Geração Automática de Objetivos e Casos de Teste a partir de Redes de Petri Orientadas a Objetos*

André L. L. Figueiredo and Patrícia D. L. Machado and Emanuela G. Cartaxo and Jorge C. A. Figueiredo and Paulo E. S. Barbosa

[1]GMF/DSC - Universidade Federal de Campina Grande (UFCG), Brasil

{andrel, patricia, emanuela, abrantes, paulo}@dsc.ufcg.edu.br

Resumo. *Este artigo apresenta um método para a seleção e geração automática de objetivos de teste e seus respectivos casos de teste a partir de modelos abstratos de sistemas reativos. É dado enfoque a modelos de Redes de Petri Orientadas a Objetos (RPOO). Configurações e seqüências de ativação das redes são adotadas como objetivos de teste. Estes são convertidos para os Sistemas de Transição Rotulados (LTSs) correspondentes que podem então ser usados como entrada para ferramentas de geração de casos de teste.*

1. Introdução

Teste funcional é um tipo de teste de software que objetiva observar se um dado sistema contempla de forma correta as funcionalidades descritas para ele [Jorgensen 2002]. Com base na sua respectiva especificação, um sistema é exercitado e suas funcionalidades são observadas através de seu comportamento, onde os possíveis defeitos são revelados. Dentre os diversos tipos de teste, um tipo de teste funcional se destaca - teste formal. Este tipo de teste permite avaliar, de forma automática, se um determinado software está de acordo com sua especificação formal [Tretmans 1999, Gaudel 1995]. Neste sentido, é possível observar de forma automática se um software realiza corretamente as funcionalidades contidas em sua especificação.

Uma das técnicas de teste formal mais aplicadas para a geração dos casos de teste é a baseada em verificação de modelos [Clarke et al. 1999]. 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 [Fernandez et al. 1996].

As propriedades utilizadas para a geração dos casos de teste são chamadas de objetivos de teste (*test purpose*). Estes 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.

*Este trabalho recebeu apoio financeiro do CNPq/FAPESQ, Projeto 060/03.

Pesquisas em seleção e geração de casos de teste baseada em objetivos de teste já produziram resultados importantes, incluindo modelos formais e ferramentas. Como exemplo, podemos citar a ferramenta TGV [Jard and Jéron 2004] desenvolvida pelo projeto AGEDIS [Hartman and Nagin 2004] e a ferramenta LTS-BT [Cartaxo et al. 2008]. No entanto, as soluções atuais demandam que o testador defina os objetivos, usualmente, de forma *ad-hoc* e escritos diretamente em uma notação de baixo nível tal como sistemas de transição rotulados (LTSs) que são amplamente adotados como modelos para a geração de casos de teste. Isto pode limitar a aplicabilidade de objetivos de teste, bem como comprometer a qualidade e confiabilidade de processos de teste. Objetivos de teste são essencialmente abstrações sobre comportamentos e propriedades dos modelos.

Este artigo apresenta um método para a seleção e geração automática de objetivos de teste em modelos LTSs a partir de modelos formais abstratos de sistemas reativos. É dado enfoque a modelos de Redes de Petri Orientadas a Objetos (RPOO)[Guerrero 2002]. Configurações e seqüências de ativação das redes, possivelmente selecionadas durante o processo de simulação, são adotadas como objetivos de teste, garantindo que comportamentos simulados poderão ser testados na implementação final. Estes são convertidos automaticamente para os LTSs correspondentes que podem então ser usados como entrada para ferramentas de geração de casos de teste. O método utiliza ferramentas para verificação e simulação de modelos a fim de validar e gerar os modelos do sistema e dos objetivos de teste como LTSs: o modelo gerado para verificação é o modelo a partir do qual casos de teste serão gerados. Um estudo de caso no domínio de aplicações de agentes móveis é apresentado a fim de ilustrar os passos de aplicação do método. Os objetivos de teste gerados são consistentes e completos com relação ao modelo do sistema. Finalmente, o conjunto de casos de teste é consistente.

O artigo está estruturado da seguinte forma. A Seção 2 apresenta fundamentação teórica sobre teste formal, objetivos de teste e RPOO, incluindo ferramentas. A Seção 3 apresenta o método proposto. A Seção 4 ilustra a aplicação do método através de um estudo de caso. Por fim, a Seção 5 discute trabalhos relacionados e a Seção 6 apresenta comentários finais sobre este trabalho e apontadores para trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta conceitos fundamentais e ferramentas utilizadas neste trabalho com enfoque em teste formal e redes de Petri orientadas a objetos.

