UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Dissertação de Mestrado

Geração de Objetivos de Teste para Sistemas Reativos Baseada na Técnica de Verificação de Modelos CTL

Daniel Aguiar da Silva

Campina Grande, PB, Brasil
Maio - 2006

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Geração de Objetivos de Teste para Sistemas Reativos Baseada na Técnica de Verificação de Modelos CTL

Daniel Aguiar da Silva

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: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
Orientadora

Campina Grande, PB, Brasil
Maio - 2006

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586g 2006

Silva, Daniel Aguiar

Geração de Objetivos de Teste para Sistemas Reativos Baseada na Técnica de Verificação de Modelos CTL/ Daniel Aguiar da Silva. – Campina Grande, 2006

91.: il.

Referências.

Dissertação (Mestrado em Informática) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadora: Patrícia Duarte de Lima Machado

1– Engenharia de Software 2– Teste Formal 3– Métodos Formais 4– Verificação de Modelos I-Título

CDU 004.41

Resumo

Técnicas e ferramentas de testes formais baseados em modelos têm sido desenvolvidas para tornar mais rigoroso e eficiente o processo de teste de sistemas reativos com características de distribuição e concorrência. O não-determinismo inerente a estes sistemas torna-os difíceis de serem testados, devido à alta complexidade de obtenção das configurações necessárias para execução dos casos de teste. As propriedades especificadas para estes sistemas são a base para a geração dos casos de teste de conformidade, que devem avaliar a correspondência entre modelo e código. Estas propriedades, denominadas objetivos de teste, devem ser especificadas de maneira a guiar a geração dos casos de teste. Entretanto, a especificação dos objetivos de teste a partir de modelos complexos como os destes sistemas ainda carece de técnicas e ferramentas apropriadas, tornando esta atividade propensa a erros. Os casos de teste podem assim, ter efetividade afetada em caso de erros na especificação dos objetivos de teste. Com o objetivo de contribuir para a solução deste problema, este trabalho apresenta técnica de geração de objetivos de teste para sistemas reativos, baseando-se na técnica de verificação de modelos CTL. A técnica proposta visa usufruir da eficiência dos algoritmos da verificação de modelos, por meio de sua adaptação para a análise destes, para a geração dos objetivos de teste.

Abstract

Techniques and tools for model based testing have been developed to make the process of testing distributed concurrent reactive systems more efficient and rigorous. The inherent nondeterminism of these systems can make it difficult to test them due to the complex process of obtaining test cases configurations from models. To better guide the testing process, properties specified to these systems are used as basis for the test case generation. Such properties, called test purposes, shall be exhibited by the implementation under test through test case execution. However, specifying test purposes from the common complex and large models of these systems suffers from the lack of appropriated tools and techniques, making it error-prone and inadequate. Thus, test cases based on such test purposes may be affected, getting no desirable soundness. Aiming at solving this problem, we present a technique for test purpose generation for reactive systems based on the CTL model checking technique. We aim at taking benefit from the efficiency of model checking algorithms to better analyze the models to generate the test purposes.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas reativos são caracterizados pela interação com o ambiente ao qual estão inseridos. A reatividade caracteriza-se pela interação do *software* com seu ambiente por meio de entradas e de saídas produzidas pelo *software*. Ambientes complexos geralmente exigem interações remotas, caracterizadas por distribuição, não-determinismo e concorrência, assim como a continuidade indefinida de sua execução. Especificar sistemas com estas características é uma tarefa complexa, exigindo grande esforço na produção de modelos corretos para tais sistemas. Para auxilia-la, muitos formalismos têm sido utilizados (e.g. [Jen92; Hoa85; Hol97]) na especificação dos sistemas, produzindo modelos precisos. Assim, a verificação de propriedades do modelo por meio de métodos formais torna-se propícia, possibilitando sua automação e maior precisão.

O sucesso da aplicação de técnicas de verificação formal de modelos tem sido cada vez maior à medida que evoluem algoritmos e ferramentas. Propriedades que se fazem necessárias em modelos são verificadas de maneira eficiente e automatizada [CGP99], mesmo em modelos grandes e complexos (característicos dos sistemas supracitados), contribuindo para a produção de modelos corretos.

Apesar de a verificação de modelos ser de grande contribuição na produção de sistemas mais robustos e confiáveis, não assegura que a implementação dos sistemas corresponda de maneira fiel ao modelo especificado. Um motivo que pode ser citado para tal inconsistência é a falha do componente humano introduzido no processo de codificação do *software*. Assim, técnicas de verificação e validação devem ainda ser aplicadas sobre a implementação a fim de detectar possíveis faltas.

Dentre as técnicas de verificação e validação aplicadas à implementação, teste é a mais popular. O teste é caracterizado pela realização de experimentos, denominados casos de teste, a partir da interação direta com o *software*. Para sistemas reativos estas interações são realizadas por intermédio pontos de observação e controle (PCO's) [JJ04], que são baseados em entradas e saídas do sistema, e produzem resultados que são então monitorados, permitindo posterior análise para validação.

O teste baseado em modelo, chamado teste de conformidade, é utilizado para avaliar a conformidade do comportamento da implementação em relação ao modelo especificado [FMP04]. Esta técnica consiste basicamente na análise do modelo e posterior definição dos casos de teste a partir da análise. Uma abordagem para a geração dos casos de teste baseia-se na especificação de propriedades desejáveis para o modelo, conhecidas como objetivos de teste, a partir das quais os casos de teste são projetados [JJ04]. Os objetivos de teste geralmente provêem foco específico em partes do modelo, a partir dos quais originam-se casos de teste que devem demonstrar correspondência entre os comportamentos especificados e implementados. Além de permitir que o teste seja melhor planejado com base em análises de risco, esta abordagem possibilita o estabelecimento de prioridades no processo de teste, importante fator para a indústria, que deve lidar com restrições de tempo no desenvolvimento de produtos.

A aplicação de teste de conformidade a sistemas reativos que operam em ambientes complexos, quando pautada em mecanismos informais, pode tornar-se cara e ineficiente devido ao não-determinismo inerente a estes sistemas. A possibilidade de se obter diferentes respostas da implementação, para uma determinada entrada, pode influenciar no resultado dos casos de teste, podendo prejudicar o processo. O desenvolvimento de mecanismos que provejam maior rigor ao teste faz-se assim, necessário.

A aplicação de técnicas de teste de conformidade a partir de modelos formais tem caracterizado parte dos esforços para agregar maior rigor e eficiência ao teste de sistemas reativos [Tre96; Gau95]. Técnicas e ferramentas [JJ04; dVT98; SEG+98] têm sido desenvolvidas e até aplicadas em projetos da indústria (e.g. [FJJV97]). No entanto, a aplicação destas técnicas ao processo de teste de sistemas tem enfrentado barreiras em relação à especificação de objetivos de teste devido à falta de técnicas e ferramentas para sua obtenção. Caso os objetivos de teste não descrevam adequadamente as propriedades a serem testadas, os casos

de teste gerados podem ter eficiência comprometida.

A especificação manual de objetivos de teste tem se mostrado inviável. Com especificações formais, eles geralmente são descritos por formalismos de baixo nível, portanto, difíceis de serem entendidos. Adicionalmente, a manutenção destes artefatos com base nas geralmente grandes especificações dos sistemas é uma tarefa extremamente laboriosa, além de propensa a erros.

Em [HLU03] é apresentado um algoritmo para geração de objetivos de teste na forma de MSC's a partir de estruturas de eventos rotuladas que representem o comportamento de sistemas. A técnica consiste em identificar em modelos, representados por máquinas de estado de comunicação assíncrona, os comportamentos definidos como significantes. A partir de tais comportamentos significantes os objetivos de teste são definidos, visando gerar um caso de teste para cada comportamento. Apesar da característica de automação desta técnica e de sua capacidade de identificar diretamente nos modelos os comportamentos significantes do sistema, o conjunto de casos de teste gerado tende a ser reduzido, não garantindo sua exaustão. Isto acontece pelo fato de o método não prover maior nível de abstração dos objetivos de teste em relação ao grafo de representação dos estados do modelo. Os objetivos de teste gerados também não provêem informações sobre os comportamentos desejáveis para os casos de teste gerados.

Apesar da solução proposta em [HLU03], o problema da geração automática de objetivos de teste permanece aberto, uma vez que possibilitar a um objetivo de teste descrever de maneira abstrata as propriedades de um sistema, e não apenas com cenários específicos, de modo a permitir a geração ampla de casos de teste é forte requisito. Outro requisito importante é a possibilidade de um engenheiro de teste definir claramente quais características do *software* deseja validar. Assim, deve dispor de formalismo e ferramentas capazes de prover auxílio na especificação criteriosa e detalhada de propriedades do sistema.

Um formalismo capaz de prover poder de abstração adequado para representar diversos cenários concretos é a lógica temporal ramificada (CTL) [CE81]. Este formalismo, além de ser mais adequado ao manuseio humano que formalismos de baixo nível, como sistemas de transições rotuladas, CTL permite a definição de forma clara e precisa de propriedades com a noção de ordem temporal. Tais características são usufruídas pelas técnicas de verificação de modelos na especificação de propriedades a serem verificadas, compondo um mecanismo

eficiente e preciso na análise sobre modelos.

Como uma implementação deve atender aos mesmos requisitos de seu modelo, ou seja, deve satisfazer as mesmas propriedades, o que precisa ser verificado em um modelo também deve ser testado em uma implementação. Desta maneira, casos de teste devem ser gerados por meio destas propriedades, tornando os objetivos de teste equivalentes às mesmas.

Com base na equivalência entre as propriedades especificadas para verificação de modelos e os objetivos de teste, sua geração pode basear-se de maneira eficiente no processo de verificação de modelos. Assim, a partir de análises detalhadas sobre o modelo, pode-se obter objetivos de teste que permitam extrair do modelo casos de teste completos e eficientes.

1.1 Objetivos

Com base no exposto acima, o objetivo deste trabalho é apresentar técnica de geração automática de objetivos de teste para sistemas reativos a partir da técnica de verificação de modelos. A técnica baseia-se na especificação de propriedades por intermédio de fórmulas descritas em CTL e na utilização de um verificador de modelos adaptado para a geração de objetivos de teste. A síntese dos objetivos de teste é realizada com base na análise sobre exemplos e contra-exemplos extraídos do modelo pelo verificador. Os objetivos de teste gerados são especificados como sistemas de transições rotuladas e devem prover informações sobre os comportamentos aceitos ou não pelos casos de teste, provendo base formal para a definição de veredictos sobre a execução destes.

1.2 Resultados

O principal resultado obtido neste trabalho é uma técnica de geração automática de objetivos de teste, que se propõe a suprir um sério problema em aberto da área de teste. O trabalho desenvolvido tem grande relevância para a área de teste, assim como para o processo de desenvolvimento de *software* com base em métodos formais. Não há relatos na literatura sobre a aplicação de formalismos de lógica temporal na especificação de objetivos de teste, caracterizando a contribuição deste trabalho como inovadora para a área. A aplicação de técnicas de verificação de modelos ao teste não é completamente nova, porém, sua aplicação

à etapa de geração de objetivos de teste também é inovadora, promovendo uma importante integração das duas técnicas. Uma breve descrição dos resultados obtidos é dada a seguir:

Técnica de geração automática de objetivos de teste A automação do processo de geração de objetivos de teste poderá contribuir de forma significativa para uma consistência e eficiência do processo de geração de casos de teste, excluindo um ponto crítico, em geral, manual e suscetível a erros do processo. Tal proposta de automação mostra-se assim, necessária e de grande impacto no processo de teste, uma vez que o objetivo de teste é peça-chave para o sucesso do processo de teste formal baseado em propriedades.

