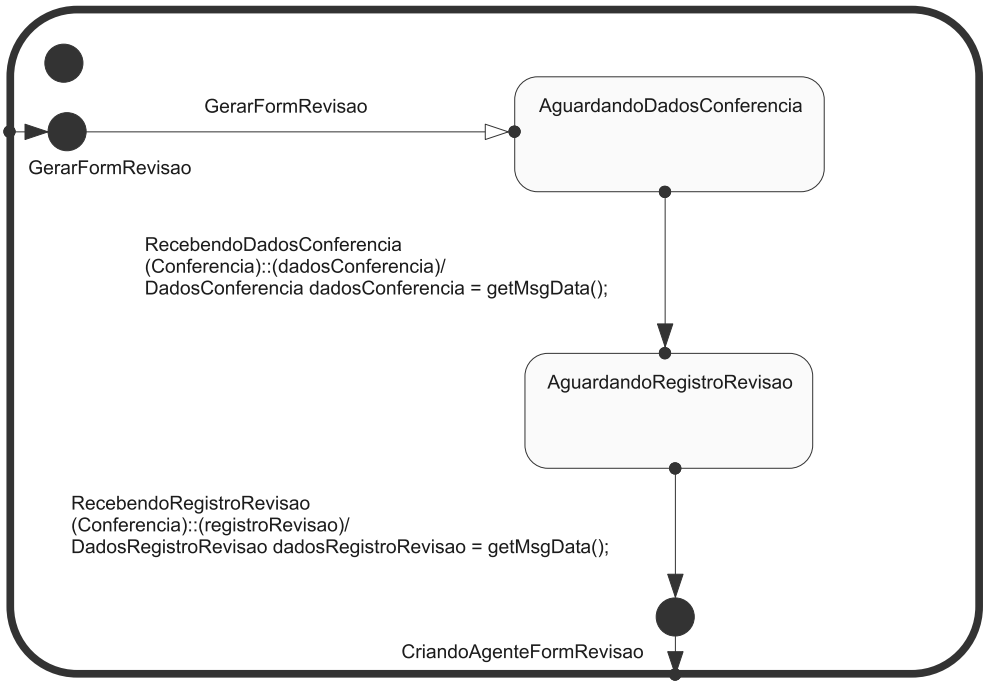

Figura A.4: Diagrama de Estados UML-RT que Detalha o Estado **ObtendoDadosConferenciaRegistro** da Cápsula **AgenteCoordenador**

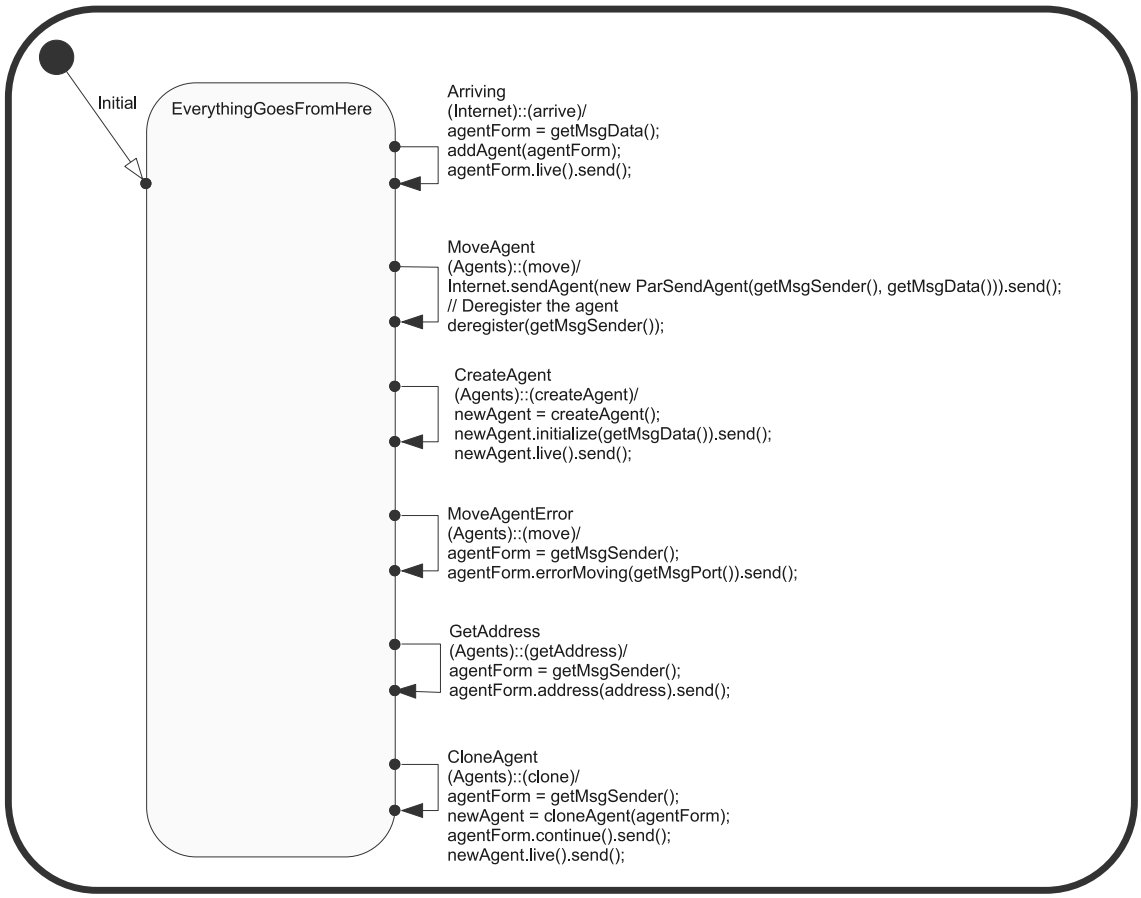

Figura A.5: Diagrama de Estados UML-RT da Cápsula **Agency**

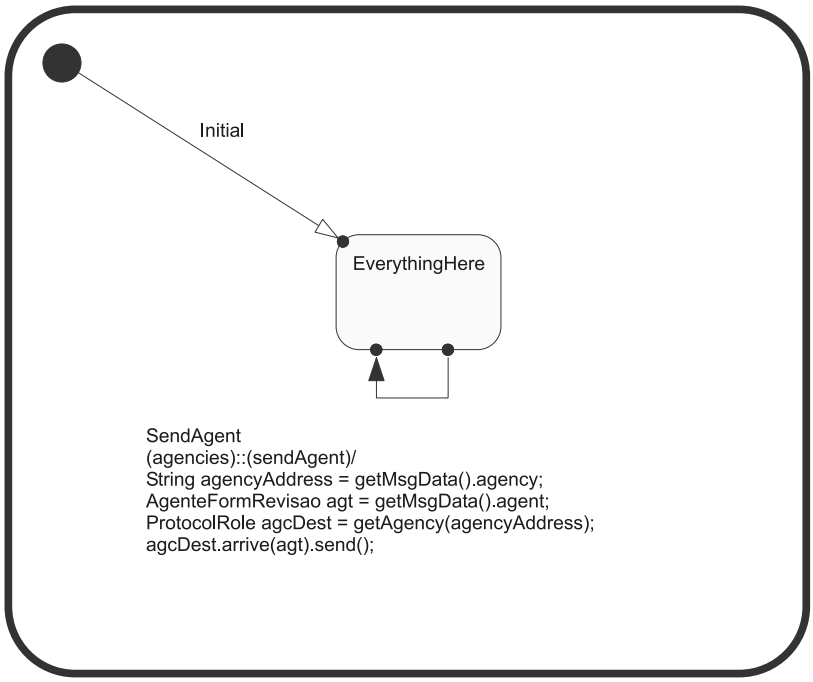

Figura A.6: Diagrama de Estados UML-RT da Cápsula **Internet**

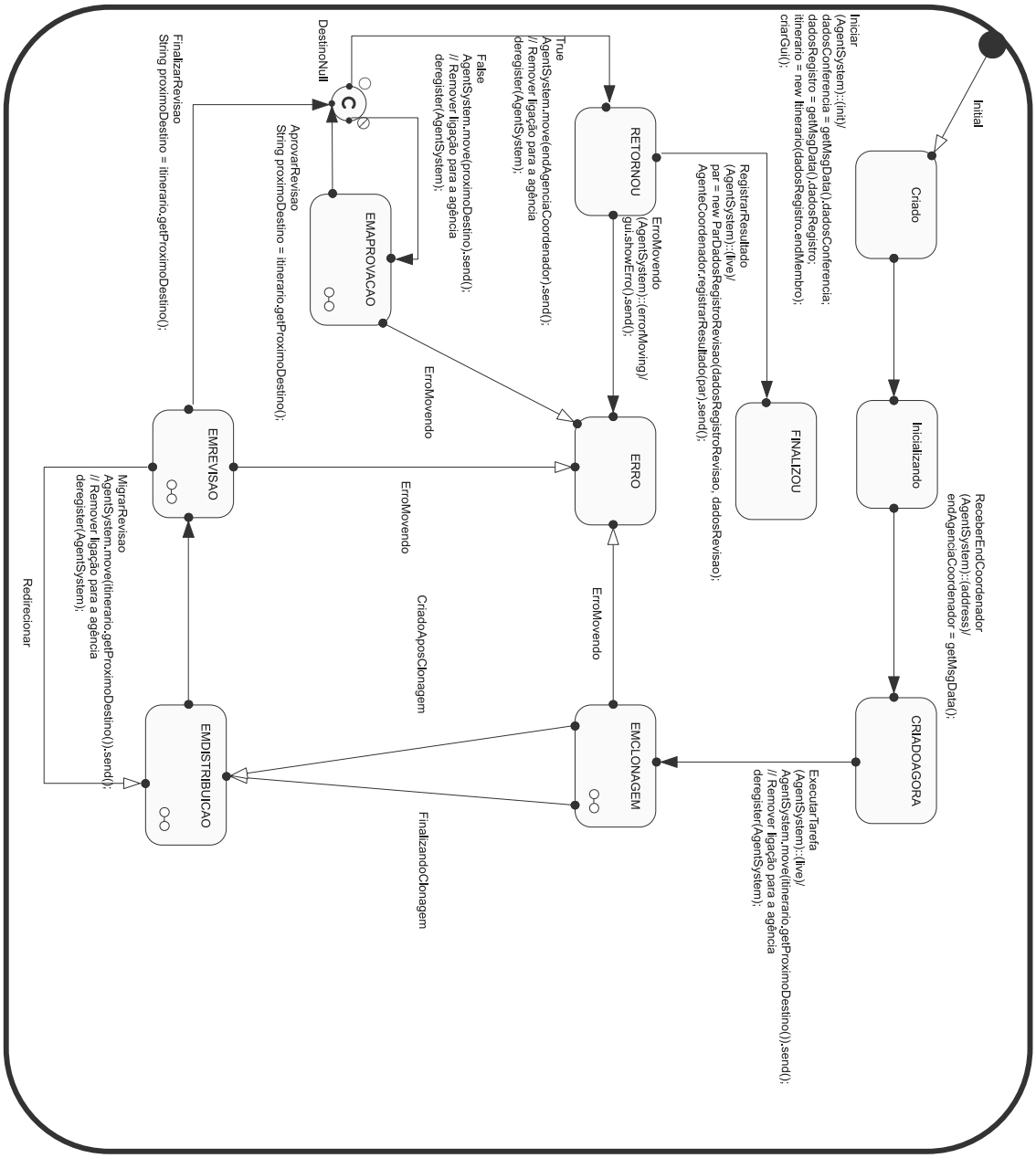

Figura A.7: Diagrama de Estados UML-RT da Cápsula **AgenteFormRevisao**

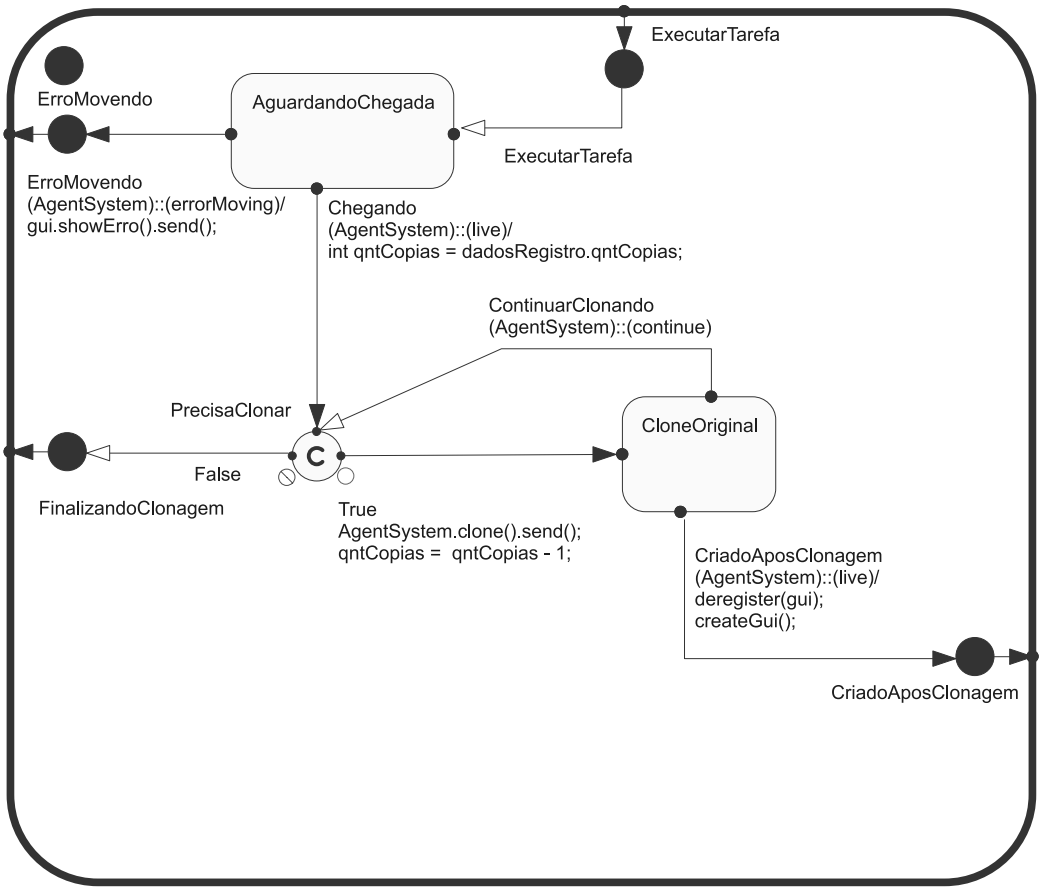

Figura A.8: Diagrama de Estados UML-RT que Detalha o Estado **EMCLONAGEM** da Cápsula **AgenteFormRevisao**

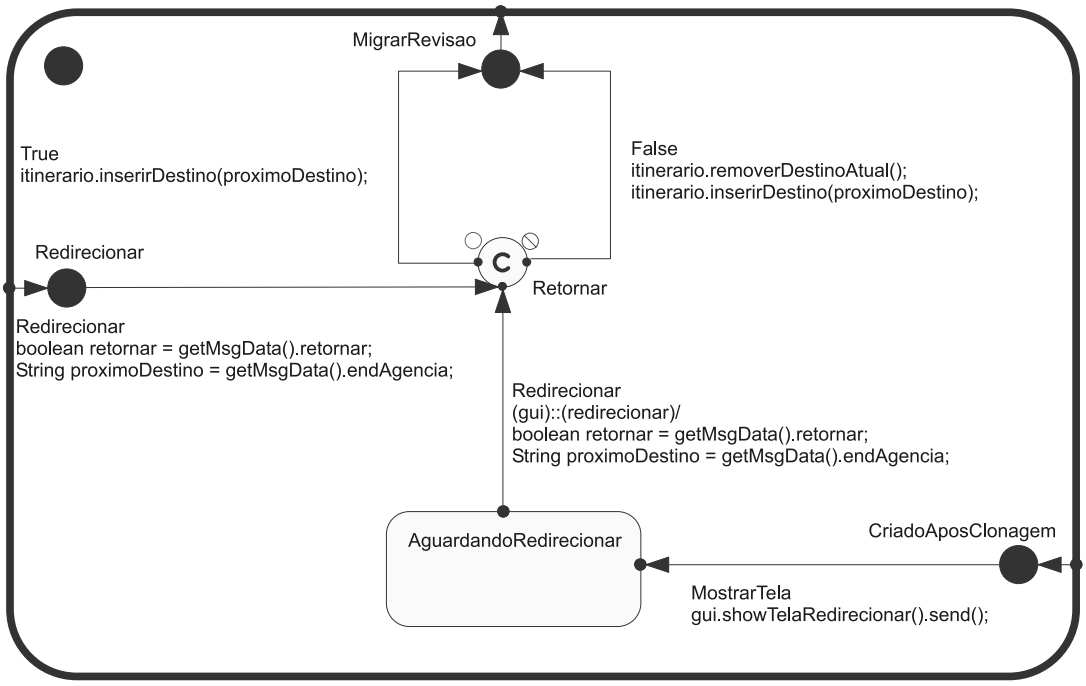

Figura A.9: Diagrama de Estados UML-RT que Detalha o Estado **EMDISTRIBUICAO** da Cápsula **AgenteFormRevisao**