2.1. Teste Formal e Objetivos de Teste

Tretmans [Tretmans 1999] propôs um framework 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. Nesta seção, conceitos básicos em teste formal, com base neste framework serão brevemente descritos.

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, decidir sobre sua correção. Neste sentido, são gerados os seguintes artefatos: (i) casos de teste, descrevem um comportamento esperado do sistema; (ii) dados de teste,

as entradas do sistema para a execução dos casos de teste; (iii) 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, casos de teste gerados são implementados e executados sobre o sistema sendo testado. Ao final, os resultados da execução dos testes são analisados. O foco deste trabalho é a geração de casos de teste com base em objetivos de teste.

Um conceito bastante utilizado na comunidade de teste formal é a relação de conformidade chamada ioco [Tretmans 1999], 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-determinísticos.

Para conjuntos de casos de teste denominados de *completos* os casos de teste são executados com sucesso se e somente se a implementação está em conformidade com a especificação. Contudo, estes conjuntos são muito difíceis de serem construídos na prática. Portanto, busca-se a construção de conjuntos de teste chamados *consistentes* os quais só rejeitam uma implementação se esta de fato não está em conformidade com a especificação. No entanto, implementações que não estão em conformidade também podem ser aceitas. Em outras palavras, podem detectar apenas não-conformidade.

Em teste formal, 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. 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 modelos utilizados pelas ferramentas de geração os casos de teste são, em geral, LTS's.

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.

TGV (*Test Generation with Verification technology*) [Jard and Jéron 2004] é uma ferramenta de teste funcional que gera automaticamente casos de teste de conformidade para sistemas reativos. Além de se basear em fundamentação teórica sólida, em particular

a teoria ioco apresentada em [Tretmans 1999], a ferramenta já foi aplicada com sucesso na indústria [Hartman and Nagin 2004]. A geração de casos de teste é baseada em técnicas de verificação de modelos tais como produto síncrono e verificação *on-the-fly*.

2.2. Redes de Petri Orientadas a Objetos

RPOO (Redes de Petri Orientadas a Objetos) [Guerrero 2002] é uma linguagem de modelagem e especificação de sistemas distribuídos e concorrentes baseada em redes de Petri e orientação a objetos. RPOO pode ser vista como uma extensão OO para redes de Petri de alto nível ou como um meio de prover semântica formal de concorrência a modelos OO. A idéia básica de RPOO é permitir a modelagem em dois níveis diferentes de abstração: um nível de interação entre objetos e um nível de comportamento interno dos objetos.

Um modelo RPOO é formado por um conjunto de classes e suas redes de Petri. As classes e seus relacionamentos com outras classes são descritas nos moldes convencionais da maioria dos modelos OO. Para cada classe, há exatamente uma rede de Petri correspondente no modelo RPOO. Desta maneira, uma classe e sua rede correspondente constituem um modelo formal para um conjunto de objetos. O comportamento dos objetos são descritos por redes de Petri coloridas modificadas com inscrições de interação.

Objetos podem interagir através de mensagens. Em um modelo RPOO, a troca de mensagens entre objetos se dá através de inscrições de interação anotadas junto às transições das redes que descrevem as classes. As inscrições descrevem ações que são efetuadas pelo objeto quando uma transição dispara. Os principais tipos de ações que podem ser associadas às transições são: (1) instanciação de uma classe, usada para criar objetos (x = new X); (2) entrada de dados – recebimento de uma mensagem por um objeto (*x?mensagem*); (3) saída síncrona de dados – envio de uma mensagem, de forma síncrona, de um objeto para outro (*x!mensagem*); (4) saída assíncrona de dados – envio de uma mensagem, de forma assíncrona, de um objeto para outro (*x.mensagem*); (5) desligamento, desfaz ligações entre objetos (-*x*); e (6) terminação da execução de um objeto (end). Uma inscrição pode conter várias ações e todas as ações de uma inscrição são executadas atomicamente quando a transição que contém a inscrição dispara.