Especificação de objetivos de teste por meio de propriedades CTL A dificuldade de especificar e manter objetivos de teste com formalismos de baixo nível pode aumentar sobremaneira os custos do teste, tornando a sua realização impraticável. Além de dificultar a legibilidade dos mesmos, se especificados com o mesmo nível do modelo, podem manter um acoplamento perigoso, no qual mudanças sobre o modelo implicam em revisões e possíveis mudanças sobre os objetivos de teste. A utilização de formalismos de mais alto nível para a realização destas tarefas soluciona tais problemas permitindo maior legibilidade por parte das equipes de teste e desenvolvimento. Permite-se também, maior nível na especificação das propriedades, minimizando o acoplamento em relação ao modelo, o que possibilita menores custos com manutenção. Com sua notação matemática, CTL permite às propriedades maior legibilidade que formalismos de baixo nível, como sistemas de transição rotuladas, além de menor acoplamento em relação ao modelo, diminuindo assim, a necessidade de manutenção sobre os objetivos de teste. Como formalismo baseado em lógica temporal, CTL tem grande poder de expressividade, podendo abranger diversos tipos de propriedade funcionais e não funcionais de um sistema.

Integração dos processos de verificação de modelos e teste A geração de objetivos de teste por intermédio da verificação de modelos promove uma integração importante entre as duas principais técnicas de verificação & validação de *software*. A análise eficiente sobre um modelo da implementação, realizada pela técnica de verificação de modelos, contribui para uma especificação mais rigorosa dos objetivos de teste, provendo assim, maior rigor à etapa de geração de casos de teste. Esta integração permite maior consistência no processo

de desenvolvimento de *software* com base em metodologia formal.

1.3 Estrutura da Dissertação

O restante deste documento está estruturado da seguinte maneira:

Capítulo 2: Fundamentação Teórica Este capítulo apresenta os principais conceitos que baseiam o trabalho desenvolvido. Dentre eles estão os conceitos de teste e teste formal, bem como ferramentas e técnicas existentes na área. A aplicação de técnicas de verificação de modelos ao teste é discutida de forma a mostrar, com a apresentação de trabalhos na área, sua viabilidade, de modo a dar suporte à motivação para o desenvolvimento da técnica. São também apresentadas as ferramentas utilizadas no estudo de caso apresentado nesta dissertação, bem como sua teoria relacionada.

Capítulo 3: Geração de Objetivos de Teste Este capítulo apresenta a principal contribuição deste trabalho, uma técnica de geração automática de objetivos de teste com base em especificações de fórmulas CTL. A estrutura da técnica é apresentada, bem como é feito um detalhamento de sua solução algorítmica para o problema de geração de objetivos de teste. Por último, o embasamento matemático dos algoritmos é demonstrada por meio de sua formalização.

Capítulo 4: Implementação de um Protótipo: Adaptação sobre o Veritas Este capítulo apresenta a abordagem algorítmica utilizada para a realização da adaptação do verificador de modelos utilizado no estudo de caso. A implementação é discutida, com base em um paralelo estabelecido em relação à implementação original do verificador de modelos.

Capítulo 5: Estudo de Caso: Geração de Objetivos de Teste para o Protocolo IP Móvel Este capítulo apresenta um estudo de caso realizado para demonstrar o funcionamento da técnica desenvolvida, assim como validar o trabalho desenvolvido. Este estudo de caso é realizado com o auxílio de algumas ferramentas apresentadas no Capítulo 2, envolvendo um verificador de modelos e um gerador automático de casos de teste.

Capítulo 6: Conclusão Este capítulo apresenta as conclusões acerca do trabalho desenvolvido discutindo as principais questões envolvidas à área e apontando possíveis desdobramentos em trabalhos futuros.

Capítulo 2

Fundamentação Teórica

A técnica proposta nesta dissertação envolve conceitos como o de teste e verificação de modelos. O melhor entendimento deste trabalho por parte do leitor requer um prévio conhecimento de tais conceitos. Deste modo, apresentamos neste capítulo a fundamentação teórica necessária, bem como discutimos o estado da arte, abordando os trabalhos relacionados que aplicam técnicas de verificação de modelos ao teste de sistemas. Dentre estes, apresentamos com maiores detalhes a ferramenta TGV [JJ04], utilizada para gerar os casos de teste do estudo de caso desta dissertação.

2.1 Teste

Teste tem sido importante ferramenta utilizada na validação de funcionalidades e qualidade de sistemas. É caracterizado pela realização de experimentos a partir da interação direta com o *software*. Estas interações produzem resultados que são observados, permitindo posterior análise para a validação [Bei90]. A validação deve ser efetuada de acordo com determinados critérios estabelecidos durante o planejamento do teste.

Existem diversos tipos de teste (e.g. [Mye79; Bei90; MS01; Bin03]), os quais devem ser aplicados de acordo com o aspecto do sistema (e.g. interface, desempenho, carga) a ser testado, bem como o seu tipo (e.g. sistemas orientados a objetos, distribuídos e reativos). Os tipos de teste podem ser classificados em dois grandes grupos, denominados teste estrutural e teste funcional.

Teste estrutural, também conhecido como teste de caixa branca, ou *white-box testing*,

baseia-se em implementações. Seu principal objetivo é testar detalhes procedimentais da implementação [Mye79], de onde os requisitos do teste devem ser extraídos. Os critérios de cobertura desta técnica baseiam-se na utilização de grafos como ferramenta para a descrição de caminhos, como os definidos pelos fluxos de dados e controle da implementação.

Teste funcional, também conhecido como teste de caixa preta, ou *black-box testing*, tem como objetivo validar as funcionalidades de um sistema. Baseia-se na especificação do sistema para sua definição e aplicação. Ao contrário do teste estrutural, esta técnica é, em geral, independente de implementação, o que permite sua definição durante a fase de projeto do sistema. Isto permite que, em processos de desenvolvimento *top-down*, nos quais a codificação do sistema baseia-se em especificações previamente definidas, o desenvolvimento dos casos de teste contribui para melhor entendimento e correção de modelos desde as etapas iniciais dos processos, evitando detecções de problemas de maneira tardia, diminuindo assim, o impacto e custos de eventuais mudanças.

A técnica de avaliar a correção do comportamento da implementação em relação à sua especificação por meio de experimentos diretos chama-se teste de conformidade. Esta técnica consiste na análise da especificação do sistema, definindo quais funcionalidades serão testadas e como serão testadas; geração e execução dos experimentos, denominados casos de teste; análises e posterior conclusão acerca dos resultados obtidos.

Quando a especificação do sistema é definida em termos de modelos, denomina-se o teste como teste baseado em modelos [EFW01]. A geração dos casos de teste desta técnica pode ser baseada em critérios de cobertura sobre o modelo, com os quais busca-se definir conjunto de casos de teste para o modelo como um todo [SJPLP99]. Outra abordagem para esta geração baseia-se na especificação de propriedades desejáveis para o modelo, conhecidas como objetivos de teste, a partir das quais os casos de teste são projetados [JJ04]. Tal abordagem é também conhecida como teste baseado em propriedades [MSM05]. Os objetivos de teste geralmente provêem foco específico de partes do modelo, fazendo com que os casos de teste gerados a partir destes, sejam específicos e atinjam apenas tais partes.

Após a geração dos casos de teste, a execução dos mesmos é realizada. Esta etapa deve ser precedida pela geração dos dados de teste. Os dados de teste são as entradas que devem ser fornecidas ao *software* para que se possa executar os experimentos definidos nos casos de teste. A análise sobre os resultados obtidos após a execução dos casos de teste deve ser

concluída com um veredito sobre a correção do comportamento exibido pela implementação em relação à sua especificação. Este procedimento é realizado por oráculos de teste, de maneira automatizada, por um *software*, ou manual, por um ser humano.

2.2 Teste Formal

A aplicação *ad hoc* do teste no desenvolvimento de sistemas tem tornado o processo de validação caro e ineficiente. Em processos de desenvolvimento de sistemas críticos (e.g. controle aero-espacial), os custos da fase de validação com teste oscilam entre 50% e 70% dos custos totais do projeto [HRV+03], fato que evidencia a grande importância de teste no processo de desenvolvimento.

Pesquisas têm sido desenvolvidas com a finalidade de tornar o processo de teste mais eficiente e com menores custos. Dentre as pesquisas realizadas com o intuito de prover um maior rigor ao teste, a aplicação de métodos formais compõe importante linha. Apesar das crenças de que a característica empírica dos experimentos sobre um artefato não-formal, como uma implementação, não comportaria a aplicação rigorosa de métodos formais, alguns trabalhos (e.g. [Tre96; Gau95; Tre99]) demonstraram o contrário. A aplicação de métodos formais ao teste seria, assim, possível.

Conhecido como teste formal, o teste baseado em métodos formais possibilita a especificação e análise rigorosas dos casos de teste de maneira automática. Diversas abordagens foram desenvolvidas com o intuito de prover maior rigor aos testes. Dentre elas, destacam-se as técnicas baseadas em modelos formais (e.g. baseadas em máquinas de estados [Cho78; FvBK+91; PBG04] e em sistemas de transições rotuladas [JJ04; Tre99; FMP04]), por meio dos quais deve-se estabelecer a conformidade com a implementação [JJ04].

A teoria na qual a técnica proposta nesta dissertação se fundamenta, baseia-se no *framework* formal de teste de conformidade apresentado em [Tre99]. Este *framework* apresenta formalmente os conceitos utilizados no processo de teste de conformidade, bem como provê mecanismos para avaliação dos casos de teste. Uma extensão apresentada em [dVT01] introduz o conceito de objetivos observáveis como a formalização para o conceito de objetivos de teste.

2.2.1 Framework Formal de Teste de Conformidade

Em busca da formalização do processo de teste de conformidade devemos estabelecer representação formal da relação de conformidade entre implementação e especificação. Diferentemente das especificações, implementações pertencem ao mundo informal, impossibilitando assim o estabelecimento de relação formal entre ambos. No entanto, um artifício pode ser utilizado para a realização desta tarefa. A relação de conformidade entre uma IUT e sua especificação pode ser formulada por uma relação, chamada de relação de implementação, entre um objeto formal que represente o modelo da IUT e a especificação. A suposição da existência de um modelo formal para uma IUT é denominada de hipótese de teste [Ber91].

Definindo as relações supracitadas, denominamos $SPECS$ o universo das especificações formais e $IMPS$ o universo das IUT's. Definimos a relação de conformidade através de **conforms-to** $\subseteq IMPS \times SPECS$. Desta maneira, para uma $IUT \in IMPS$ e $spec \in SPECS$, a correção entre a IUT e a especificação $spec$ é expressada através de IUT **conforms-to** $spec$. Para definir a relação que representa formalmente a relação **conforms-to** definimos $MODS$ como o universo dos modelos. Assim, relação de implementação entre modelos e especificações é dada por **imp** $\subseteq MODS \times SPEC$. Portanto, para um modelo $i_{IUT} \in MODS$ da implementação IUT:

$$IUT \textbf{ conforms-to } spec \iff i_{IUT} \textbf{ imp } spec \tag{2.1}$$

Para concluir acerca da relação de conformidade entre implementação e especificação é necessário efetuar testes sobre a implementação e então observar seu comportamento. A execução dos testes dá-se pela definição e execução dos casos de teste sobre a IUT. A formalização deste processo passa pela definição formal dos casos de teste e das observações realizadas sobre sua execução. As observações são utilizadas para a formalização do conceito de execução dos casos de teste, que, assim como as implementações, faz parte do mundo informal.