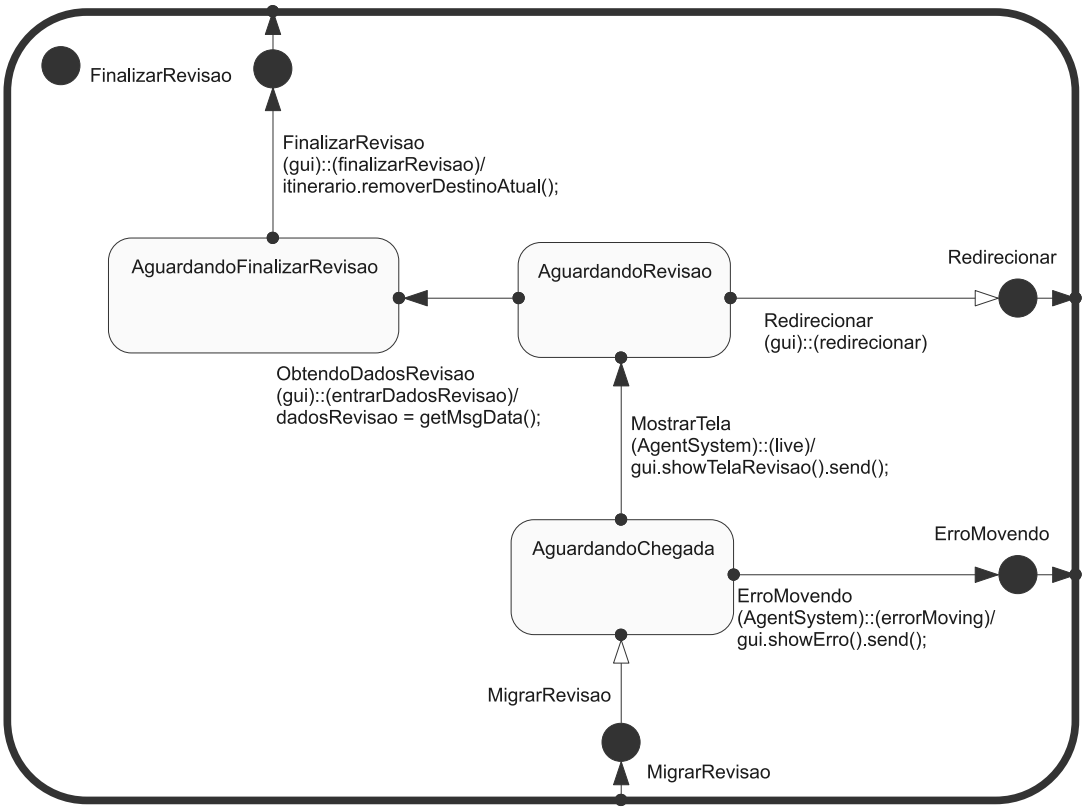

Figura A.10: Diagrama de Estados UML-RT que Detalha o Estado **EMREVISAO** da Cápsula **AgenteFormRevisao**

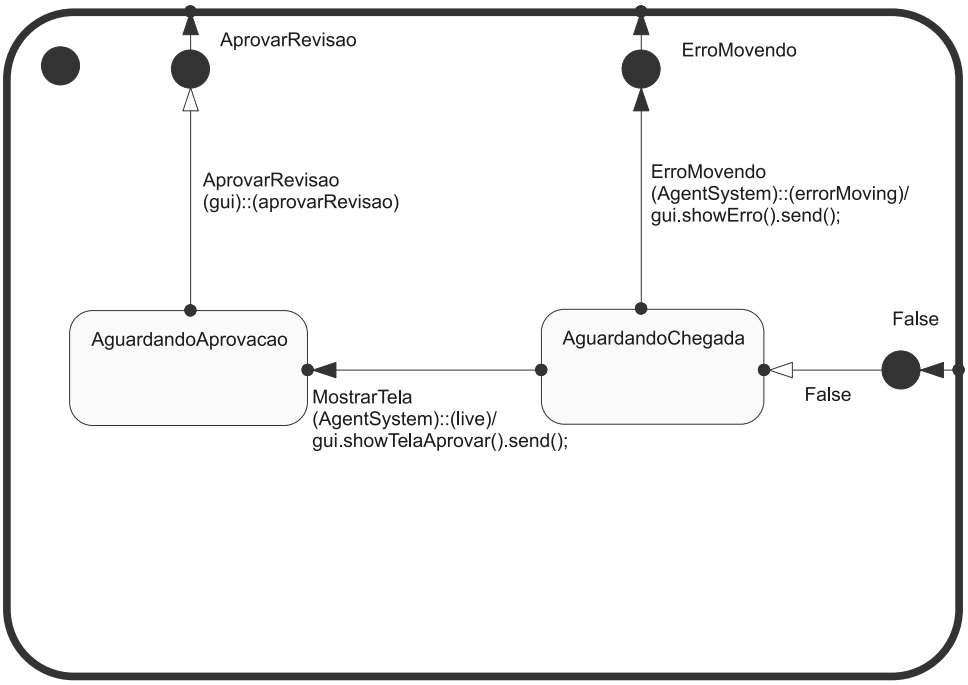

Figura A.11: Diagrama de Estados UML-RT que Detalha o Estado **EMAPROVACAO** da Cápsula **AgenteFormRevisao**