Embora o comportamento de cada objeto seja descrito por sua rede de Petri, o comportamento de um modelo RPOO deve também considerar as interações e ligações no nível de objeto. Uma *configuração* em RPOO define um retrato do estado do sistema de objetos em um dado instante de seu funcionamento. Um configuração consiste do conjunto de objetos do sistema, as ligações entre estes objetos, o conjunto de mensagens pendentes trocadas por estes objetos, além dos estados internos das redes que modelam os objetos. A visão OO de uma configuração é definida como uma estrutura de objetos. O comportamento de um sistema RPOO pode ser observado a partir da definição de uma configuração inicial. Novas configurações são alcançadas a partir da configuração inicial e representam novos estados do sistema, resultado da ocorrência de ações descritas nos modelos dos objetos. Para cada tipo de ação definida para o sistema de objetos, existem regras que guiam a mudança de configuração de uma instância do modelo. Como exemplo de regra temos que um objeto não pode enviar uma mensagem para um outro objeto que ele não mantém uma ligação.

O espaço de estados de um modelo RPOO é um grafo que representa todas as configurações alcançáveis a partir da configuração inicial. Um nó representa uma

configuração e um arco ligando duas configurações indica a ação que foi executada para transformar uma configuração em outra.

2.2.1. Simulador de Sistemas de Objetos

O Simulador de Sistemas de Objetos (SSO) [Santos 2003] é uma ferramenta que permite simular as mudanças ocorridas em um sistema de objetos, através da execução de eventos. Um evento corresponde a uma ação ou conjunto de ações que ocorrem de forma atômica. Assim é possível exercitar a visão OO de um modelo RPOO, simulando diferentes situações de interação entre os objetos de um sistema. A análise efetuada sobre as estruturas de objetos alcançadas descreve o comportamento do sistema em termos da comunicação entre seus elementos, abstraindo detalhes do comportamento interno. A ferramenta, desenvolvida em Java, possui um conjunto de regras que determinam estados dos sistemas de objetos em função da aplicação de eventos a um determinado estado do sistema de objetos. A simulação é apresentada graficamente ao usuário. A ferramenta permite ainda a interação com o usuário através de linha de comando. Neste caso, o usuário utiliza uma forma textual para definir a estrutura inicial e as ações que serão executadas. Os cenários de simulação exercitados podem ser guardados em arquivos para serem usadas com outras ferramentas para diferentes propósitos.

2.2.2. JMobile

JMobile [Silva 2005] consiste de uma nova notação para RPOO e um framework que dá suporte a ferramentas de simulação de modelos RPOO e de geração de espaços de estados de modelos RPOO. Do ponto de vista da notação, JMobile utiliza uma sintaxe baseada na linguagem de programação Java para definir as inscrições dos modelos de Rede de Petri. Do ponto de vista do framework, JMobile disponibiliza em sua API, classes e métodos responsáveis por prover acessos às informações dos modelos e pela simulação destes modelos. A partir do framework JMobile, um conjunto de ferramentas (JMobile Tools) foi desenvolvido, incluindo um simulador e um gerador de espaço de estados para modelos RPOO. Diferente do SSO, o simulador considera além da visão OO o estado interno das redes que modelam os objetos. O gerador de espaço de estados, por sua vez, recebe como entrada um modelo RPOO e sua configuração inicial, gerando o espaço de estado correspondente.

3. Método para a Geração de Objetivos e Casos de Teste

Geração automática de casos de teste a partir de especificações formais pode aumentar a qualidade e a confiabilidade de processos de teste, visto que casos de teste tendem a ser consistentes, i.e., não rejeitam implementações corretas e podem ser convertidos diretamente em código, minimizando a introdução de defeitos de codificação no caso de serem voltados a execução automática. Conjuntos de teste podem ser mais precisamente avaliados com relação à métricas de cobertura de requisitos.

No entanto, alguns desafios ainda precisam ser vencidos. Especificações raramente são completas e com isto fica difícil assegurar que um software foi efetivamente testado apenas com base nos casos de teste gerados [Fernandez et al. 2004]. Além disso,

o processo é fortemente dependente da qualidade da especificação: ambigüidades e inconsistências são passadas diretamente para os casos de teste, invalidando todo o processo. Além de consistentes, conjuntos de caso de teste precisam refletir propriedades de interesse do sistema. Visto que não é possível garantir corretude com testes, a meta é detectar o maior número possível de defeitos relevantes. Métodos formais convencionais são voltados a dar suporte a verificação e, portanto, não apoiam diretamente as metas e atividades de um processo de teste. A escolha e direcionamento de casos de teste continua a ser uma atividade empírica onde diferentes fatores não formalizáveis vão nortear a escolha dos melhores casos de teste.