Definimos o universo dos casos de teste como $TESTS$ e a execução de um caso de teste $t \in TESTS$ sobre uma $IUT \in IMPS$ através de um procedimento operacional denominado $exec(t, IUT)$. O domínio das observações efetuadas sobre o procedimento operacional denominamos OBS. A formalização do processo de execução dos casos de teste pode ser por uma função de observação $obs : TESTS \times MODS \longrightarrow P(OBS)$. Assim, $obs(t, i_{IUT})$ representa formalmente a execução $exec(t, IUT)$. Diante da relação estabelecida entre o

procedimento operacional $exec$ e a função de observação obs e da hipótese de teste na qual se baseia a relação entre implementação e modelo, podemos estabelecer que:

$$\forall IUT \in IMPS\ \exists i_{IUT} \in MODS\ \forall t \in TESTS : exec(t, IUT) = obs(t, i_{IUT}) \quad (2.2)$$

As observações realizadas sobre a execução dos casos de teste levam a conclusões acerca da correção do comportamento da implementação em relação ao comportamento especificado. Estas conclusões compreendem vereditos que definem se a execução foi correta ou errada. Tais vereditos devem ser dados por funções, chamadas funções de veredito, definidas por $verd_t : P(OBS) \longrightarrow \{\textbf{fail}, \textbf{pass}\}$. Assim, para um caso de teste t definimos que a implementação IUT é correta em relação a t caso o veredito sobre sua execução tem como resultado **pass**:

$$IUT\ \textbf{passes}\ t =_{def} verd_t(exec(t, IUT)) = \textbf{pass} \quad (2.3)$$

A relação de conformidade entre implementação e especificação obtida através do veredito positivo em relação à execução de um conjunto de casos de teste $T \subseteq TESTS$: IUT **passes** $T \iff \forall t \in T : IUT$ **passes** t. Assim:

$$IUT\ \textbf{conforms-to}\ spec \iff IUT\ \textbf{passes}\ T \quad (2.4)$$

Um conjunto de casos de teste que satisfaz a propriedade definida em 2.4 é chamado completo. Conjuntos de casos de teste com estas características são considerados efetivos e exaustivos. Efetividade e exaustividade significam que apenas implementações corretas passarão nos testes e que todas implementações não-conformes serão detectadas, respectivamente. Porém, a obtenção de conjuntos de casos de teste completos na prática torna-se impossível, uma vez que estes tendem a ser infinitos. Considerando este fato, a definição de efetividade e exaustão têm seu rigor diminuído. Um conjunto de casos de teste efetivo implicará que todas as implementações corretas e, possivelmente algumas incorretas, passarão nos testes. Já um conjunto exaustivo implicará que algumas implementações não-conformes poderão não ser detectadas. Para demonstrar efetividade, ou exaustividade, de um conjunto de casos de teste particular:

$$\forall i \in MODS : (i\ \textbf{imp}\ s \iff \forall t \in TESTS : verd_t(obs(t, i)) = \textbf{pass}) \quad (2.5)$$

Uma vez a equação 2.5 é demonstrada, podemos assumir que:

$$IUT \textbf{ passes } T$$

sse (Definição de IUT **passes** T)

$$\forall\, t \in T : IUT \textbf{ passes } t$$

sse (Definição de IUT **passes** t)

$$\forall\, t \in T : verd_t(EXEC(t, IUT)) = \textbf{pass}$$

sse (Hipótese de teste (2.2))

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

sse (Completeza dos modelos (2.5) aplicada a i_{IUT})

$$i_{IUT} \textbf{ imp } s$$

sse (Definição de conformidade)

$$IUT \textbf{ conforms-to } s$$

Desta maneira, se a propriedade de completeza (2.4) for demonstrada para o nível dos modelos e, se há base para assumir a hipótese de teste, então a relação de conformidade entre implementação e especificação pode ser estabelecida por intermédio de procedimentos de teste.

A derivação de casos de teste completos deve ser realizada por um procedimento definido pela função $der_{imp} : SPECS \rightarrow \mathcal{P}(TESTS)$. Atendendo aos requisitos de efetividade, este procedimento deve satisfazer a equação 2.4 da esquerda para a direita. Já a exaustividade deve satisfazer tal equação da direita para a esquerda.

2.2.2 Objetivos Formais de Teste de Conformidade

A abordagem de definição dos casos de teste a partir da especificação de propriedades implica na observação de tais propriedades durante a execução dos casos de teste sobre a IUT. A relação de conformidade deve ser estabelecida então por meio destas observações [dVT01], que podem apenas ser realizadas caso a IUT seja capaz de exibi-las durante sua execução.

Definimos o universo dos objetivos de teste como $TOBS$ e a relação de exibição entre uma IUT e os objetivos de teste **exhibits** $\subseteq IMPS \times TOBS$. No entanto, para representar esta relação no mundo formal, devemos estabelecer relação entre modelos de implementação e os objetivos de teste. Assim, definimos uma relação de revelação **rev** $\subseteq MODS \times TOBS$. Uma IUT deve satisfazer um objetivo de teste, exibindo-o durante sua execução, se e somente

se o modelo da IUT também satisfaz tais propriedades, ou seja, a relação de revelação entre modelo e objetivo de teste é válida. Formalmente, para uma implementação $IUT \in IMPS$, um objetivo de teste $e \in TOBS$ e um modelo da IUT $i_{IUT} \in MODS$:

$$IUT \textbf{ exhibits } e \iff i_{IUT} \textbf{ rev } e \tag{2.6}$$

O veredito sobre as exibições efetuadas pela implementação em relação a um objetivo de teste pode ser definido por uma função, denominada *hit*: $H_e : P(OBS) \longrightarrow \{\textbf{hit}, \textbf{miss}\}$. Assim, a seguinte definição pode ser estabelecida, dado T_e um conjunto de casos de teste obtido com base em um objetivo de teste e:

$$IUT \textbf{ hits } e \textbf{ by } T_e =_{def} H_e(\bigcup\{exec(t, IUT) \mid t \in T_e\}) = \textbf{hit} \tag{2.7}$$

Em relação à observação de propriedades durante a execução de casos de teste, podemos então concluir que uma implementação exibe determinada propriedade e quando a execução do conjunto de casos de teste T_e aponta veredito positivo (**hit**) durante sua execução. Assim:

$$IUT \textbf{ exhibits } e \iff IUT \textbf{ hits } e \textbf{ by } T_e \tag{2.8}$$

A propriedade 2.8 define a noção de completude em relação a um objetivo de teste, denominada e-completude. Um conjunto de casos de teste que satisfaz esta propriedade é denominado e-completo. Conjuntos de casos de teste com estas características são considerados e-efetivos e e-exaustivos, significando que apenas implementações que exibam a propriedade passarão nos testes e que todas implementações que não exibam serão detectadas, respectivamente. Para demonstrar e-efetividade, ou e-exaustividade, de um conjunto de casos de teste particular:

$$\forall i \in MODS : (\ i \textbf{ rev } e \iff H_e(\bigcup\{obs(t_e, i) \mid t_e \in T_e\}) = \textbf{hit}\ ) \tag{2.9}$$

Uma vez a equação 2.9 é demonstrada, podemos assumir que:

$$IUT \textbf{ hits } e \textbf{ by} T_e$$
$$\textit{sse} \text{ (Abreviação } \textbf{hits } e \textbf{ by} T_e)$$
$$H_e(\bigcup\{exec(t_e, IUT) \mid t_e \in T_e\}) = \textbf{hit}$$
$$\textit{sse} \text{ (Hipótese de teste (2.2))}$$
$$H_e(\bigcup\{obs(t_e, i_{IUT}) \mid t_e \in T_e\}) = \textbf{hit}$$

sse (e-completeza dos modelos (2.9) aplicada a i_{IUT})

$$i_{IUT} \textbf{ rev } e$$

sse (Definição de exibição)

$$IUT \textbf{ exhibits } e$$

Pode-se assim, definir quando uma implementação exibe um determinado objetivo de teste por meio de testes, uma vez demonstrada a propriedade de completeza e a hipótese de teste.

Os conceitos de conformidade e exibição, aqui apresentados, podem ser relacionados. Um objetivo de teste pode ser utilizado como critério para a seleção de conjuntos de casos de teste que sejam e-completos e efetivos. A e-efetividade garante que uma implementação que obtém resultado positivo em uma execução exibe o comportamento desejado. Por outro lado, e-exaustividade garante-nos encontrar todas as implementações capazes de exibir tal comportamento. A efetividade provê a capacidade de detectar implementações não-conformes. Assim, toda implementação incorreta será detectada.

Um objetivo de teste pode ser revelado tanto por implementações conformes, como por implementações não-conformes em relação a uma dada especificação. Uma situação ideal, apesar de impraticável, consideraria um objetivo de teste e quando i **rev** $e \supseteq i$ **passes** T, onde $T \in TESTS$. Entretanto, os objetivos de teste são escolhidos de forma que $\{i \mid i \textbf{ rev } e\} \cap \{i \mid i \textbf{ imp } s\} \neq \emptyset$. Neste caso, para um caso de teste $T_{s,e}$ e-completo e efetivo, a obtenção de resultados **fail** significa a identificação de não-conformidade, uma vez que casos de testes e-efetivos não rejeitam implementações conformes e sua e-completeza garante a distinção entre todas as implementações capazes de exibir ou não o objetivo de teste. Por fim, se uma execução atinge resultado (**pass,hit**), a confiança em correção da implementação torna-se alta, uma vez que **hit** provê possibilidade de conformidade. Por outro lado, um resultado (**pass,miss**) não define prova de não-conformidade de uma implementação, mas apenas que não encontrou as evidências de confiança na correção da implementação.

2.3 Verificação de Modelos

A verificação de modelos é uma técnica utilizada para verificar, de maneira rigorosa e automatizada, a correção de modelos de sistemas reativos, concorrentes e distribuídos [CGP99].

Por meio de um modelo formal (e.g. [Jen92; Hoa85; Hol97]) e uma propriedade especificada em formalismos de lógica temporal (e.g. LTL [Pnu77], CTL [CE81]), um *software*, denominado verificador de modelos, deve efetuar a verificação da satisfazibilidade da fórmula em relação ao modelo. Esta verificação de satisfazibilidade em relação ao modelo dá-se sobre o seu espaço de estados. O espaço de estados é a representação de todos os estados possíveis sobre o modelo, normalmente em uma linguagem abstrata de baixo nível, como tipos especiais de máquinas de estados finitos, como estruturas de Kripke.

A Figura 2.1 mostra o esquema de funcionamento do processo de verificação de modelos. O verificador efetua uma busca exaustiva sobre o espaço de estados do modelo para concluir acerca da satisfação da propriedade em relação ao modelo. Em caso de o modelo satisfazer a propriedade, um resultado positivo deve ser emitido, certificando a correção do modelo. Em caso de o modelo não satisfazer a propriedade, o verificador emite o resultado negativo e um contra-exemplo, mostrando o porquê de a propriedade não ser satisfeita pelo modelo. O contra-exemplo emitido pelo verificador constitui-se de um caminho extraído do grafo do espaço de estados do modelo.

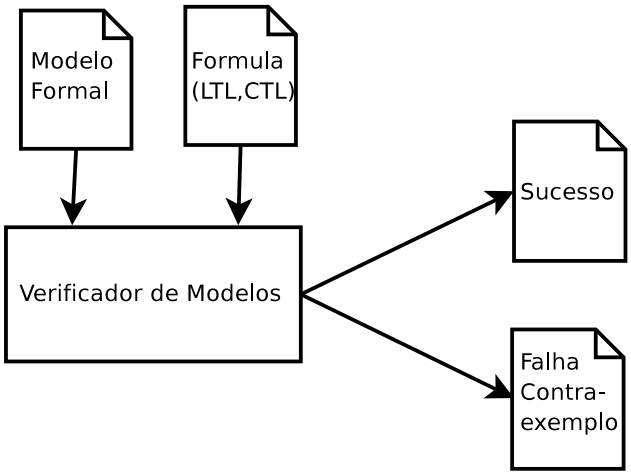


Figura 2.1: Verificação de Modelos

Inicialmente, a verificação de modelos foi desenvolvida para ser aplicada ao desenvolvimento de *hardware*. O sucesso de sua aplicação levou a comunidade acadêmica a aplicá-la ao desenvolvimento de *software*, de maneira a tornar mais rigoroso o desenvolvimento de sistemas. Entretanto, a aplicação da técnica por meio da enumeração exaustiva do espaço de estados de sistemas reativos apresentou sérias restrições em relação ao tamanho da represen-

tação do sistema.

Os sistemas reativos, caracterizados pela interação com o ambiente, são tipicamente concorrentes e distribuídos. Sistemas com tais características normalmente não terminam, tendo execução infinita. A representação do espaço de estados destes sistemas torna-se impraticável, devendo ser limitada. No entanto, mesmo representações finitas podem sofrer sérias restrições, devido a limitações de memória. Tais limitações acarretam em um problema chamado, problema da explosão do espaço de estados [CGP99], no qual a verificação de determinados modelos torna-se impraticável.

Diversas técnicas têm sido propostas para amenizar o problema do espaço de estados, tais como redução de ordem parcial e verificação *on-the-fly* [CGP99]. Porém, o problema da explosão de espaço de estados permanece para uma enumeração exaustiva dos mesmos. Com o intuito de solucionar tal problema, uma técnica de verificação de modelos baseada na representação simbólica do espaço de estados foi proposta em [Mcm93].