A.2 Modelos em RPOO

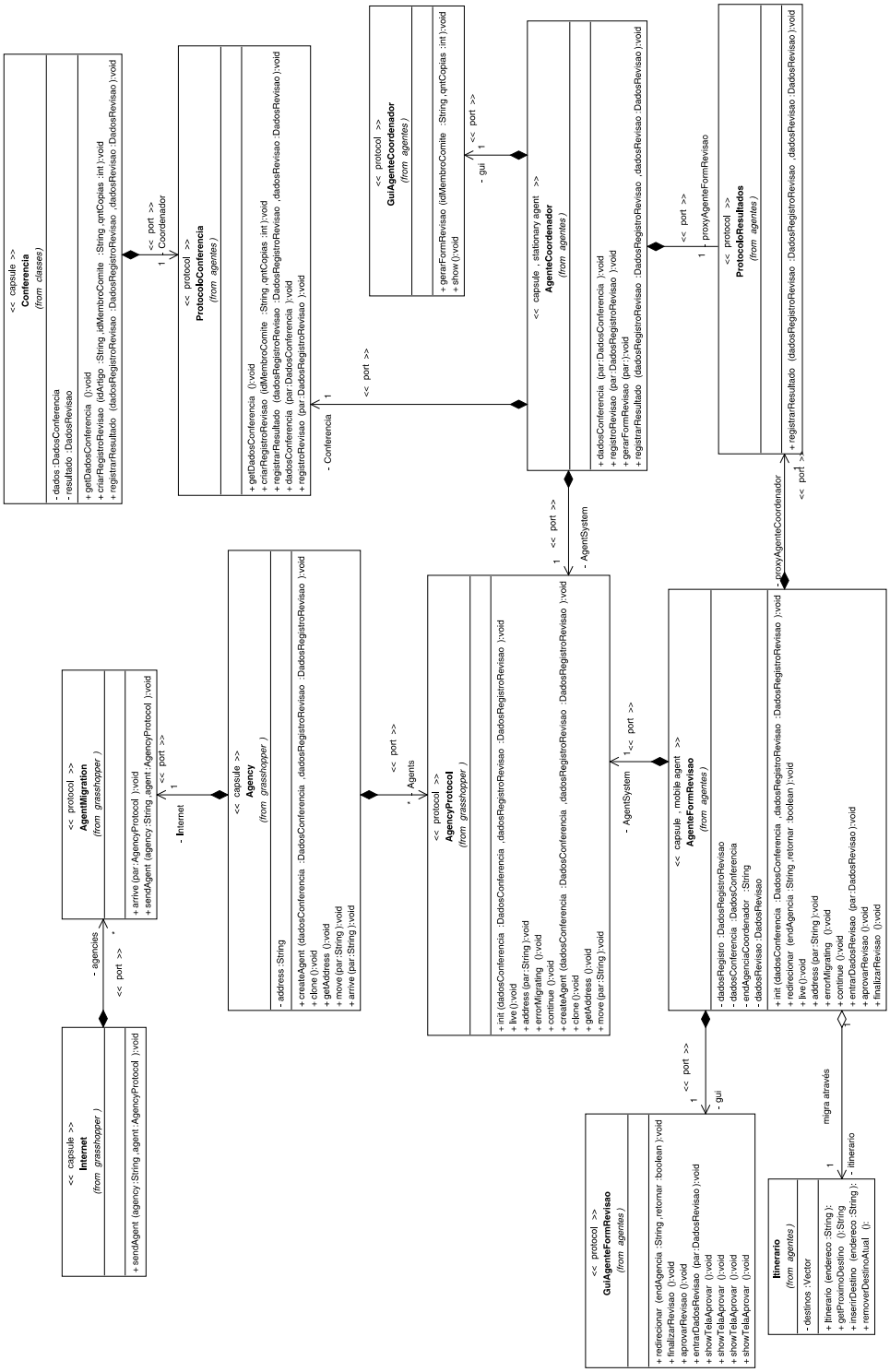

Figura A.12: Diagrama de Classes RPOO do Sistema de Conferências

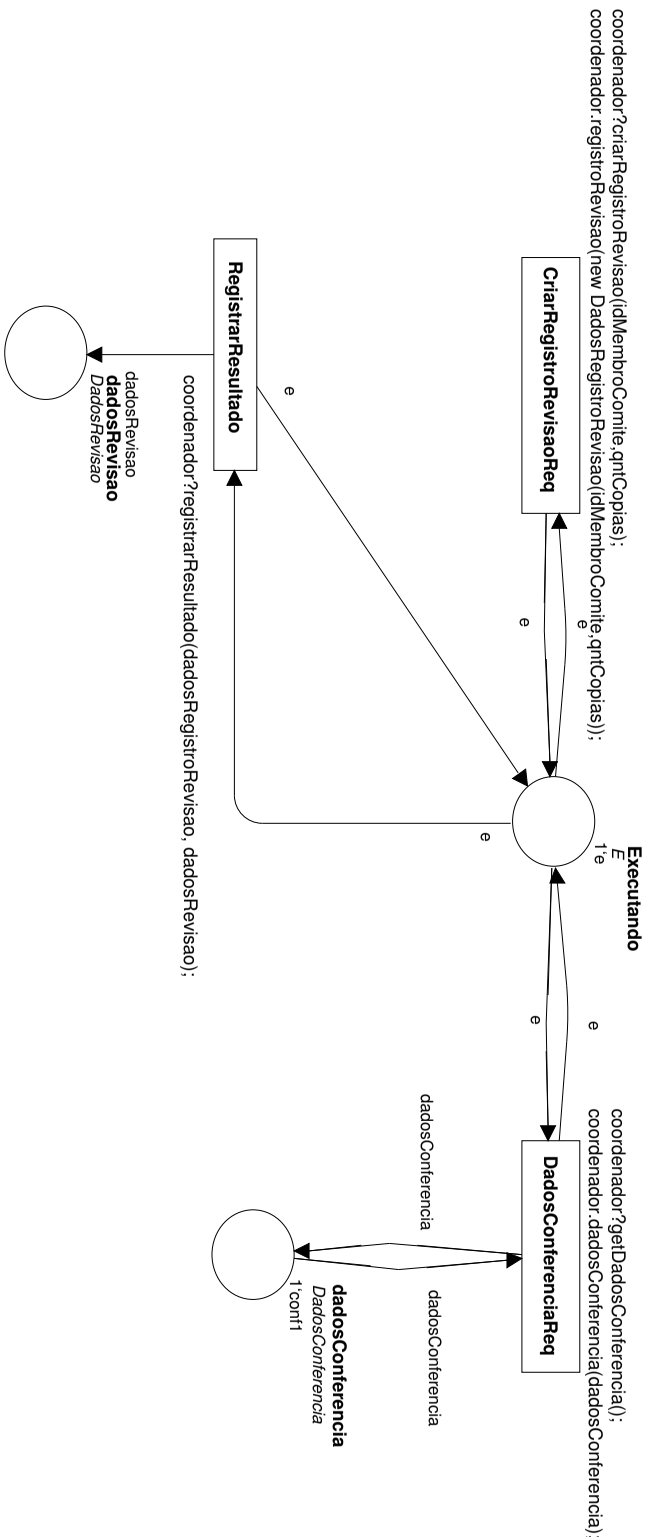

Figura A.13: Rede de Petri da Classe **Conferencia**

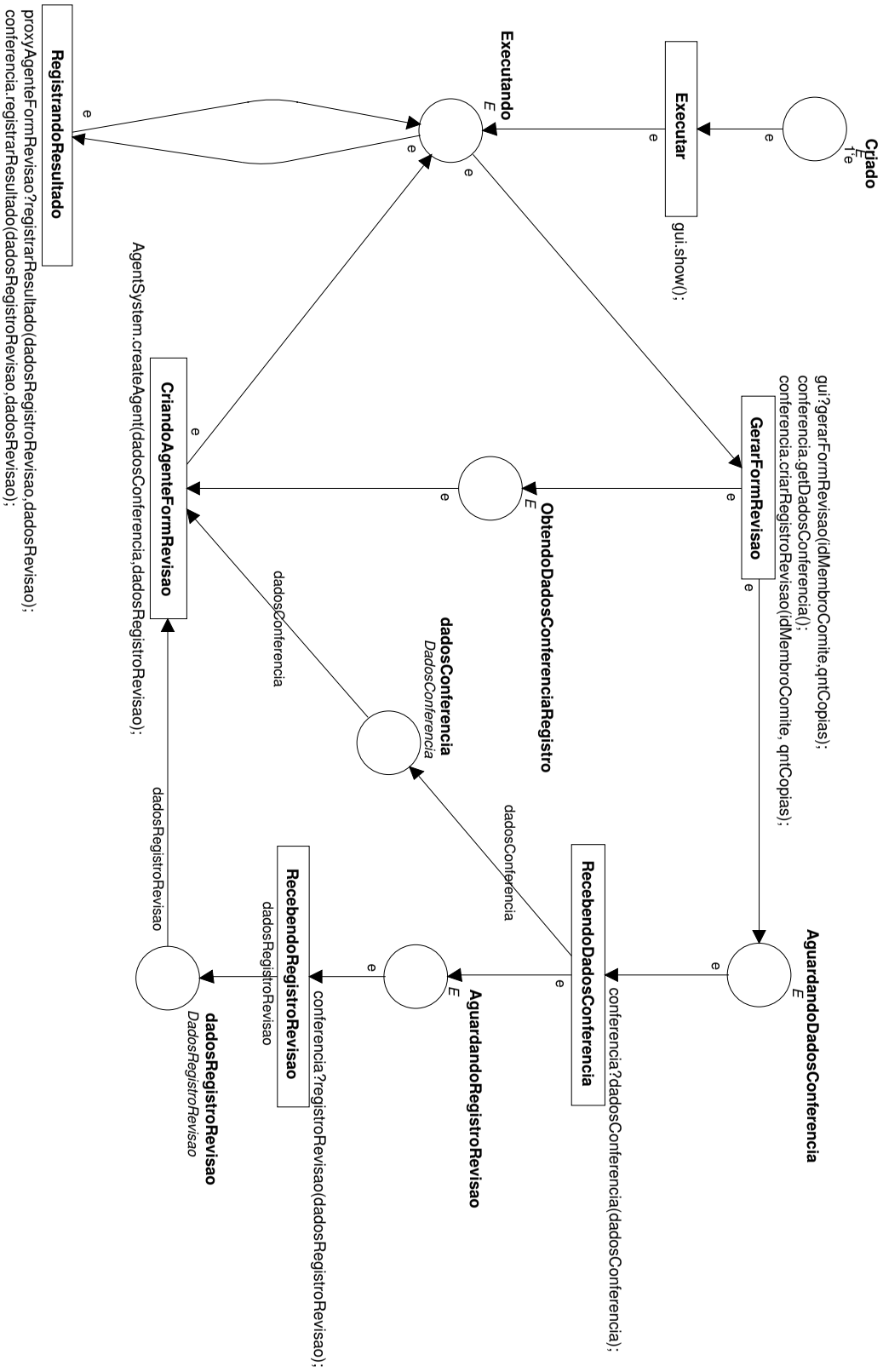

Figura A.14: Rede de Petri da Classe **AgenteCoordenador**

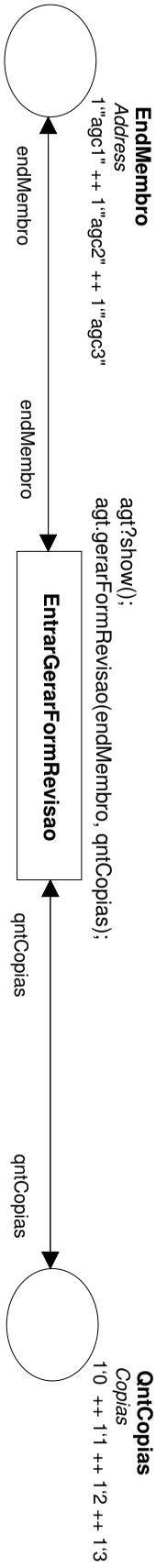

Figura A.15: Rede de Petri da Classe **GuiAgenteCoordenador**

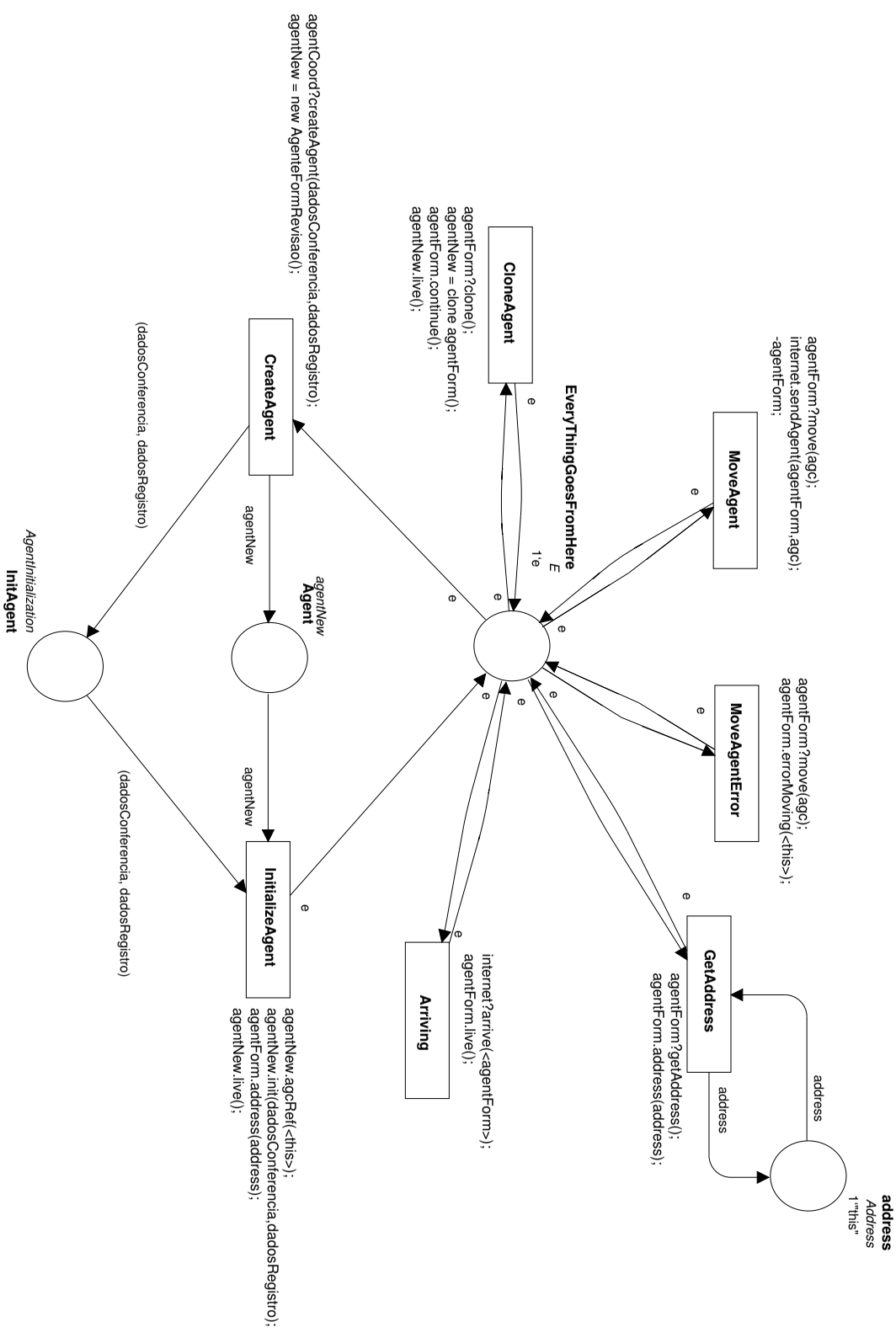

Figura A.16: Rede de Petri da Classe **Agency**

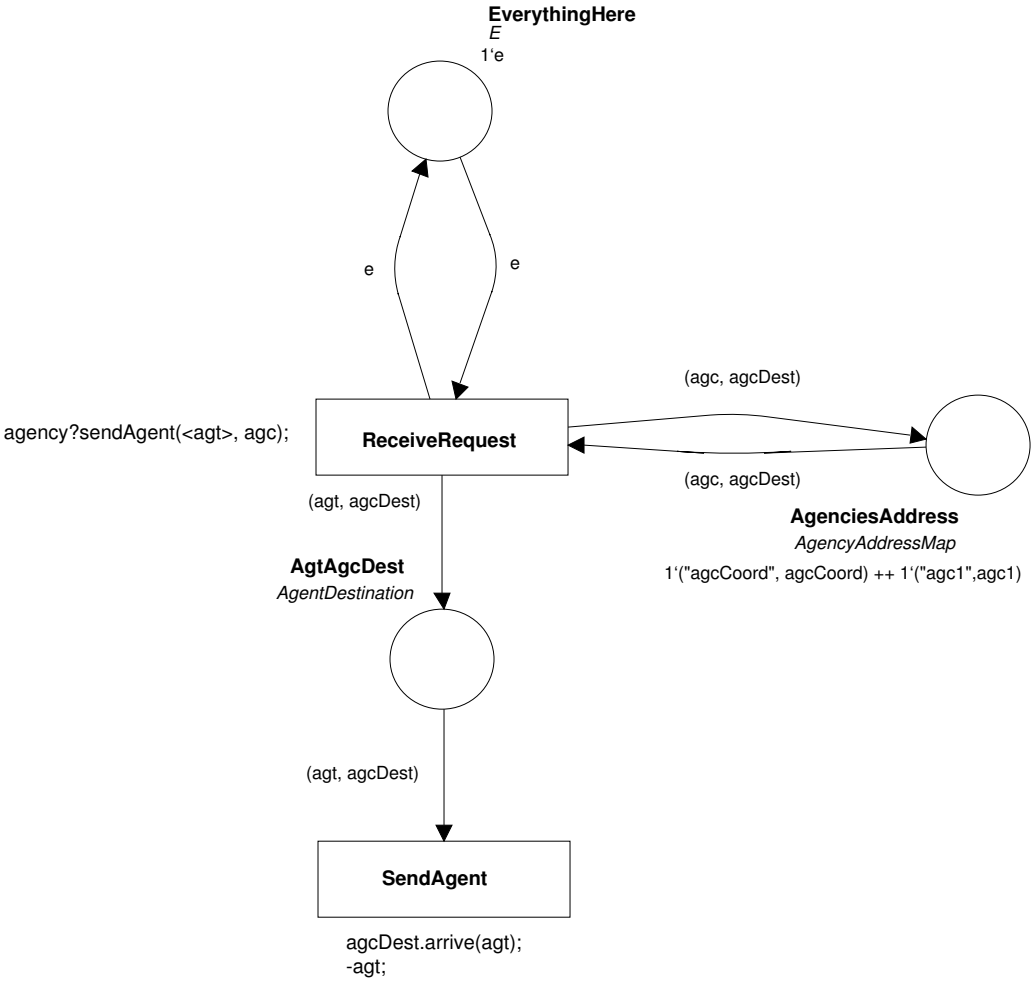

Figura A.17: Rede de Petri da Classe **Internet**

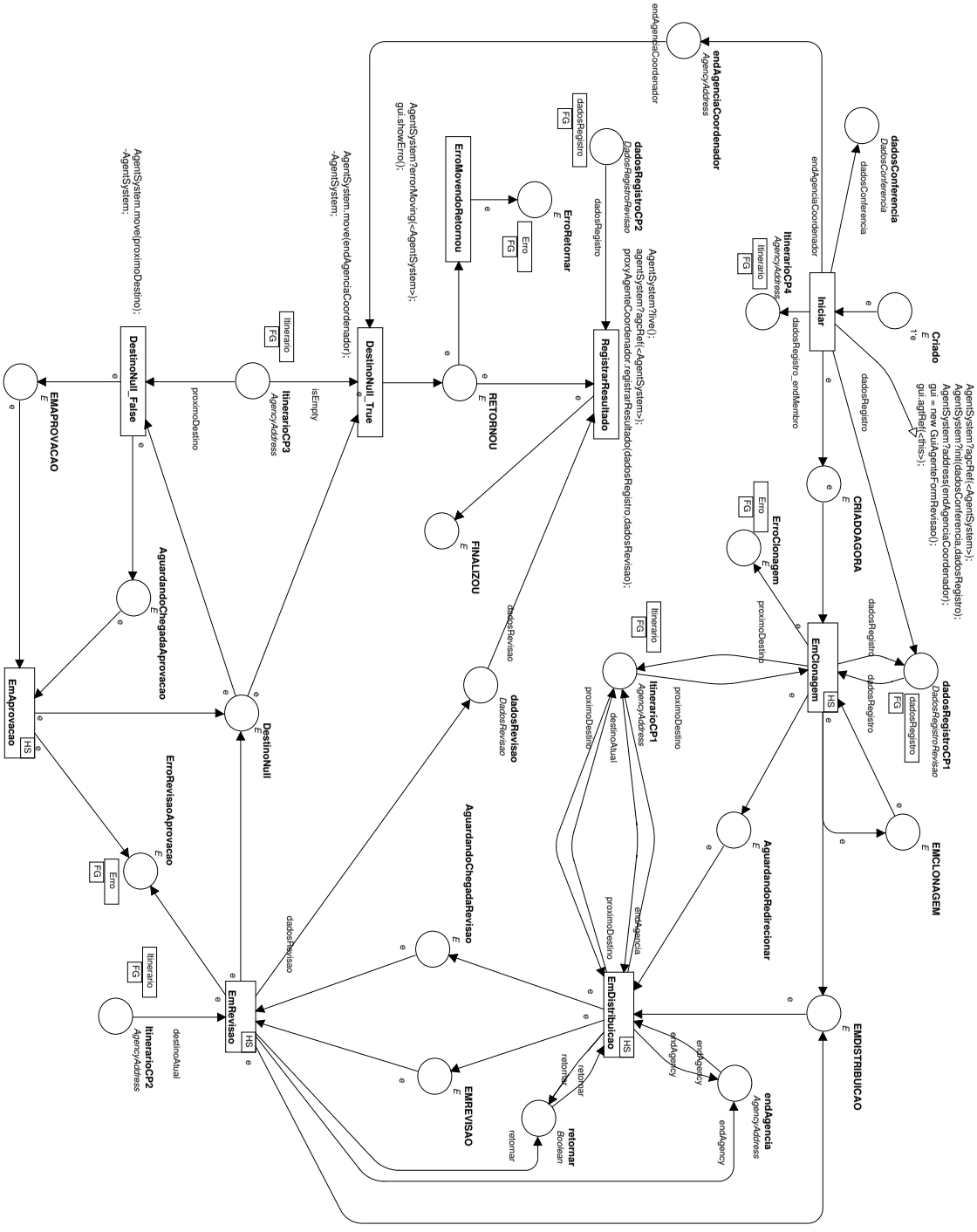

Figura A.18: Página Principal da Rede de Petri da Classe **AgenteFormRevisao**

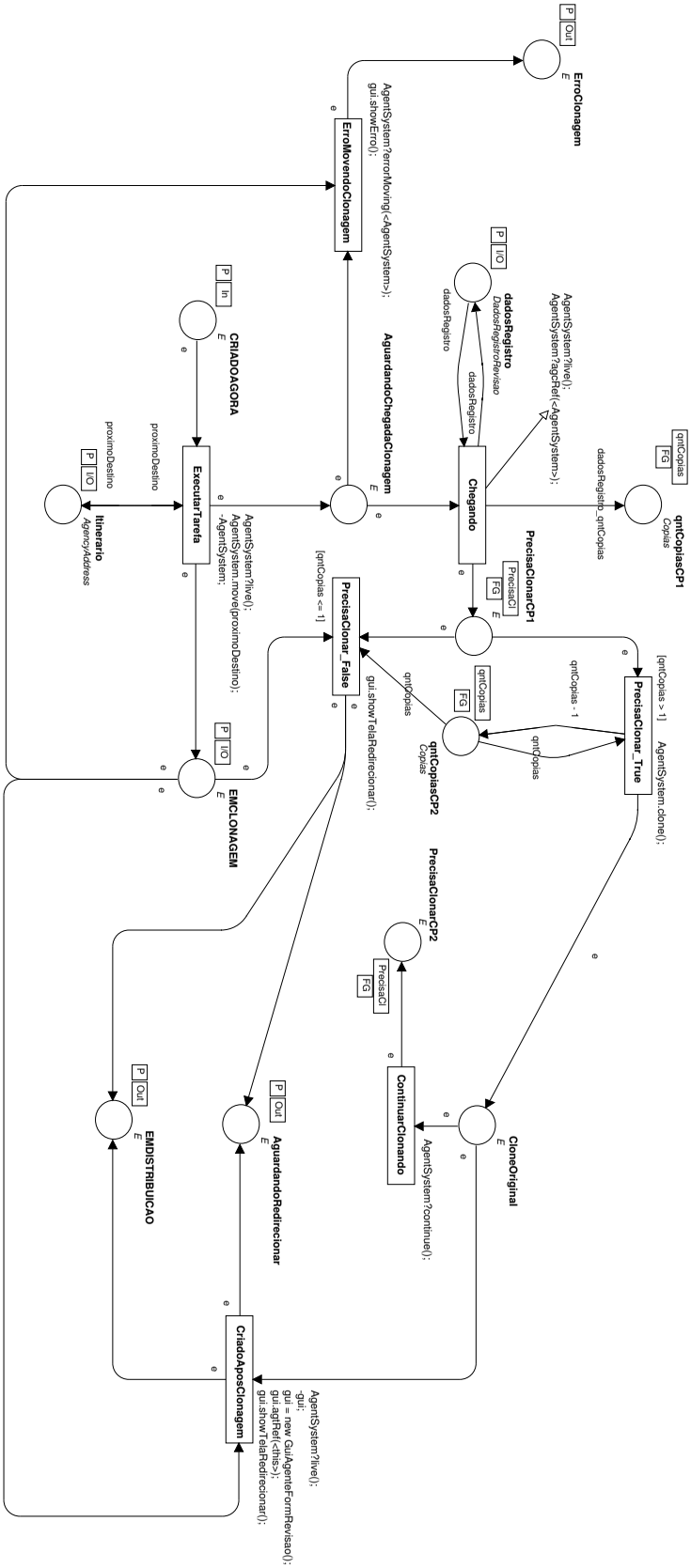

Figura A.19: Página *EmClonagem* da Rede de Petri da Classe **AgenteFormRevisao**