Neste cenário, os principais requisitos para um método prático de teste formal são [Tretmans 1999, Jard and Jéron 2004]:

- Geração de conjuntos de casos de teste consistentes. Isto é diretamente influenciado pelos formalismos e algoritmos de geração adotados que devem seguir uma boa fundamentação teórica e exigir um mínimo de trabalho manual;
- Geração de conjuntos de casos de teste válidos, i.e., que reflitam propriedades desejadas por seus usuários finais. Isto é diretamente influenciado pela qualidade dos processos de verificação e validação das especificações;
- Geração a partir de especificações parciais, visto que a geração total é usualmente impossível e tem pouca utilidade, pois os conjuntos de casos de teste tendem a ser muito grandes e repetitivos. Além disto, especificações completas geralmente não estão disponíveis;
- Seleção de casos de teste com base em propriedades ou funcionalidades específicas de interesse. Isto é fortemente dependente do uso de notações comumente adotadas e da disponibilidade de visões abstratas que tornem possível a investigação de funcionalidades e propriedades do sistema.
- A necessidade de construção de novos artefatos deve ser minimizada, pois o processo de construção de uma especificação formal já é por si de altos custos. Para tal, é desejável que modelos de teste sejam derivados diretamente dos construídos em processos de desenvolvimento existentes.

Apesar de aspectos teóricos de teste formal já estarem bastante consolidados e ferramentas de suporte já existirem, as abordagens atuais ainda não cobrem todos estes pontos em conjunto de forma satisfatória. Visando atingi-los, um método de teste formal é proposto neste artigo. As principais características são:

- Um mesmo modelo é usado no processo de verificação de modelos e para a geração de casos de teste, a fim de garantir que este modelo será devidamente verificado e validado;
- A geração de casos de teste é feita com base em algoritmos baseados em teoria consolidada de teste, garantindo a geração de conjuntos de casos de teste consistentes;
- A geração é dirigida por objetivos de teste, tornando possível o enfoque em partes específicas do modelo;
- Objetivos de teste são definidos com base em abstrações do modelo de teste e podem ser validados.

Para tal, RPOO foi adotado com formalismo base. As diferentes visões de RPOO e o conjunto de ferramentas já disponível e sua fácil integração com geradores de casos

de teste como TGV foram determinantes nesta escolha. Objetivos de teste podem ser gerados em um processo intuitivo por projetista de teste a partir da simulação do modelo de objetos, onde são considerados apenas conceitos de orientação a objetos referentes a criação de objetos e passagem de mensagens. Cada objetivo é representado por uma seqüência válida de eventos a partir de uma certa configuração do sistema.

A estrutura geral do método é apresentada na Figura 1, onde são ilustradas as principais entradas/saídas produzidas e as ferramentas adotadas. Os sistemas a serem testados precisam estar modelados em RPOO. Assumimos que estes modelos foram verificados usando ferramentas para verificação de modelos RPOO tal como a ferramenta Veritas [Rodrigues et al. 2004]. Utilizamos JMobile (Seção 2) para dar suporte a geração de espaços de estados que é o modelo a partir do qual casos de teste são gerados.