As técnicas de verificação de modelos mais conhecidas e aplicadas são as técnicas baseada em formalismos lógica temporal linear (LTL) e lógica temporal ramificada (CTL). Estes formalismos são capazes de expressar relações de ordem sem a necessidade de recorrer de maneira explícita à noção de tempo. A abordagem baseada em LTL considera que uma propriedade deve ser satisfeita em todas as execuções do sistema. A abordagem baseada em CTL considera que uma propriedade deve ser satisfeita para uma ou todas as execuções do sistema, a depender de suas restrições. Como a base do trabalho proposto é operar com base na verificação de modelos CTL, a seguir apresentaremos resumidamente apenas tal abordagem e seu formalismo base.

2.3.1 Lógica Temporal Ramificada - CTL

Ao contrário da lógica temporal linear, que considera, a cada momento no tempo apenas um único futuro, a lógica temporal ramificada [CE81] pode considerar diferentes futuros possíveis. A execução do sistema pode ser representada com base em ramificações, como em uma árvore, podendo haver grande número possibilidades. Tais possibilidades revelam execuções que podem ser quantificadas com base em uma ou em todas as execuções do sistema.

Informalmente, fórmulas em CTL podem compostas por operadores especiais, opera-

dores de lógica proposicional e proposições atômicas. Os operadores desta lógica ocorrem sempre em pares podendo ser formados pelos quantificadores universal A ou existencial E, seguidos pelos operadores F,G,U ou X. O quantificador universal indica que a fórmula especificada deve considerar todas as execuções da árvore de possibilidades do sistema. O quantificador existencial indica que a fórmula especificada pode considerar apenas uma execução da árvore, determinando a obrigatoriedade da existência de ao menos uma execução.

A noção de ordem temporal expressada pelos operadores F,G,U e X baseia-se na concepção de estados e relação de predecessão, definida através da estrutura de Kripke. Formalmente, uma estrutura de Kripke é uma tupla $\mathcal{M} = (S, I, R, L)$, onde S é um conjunto finito de estados, $I \subseteq S$ o conjunto de estados iniciais, $R \subseteq S \times S$ a relação de transição satisfazendo $\forall\, s \in S.(\exists\, s' \in S.(s, s') \in R)$.

O operador F indica a idéia de futuro. Por exemplo, a fórmula $EF\phi$ determina a existência, em algum caminho, de um estado futuro no qual a proposição atômica ϕ é satisfeita. A fórmula $AF\phi$ determina a existência, em todos os caminhos, de um estado futuro no qual a proposição atômica ϕ é satisfeita.

O operador G implica que a fórmula deve ser satisfeita globalmente, ou seja, durante toda a execução do sistema. Por exemplo, a fórmula $EG\phi$ determina a existência, de algum caminho, para o qual a proposição atômica ϕ é satisfeita em todos os estados. A fórmula $AG\phi$ determina que, para todos os caminhos, a proposição atômica ϕ é satisfeita em todos os estados.

O operador U é um operador binário, o qual implica que o primeiro argumento deve ser verdadeiro até que o segundo o seja, não importando o valor do primeiro argumento após esta condição. Por exemplo, a fórmula $EU(\phi, \theta)$ determina a existência de algum caminho, no qual ϕ seja verdadeira até que θ o seja. A fórmula $AU(\phi, \theta)$ determina que, em todos os caminhos, ϕ seja verdadeira até que θ o seja.

O operador X implica a satisfação da fórmula no estado imediatamente posterior ao atual. Por exemplo, a fórmula $EX\phi$ determina a existência de algum caminho, no qual ϕ é satisfeita no estado sucessor imediato. A fórmula $AX\phi$ determina que, em todos os caminhos, ϕ é satisfeita no estado sucessor imediato.

A sintaxe de CTL é formalmente dada pela seguinte definição [CGP99]: Seja AP o conjunto de proposições atômicas, então:

1. cada proposição $p \in AP$ é uma fórmula CTL;

2. se ϕ e ψ são fórmulas CTL, então $\neg\phi$, $\phi \wedge \psi$, $EX\phi$, $AX\phi$, $E[\phi U \psi]$, $A[\phi U \psi]$, $EG\phi$, $AG\phi$, $EF\phi$ e $AF\phi$ também o são.

Cada um dos operadores pode ser descrito através dos operadores $EX\phi$, $EG\phi$ e $E[\phi U \theta]$:

- $AX\phi \equiv \neg EX\neg\phi$;
- $EF\phi \equiv E[trueU\phi]$;
- $AG\phi \equiv \neg EF\neg\phi$;
- $AF\phi \equiv \neg EG\neg\phi$;
- $A[\phi U\theta] \equiv \neg E[\neg\theta U(\neg\phi \wedge \neg\theta)] \wedge \neg EG\neg\theta$;

A semântica de CTL é formalmente dada pela seguinte definição [CGP99]: Sejam AP o conjunto das proposições atômicas, $p \in AP$, o modelo $\mathcal{M}$ formalmente dado pela estrutura de Kripke $\mathcal{M} = (S, R, I, L)$, $s \in S$ e ϕ e ψ fórmulas CTL. A relação satisfaz, denotada por $\models$, é definida por:

$s_0 \models p \quad \Leftrightarrow p \in L(s_0)$

$s_0 \models \neg\phi \quad \Leftrightarrow s_0 \models \neg\phi$

$s_0 \models \phi \wedge \psi \quad \Leftrightarrow (s_0 \models \phi)\ and\ (s_0 \models \psi)$

$s_0 \models EX\phi \quad \Leftrightarrow$ para algum estado t tal que $(s_0, t) \in R$, $t \models \phi$

$s_0 \models E[\phi U\psi] \quad \Leftrightarrow \exists\sigma \in Caminhos(s_0),\ (\exists j \geq 0, (\sigma^j \models \psi \wedge (\forall 0 \leq k < j,\ \sigma^k \models \phi))$

$s_0 \models EG\phi \quad \Leftrightarrow\ \exists\sigma\ \in\ Caminhos(s_0)$ tal que $\sigma^0, \sigma^1, ..., \sigma^n, \sigma^k,\ 0 \leq k \leq n \wedge (\sigma^n, \sigma^k) \in R, \forall i[0 \leq i \leq n, \sigma^i \models \phi]$

O conceito formal de caminhos é dado da seguinte maneira: Um caminho em $\mathcal{M}$ é uma seqüência infinita de estados $s_0,\ s_1,\ s_2,\ ...$ tal que $s_0 \in I$ e $(s_i, s_{i+1} \in R$ para todo $i \geq 0)$.

Informalmente, um caminho é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. $\sigma[i]$ denota o (i+1)-ésimo estado de σ e σ^i representa o sufixo de σ obtido pela remoção do(s) i-primeiro(s) estados de σ. A função $Caminhos(s)$ determina todos os possíveis caminhos da estrutura $\mathcal{M}$ que se iniciam no estado s.

2.3.2 Verificação de Modelos CTL

A verificação de modelo CTL [CGP99] baseia-se na verificação de espaços de estados de representação ramificada. O processo de verificação de uma fórmula ϕ reduz-se a um problema de busca exaustiva, no qual o verificador de modelos percorre todo o espaço de estados em busca dos estados que satisfazem ϕ. Para possibilitar esta busca, os estados do modelo devem ser rotulados com as sub-fórmulas de ϕ que são válidas nestes estados.

O conjunto de sub-fórmulas de ϕ é denotado por $Sub(\phi)$ e definido de forma indutiva da seguinte maneira: Sejam AP o conjunto das proposições atômicas, $p \in AP$, ϕ e ψ fórmulas CTL, então:

$$\begin{aligned} Sub(p) &= \{p\} \\ Sub(\neg\phi) &= Sub(\phi) \cup \{\neg\phi\} \\ Sub(\phi \wedge \psi) &= Sub(\phi) \cup Sub(\psi) \cup \{\phi \wedge \psi\} \\ Sub(EX\phi) &= Sub(\phi) \cup \{EX\phi\} \\ Sub(E[\phi U \psi]) &= Sub(\phi) \cup Sub(\psi) \cup \{E[\phi U \psi]\} \\ Sub(EG\phi) &= Sub(\phi) \cup \{EG\phi\} \end{aligned}$$

Assim, se um determinado $s \in S$ satisfaz ϕ: $\mathcal{M}, s \models \phi$, então s é rotulado com ϕ. Formalmente: $\mathcal{M}, s \models \phi \iff \phi \in L(s)$.

2.4 Testes e Verificação de Modelos

Uma linha de pesquisa muito ativa atualmente é a união da técnica de verificação de modelos aos testes. Durante muito tempo pensou-se que com o advento de técnicas de verificação de modelos, a necessidade de execução de testes para validação de sistemas passaria a ser nula. Porém, a verificação de propriedades da especificação não garante a correção do respectivo código produzido, uma vez que o último é obtido, predominantemente, de maneira informal, podendo haver ruídos no entendimento do desenvolvedor do sistema. Além da reconhecida necessidade de testes para validação da implementação, há também o reconhecimento de questões chaves no que diz respeito à correção do sistema operacional que dará suporte ao

software, bem como do compilador utilizado e do próprio verificador de modelos utilizado na verificação do modelo do sistema [Tre99], fato que reforça a necessidade da realização de testes inclusive na validação do modelo.

Diante do reconhecimento da característica complementar das técnicas de verificação e teste formais, esforços têm sido despendidos no intuito de integrá-las, tornando o processo de depuração do *software* mais eficiente. Assim, técnicas de verificação de modelos têm sido aplicadas ao processo de teste através de métodos que têm sido propostos [PEAM98; GH99; JCE96; JJ04; dVT98]. Alguns métodos se propõem a extrair casos de testes através da conversão direta de contra-exemplos gerados por verificadores de modelos [PEAM98; GH99; JCE96], no entanto sem se basear em uma teoria sólida de testes. Estas técnicas não são apropriadas para sistemas não-deterministas [JJ04], uma vez que tendem a produzir casos de teste que exercitem apenas uma alternativa de execução dentre várias, sendo portanto, ineficientes e não exaustivos, provendo baixa cobertura. Outros métodos baseiam-se em teoria de teste formal [JJ04; dVT98], e, através da adaptação de algoritmos e ferramentas de verificação de modelos, provêem processo exclusivo para a geração de casos de testes. A ferramenta apresentada em [JJ04], chamada TGV, utiliza o conceito de objetivo de testes para a geração de casos de teste. Esta ferramenta será utilizada no desenvolvimento do estudo de caso desta dissertação e será apresentada na Seção 2.5.

Em [FMP04], é apresentado procedimento e estrutura algorítmica para a geração automática de casos de teste a partir de propriedades em lógica temporal linear. O procedimento se baseia na especificação, possivelmente parcial, do sistema em um tipo especial de sistemas de transições rotuladas (LTS) chamado IOLTS. Especifica-se para a representação da propriedade temporal um observador, que é um reconhecedor de linguagens na forma de um tipo especial de máquinas de estados finitos denominado autômato de Rabin [Rab72]. A geração dos casos de teste dá-se por meio de um produto assimétrico entre especificação e observador, com posterior extração dos caminhos que representam os casos de teste através de procedimento definido através de heurísticas que definem os caminhos "mais promissores". A especificação parcial do sistema permite maior flexibilidade na geração e execução dos casos de teste em relação à teoria *ioco*, apresentada apresentada em [Tre96], que exige uma especificação exaustiva do sistema para não permitir a emissão de vereditos negativos devido à não especificação de comportamentos.