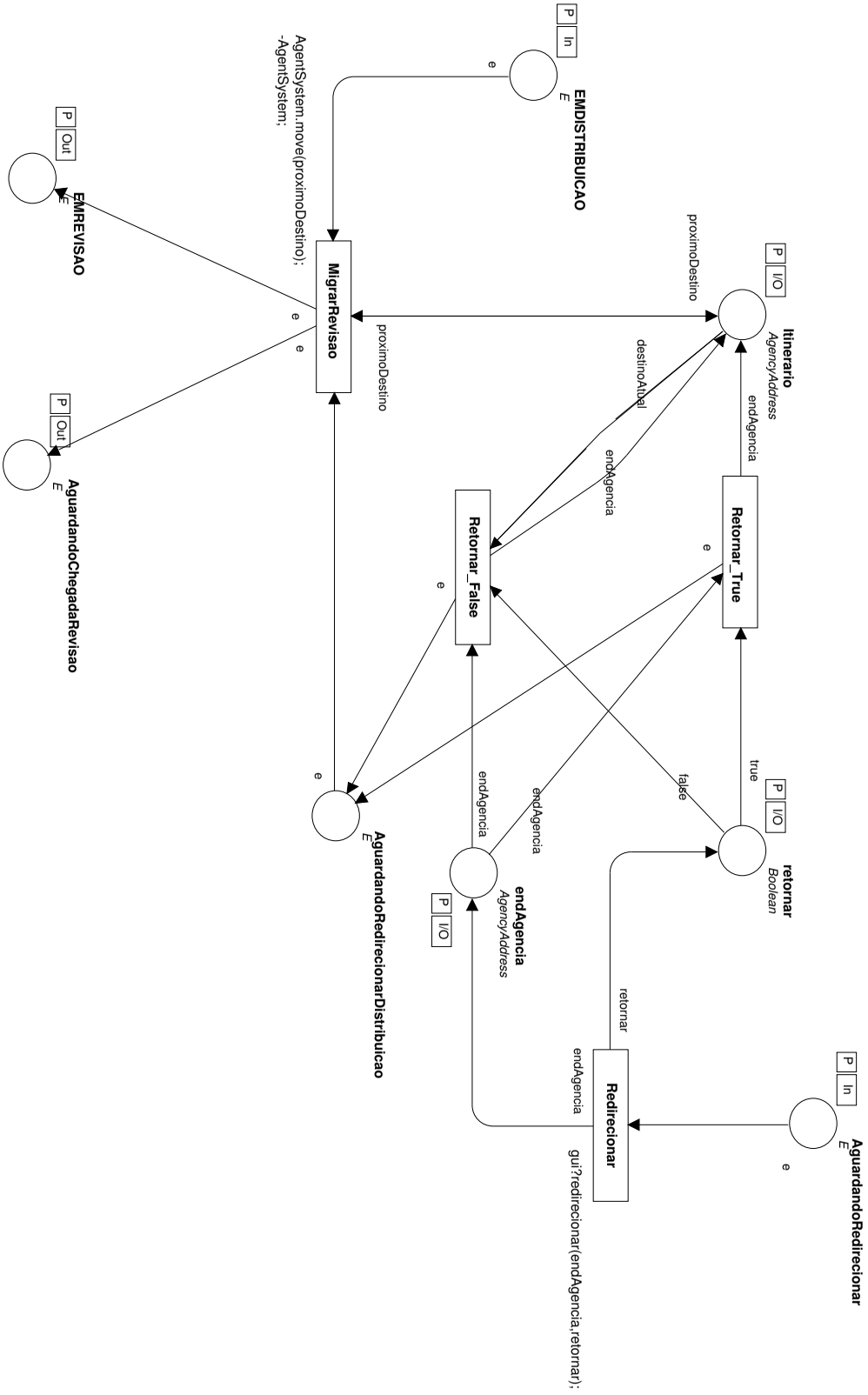

Figura A.20: Página *EmDistribuicao* da Rede de Petri da Classe **AgenteFormRevisao**

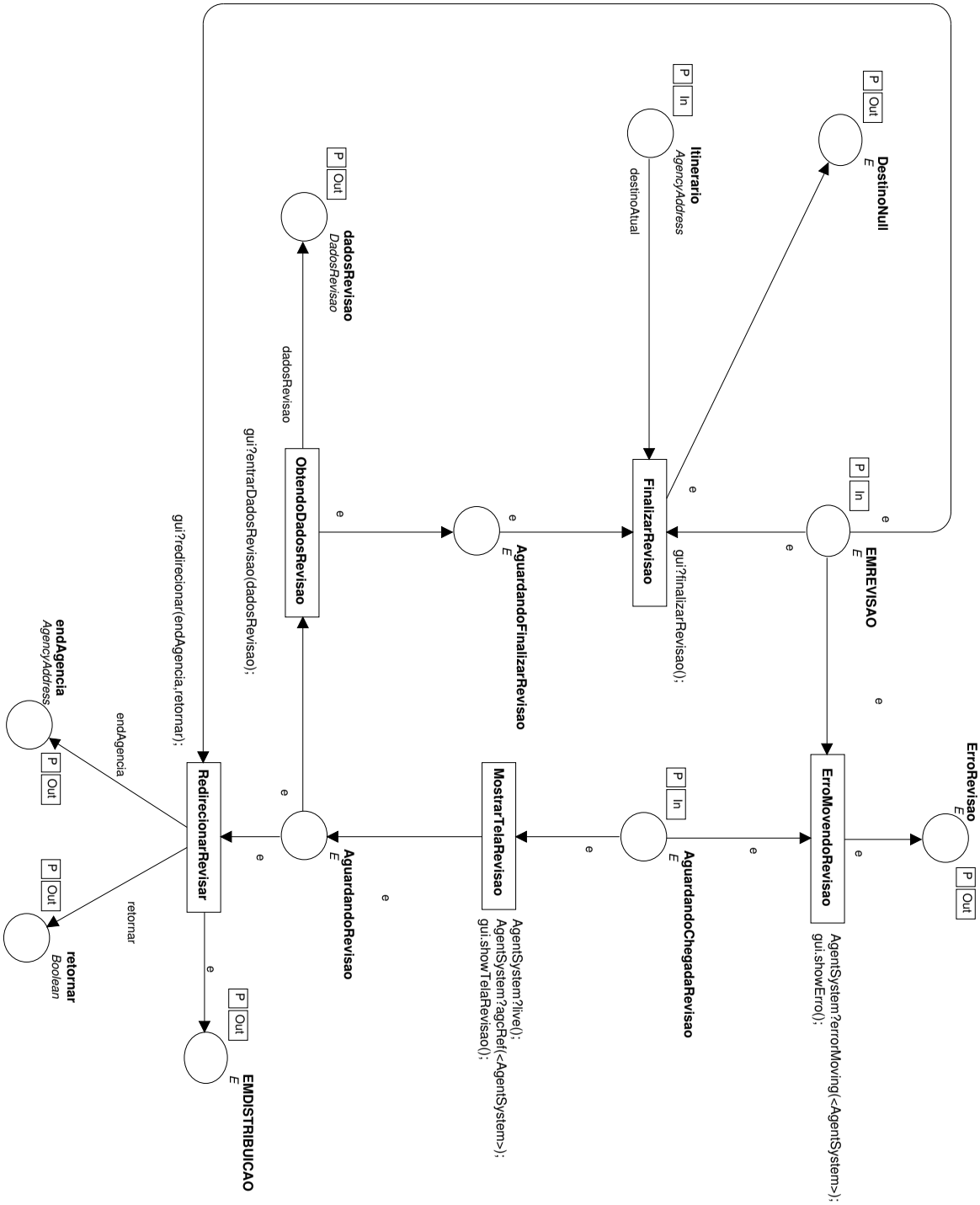

Figura A.21: Página *EmRevisao* da Rede de Petri da Classe **AgenteFormRevisao**

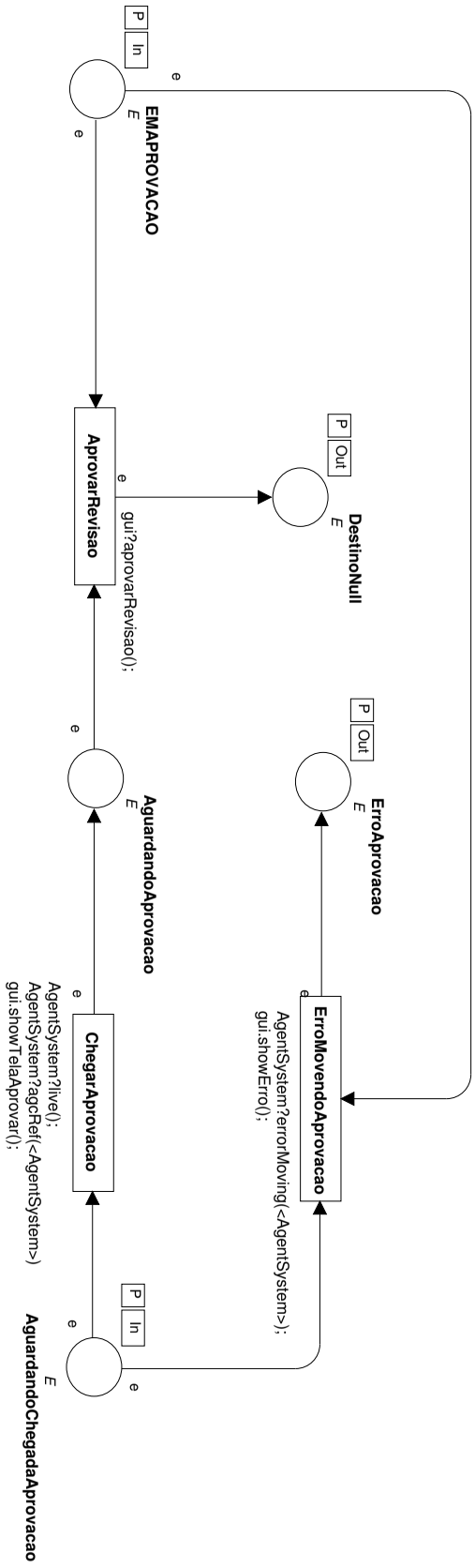

Figura A.22: Página *EmAprovacao* da Rede de Petri da Classe **AgenteFormRevisao**

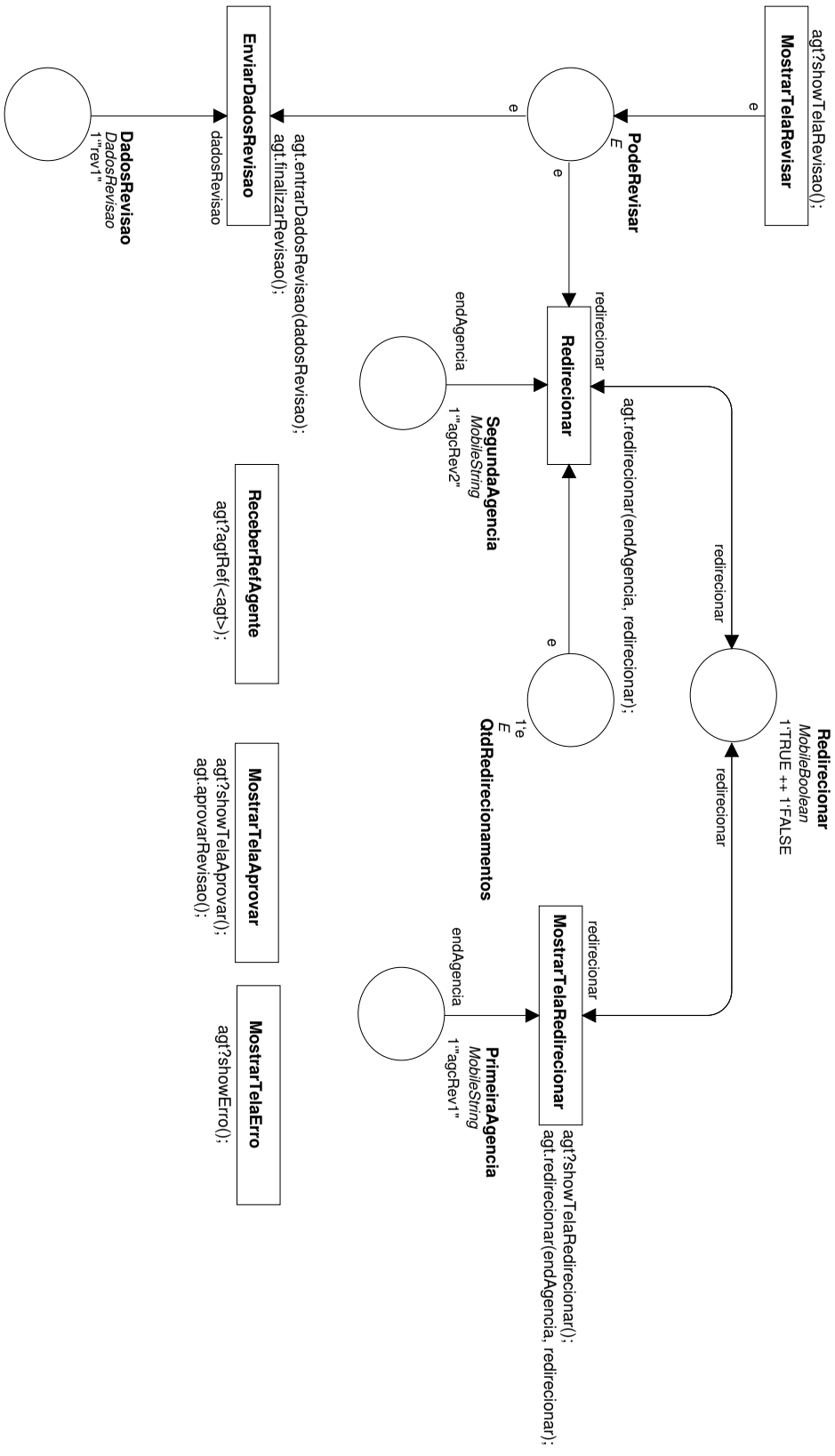

Figura A.23: Rede de Petri da Classe **GuiAgenteFormRevisao**

Apêndice B

Casos de Teste para o Sistema de Conferências

Este apêndice apresenta os casos de teste utilizados para testar o sistema de conferências durante a realização do estudo de caso. Cada caso de teste é descrito como uma sequência de entradas e saídas do sistema. Primeiro são apresentados os casos de teste para os padrões de projeto *Itinerary*, *MasterSlave* e *Branching*, e depois s casos de teste do objetivos de teste *Error Moving*.