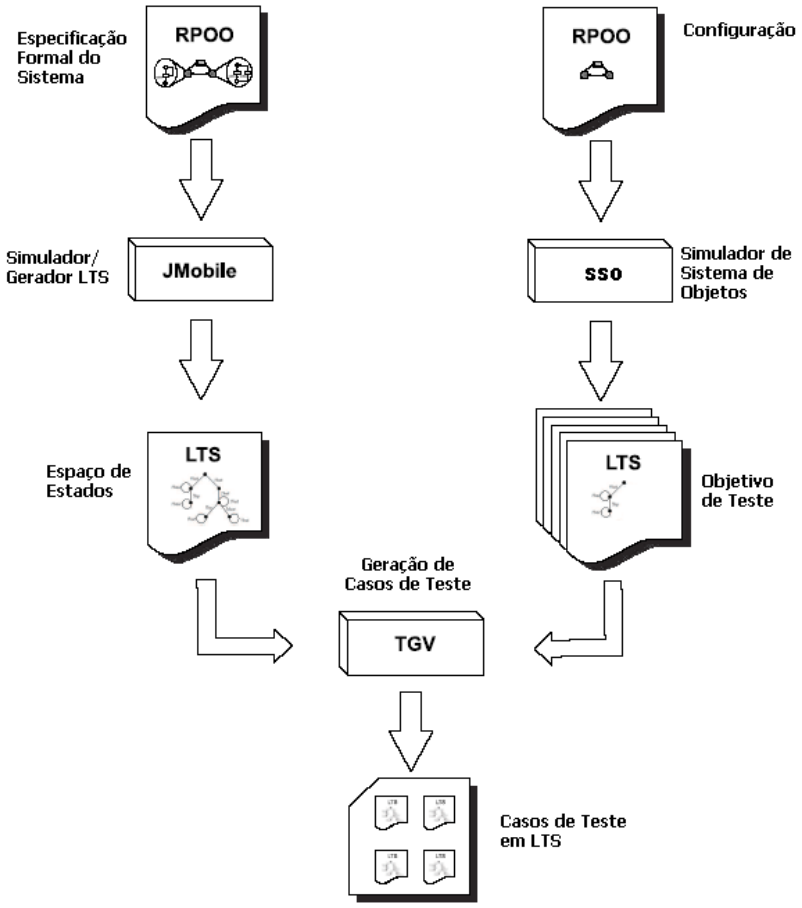

Figura 1. Visão Geral do Método para Geração Automática de Casos de Teste

JMobile 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 e outro utilizado pela ferramenta Aldebaran [Fernandez 1989]. Este último formato é utilizado no nosso método, já que a ferramenta de geração de casos de teste TGV recebe suas entradas em tal formato.

O objetivo de teste (parte direita da Figura), que constitui um comportamento específico da aplicação que se deseja testar, é obtido automaticamente através de simulações. Para isto utilizamos a ferramenta SSO (Seção 2). SSO permite simular as

alterações ocorridas no modelo da aplicação dado uma configuração, gerando possíveis objetivos de teste como seqüências de eventos. As entradas para esta ferramenta são: (i) a configuração de um sistema de objetos, que é formada pelos objetos que compõem o sistema e suas relações; e, (ii) o conjunto de eventos da execução a qual se pretende simular. A saída também é processada pela ferramenta Aldebaran a fim de produzir um IOLTS que é dado como entrada para TGV, para cada objetivo de teste.

Grande parte das ferramentas de geração de casos de teste recebe como entrada o sistema especificado em LTS e aplica sua técnica através de seus algoritmos. No nosso método, a ferramenta TGV é utilizada para a geração dos casos de teste propriamente dita. Esta ferramenta 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. O algoritmo utilizado por esta ferramenta é o de busca em profundidade (Depth First Search - DFS). 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 seqüência de UML.

O uso de TGV e o fato de que os objetivos de teste são seqüências válidas da especificação garantem que o conjunto de casos de teste gerado é *consistente*, segundo [Tretmans 1999], e que os objetivos de teste são *e-completos*, segundo [de Vries and Tretmans 2001], i.e., os casos de teste gerados a partir destes objetivos conseguem distinguir implementações que exibem e que não exibem o comportamento que se deseja observar.

4. Estudo de Caso

Esta seção mostra uma aplicação do método proposto na Seção 3. 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.

4.1. Visão Geral

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 desenvolvida sobre o paradigma de Agentes Móveis [Fuggeta et al. 1998]. 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. 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.

4.2. Modelos RPOO

Os modelos RPOO completos, contendo todos os diagramas, redes de Petri e suas respectivas explicações podem ser encontrados na dissertação de Figueiredo [Figueiredo 2005]. No geral, agentes e agências são modelados como objetos que se relacionam e trocam mensagens entre si. As principais entidades envolvidas neste estudo de caso são:

- **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.
- **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 artigo.
- **Agência Revisor**: Agência por onde o **Agente Formulário Revisão** irá passar e onde estará situado um possível revisor de um artigo.

Para cada classe do modelo, temos uma rede de Petri que descreve o comportamento de seus objetos.

4.3. Geração e Seleção de Objetivos de Teste

De acordo com o método proposto, quaisquer seqüências de eventos geradas por SSO podem ser utilizadas como objetivos de teste que denotam um comportamento de interesse que se quer testar no sistema. Em particular, neste estudo de caso, optamos por testar o comportamento esperado para padrões de projeto para agentes móveis que foram adotados no desenvolvimento da aplicação. Desta forma, SSO foi utilizado para simular o comportamento dos padrões e as seqüências de eventos obtidas foram utilizadas para definir os objetivos de teste.