O problema da explosão do espaço de estados [CGP99] causados por técnicas que envolvem a enumeração dos estados de um sistema, como técnicas baseadas em LTS (e.g. [JJ04; FMP04; Tre96]), tem levado a busca por alternativas para a representação dos modelos. Uma linha de pesquisa que tem concentrado esforços é a geração simbólica de casos de teste (e.g. [CJRZ02; FTW05]). Em [CJRZ02] é apresentada uma ferramenta, chamada STG, para a geração de casos de teste simbólicos para sistemas reativos. O modelo no qual se baseiam as especificações simbólicas é o sistema de transições simbólicas de entrada e saída (IOSTS) [RdBJ00]. Como em [JJ04], tal ferramenta baseia-se na seleção de casos de teste através da especificação de objetivos de teste, no entanto especificados também em IOSTS. Já em [FTW05] os autores propõem um algoritmo para a geração de casos de teste baseada na teoria apresentada em [Tre96]. No entanto, tal teoria foi adaptada para se adequar aos modelos simbólicos propostos, chamados sistemas de transições simbólicas (STS).

2.4.1 Relação entre Objetivos de Teste e Propriedades de Verificação de Modelos

O problema da verificação de modelos é definido em [CGP99] da seguinte maneira: Dada uma estrutura de Kripke M que representa um sistema concorrente de estados finitos e uma fórmula de lógica temporal f expressando uma propriedade, identificar o conjunto de estados S da estrutura que satisfazem f. Formalmente: $S = \{s \in S \mid M, s \models f\}$.

Considerando um modelo de uma IUT, $i_{IUT} \in MODS$, na forma de uma estrutura de Kripke m_{IUT} como uma representação finita dos estados da IUT, podemos estabelecer que para um determinado objetivo de teste e haverá um conjunto de estados em m_{IUT} em que e é satisfeito, possibilitando à i_{IUT} revelar tal objetivo de teste. Assim, de forma análoga ao problema da verificação de modelos, podemos estabelecer que a dada estrutura de Kripke m_{IUT} e um objetivo de teste e, devemos identificar o conjunto de estados S da estrutura m_{IUT} que satisfazem e. Assumindo que podemos especificar um objetivo de teste e por meio de uma fórmula de lógica temporal f, estabelecemos que:

$$i_{IUT} \textbf{ rev } e \iff \exists s \in S : m_{IUT}, s \models f \tag{2.10}$$

Para uma implementação IUT capaz de exibir o comportamento especificado por um objetivo de teste e por intermédio de um conjunto de casos de teste T_e baseado em e, dado que

a relação de implementação m_{IUT}/i_{IUT} **imp** $spec$ é satisfeita, se o seu modelo satisfaz o objetivo de teste e, a IUT exibe e (*cf.* propriedade 2.6).

2.5 Seleção de Casos de Teste com TGV

TGV (Test Generation with Verification technology) [JJ04] é uma ferramenta de suporte a testes de caixa preta de conformidade que provê síntese automática de casos de teste para sistemas reativos não-deterministas. Com síntese de casos de teste baseada em técnicas de verificação de modelos, tais como produto síncrono, verificação *on-the-fly* e algoritmos de busca, esta ferramenta foi utilizada com sucesso em experimentos da indústria [FJJV97].

TGV baseia-se na teoria *ioco* apresentada em [Tre96]. Esta teoria define uma relação de conformidade entre implementação e especificação, na qual uma IUT é ioco-correta em relação a sua especificação se (i) a IUT nunca produz uma saída que não poderia ser produzida pela especificação na mesma situação (i.e. após a mesma seqüência de entradas e saídas), ou (ii) a IUT pode tornar-se incapaz de evoluir (e.g. em *deadlock*) se a especificação também o pode. Tal incapacidade de evoluir é denominada quiescência (o termo em inglês *quiescence*), a qual define a ausência de saídas pela IUT. Quiescência pode ser observada por temporizadores.

Como entrada, TGV recebe um modelo do comportamento do sistema e um objetivo de teste, ambos fornecidos como uma variante de sistemas de transições rotuladas (LTS), chamado sistemas de transições rotuladas de entrada e saída (IOLTS). Um IOLTS provê distinção entre os eventos do sistema que são controláveis e aqueles que são observáveis pelo ambiente, i.e. entradas e saídas, respectivamente. Ações internas do sistemas também podem ser representadas. A Figura 2.2 mostra um exemplo de um sistema de transições rotuladas de entrada e saída utilizada pelo TGV. As ações de entrada e saída são definidas através dos rótulos "*INPUT*" e "*OUTPUT*", respectivamente.

Formalmente, um IOLTS é uma tupla $M = (Q, A, \longrightarrow, q_0)$, onde Q é um conjunto finito não vazio de estados, A é o alfabeto de ações, $\longrightarrow \subseteq Q \times A \times Q$ é a relação de transição e $q_0 \in Q$ é o estado inicial. O alfabeto de ações A é particionado em três subconjuntos: $A = A_I \cup A_O \cup I$. A_I é o conjunto das entradas, A_O é o conjunto das saídas e I é o conjunto de ações internas do sistema.

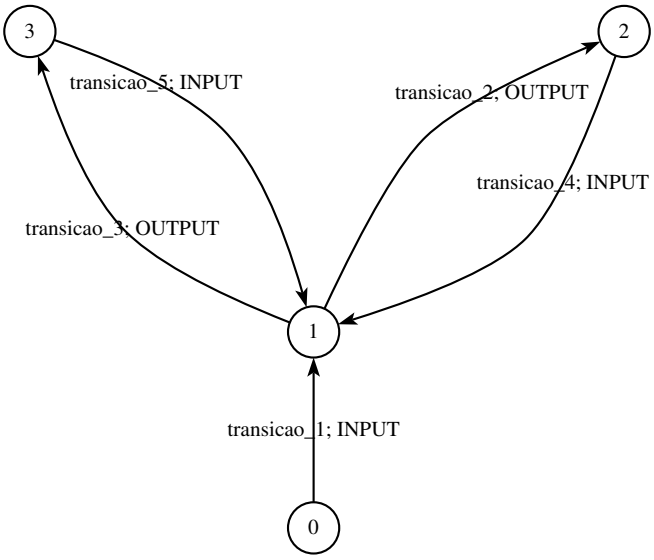


Figura 2.2: Exemplo de IOLTS utilizado pelo TGV

Os objetivos de teste são equipados com estados especiais de aceitação e rejeição (resp. *accept* e *refuse*). Estes estados devem ser utilizados para guiar processos de seleção de casos de teste. Aceitação representa seqüências que devem ser selecionadas para os casos de teste, enquanto que rejeição representa as que não devem originar casos de teste. O TGV utiliza as informações destes estados para otimizar o produto síncrono dos artefatos utilizados na entrada para a seleção de casos de teste. A Figura 2.3 mostra um exemplo de objetivo de teste utilizado pelo TGV. Este objetivo de teste define que a seqüência de transições aceitas para a execução dos casos de teste deve ter em sua composição a transição "transicao_1" seguida da transição "transicao_2". Execuções em que a transição "transicao_3" ocorra entre "transicao_1" e "transicao_2" não devem gerar casos de teste. De acordo com a relação *ioco*, transições não especificadas devem ser rejeitadas pelos casos de teste, não sendo assim, especificadas.

Os objetivos de teste definem as funcionalidades a serem testadas. A ferramenta aplica uma série de algoritmos sobre os modelos do sistema e objetivos de teste com o fim de obter os casos de teste e os respectivos vereditos. Tais algoritmos são originalmente aplicados ao problema da verificação de modelos (e.g. produto síncrono), entretanto, sofreram adaptações para possibilitar sua aplicação à geração automática de casos de teste. A Figura 2.4 ([MSM05]) mostra a arquitetura do TGV.

Como um primeiro passo do processo de seleção dos casos de teste, o TGV realiza um

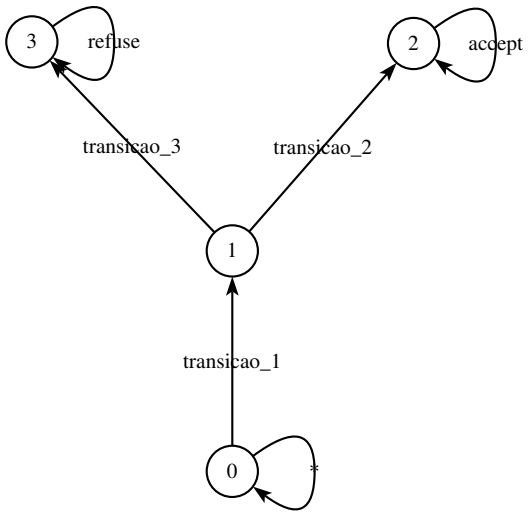


Figura 2.3: Exemplo de objetivo de teste

produto síncrono entre a especificação S do sistema e o objetivo de teste TP, dando origem a um terceiro IOLTS, chamado SP. Este produto soluciona o problema de identificar os comportamentos de S aceitos ou rejeitados por TP. Como a relação ioco pode ser estabelecida com base nos comportamentos observáveis, TGV define um outro IOLTS que contém apenas as ações observáveis do sistema (i.e. apenas ações de entrada e saída) e provê meios de definir de forma explícita e observável os estados quiescentes, em um processo denominado determinação. Este IOLS é chamado de SP^{vis}, como mostrado na Figura 2.4.

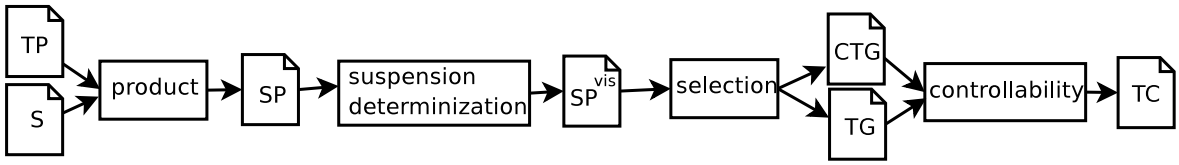


Figura 2.4: Arquitetura do TGV

O próximo passo é a construção de um novo IOLTS, denominado CTG. O CTG representa um grafo contendo todos os casos de teste referentes ao objetivo de teste fornecido. Este grafo é construído por meio da extração dos comportamentos aceitos (i.e. identificados através dos estados *accept*) e pela inversão de entradas e saídas. Esta inversão de entradas e saídas é realizada para prover a ligação entre a implementação e os casos de teste. As saídas da IUT devem ser utilizadas como entradas dos casos de teste e vice-versa. O grafo deve ser completado com entradas em todos os estados nos quais uma entrada é possível, definindo-se também os vereditos por conjuntos **Pass**, **Inconc** e **Fail**. Os estados **Pass** são definidos pelos

estados *accept* do objetivo de teste. Os estados *Inconc* são estados que não levam a estados *accept* para nenhum caminho, entretanto, é um estado sucessor de um estado que pertence a pelo menos um caminho que leve a um estado *accept*. Os estados *Fail* compõem novos estados que são alcançados quando a relação ioco é violada.

Concluindo o processo, conflitos de controlabilidade são eliminados para obter os casos de teste. Como mostrado na Figura 2.5(a), os problemas de controlabilidade são representados pela presença de opções entre saídas ou entre entradas e saídas em alguns estados. Tais conflitos podem ser solucionados com a supressão de algumas transições. Neste caso, ou uma saída é mantida e todas as outras entradas e saídas são suprimidas, ou todas as entradas são mantidas e as saídas suprimidas (Figura 2.5(a)). Opcionalmente, um grafo representando um caso de teste TG (Figura 2.4) pode ser construído durante a fase de seleção através da solução *on-the-fly* dos problemas de controlabilidade.

(a) Configurações não permitidas (b) Configurações permitidas

Figura 2.5: Conflitos de controlabilidade

Os casos de teste abstratos gerados pelo TGV são efetivos e exaustivos em relação à relação ioco. Entretanto, desde que o CTG gerado pode conter infinitos casos de teste e a ferramenta não provê um mecanismo de seleção dos mesmos, a efetividade e a exaustividade dos casos de teste tornam-se difíceis de serem alcançadas na prática. A ferramenta também não provê mecanismos para implementação e execução dos casos de teste. Entretanto, TGV faz parte de um projeto mais amplo, o qual é denominado AGEDIS [HN04]. Este projeto tem como objetivo o desenvolvimento de uma metodologia e de ferramentas para a automação da geração de casos de teste baseados em modelos, bem como sua execução para sistemas distribuídos.