- **Caso de Teste 1 do para o Padrão de Projeto *Itinerary***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao((’agcMemb’,1)); INPUT"
"conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
"conf:agentCoord.registroRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:agcCoord.createAgent((’dadosconf1’,(’agcMemb’,1),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init((’dadosconf1’,(’agcMemb’,1),agentCoord))
& agcCoord:agentFormOriginal.address(’agcCoord’)
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move(’agcMemb’); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar((’agcRev1’,true)); OUTPUT"
"agentFormOriginal:agcMemb.move(’agcRev1’); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar((’agcRev2’,false)); OUTPUT"

"agentFormOriginal:agcRev1.move('agcRev2'); INPUT"
"agcRev2:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev2.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.aprovarRevisao(); OUTPUT"
"agentFormOriginal:agcMemb.move('agcCoord'); INPUT (PASS)"
<goal state>

- **Caso de Teste 2 do para o Padrão de Projeto *Itinerary***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',1)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',1)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',1)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',1),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',1),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',true)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',true)); OUTPUT"
"agentFormOriginal:agcRev1.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')

& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.aprovarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.aprovarRevisao(); OUTPUT"
"agentFormOriginal:agcMemb.move('agcCoord'); INPUT (PASS)"
<goal state>

- **Caso de Teste 3 do para o Padrão de Projeto *Itinerary***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()
& agcMemb:agentForm0.live(); OUTPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',true)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev2',true)); OUTPUT"
"agentFormOriginal:agcRev1.move('agcRev2'); INPUT"

"agcRev2:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev2.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.aprovarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.aprovarRevisao(); OUTPUT"
"agentFormOriginal:agcMemb.move('agcCoord'); INPUT (PASS)"
<goal state>

- **Caso de Teste 1 do para o Padrão de Projeto *MasterSlave***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',1)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',1)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',1)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',1),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',1),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',false)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev2',false)); OUTPUT"
"agentFormOriginal:agcRev1.move('agcRev2'); INPUT"

"agcRev2:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev2.move('agcCoord'); INPUT"
"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado((('agcMemb',1),'rev_guirevisao-agentFormOriginal')); INPUT (PASS)"
<goal state>

- **Caso de Teste 2 do para o Padrão de Projeto *MasterSlave***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()
& agcMemb:agentForm0.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',false)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcCoord'); INPUT"

"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT"
"agentCoord:conf.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar(('agcRev1',false)); OUTPUT"
"agentForm0:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentForm0.live(); OUTPUT"
"agentForm0:guirevisao-agentForm0.showTelaRevisao(); INPUT"
"guirevisao-agentForm0:agentForm0.entrarDadosRevisao('rev_guirevisao-agentForm0')
& guirevisao-agentForm0:agentForm0.finalizarRevisao(); OUTPUT"
"agentForm0:agcRev1.move('agcCoord'); INPUT"
"agcCoord:agentForm0.live(); OUTPUT"
"agentForm0:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentForm0')); INPUT (PASS)"
<goal state>

- **Caso de Teste 3 do para o Padrão de Projeto *MasterSlave***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()
& agcMemb:agentForm0.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar(('agcRev1',false)); OUTPUT"
"agentForm0:agcMemb.move('agcRev1'); INPUT"

"agcRev1:agentForm0.live(); OUTPUT"
"agentForm0:guirevisao-agentForm0.showTelaRevisao(); INPUT"
"guirevisao-agentForm0:agentForm0.entrarDadosRevisao('rev_guirevisao-agentForm0')
& guirevisao-agentForm0:agentForm0.finalizarRevisao(); OUTPUT"
"agentForm0:agcRev1.move('agcCoord'); INPUT"
"agcCoord:agentForm0.live(); OUTPUT"
"agentForm0:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentForm0')); IN-PUT"
"agentCoord:conf.registrarResultado((('agcMemb',2),'rev_guirevisao-agentForm0')); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',false)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal') & guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcCoord'); INPUT"
"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT (PASS)"
<goal state>

- **Caso de Teste 1 do para o Padrão de Projeto *Branching***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()

& agcMemb:agentForm0.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar((’agcRev1’,false)); OUTPUT"
"agentFormOriginal:agcMemb.move(’agcRev1’); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar((’agcRev1’,false)); OUTPUT"
"agentForm0:agcMemb.move(’agcRev1’); INPUT"
"agcRev1:agentForm0.live(); OUTPUT"
"agentForm0:guirevisao-agentForm0.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao(’rev_guirevisao-agentFormOriginal’)
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move(’agcCoord’); INPUT"
"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado(((’agcMemb’,2),’rev_guirevisao-agentFormOriginal’)); INPUT"
"agentCoord:conf.registrarResultado(((’agcMemb’,2),’rev_guirevisao-agentFormOriginal’)); INPUT"
"guirevisao-agentForm0:agentForm0.entrarDadosRevisao(’rev_guirevisao-agentForm0’)
& guirevisao-agentForm0:agentForm0.finalizarRevisao(); OUTPUT"
"agentForm0:agcRev1.move(’agcCoord’); INPUT"
"agcCoord:agentForm0.live(); OUTPUT"
"agentForm0:agentCoord.registrarResultado(((’agcMemb’,2),’rev_guirevisao-agentForm0’)); INPUT (PASS)"
<goal state>

- **Caso de Teste 2 do para o Padrão de Projeto *Branching***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao((’agcMemb’,2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao((’agcMemb’,2)); INPUT"
"conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
"conf:agentCoord.registroRevisao((’agcMemb’,2)); OUTPUT"
"agentCoord:agcCoord.createAgent((’dadosconf1’,(’agcMemb’,2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init((’dadosconf1’,(’agcMemb’,2),agentCoord))

& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()
& agcMemb:agentForm0.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',false)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar(('agcRev1',false)); OUTPUT"
"agentForm0:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentForm0.live(); OUTPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRevisao(); INPUT"
"guirevisao-agentForm0:agentForm0.entrarDadosRevisao('rev_guirevisao-agentForm0')
& guirevisao-agentForm0:agentForm0.finalizarRevisao(); OUTPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev2',false)); OUTPUT"
"agentFormOriginal:agcRev1.move('agcRev2'); INPUT"
"agentForm0:agcRev1.move('agcCoord'); INPUT"
"agcRev2:agentFormOriginal.live(); OUTPUT"
"agcCoord:agentForm0.live(); OUTPUT"
"agentForm0:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentForm0')); INPUT"
"agentCoord:conf.registrarResultado((('agcMemb',2),'rev_guirevisao-agentForm0')); INPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev2.move('agcCoord'); INPUT"
"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT (PASS)"
<goal state>

- **Caso de Teste 1 do para o Objetivo de Teste *Error Moving***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao((’agcMemb’,1)); INPUT"
"conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
"conf:agentCoord.registroRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:agcCoord.createAgent((’dadosconf1’,(’agcMemb’,1),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init((’dadosconf1’,(’agcMemb’,1),agentCoord))
& agcCoord:agentFormOriginal.address(’agcCoord’)
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move(’agcMemb’); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar((’agcRev1’,true)); OUTPUT"
"agentFormOriginal:agcMemb.move(’agcRev1’); INPUT"
"agcMemb:agentFormOriginal.errorMoving(agcMemb); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showErro(); INPUT (PASS)"
<goal state>

- **Caso de Teste 2 do para o Objetivo de Teste *Error Moving***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao((’agcMemb’,1)); INPUT"
"conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
"conf:agentCoord.registroRevisao((’agcMemb’,1)); OUTPUT"
"agentCoord:agcCoord.createAgent((’dadosconf1’,(’agcMemb’,1),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init((’dadosconf1’,(’agcMemb’,1),agentCoord))
& agcCoord:agentFormOriginal.address(’agcCoord’)
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move(’agcMemb’); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar((’agcRev1’,false)); OUTPUT"
"agentFormOriginal:agcMemb.move(’agcRev1’); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"

"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcCoord'); INPUT"
"agcRev1:agentFormOriginal.errorMoving(agcRev1); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showErro(); INPUT (PASS)"
<goal state>

- **Caso de Teste 3 do para o Objetivo de Teste *Error Moving***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcCoord:agentFormOriginal.errorMoving(agcCoord); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showErro(); INPUT (PASS)"
<goal state>

- **Caso de Teste 4 do para o Objetivo de Teste *Error Moving***

<initial state>
"agentCoord:guicoord.show(); INPUT"
"guicoord:agentCoord.gerarFormRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:conf.getDadosConferencia()
& agentCoord:conf.criarRegistroRevisao(('agcMemb',2)); INPUT"
"conf:agentCoord.dadosConferencia('dadosconf1'); OUTPUT"
"conf:agentCoord.registroRevisao(('agcMemb',2)); OUTPUT"
"agentCoord:agcCoord.createAgent(('dadosconf1',('agcMemb',2),agentCoord)); INPUT"
"agcCoord:agentFormOriginal.init(('dadosconf1',('agcMemb',2),agentCoord))
& agcCoord:agentFormOriginal.address('agcCoord')
& agcCoord:agentFormOriginal.live(); OUTPUT"

"agentFormOriginal:agcCoord.move('agcMemb'); INPUT"
"agcMemb:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agcMemb.clone(); INPUT"
"agcMemb:agentFormOriginal.continue()
& agcMemb:agentForm0.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRedirecionar(); INPUT"
"agentForm0:guirevisao-agentForm0.showTelaRedirecionar(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.redirecionar(('agcRev1',false)); OUTPUT"
"agentFormOriginal:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:guirevisao-agentFormOriginal.showTelaRevisao(); INPUT"
"guirevisao-agentFormOriginal:agentFormOriginal.entrarDadosRevisao('rev_guirevisao-agentFormOriginal')
& guirevisao-agentFormOriginal:agentFormOriginal.finalizarRevisao(); OUTPUT"
"agentFormOriginal:agcRev1.move('agcCoord'); INPUT"
"agcCoord:agentFormOriginal.live(); OUTPUT"
"agentFormOriginal:agentCoord.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT"
"agentCoord:conf.registrarResultado((('agcMemb',2),'rev_guirevisao-agentFormOriginal')); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar(('agcRev1',false)); OUTPUT"
"agentForm0:agcMemb.move('agcRev1'); INPUT"
"agcRev1:agentForm0.live(); OUTPUT"
"agentForm0:guirevisao-agentForm0.showTelaRevisao(); INPUT"
"guirevisao-agentForm0:agentForm0.redirecionar(('agcRev2',false)); OUTPUT"
"agentForm0:agcRev1.move('agcRev2'); INPUT"
"agcRev1:agentForm0.errorMoving(agcRev1); OUTPUT"
"agentForm0:guirevisao-agentForm0.showErro(); INPUT (PASS)"
<goal state>

Apêndice C

Padrões de Teste para Sistemas Baseados em Agentes Móveis

Este apêndice traz um conjunto de padrões de teste identificados a partir de especificações de padrões de projeto.

Padrão Traveller

- **Nome:** Traveller
- **Contexto:** Implementações que utilizam o padrão Itinerary. Este padrão é usado no contexto em que podemos garantir as as agências estão on-line em qualquer instante de tempo.
- **Objetivo:** Submeter os agentes das aplicações a um cenário de agências válidas. Assim, verificar se o agente móvel está percorrendo todo o itinerário e executando a tarefa que lhe foi confiada.
- **Estrutura:**

  – Modelo de Teste: As figuras abaixo representam os diagramas de classes e de seqüencias independentes de plataforma.

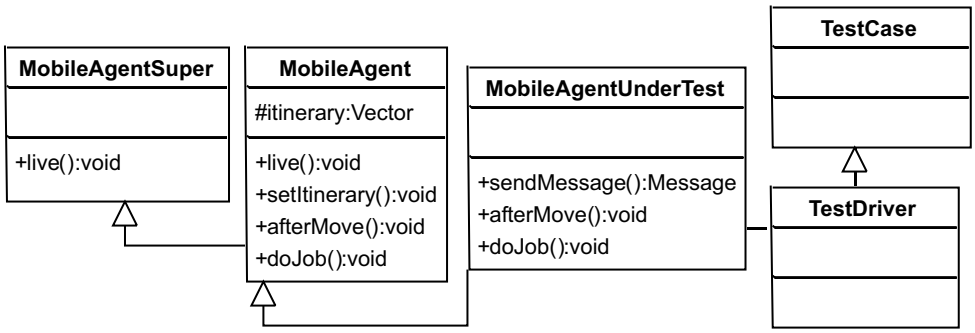

Figura C.1: Diagrama de classes do padrão Traveller independente de plataforma

  – Procedimentos:

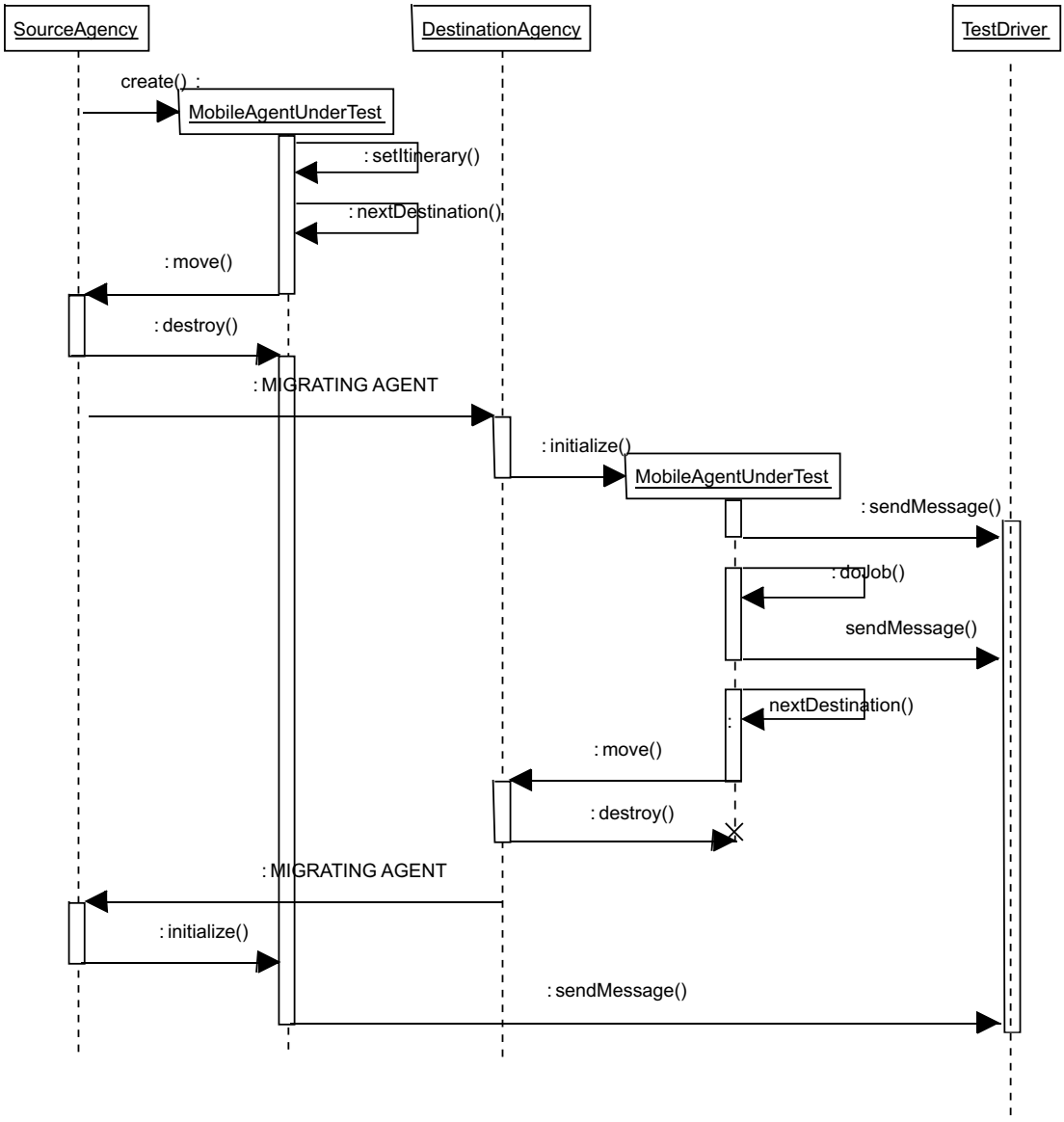

Figura C.2: Diagrama de seqüencia do padrão Traveller independente de plataforma (Basic Execution)

- Usando herança, construa um agente *wrapper* para o agente móvel, o agente "UnderTest";

- Sobrescreva os métodos afterMove e doJob, chamando os respectivos métodos das superclasses, e adicionando o envio de mensagens para o *test driver*;

- *Start* as agências e execute a aplicação configurando o itinerário do agente com as devidas agência válidas.

– Oráculo: Com as mensagens recebidas, o oráculo deve verificar:

- O *test driver* recebeu uma mensagem informando a chegada do agente em cada agência

de seu itinerário;

- O *test driver* recebeu uma mensagem informando a execução da tarefa em cada agência de seu itinerário;

- O *test driver* recebeu uma mensagem informando a chegada do agente na agência de origem.

- **Implementação:**

  – Plataforma Grasshopper: As figuras abaixo apresentam a solução na plataforma Grasshopper:

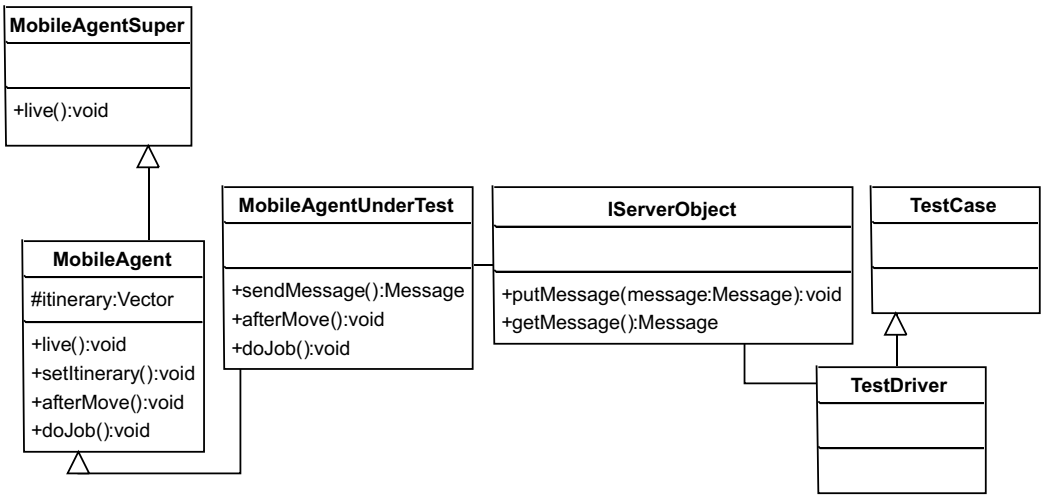

Figura C.3: Diagrama de classes do padrão Traveller dependente da plataforma Grasshopper

- Para a troca de mensagens entre o agente e o *test driver*, utilizou-se a solução proposta pela documentação da plataforma Grasshopper. A idéia principal é colocar um *IServerObject* entre eles e que é acessado via *proxy*. Este colaborador, recebe as mensagens dos agentes e a repassa para o *test driver*.