Diferentes padrões de projeto foram considerados para obter objetivos de teste e, conseqüentemente, obter casos de teste para o sistema apresentado acima. Neste artigo, iremos nos ater ao padrão *Itinerary*. Este padrão define 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.

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. A Figura 2 apresenta um objetivo de teste obtido para o padrão *Itinerary*. Para realizar a simulação que propiciou sua geração, 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 de interesse estão no grafo, deixando as demais abstraídas nas transições com rótulo *.

Pela Figura 2, vemos 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.

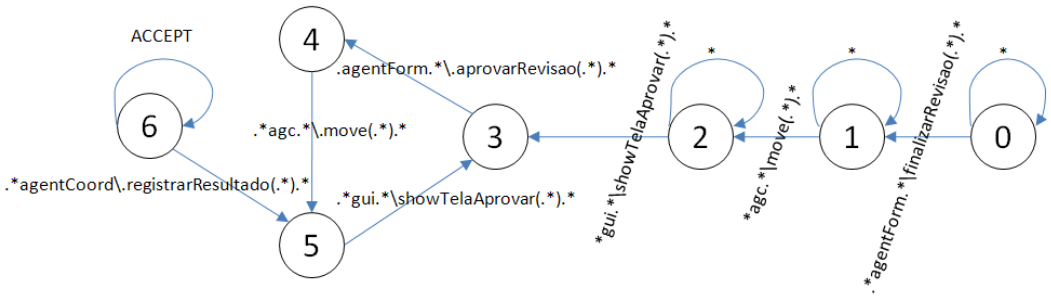

Figura 2. LTS do Objetivo de Teste para o Padrão de Projeto *Itinerary*

4.4. Resultados da Geração dos Casos de Teste

A geração de casos de teste foi realizada a partir da especificação do sistema (espaço de estados) e com base nos objetivos de teste elaborados. Tal geração se deu com o uso da ferramenta 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 objetivo de teste.

A geração dos casos de teste com TGV é simples, já que consiste em fornecer um LTS do sistema a ser testado e um objetivo de teste como entradas, ambos no formato Aldebaran. A ferramenta retorna 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 1 apresenta algumas informações para cada grafo gerado a partir de alguns objetivos de teste.

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.

Tabela 1. 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 trabalho é o de apresentar a viabilidade de implementação dos casos de teste gerados pelo método e de mostrar a eficiência dos mesmos em revelar defeitos nas aplicações, selecionamos um conjunto de casos de teste para mostrar sua implementação e execução, mostrando os defeitos encontrados, 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 seqüências de entradas e saídas do sistema. O testador fornecerá as entradas ao sistema a ser testado e é esperado que este retorne as saídas definidas. A cada rótulo de transição do caso de teste, a ferramenta acrescenta a palavra *INPUT* ou a palavra *OUTPUT*. A transição que contém a palavra *INPUT* indica ao testador que este deve esperar aquela ação como uma entrada, ou seja, uma saída do sistema a ser testado. Já a transição que possui a palavra *OUTPUT* indica ao testador que este deve produzir aquela ação para que sirva de entrada para o sistema a ser testado. Note que os termos *INPUT* e *OUTPUT* se referem ao programa que irá testar a aplicação, comumente chamado de *test driver*. A Figura 3 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 seqüência de ações procede até que, na linha 25, o *agentFormOriginal* solicita que seja migrado para a agência *agcCoord*.

```
 ------------------------------------------------------------------------
 1 <initial state>
 2 "agentCoord:guicoord.show(); INPUT"
 3 "guicoord:agentCoord.gerarFormRevisao((’agcMemb’,1)); OUTPUT"
 4 "agentCoord:conf.getDadosConferencia() &
    agentCoord:conf.criarRegistroRevisao((’agcMemb’,1)); INPUT"
 5 "conf:agentCoord.dadosConferencia(’dadosconf1’); OUTPUT"
 6 "conf:agentCoord.registroRevisao((’agcMemb’,1)); OUTPUT"
 .
 .
 .
23 "agentFormOriginal:
    guirevisao-agentFormOriginal.showTelaAprovar(); INPUT"
24 "guirevisao-agentFormOriginal:
    agentFormOriginal.aprovarRevisao(); OUTPUT"
25 "agentFormOriginal:agcMemb.move(’agcCoord’); INPUT (PASS)"
26 <goal state>
 ------------------------------------------------------------------------
```

Figura 3. Trecho de um Caso de Teste para o Sistema de Conferências

4.5. Implementação e Execução dos Casos de Teste

A abordagem escolhida para a implementação dos casos de teste foi 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 é criado (arquivo de *log*). Este foi usado para avaliar as execuções do sistema, pois as ações ali contidas devem ser as mesmas esperadas pelo caso de teste, para uma execução correta.