2.6 Considerações Finais

Este capítulo apresentou a base teórica necessária para o entendimento deste trabalho. Foram discutidos os conceitos de testes, testes formais, verificação de modelos e a viabilidade da aplicação de técnicas de verificação de modelos aos testes, bem como o estado da arte na área. Os trabalhos apresentados demonstram a viabilidade da união das duas técnicas, mostrando uma evolução tal, que os testes formais tornaram-se uma realidade mesmo em projetos da indústria, como discutido em [FJJV97].

Dentre as ferramentas apresentadas durante a discussão do estado da arte, a ferramenta de geração de casos de teste TGV foi apresentada com maiores detalhes, na Seção 2.5. Esta ferramenta é utilizada no estudo de caso, apresentado no Capítulo 5, para a geração dos casos de teste baseados nos objetivos de teste gerados pela técnica proposta nesta dissertação.

Capítulo 3

Geração de Objetivos de Teste

Este Capítulo apresenta a técnica de geração de objetivos de teste proposta nesta dissertação. As etapas definidas para o processo de geração são apresentadas com suas justificativas, seguindo-se da realização de uma análise sobre o funcionamento da técnica com base nos tipos de fórmula CTL. Em seguida, os algoritmos definidos pela técnica, nos quais uma implementação deve guia-se, são delineados. Por fim, são apresentadas considerações finais, com uma breve discussão incluindo restrições e aplicações de propriedades de lógica temporal linear para a técnica.

3.1 Visão Geral da Técnica

A verificação de propriedades por meio de técnicas de verificação de modelos[1] tem obtido sucesso em relação a sistemas concorrentes [Kat99; CGP99]. No entanto, o mesmo rigor nem sempre é aplicado em relação à geração de casos de testes para a implementação, deixando grande discrepância entre os dois processos, possibilitando falhas na implementação em pontos que o modelo foi bem sucedido.

Com base na similaridade entre as propriedades utilizadas nas técnicas de verificação e os objetivos de teste utilizados nos testes de conformidade, a técnica proposta visa gerar objetivos de teste a partir de tais propriedades especificadas em lógica temporal [CGP99]. A justificativa está no fato de que o que se deseja verificar em um modelo, deseja-se testar em uma implementação, definindo a correspondência entre os objetivos de teste a serem gerados

[1] A partir deste ponto podemos nos referir apenas como técnicas de verificação.

e as fórmulas de lógica temporal definidas para a verificação. Cria-se assim, uma ligação entre os processos de verificação e testes, diminuindo o intervalo existente entre ambos. O rigor aplicado à verificação do modelo pode ser então aplicado de maneira similar aos testes.

Para alcançar este objetivo, a técnica proposta consiste em efetuar análises sobre espaços de estados como a técnica de verificação de modelos o faz, no entanto, com o objetivo de obter informações sobre o modelo a fim de especificar objetivos de teste. O processo de verificação de modelos é adaptado e, ao invés de realizar a análise sobre correção, realiza a extração de informações com base em uma propriedade especificada como uma fórmula de lógica temporal. Estas informações são obtidas por meio da extração de caminhos do modelo relacionados à fórmula (i.e. exemplos e contra-exemplos). Estes caminhos serão utilizados como informação base para guiar a geração do objetivo de teste correspondente à propriedade. Uma vez obtido com informações do modelo, tal objetivo de teste deve, assim, refletir a propriedade especificada em relação ao mesmo de maneira específica. A Figura 3.1 ilustra o processo de geração da técnica proposta.

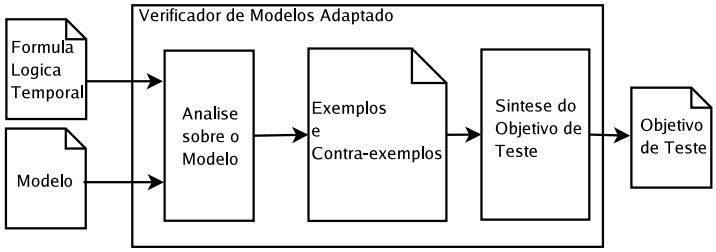


Figura 3.1: Processo de geração do objetivo de teste

Para a aplicação desta técnica pressupõe-se a verificação prévia do modelo em relação à propriedade que dará origem ao objetivo de teste, devendo esta ser satisfeita pelo modelo. Os objetivos gerados são definidos na forma de sistemas de transições rotuladas (LTS).

Formalmente, um objetivo de teste é uma tupla $M = (Q, A, \longrightarrow, q_0)$, onde Q é um conjunto finito não vazio de estados, A é o alfabeto de ações, $\longrightarrow \subseteq Q \times A \times Q$ é a relação de transição e $q_0 \in Q$ é o estado inicial. São equipados com estados especiais de aceitação e rejeição (resp. *accept* e *refuse*). Estes estados devem ser utilizados para guiar processos de seleção de casos de teste. Aceitação representa seqüências que devem ser selecionadas para os casos de teste, enquanto que rejeição representa as que não devem originar casos de teste.

Como apresentado no Capítulo 2, o teste de conformidade é um tipo de teste de caixa

preta, no qual apenas o comportamento do *software* é conhecido, sendo observado através de sua interface com o ambiente. Esta interface é definida através de pontos de observação e controle, nos quais é possível controlar a execução do software através de entradas e observar sua resposta através de saídas ao ambiente. Assim, por basear-se na especificação de ações (i.e. entradas e saídas) do sistema, LTS constitui um formalismo adequado para o processo de testes de conformidade, sendo utilizado como base em diversas técnicas (e.g. [JJ04; Tre96; CJRZ02; FTW05]).

3.2 Análise sobre o Modelo

A especificação de objetivos de teste é guiada, geralmente, por análises sobre modelos. Entretanto, a análise sobre modelos formais tende a ser de alta complexidade e propensa a erros. As representações de modelos formais são dadas, em geral, em linguagem pouco natural e intuitiva ao ser humano (e.g. LTS). O tamanho dos modelos de sistemas reativos com características de distribuição e concorrência tende a ser muito grande, dificultando sobremaneira a análise humana.

A técnica de verificação de modelos provê mecanismos eficientes para análise de modelos de sistemas reativos com características de distribuição e concorrência [CGP99]. Fundamentando-se na adaptação da verificação de modelos para a realização de análises sobre os modelos, a técnica aqui proposta tem como importante meta usufruir destes mecanismos, de modo a possibilitar a realização de análises exaustivas para uma rigorosa coleta de dados para o processo de geração dos objetivos de teste.

3.3 Obtenção de Exemplos e Contra-exemplos

A adaptação no processo de verificação de modelos dá-se com a mudança dos algoritmos de busca [CVWY92] de um verificador, que, ao invés de extrair apenas um exemplo ou contra-exemplo, passará a extrair maior número de ambos, exemplos e contra-exemplos, como mostrado na Figura 3.1. Assim, permite-se uma maior amostragem do espaço de estados para a obtenção de informações suficientes para a elaboração do objetivo de teste. A extração de contra-exemplos, caso existam, se justifica por dois principais motivos:

- O primeiro baseia-se na mudança de formalismos para representação entre o modelo e o objetivo de teste a ser produzido com base no modelo. As propriedades analisadas sobre os modelos, que são representados através de estruturas de kripke, que rotulam estados e transições, devem ser refletidas fielmente através de estruturas que rotulam apenas transições (i.e. LTS). Como o artefato resultante em LTS (o objetivo de teste) ainda provê abstrações na seqüência de ações (comumente representadas através de "*"), a restrição em relação a estados resultantes para o sistema tem seu rigor diminuído. Distorções podem assim, ser causadas através de transições indesejáveis que levem a implementação sob teste a estados que violem nossas propriedades. Os contra-exemplos são utilizados nestes casos para fornecer restrições e manter equivalência entre as representações LTS e de kripke.

- O segundo motivo baseia-se no não-determinismo dos sistemas especificados. Ao especificar um objetivo de teste desejamos focar os casos de teste no intuito de que estes venham a fazer com que a implementação exiba um determinado comportamento. Porém, por uma questão de não-determinismo, a implementação pode, ao receber um estímulo, exibir um outro comportamento diferente do especificado pelo caso de teste. Este pode ser um comportamento correto especificado para a implementação, no entanto, não condizente ao objetivo de teste inicialmente especificado e fora do seu foco. Os casos de teste, por sua vez, ao não alcançarem o seu objetivo não devem fornecer veredito sobre a correção de tal comportamento, uma vez que este não diz respeito ao comportamento especificado como objetivo de teste. Desta forma, os contra-exemplos são utilizados para extrair informação acerca de tais comportamentos, corretos, porém não focados, para que sejam especificados nos objetivos de teste, evitando que os respectivos casos de teste forneçam falsos veredictos.

Durante a busca por caminhos, o verificador deve realizar uma triagem sobre estes. Para cada caminho que satisfaz a propriedade especificada um exemplo é selecionado. Para cada caminho que viola a propriedade durante a busca pelos exemplos, um contra-exemplo é selecionado. Contra-exemplos podem ser obtidos a partir de fórmulas que contenham conectivos baseados em quantificadores existenciais (e.g. EF, EU e EX).

3.4 Processo de Síntese dos Objetivos de Teste

Sobre os exemplos e (possivelmente) contra-exemplos obtidos por meio da busca sobre o modelo, devemos prover mecanismo para a geração do objetivo de teste correspondente. O objetivo de teste deve, neste caso, reunir toda a informação contida nos caminhos. No entanto, esta informação deve ser tão compacta e abstrata quanto possível, contida em apenas um grafo.

3.4.1 Abstração sobre Exemplos e Contra-exemplos

O mecanismo proposto para a geração de um grafo abstrato baseia-se na realização de uma segunda triagem, porém, sobre as transições contidas nos exemplos e contra-exemplos. As transições são analisadas, sendo classificadas segundo sua relevância no alcance da propriedade definida. Para tornar possível esta análise sobre transições, optamos por fazer representações mais abstratas dos caminhos extraídos do modelo. Estas representações baseiam-se na simplificação da representação dos estados. O modelo utilizado para tal representação baseia-se em máquina de estados finitos, formalmente definido pela tupla $(Q, \Sigma, \delta, q_0, F)$, onde Q é um conjunto finito e não vazio de estados, Σ um conjunto finito de símbolos que compõem o alfabeto aceito pela máquina, $q_0 \in Q$ o estado inicial, e $F \subseteq Q$ o conjunto de estados finais, denominados estados de aceitação. Caminhos que definem exemplos e contra-exemplos compõem seqüências bem definidas de estados, caracterizando seu determinismo. Por serem deterministas, suas transições devem ser definidas por meio de uma função, aqui denominada δ. Assim, a função de transição δ é definida como $\delta : Q \times \Sigma \longrightarrow Q$.

O rótulo sobre os estados baseia-se na relação de satisfação entre o estado e a fórmula. Cada estado de um caminho extraído deve receber como rótulo, o rótulo da proposição atômica (integrante da fórmula) à qual satisfaz. O último estado da cadeia que compõe um caminho deve ser rotulado apenas como estado final, não recebendo o rótulo da proposição que satisfaz (ou que viola, no caso de ser um contra-exemplo). A Figura 3.2 mostra um exemplo extraído, com sua representação de estados simplificada. Os estados não finais do exemplo são rotulados como p e q, relativos às proposições às quais satisfazem, e o estado final, rotulado como aceitação, com o rótulo *accept*.

A Figura 3.3 mostra um grafo referente a um contra-exemplo, com estados não finais

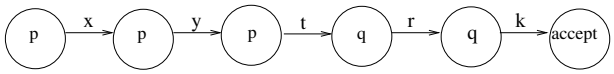


Figura 3.2: Representação simplificada de um exemplo

rotulados por p e q, relativos às proprosições que satisfazem, e o estado final, com rótulo *refuse*.