– Plataforma JADE: As figuras abaixo apresentam a solução na plataforma JADE:

- Para a troca de mensagens entre o agente e o *test driver*, utilizou-se a solução proposta pela documentação da plataforma JADE. A idéia principal é colocar um agente como classe interna do *test driver*, o *InternalAgent*. Este agente colabora, recebendo as mensagens dos agentes e as repassa para o *test driver*.

- **Forças:** Iniciar certos serviços de uma plataforma automaticamente é uma tarefa não trivial.

- **Modelo de falha:**

  – O agente pode não realizar sua tarefa;

  – O agente pode se perder em seu itinerário;

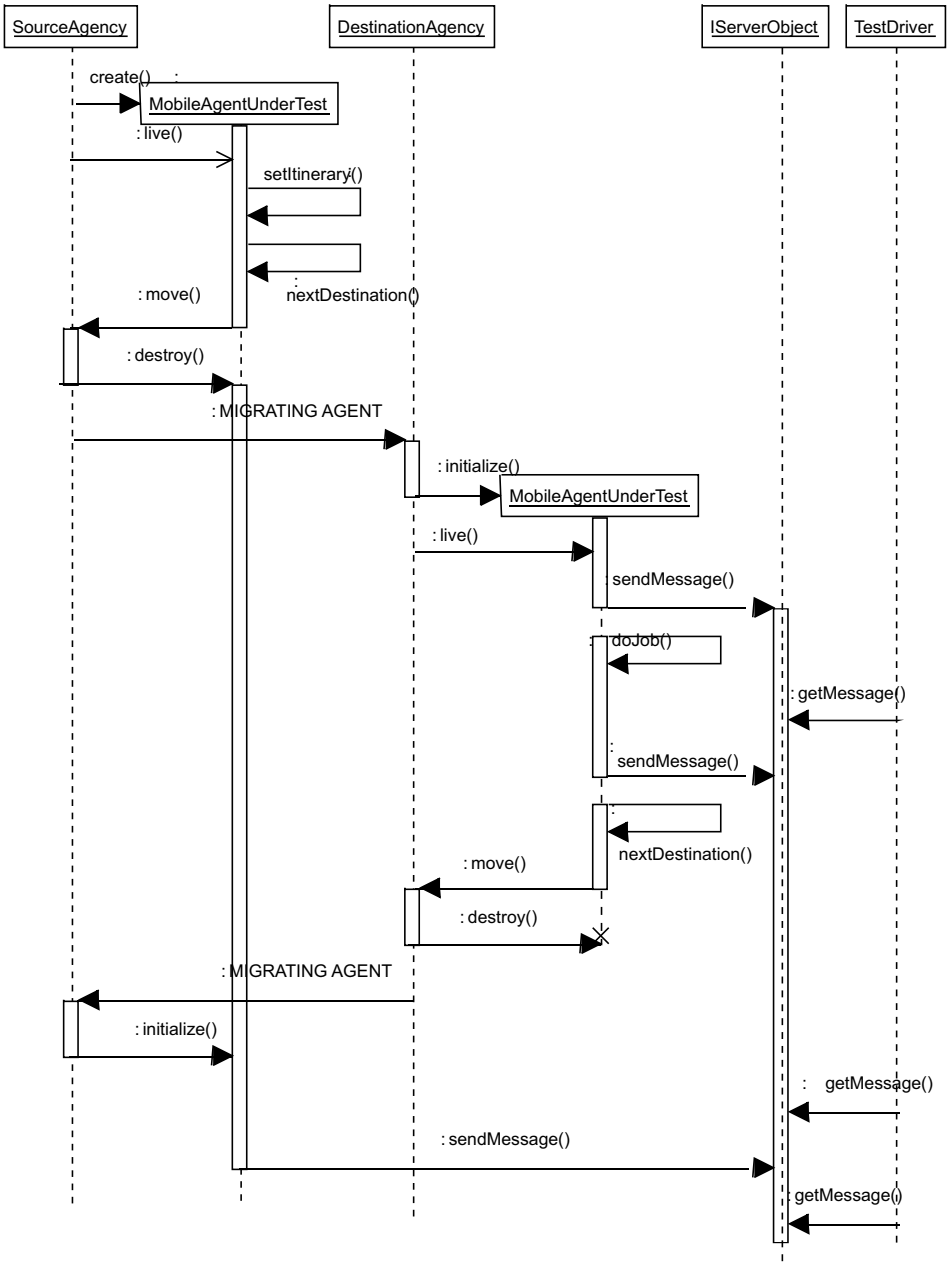

Figura C.4: Diagrama de seqüencia do padrão Traveller dependente da plataforma Grasshopper

– Pode ocorrer uma parada da aplicação.

Padrão Repeated

- **Nome:** Repeated
- **Contexto:** Implementações que utilizam o padrão Itinerary. Este padrão é usado no contexto em que podemos garantir as as agências estão on-line em qualquer instante de tempo. Nele o agente deverá possui agências repetidas(seqüenciais ou não) em seu itinerário para a realização

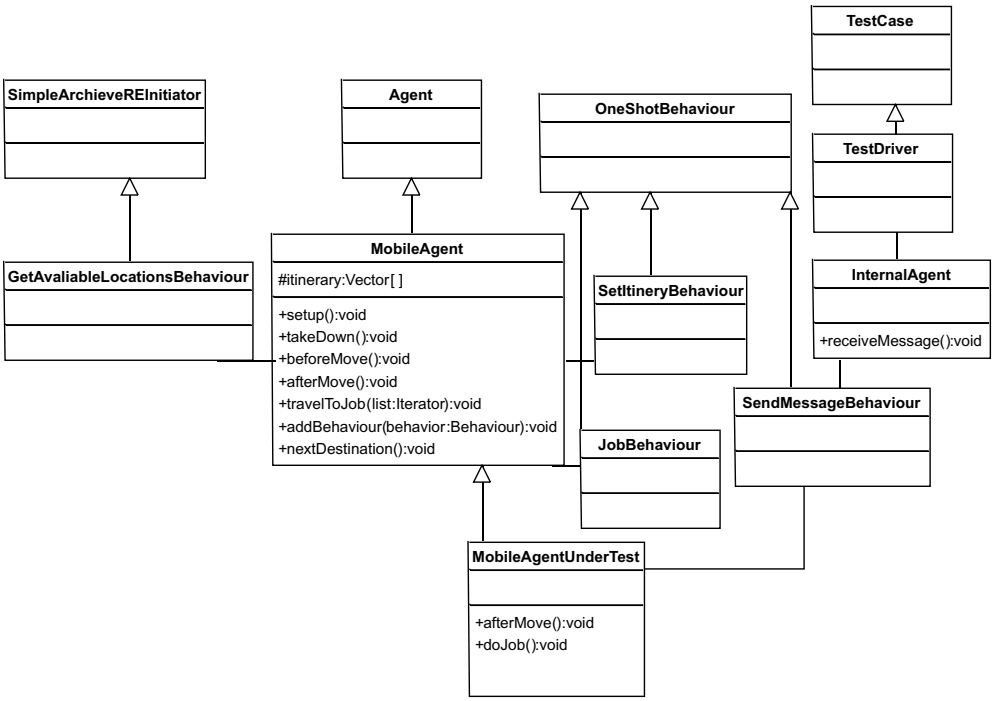

Figura C.5: Diagrama de classes do padrão Traveller dependente da plataforma JADE

de sua tarefa.

- **Objetivo:** Submeter os agentes das aplicações a um cenário de agências ativas. Assim, verificar o comportamento do agente móvel ao submetê-lo a realização de uma tarefa em uma agência já visitada.

- **Estrutura:**

  – Modelo de Teste: As figuras abaixo representam os diagramas de classes e de seqüencias independentes de plataforma.

  – Procedimentos:

  - Usando herança, construa um agente *wrapper* para o agente móvel, o agente "Under-Test";

  - Sobrescreva os métodos afterMove e doJob, chamando os respectivos métodos das superclasses, e adicionando o envio de mensagens para o *test driver*;

  - Start as agências e execute a aplicação configurando o itinerário do agente com agências repetidas seqüencialmente e não seqüencialmente.

  – Oráculo: Com as mensagens recebidas, o oráculo deve verificar:

  - O *test driver* recebeu uma mensagem informando a execução da tarefa em cada agência de seu itinerário, sem repetições de execução;

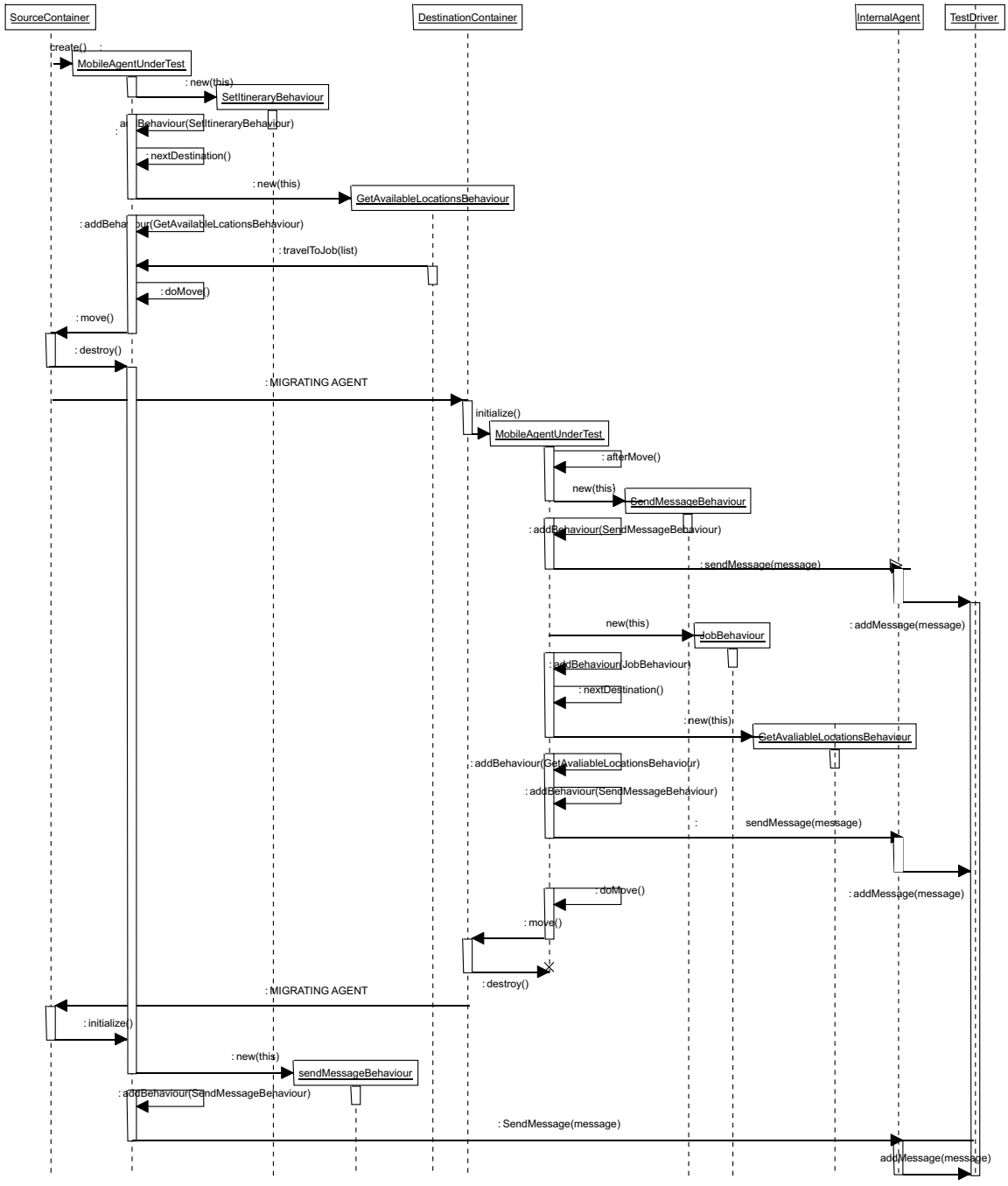

Figura C.6: Diagrama de seqüencia do padrão Traveller dependente da plataforma JADE

- O *test driver* recebeu uma mensagem informando a chegada do agente na agência de origem;

- **Implementação:**

– Plataforma Grasshopper: As figuras abaixo apresentam a solução na plataforma Grasshopper:

- Para a troca de mensagens entre o agente e o driver de teste, utilizou-se a solução