A execução dos casos de teste gerados foi feita de forma manual devido a falta de infra-estrutura para execução automática de testes na plataforma de agentes móveis escolhida. As ações descritas pelos mesmos foram entradas no sistema e suas saídas foram observadas. Na prática, 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. Na Tabela 2, são apresentados os casos de teste para o objetivo de teste (gerado a partir do padrão *Itinerary*) exercitados e os resultados das execuções. Apresentamos uma breve descrição dos casos de teste e mostramos os resultados da execução, através dos defeitos encontrados, caso

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 um defeito 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 implementada pelo padrão *Itinerary*, encontramos fortes indícios de que o padrão não foi implementado de forma correta.

Tabela 2. Casos de Teste Extraídos do Objetivo de Teste para o Padrão *Itinerary*

existam. Os casos de teste e uma descrição mais completa podem ser encontrados em [Figueiredo 2005].

A execução dos casos de teste nos permitiu avaliar o potencial destes em revelar defeitos. Especificamente em relação à característica de mobilidade, os casos de teste se mostraram úteis em revelar defeitos na implementação desta característica, como pode ser visto nos resultados dos Casos de Teste 02 e 03 do padrão *Itinerary* (Tabela 2). Assim, os objetivos de teste se mostraram interessantes quando se deseja testar partes específicas de um sistema, já que seus casos de teste detectaram defeitos com relação ao comportamento especificado pelos objetivos ou mostraram conformidade de tal comportamento.

5. Trabalhos Relacionados

Geração de casos de teste usando verificadores de modelo tem sido bastante explorada na comunidade [de Vries and Tretmans 1998, Jard and Jéron 2004, Bonifácio et al. 2006]. Casos de teste são gerados a partir de caminhos no modelo e objetivos de teste são formalizados, usualmente, usando lógica temporal [Fernandez et al. 2004, da Silva and Machado 2006], LTSs [Jard and Jéron 2004], diagramas de estado UML [Hartman and Nagin 2004], e Message Sequence Charts (MSC) [Henniger et al. 2003]. Abordagens simbólicas também têm sido propostas [Clarke et al. 2002, Frantzen et al. 2005].

A contribuição original deste trabalho é apresentar um método onde casos de teste são gerados com base em objetivos de teste que são definidos em um nível de abstração alto e também podem ser gerados automaticamente. Tal nível de abstração é fundamental para que o testador possa expressar propriedades de forma prática e intuitiva, através de seqüências de ativação de objetos. Estas seqüências podem representar instâncias de interesse do comportamento de sistemas. E podem ser reaproveitadas diretamente do processo de simulação dos modelos, sendo geradas automaticamente por ferramentas, como SSO. Em abordagens como as propostas por [Hartman and Nagin 2004] e [Henniger et al. 2003], objetivos de teste são construídos como um novo artefato independente. Já as propostas por [Fernandez et al. 2004, da Silva and Machado 2006] dão enfoque ao teste de propriedades temporais, podendo ser usadas de forma complementar

a proposta deste artigo.

6. Considerações Finais

Este artigo apresenta um método para a geração e seleção automática de casos de teste baseadas em objetivos de teste para sistemas reativos. O método é baseado em fundamentação teórica sólida e apoiado pelo uso de ferramentas. Outra característica importante é a sua integração com práticas de verificação de modelos, fazendo reuso de artefatos gerados por estas práticas. A proposta é inovadora no sentido de que objetivos de teste também podem ser gerados automaticamente. O estudo de caso apresentado mostra uma aplicação onde os objetivos de teste representam comportamentos desejados na implementação de padrões de projeto. Este comportamento pode ser obtido a partir do uso da ferramenta SSO. A ferramenta faz um log da simulação do envio de uma seqüência de eventos válida sobre o modelo de objetos e, ao final, esta seqüência é exportada como um LTS representando, no caso, o objetivo de teste desejado. Com isto, temos ganhos em confiabilidade na definição destes objetivos: são precisos e fiéis ao modelo. Temos também ganhos com versatilidade, pois o projetista de teste tem uma forma mais intuitiva para defini-los, através da animação do modelo usando SSO. Diagramas de classe RPOO são similares a diagramas de classe em UML padrão. Neste sentido, o método pode ser aplicado por testadores com conhecimento em UML e orientação a objetos, abstraindo o detalhamento das classes em modelos de redes de Petri.