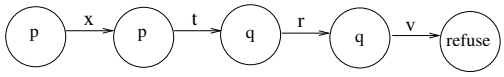


Figura 3.3: Representação simplificada de um contra-exemplo

Sobre estas representações, o algorítmo de síntese deve realizar alguns procedimentos de análise e abstrações, resumidos a seguir. Estes procedimentos são explicados com maiores detalhes, com o auxílio de um exemplo, na Subseção 3.4.2:

Classificação das transições que compõem o caminho Esta classificação baseia-se na mudança de estados da representação. As transições que causam mudança entre estados, ou seja, que relacionam estados de rótulos diferentes, são classificadas como relevantes. As transições que relacionam estados com rótulos iguais são classificadas como irrelevantes. As transições irrelevantes podem ser abstraídas na construção do objetivo de teste. Tais abstrações são representadas por auto-laços rotulados com "*" (asterisco). Estes auto-laços são denominados transições asterisco, transições estrela ou *-transições. As transições asterisco representam qualquer transição, excetuando as transições que conectam o estado que a contém e outro estado de rótulo diferente. Por simplicidade, durante a construção do objetivo de teste, as transições asterisco podem ser acrescentadas a cada estado do LTS resultante. Para tanto, supõe-se uma representação completa do modelo sob análise no processo de geração, para que não haja falhas na abstração representada por estas transições.

Detecção de transições interdependentes Há situações em que a mudança de estados está relacionada a uma seqüência de transições interdependentes. Esta interdependência é caracterizada pela presença de um conjunto de transições que causam mudança de estado de maneira alternada, nos caminhos extraídos. Desta forma, as transições interdependentes devem ser detectadas através da interseção entre os dois conjuntos criados.

A Figura 3.4 mostra um exemplo de interdependência de transições. Durante a realização do processo de análise descrito acima, para triagem das transições, as transições 'f' e 'g' devem ser classificadas como relevantes e também como irrelevantes, pertencendo aos dois conjuntos de transições. Os conjuntos definidos para as classificações de relevantes e irrelevantes são $L = \{f, g\}$ e $N = \{b, c, d, f, g\}$, respectivamente.

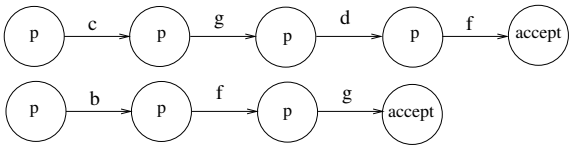


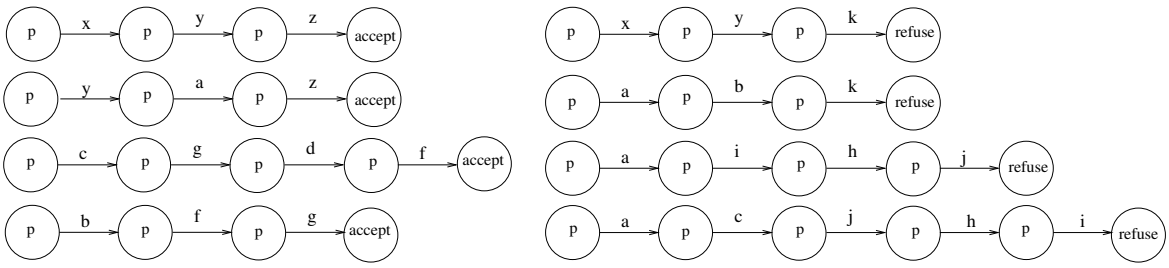
Figura 3.4: Exemplo de interdependência entre transições

Síntese do Objetivo de Teste A construção do objetivo de teste deve considerar a informação das situações previstas por exemplos e contra-exemplos. Assim, as transições classificadas como relevantes de ambos os conjuntos de caminhos devem compor o objetivo de teste correspondente. Transições provenientes de exemplos devem compor seqüências que levem ao estado de aceitação do objetivo de teste, devendo ser selecionadas como casos de teste em procedimento de geração de casos de teste posterior. As transições coletadas a partir de contra-exemplos devem compor seqüências que levem ao estado de rejeição do objetivo de teste. As transições interdependentes devem ser representadas adequadamente, devendo prover caminhos que descrevem corretamente as seqüências das transições no objetivo de teste.

3.4.2 Procedimento de Síntese de Objetivos de Teste

Como dito anteriormente, a idéia do algoritmo baseia-se na mudança de estados. Assim, deve-se classificar as transições encontradas nos exemplos em 2 conjuntos: O conjunto L formado pelas transições que causam mudança de estado e o conjunto N pelas transições que não causam mudança de estado. A Figura 3.5(a) mostra um conjunto de exemplos que representam uma dada fórmula e a Figura 3.5(b) apresenta contra-exemplos. A partir destes caminhos criaremos os conjuntos L e N.

Diante da análise feita sobre a mudança de estados, o conjunto L das transições que causam mudança de estado é $L = \{z, f, g, i, j, k\}$ e o conjunto N das transições que não causam



(a) Exemplos (b) Contra-exemplos

Figura 3.5: Caminhos relativos à fórmula

mudança de estado é $N = \{x, y, a, b, c, d, f, g, i, j, h\}$. O conjunto L é dividido em subconjuntos para cada par de conexão entre dois estados quaisquer. Ou seja, cada subconjunto contém as transições que interconectam dois estados diferentes. A fórmula analisada origina subconjuntos de transições que interconectam os estados "p" e "*accept*" e "p" e "*refuse*". Os respectivos subconjuntos são $L_{p\_accept} = \{z, f, g\}$ e $L_{p\_refuse} = \{i, j, k\}$.

Podemos perceber que há transições comuns aos dois conjuntos (L e N), causando a mudança de estado em alguns exemplos e não causando em outros. Concluímos a partir de tal fato que tais transições não causam mudança de estado de maneira independente, mas em conjunto. A partir da interseção entre os dois conjuntos podemos agrupar tais transições, que denominamos transições interdependentes. Para o exemplo acima, as transições 'f' e 'g' ocorreram de forma conjunta e em ordens alternadas, sendo complementares para a mudança de estado. De maneira idêntica acontece com as transições 'i' e 'j'. Já as transições 'z' e 'k' são suficientes para causar de forma isolada as mudanças de estado, sendo chamadas de independentes.

Para agrupar as transições interdependentes, um conjunto de interseção entre cada subconjunto de L é criado. Assim, os conjuntos $L_{p\_accept} \cap N = I_{p\_accept} = \{f, g\}$ e $L_{p\_refuse} \cap N = I_{p\_refuse} = \{i, j\}$ são definidos. As transições que ocorrem em conjunto são então, agrupadas. A combinação de ordem entre elas, presente nos exemplos e contra-exemplos, deve ser analisada para a construção do grafo do objetivo de teste. Desta forma, tuplas são criadas para computar as combinações entre as transições interdependentes. As tuplas $SUB_{If_{p\_accept}} = [g, f]$, $SUB_{Ig_{p\_accept}} = [f, g]$, $SUB_{Ii_{p\_refuse}} = [j, i]$ e $SUB_{Ij_{p\_refuse}} = [i, j]$ são criadas. Cada uma destas tuplas deverá originar um caminho entre os estados "p" e "*accept*" ou "p" e "*refuse*". Para tanto, entre cada par de transições

um estado intermediário deve ser criado a fim de compor o caminho entre os dois estados previamente definidos.

Após a definição das transições interdependentes, as transições independentes são então separadas para interconectarem diretamente os estados. Assim, os subconjuntos $S_{p\_accept} = L_{p\_accept} - I_{p\_accept} = \{z\}$ e $S_{p\_refuse} = L_{p\_refuse} - I_{p\_refuse} = \{k\}$ são criados.

Para a construção do objetivo de teste resultante, cada subconjunto que descreve as seqüências de transições interdependentes e independentes entre os estados é utilizado na definição dos caminhos aos estados *accept* e *refuse* do mesmo. O grafo que representa o objetivo de teste, originado com base nos subconjuntos e tuplas, acima é mostrado na Figura 3.6.

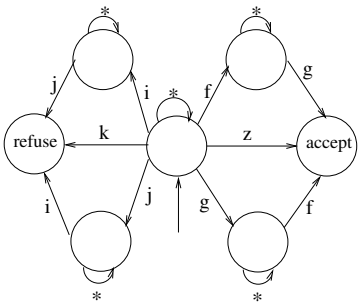


Figura 3.6: Objetivo de teste resultante

3.5 Objetivo de Teste e os Tipos de Fórmulas

Realizado com base nas informações contidas no modelo, o processo de geração dará origem a objetivos de teste com formatos peculiares, voltados ao respectivo modelo. Desta maneira, o formato dos objetivos de teste a serem originados por uma determinada fórmula não podem ser preditos de forma precisa por um padrão, a exemplo do procedimento de conversão de fórmulas LTL em autômatos de Büchi [Büc62; Kat99]. No entanto, a estrutura de exemplos e contra-exemplos obtidos com base em fórmulas de lógica temporal pode ser estabelecida em relação à representação abstrata por nós estabelecida, baseada na relação de satisfação de cada estado.

A estrutura de um objetivo de teste varia basicamente sob influência da existência ou não de contra-exemplos (e.g. fórmulas dos tipos EU, EG e EX) e de transições interdependentes.

Fórmulas idênticas aplicadas a modelos diferentes de um mesmo sistema podem originar grafos de objetivos de teste distintos, uma vez que as diferenças entre estes modelos podem gerar exemplos e contra-exemplos com transições também distintas.

3.5.1 Fórmulas Baseadas em Quantificador Existencial

As fórmulas baseadas em quantificador existencial formam a base de todas as fórmulas CTL. Três classes são especialmente importantes por derivarem outras fórmulas e, por sua ampla utilização. São elas as fórmulas baseadas em EU, EG e EX.

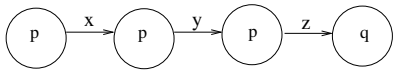
Fórmulas Baseadas em EU

Fórmulas do tipo $EU(p, q)$ produzem exemplos que seguem um determinado padrão estrutural, mostrado na Figura 3.7(a). Podemos perceber que, em relação à propriedade a ser testada, os exemplos obtidos são compostos por apenas 2 tipos de estados:

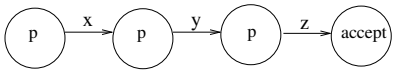
- O primeiro, que pode se repetir inúmeras vezes, compondo inúmeros estados nos quais a propriedade "p" for verdadeira, sendo chamado de estado "p". Neste caso, podemos observar que as transições que ocorrem entre os estados "p" não contribuem para a mudança de estados do ponto de vista da propriedade a ser testada. Assim, a princípio podem ser abstraídos de alguma forma. No entanto, para sistemas de transições rotuladas pode haver seqüências de transições necessárias para que haja mudança de estados, o que implica restrições.

- O segundo tipo de estado ocorre apenas uma vez, o estado em que a propriedade "q" é válida, sendo atingido após a ocorrência da transição 'z'. Este estado corresponde ao estado de aceitação da máquina de estados, ou seja, como estado final do exemplo extraído do modelo. A Figura 3.7(b) mostra um exemplo correspondente da representação abstrata utilizada para a geração dos objetivos de teste.

Os contra-exemplos refletem a violação de propriedades, que, para fórmulas do tipo $EU(p, q)$, significam a violação de "p" e "q". A Figura 3.8(a) mostra um exemplo de violação de uma fórmula $EU(p, q)$. O grafo representa um contra-exemplo em que o estado

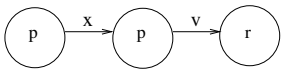


(a) Exemplo correspondente à fórmula $EU(p, q)$

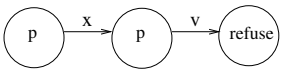


(b) Representação do exemplo da Figura 3.7(a)

Figura 3.7: Grafos de exemplos utilizados no processo



(a) Exemplo de violação de uma fórmula $EU(p, q)$



(b) Representação do contra-exemplo da Figura 3.8(a)

Figura 3.8: Grafos de contra-exemplos utilizados no processo

rotulado por "r" viola tanto "p", quanto "q". A Figura 3.8(b) mostra o grafo correspondente ao exemplo com a representação abstrata utilizada para a geração dos objetivos de teste.

Fórmulas do tipo $EF(q)$ por serem derivadas de fórmulas $EU(p, q)$, onde o valor de "p" é verdadeiro (*true*), possuem exemplos que seguem a mesma estrutura desta. No entanto, não originam contra-exemplos, uma vez que a proposição "p" de valor verdade *true*, nunca é violada pelos estados do modelo. Uma vez que pressupõe-se a validade da fórmula em relação ao modelo, a ausência de exemplos é descartada.