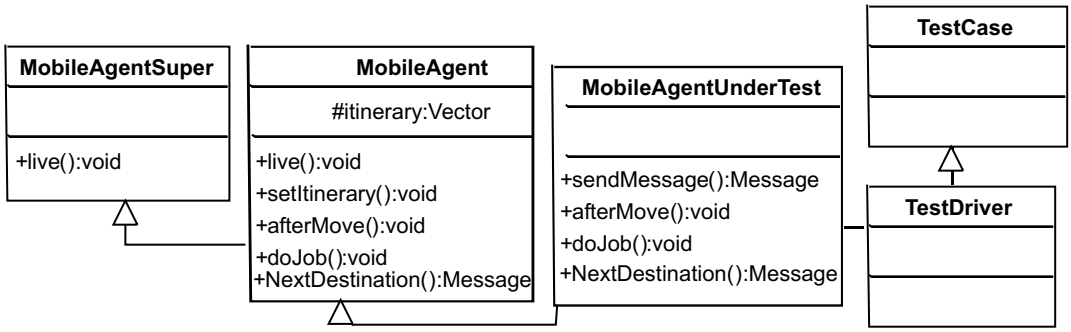

Figura C.7: Diagrama de classes do padrão Repeated independente de plataforma

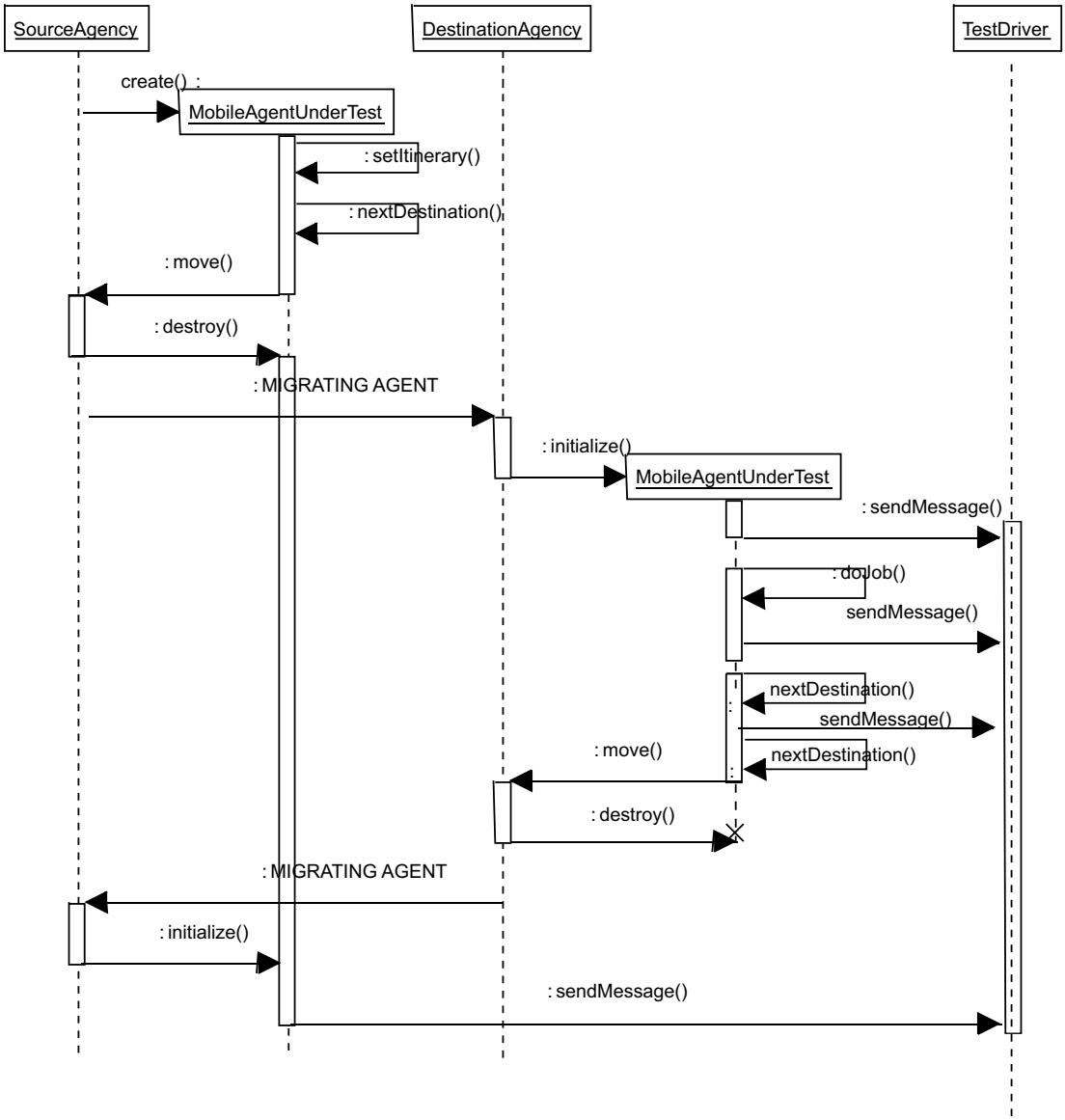

Figura C.8: Diagrama de seqüencia do padrão Repeated independente de plataforma

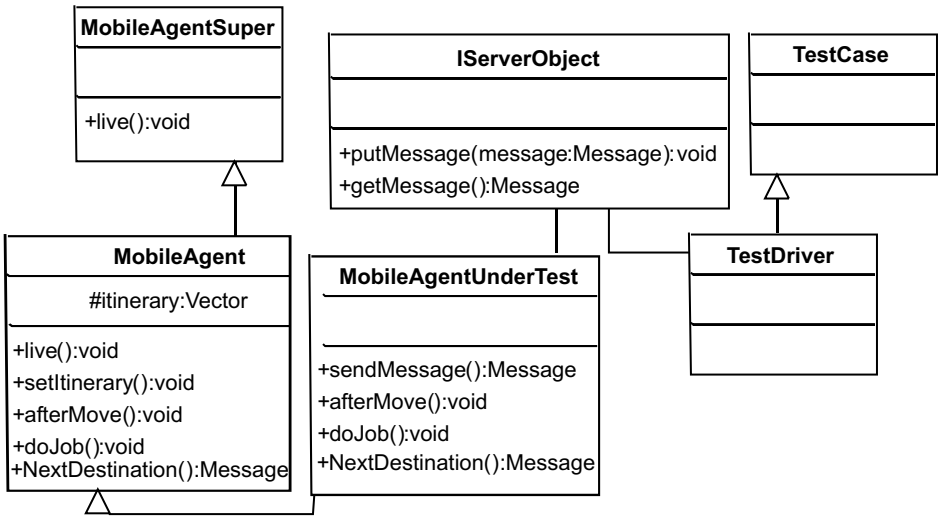

Figura C.9: Diagrama de classes do padrão Repeated dependente da plataforma Grasshopper

proposta pela documentação da plataforma Grasshopper. A idéia principal é colocar um *IServerObject* entre eles e que é acessado via *proxy*. Este colaborador, recebe as mensagens dos agentes e a repassa para o *test driver*.

– Plataforma JADE: As figuras abaixo apresentam a solução na plataforma JADE:

- Para a troca de mensagens entre o agente e o *test driver*, utilizou-se a solução proposta pela documentação da plataforma JADE. A idéia principal é colocar um agente como classe interna do *test driver*, o *InternalAgent*. Este agente colabora, recebendo as mensagens dos agentes e as repassando para o *test driver*.

- **Forças:** Iniciar certos serviços de uma plataforma automaticamente é uma tarefa não trivial.

- **Modelo de falha:**

  – O agente pode se perder em seu itinerário ao tentar migrar para a agência atual;

  – Pode ocorrer uma parada da aplicação.

Padrão Black Hole

- **Nome:** Black Hole

- **Contexto:** Pode-se utilizar este padrão em aplicações que implementam os padrões de projeto para Agentes Móveis Itinerary e Master-Slave. Este padrão é bastante útil no contexto em que as agências são instáveis, isto é, onde não se pode garantir que as agências estejam on-lines todo o tempo.

- **Objetivo:** Testar uma implementação visando verificar se ela lida de forma correta com agências que não estão disponíveis. Deseja checar se agentes não se perdem quando existem agências indisponíveis em seus itinerários.

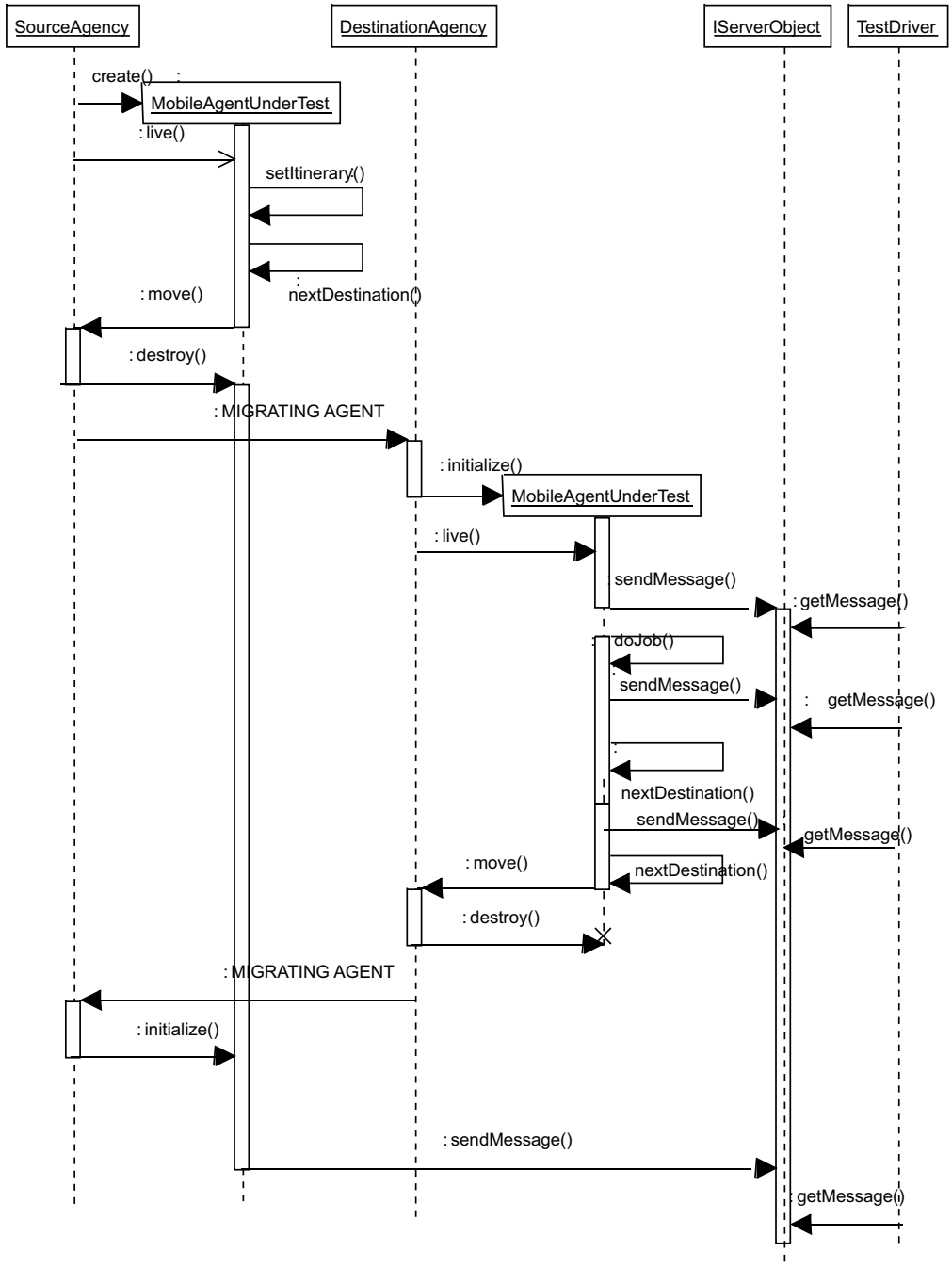

Figura C.10: Diagrama de seqüencia do padrão Repeated dependente da plataforma Grasshopper

- **Estrutura**

  – Modelo de Teste: Os modelos apresentados nas figuras C.13 e C.14 representam uma solução independente de plataforma. Onde temos, um diagrama de classe para o padrão proposto e também um diagrama de seqüencia mostrando uma execução básica do teste.

  – Procedimento:

  - Usando herança, construa um agente *wrapper* para o agente itinerário, chamado agente

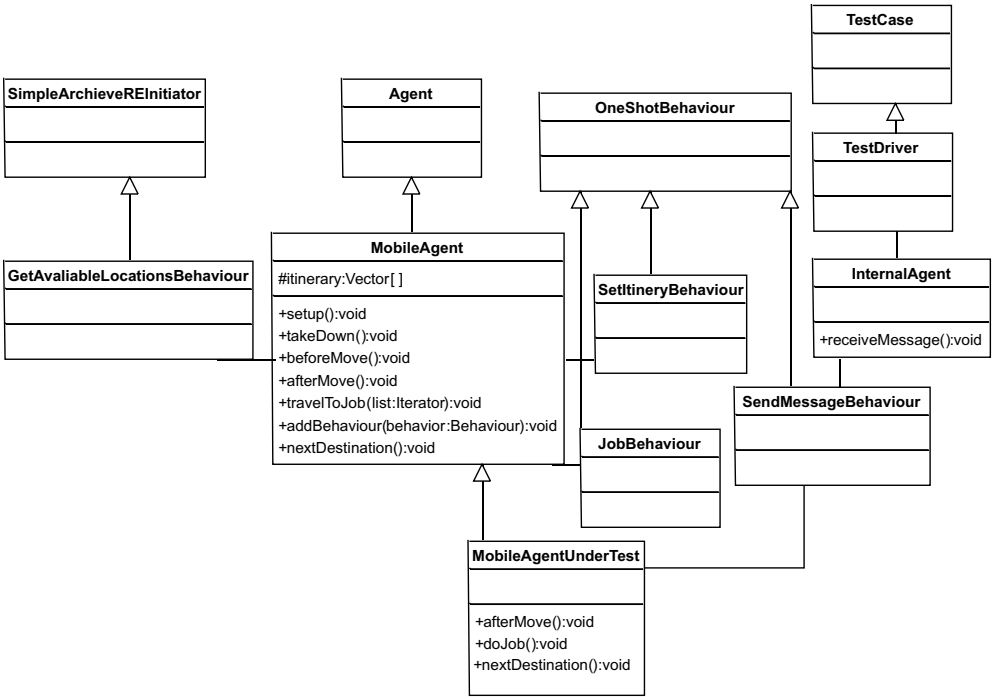

Figura C.11: Diagrama de classes do padrão Repeated dependente da plataforma JADE

"UnderTest";

- Sobrescreva os métodos *nextDestination* e *doJob*, chamando os respectivos métodos das superclasses, e adicionando o envio de mensagens para o *driver* de teste;

- Coloque o *test driver* para iniciar um determinado número de agências, onde uma delas seja inválida;

- Inicie a aplicação configurando o itinerário do agente com estas agências.

– Oráculo: O oráculo deve verificar os seguintes resultados:

- O *test drive* recebeu uma mensagem informando da chegada do agente em cada agência de seu itinerário e não na agência inválida;

- O *test driver* recebeu uma mensagem informando a execução da tarefa em cada agência de seu itinerário e não na agência inválida;

- O *test drive* recebeu uma mensagem informando a chegada do agente na agência de origem;

- **Implementação:**

– Plataforma Grasshopper: Os modelos apresentados nas figuras C.15 e C.16 representam uma solução dependente de plataforma (plataforma Grasshopper). Neles, podemos ver como aplicar este padrão em sistemas que são implementados utilizando a plataforma Grasshopper.

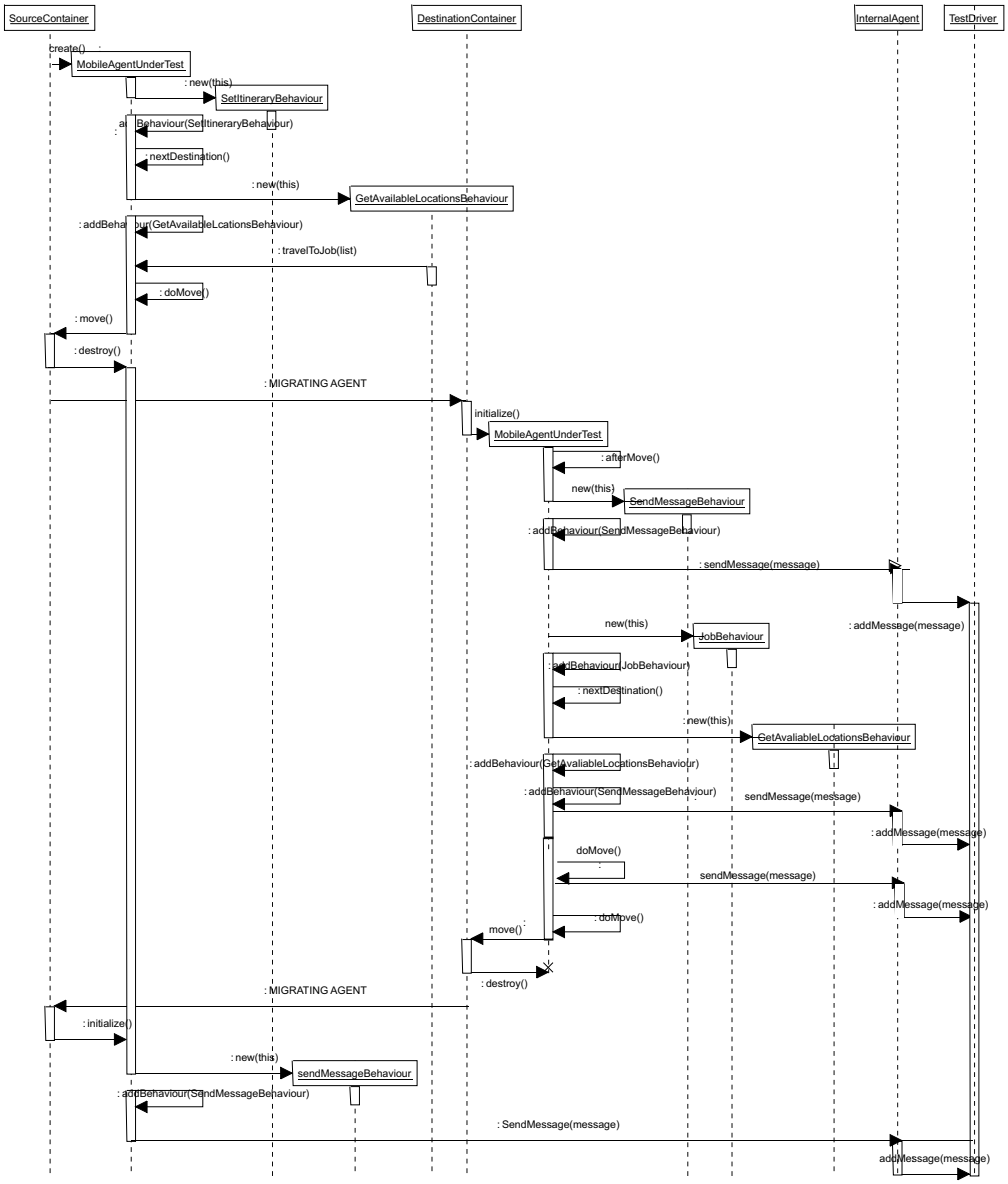

Figura C.12: Diagrama de seqüencia do padrão Repeated dependente da plataforma JADE