Como trabalhos futuros, o método será estendido a fim de considerar padrões de teste na definição de objetivos de teste. Extensões e variações do formalismo RPOO serão consideradas a fim de ampliar o escopo de uso do método para software baseado em componentes e com restrições temporais.

Referências

Bonifácio, A., Moura, A., Simão, A., and Maldonado, J. (2006). Conformance testing using timed extended finite state machines and model checking. In *Proceedings of Brazilian Symposium on Formal Methods*.

Cartaxo, E. G., Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2008). LTS-BT: A tool to generate and select functional test cases for embedded systems. In *Proceedings of 23rd Annual ACM Symposium on Applied Computing*, volume 2, pages 1540–1544.

Clarke, D., Jeron, T., Rusu, V., and Zinovieva, E. (2002). STG – A Symbolic Test Generation Tool. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

da Silva, D. A. and Machado, P. D. L. (2006). Towards test purpose generation from CTL properties for reactive systems. *Electr. Notes Theor. Comput. Sci.*, 164(4):29–40.

de Vries, R. G. and Tretmans, J. (1998). On-the-fly conformance testing using spin. In *Proceedings of Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker*, pages 115–128.

de Vries, R. G. and Tretmans, J. (2001). Towards formal test purposes. In *Proceedings of 1st International Workshop on Formal Approaches to Testing of Software (FATES)*, pages 61–76, Aalborg, Denmark.

Fernandez, J., Mounier, L., and Pachon, C. (2004). Property oriented test case generation. In *Proceedings of Formal Approaches to Software Testing*, volume 2931 of *LNCS*, pages 147–163. Springer.

Fernandez, J.-C. (1989). Aldebaran: A Tool for Verification of Communicating Processes. Technical report, Rapport SPECTRE, C14, Laboratoire de Génie Informatique - Institut IMAG, Grenoble - França.

Fernandez, J.-C., Jard, C., Jeron, T., and Viho, C. (1996). Using on-the-fly Verification Techniques for the Generation of Test Suites. In *Proc. of the* 8^{th} *Int. Conf. on Computer Aided Verification CAV*, volume 1102 of *LNCS*, pages 348–359. Springer.

Figueiredo, A. L. L. (2005). Geração automática de casos de teste para sistemas baseados em agentes móveis. Master's thesis, COPIN/Universidade Federal de Campina Grande. http://www.dsc.ufcg.edu.br/ andrel/arquivos/dissertacao.pdf.

Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2005). Test generation based on symbolic specifications. In Grabowski, J. and Nielsen, B., editors, *Proceedings of FATES'04*, volume 3395 of *LNCS*, pages 1–15. Springer.

Fuggeta, A., Picco, G., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24:342–361.

Gaudel, M.-C. (1995). Testing can be formal, too. In *Proceedings of TAPSOFT*, LNCS, pages 82–96, London, UK. Springer.

Guerrero, D. D. S. (2002). *Redes de Petri Orientadas a Objetos*. PhD thesis, COPELE/Universidade Federal de Campina Grande.

Hartman, A. and Nagin, K. (2004). The AGEDIS tools for model based testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 129–132, New York, NY, USA. ACM Press.

Henniger, O., Lu, M., and Ural, H. (2003). Automatic generation of test purposes for testing distributed systems. In *Formal Approaches to Software Testing, Proceedings of FATES'03*, volume 2931 of *LNCS*, pages 178–191. Springer.

Jard, C. and Jéron, T. (2004). Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6.

Jorgensen, P. C. (2002). *Software Testing - A Craftsman's Approach*. CRC Press.

Rodrigues, C. L., Barbosa, P. E. S., Guerrero, D. D. S., and de Figueiredo, J. C. A. (2004). Rpoo model checker. In *Anais do Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, Brasília - Brasil.

Santos, J. A. M. (2003). Suporte à análise e verificação de modelos rpoo. Master's thesis, COPIN/Universidade Federal de Campina Grande, UFCG.

Silva, T. M. (2005). Simulação automática e geração de espaço de estados de modelos em redes de petri orientadas a objetos. Master's thesis, COPIN/Universidade Federal de Campina Grande.

Tretmans, J. (1999). Testing concurrent systems: A formal approach. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 46–65. Springer.