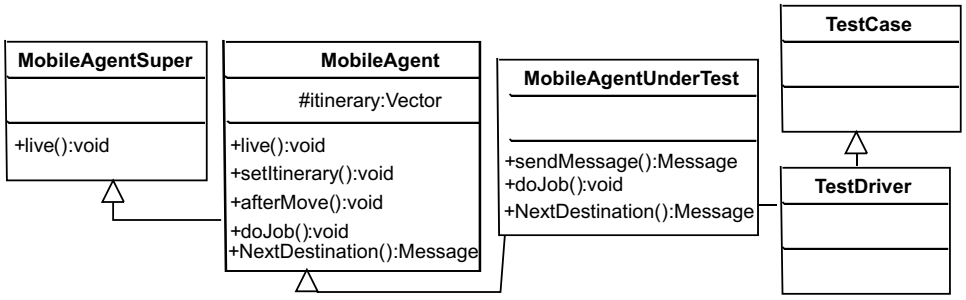

Figura C.13: Diagrama de classes independente de plataforma do padrão Black Hole

- Para a troca de mensagens entre o agente e o driver de teste, utilizou-se a solução

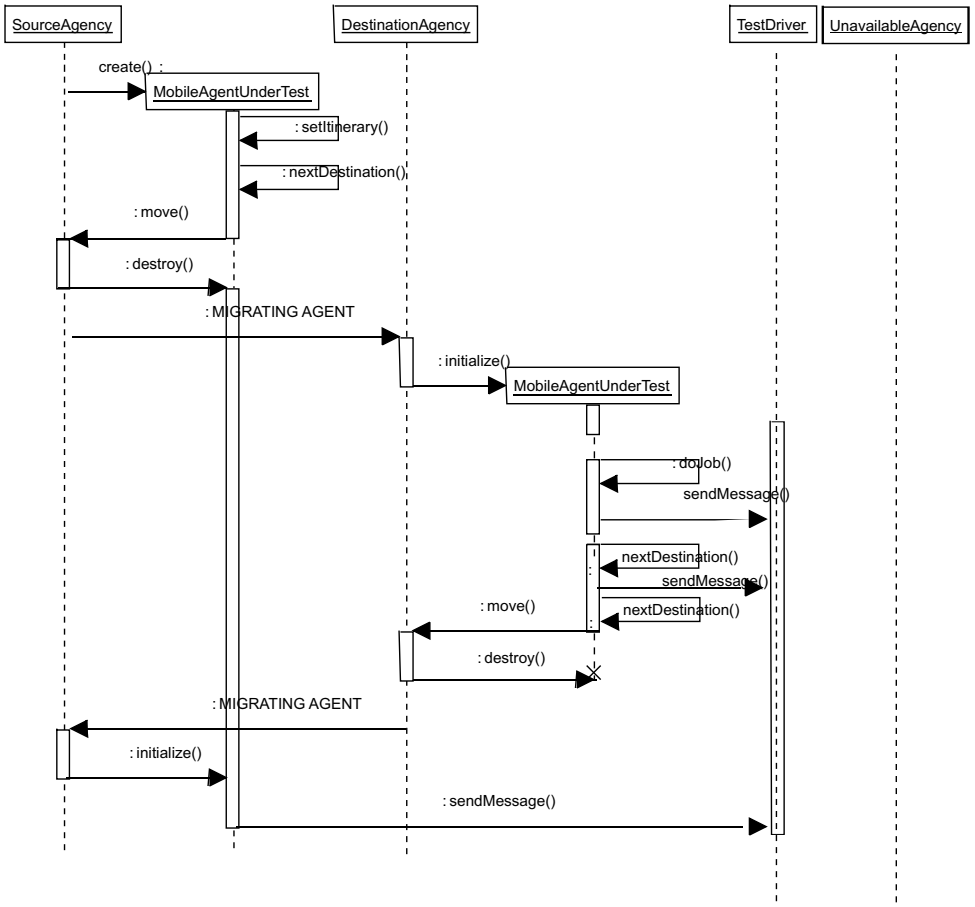

Figura C.14: Diagrama de seqüencia independente de plataforma do padrão Black Hole

proposta pela documentação da plataforma Grasshopper. A idéia principal é colocar um *IServerObject* entre eles e que é acessado via *proxy*. Este colaborador, recebe as mensagens dos agentes e a repassa para o *test driver*.

– Plataforma JADE: Os modelos apresentados nas figuras C.17 e C.18 representam uma solução dependente de plataforma (plataforma JADE). Neles, podemos ver como aplicar este padrão em sistemas que são implementados utilizando a plataforma JADE.

- Para a troca de mensagens entre o agente e o *test driver*, utilizou-se a solução proposta pela documentação da plataforma JADE. A idéia principal é colocar um agente como classe interna do *test driver*, o *InternalAgent*. Este agente colabora, recebendo as mensagens dos agentes e as repassando para o *test driver*.

- **Forças:** (1) Iniciar uma plataforma indisponível de forma automática não é uma tarefa trivial. (2) Geralmente, plataformas lançam exceções quando um agente tenta migrar para uma agência inválida, as quais não são, comumente, tratados no código.

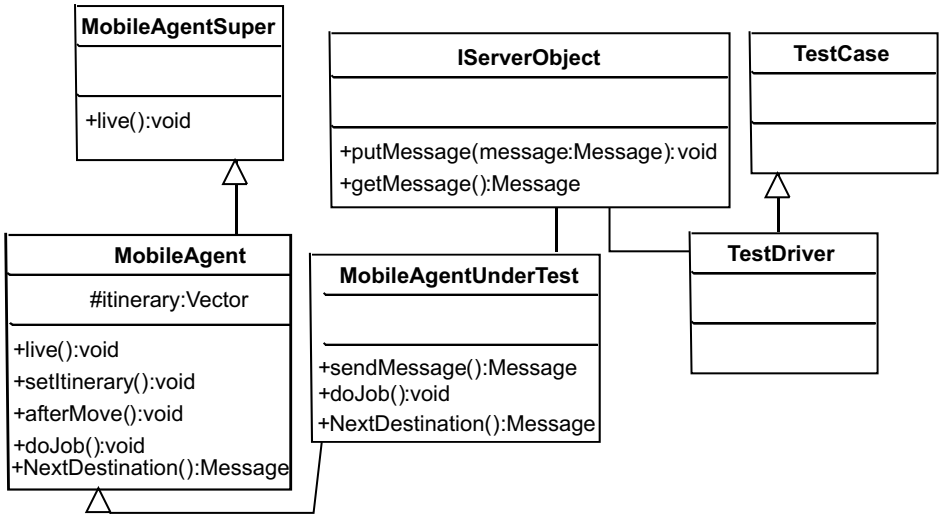

Figura C.15: Diagrama de classes do padrão Black Hole dependente da plataforma Grasshopper

- **Modelo de falha:**
  - Exceções não tratadas devido a tentativa de migrar para uma agência inválida;
  - Não tratamento das demais agência válidas após tentar migrar para agência inválida;
  - Perda do agente itinerário após a tentativa de migrar para uma agência inválida.

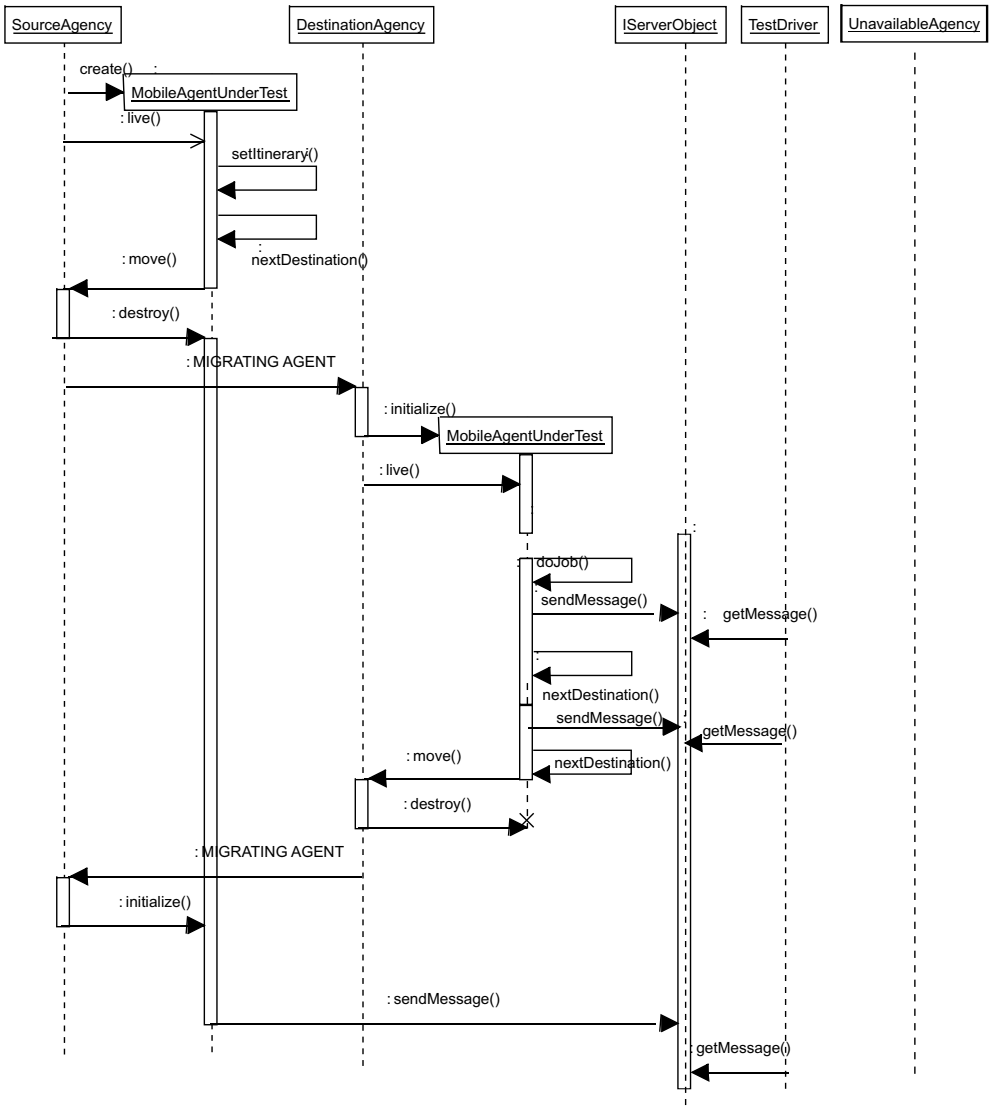

Figura C.16: Diagrama de seqüencia do padrão Black Hole dependente da plataforma Grasshopper

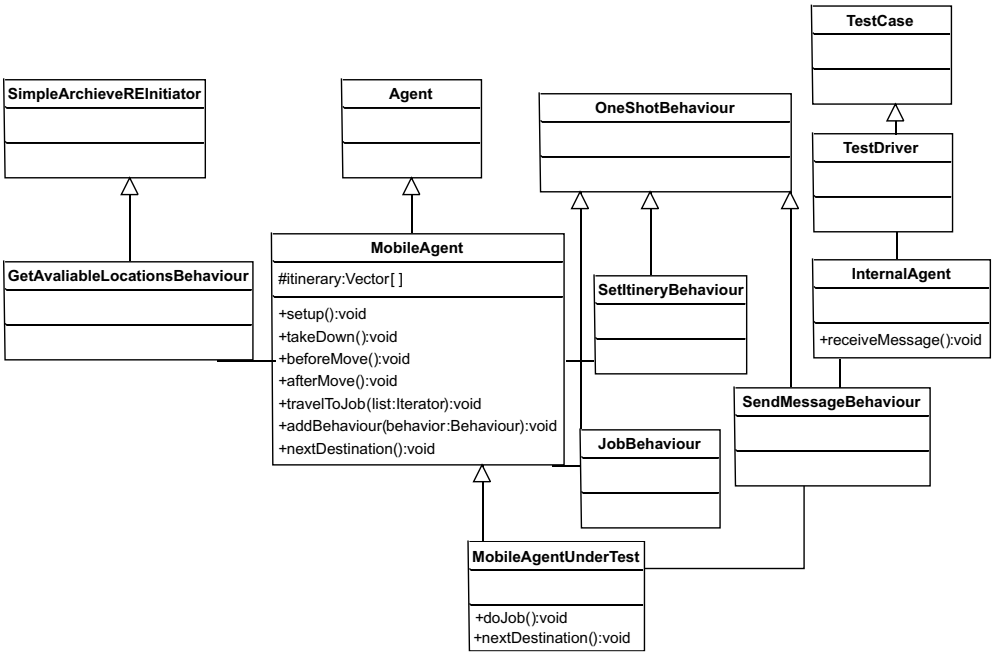

Figura C.17: Diagrama de classes do padrão Black Hole dependente da plataforma JADE

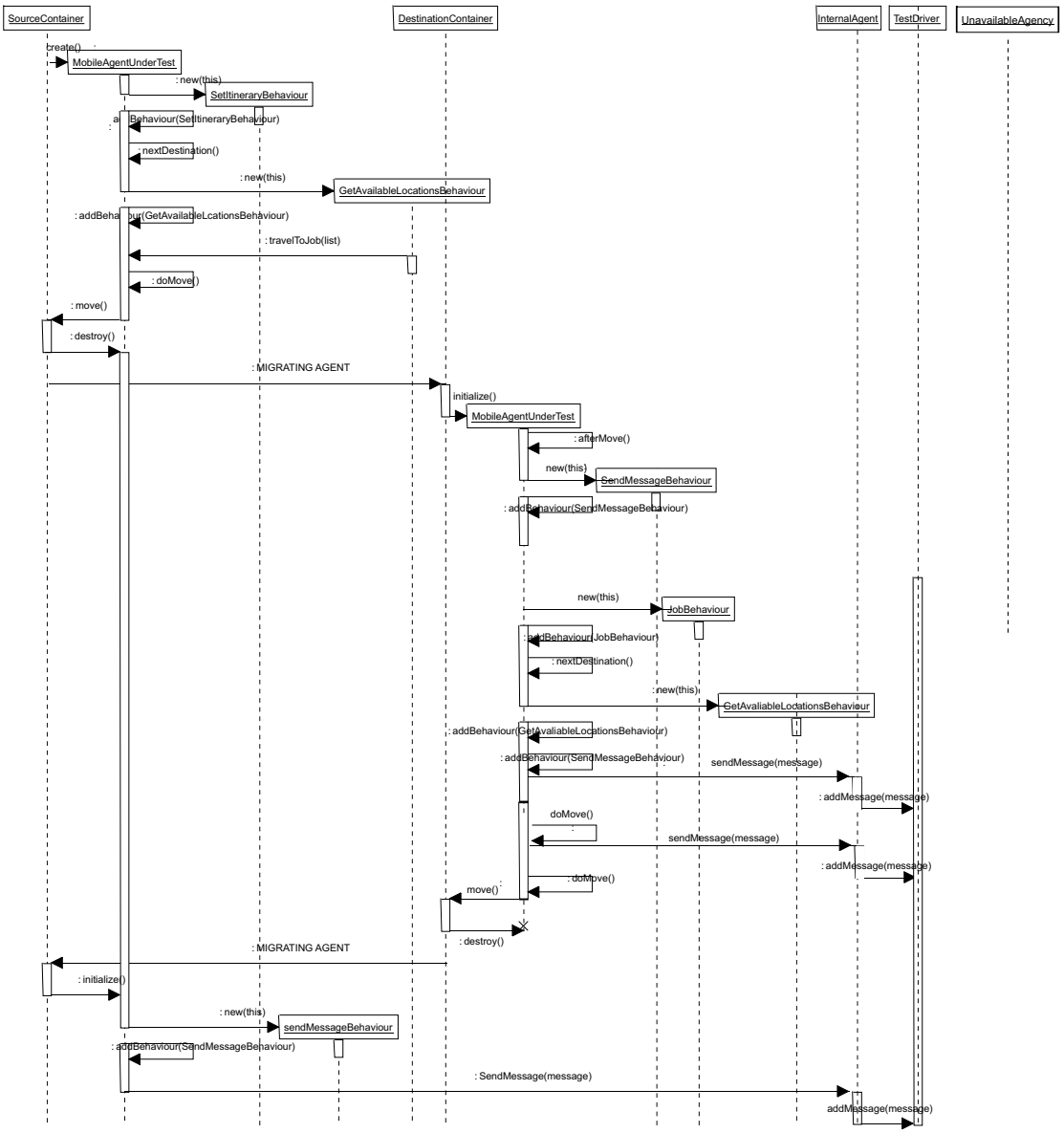

Figura C.18: Diagrama de seqüencia do padrão Black Hole dependente da plataforma JADE