Fórmulas Baseadas em EG

O padrão de exemplos originados com base em fórmulas do tipo $EG(f)$ definem apenas um tipo de estado, como mostrado na Figura 3.9. Os algoritmos de busca implementados para estas fórmulas geralmente baseiam-se na detecção de laços. Desta forma, os exemplos podem ser limitados por meio da repetição de estados. O estado final de um exemplo é formado por um estado anteriormente visitado durante a busca e já incluído no caminho que compõe o exemplo.

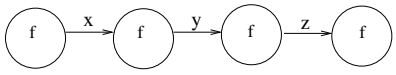


Figura 3.9: Exemplo correspondente à fórmula $EG(f)$

A formação de um exemplo para este tipo de fórmula, permite a criação de objetivos de teste que contenham laços e, assim, permitam a geração de casos de teste de execução

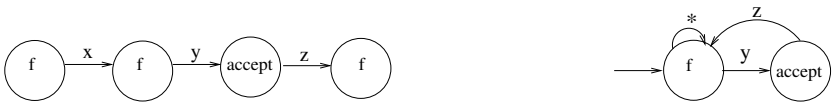
infinita. Execuções infinitas devem, nestes casos, ser tratadas pelas técnicas de geração e/ou execução de casos de teste, uma vez que execuções infinitas não têm valor prático para os testes.

A abordagem adotada neste trabalho define que o estado final de um caminho extraído do modelo corresponde ao último estado do caminho, sendo este rotulado como *accept* ou *refuse*. Desta maneira, a repetição de um estado em um exemplo, ilustrada através da Figura 3.10(a), não é representada como uma repetição de fato. O estado final possui rótulo diferenciado (Figura 3.10(b)), desta maneira, o laço é descaracterizado, não sendo propagado ao objetivo de teste.

(a) Exemplo original extraído (b) Representação simplificada

Figura 3.10: Caminhos relativos à fórmula $EG(f)$

Para possibilitar a geração de laços nos objetivos de teste, é necessário manter a representação dos mesmos na representação simplificada adotada inicialmente. No entanto, uma alteração se faz necessária. O estado final de um exemplo deixa de ser necessariamente o último estado do caminho extraído, mas sim, o último estado diferente que satisfaça a propriedade. Neste caso, para estas fórmulas, o penúltimo estado deve ser considerado como final e, a transição que parte deste caracterizará o laço para um estado antecessor. As Figuras 3.11(a) e 3.11(b) ilustram a representação de um exemplo baseada em tais alterações e um objetivo de teste originado a partir de tal exemplo, respectivamente. Apesar de uma adaptação do processo ser supostamente simples e demandar poucos esforços, sua realização merece atenção especial e realização dos devidos experimentos.

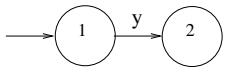


(a) Exemplo (b) Objetivo de teste resultante

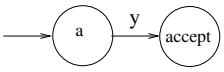
Figura 3.11: Representação alternativa de grafos baseados em $EG(f)$

Fórmulas Baseadas em EX

Os algoritmos de busca baseados neste tipo de fórmula realizam, geralmente, uma busca limitada apenas a estados imediatamente sucessores a um estado fornecido como "estado atual"[Hel97; VL93]. Desta forma, os caminhos extraídos devem possuir tamanho limitado a um estado além do estado pelo qual iniciou-se a busca. Para uma fórmula $EX(p)$, onde p é uma proposição atômica, os caminhos deverão conter apenas 2 estados em sua composição. Neste caso, um rótulo qualquer pode ser definido para o estado inicial, para sua definição de acordo com a representação simpliicada, definida previamente. As Figuras 3.12(a) e 3.12(b) mostram as composições de um exemplo e um objetivo de teste referentes a uma fórmula $EX(p)$, respectivamente.



(a) Exemplo original extraído



(b) Objetivo de teste resultante

Figura 3.12: Representação alternativa de grafos baseados em $EX(p)$

3.5.2 Fórmulas Baseadas em Quantificador Universal

Uma vez que as fórmulas a serem utilizadas na geração dos objetivos de teste devem ser satisfeitas no modelo, as fórmulas baseadas em quantificador universal, por envolverem todos os caminhos da árvore do espaço de estados, não originam contra-exemplos. A geração dos objetivos de teste a partir destas fórmulas devem assim, basear-se apenas em exemplos, devendo possuir como estados finais apenas estados de aceitação. A geração de casos de teste baseada nestes objetivos de teste deve ser rigorosa de forma a não permitir comportamentos que violem o comportamento por estes descrito.

3.5.3 Conectivos Mínimos da CTL

Os conectivos mínimos CTL de conjunção e disjunção, definidos por $\wedge$ e $\vee$, apresentam interesse prático para o teste quando utilizados em fórmulas comportas por quantificadores existenciais e/ou universais, definindo fórmulas compostas. Os caminhos extraídos do modelo com base nestas fórmulas seguem os padrões estruturais discutidos anteriormente, neste

capítulo. Estes caminhos devem, portanto, ser tratados de acordo com o tipo de fórmula ao qual correspondem.

De maneira análoga, o conectivo de negação NOT não define composições específicas de caminhos Entretanto, este conectivo tem grande importância, uma vez que é utilizado na definição das fórmulas baseadas em quantificador universal, como visto no Capítulo 2.

3.6 Algoritmo de Síntese

O algoritmo de síntese de objetivos de teste que será descrito nesta seção será subdividido para melhor entendimento. O Algoritmo 1 mostra o procedimento que deve ser efetuado durante a análise dos caminhos extraídos do modelo e triagem das transições destes caminhos.

A triagem das transições é realizada entre as linhas 2 e 10. As transições irrelevantes, que não causam mudança de estado, são adicionadas ao conjunto denominado N (linhas 3 e 4). As transições relevantes são armazenadas em conjuntos de acordo com as informações dos tipos de estados[2] que interconectam (linhas 6 a 9). Os estados interconectados são denominados estados de entrada e saída como estados antecessor e posterior à ocorrência da transição, respectivamente. Um conjunto deve ser criado para cada tipo de interconexão entre estados, assim, uma determinada transição relevante deverá ser armazenada em um conjunto cujo rótulo contém informações do estado de entrada e do estado de saída.

Com base nas transições classificadas como relevantes deverão ser criados os caminhos que comporão o grafo abstrato do objetivo de teste. Tais caminhos devem levar a estados finais, sendo estes de aceitação (i.e. *accept*) ou rejeição (i.e. *refuse*). A definição destes caminhos é realizada com base em uma terceira triagem baseada na existência ou não de interdependência entre transições na implicação de mudança de estados. Esta triagem é realizada entre as linhas 12 e 23. Para cada conjunto de interconexão entre dois estados quaisquer, as transições independentes e as interdependentes devem ser separadas (linhas 13 e 14). As interdependentes, como dito anteriormente, devem ser determinadas a partir da interseção com com o conjunto de transições irrelevantes (linha 13). As transições independentes causam a mudança de estado de maneira isolada.

As combinações de ordem entre as transições interdependentes são definidas entre as

[2]Cada rótulo define um tipo de estado.

Algoritmo 1 Análise sobre Exemplos e Contra-exemplos

```
1: for all e ∈ traces do
2:   for all t ∈ e do
3:     if ¬ changeState(t,e) then
4:       add(t,N)
5:     else
6:       i = inState(t,e)
7:       o = outState(t,e)
8:       add(t,L_io)
9:       add(L_io,L)
10:    end if
11:  end for
12:  for all L_io ∈ L do
13:    I_io = L_io ∩ N
14:    S_io = L_io − I_io
15:    add(S_io,S)
16:    add(I_io,I)
17:    for all t ∈ I_io do
18:      SUB_It_io = ⊘
19:      for all p ∈ predecessors(t) do
20:        add(p,SUB_It_io)
21:      end for
22:    end for
23:  end for
24: end for
```

linhas 17 e 22. Estas combinações devem formar diferentes caminhos entre dois estados interconectados pelas transições. Assim, a relação de predecessão entre as transições define a ordem entre as mesmas para cada caminho (linhas 19 a 21).

O Algoritmo 2 descreve a geração do objetivo de teste propriamente dita. Os conjuntos criados pelo Algoritmo 1 devem dar origem aos caminhos do objetivo de teste. Os caminhos entre dois estados quaisquer, interligados por transições independentes, são criados entre as linhas 3 e 7.

Para a criação dos caminhos que envolvem transições interdependentes (linhas 8 a 24) estados intermediários são criados para representar as seqüências entre as transições (linha 12). A possibilidade de combinações que envolvam relações de predecessão entre transições dependentes e independentes devem ser detectadas entre as linhas 13 e 17. Havendo tal relação, detectada pelo procedimento *predecessors* (linha 14), a abreviação entre o estado intermediário e o estado de saída deve ser realizada por meio da transição independente (linha 15).

Algoritmo 2 Síntese do Objetivo de Teste

```
1:  TestPurpose = ∅
2:  x = 0
3:  for all S_io ∈ S do
4:     for all t ∈ S_io do
5:        add((i,t,o), TestPurpose)
6:     end for
7:  end for
8:  for all I_io ∈ I do
9:     for all t ∈ I_io do
10:       for all s ∈ SUB_{It_io} do
11:          if s ≠ t then
12:             add((i,s,i_x),TestPurpose)
13:             for all j ∈ S_io do
14:                if s ∈ predecessors(j) then
15:                   add((i_x,j,o),TestPurpose)
16:                end if
17:             end for
18:          else
19:             add((i_x,s,o),TestPurpose)
20:          end if
21:          x = x + 1
22:       end for
23:    end for
24: end for
```

3.7 Considerações Finais

A técnica apresentada estabelece uma solução para o problema da geração automática de objetivos de testes. A grande vantagem desta técnica se fundamenta na utilização de formalismos abstratos de alto nível, como os formalismos de lógica temporal, para a especificação de objetivos de teste em um formalismo de baixo nível (i.e. LTS) pronto para a utilização em técnicas de geração de casos de testes existentes (e.g. [JJ04]).

A utilização de formalismos de lógica temporal na especificação de objetivos de teste torna-se mais adequada que a utilização de formalismos como LTS, devido à sua maior proximidade à linguagem humana. Especificar e manter um objetivo de teste tornam-se tarefas menos dispendiosas e propensas a erros. No entanto, apesar de prover linguagem mais adequada ao manuseio humano, formalismo de lógica temporal pressupõem conhecimento de sua base teórica e experiência na construção de fórmulas, requerindo investimentos para treinamento de uma equipe técnica.

Fórmulas com quantificadores universais (e.g. AG, AU) não produzem contra-exemplos. A satisfação de tais fórmulas implica que todos os caminhos iniciados a partir de um estado inicial possam servir de exemplo, podendo cobrir todo o modelo. No entanto, a estratégia adotada para a extração de exemplos pode se basear em um limite da quantidade destes, para tornar factível a síntese dos objetivos de teste. O objetivo para um funcionamento ótimo da técnica é a utilização da menor quantidade de informações possível para a geração de um objetivo de teste representativo que corresponda à fórmula especificada.

Propriedades que devem ser representadas por fórmulas compostas e/ou aninhadas são contempladas pela técnica. Possibilita-se assim, a sua aplicação aos padrões de fórmulas mais utilizados [DAC99]. Propriedades de vivacidade (i.e. *liveness*), que geralmente implicam em execuções infinitas (e.g. fórmulas baseadas em EG) são representadas de forma finita pelos objetivos de teste gerados. Casos de teste gerados para tais tipos de propriedades devem assim, possuir número finito de ações, tendo execução finita.

Uma abordagem para a representação infinita destas propriedades pelos objetivos de teste foi apresentada. Técnicas de geração e/ou execução de casos de teste baseadas nesta abordagem devem considerar tal representação e limitar assim o tamanho dos casos de teste. Há técnicas que efetuam tal limitação por intermedio de temporizadores [JJ04].

Os algoritmos apresentados neste capítulo, apesar de enfocarem estruturas baseadas em lógica temporal ramificada (CTL), podem também ser aplicados a especificações baseadas em lógica temporal linear (LTL). No entanto, a diferente estrutura dos mecanismos de verificação de modelos baseados em LTL [CGP99; Kat99], exige uma maior investigação para sua adaptação.

A técnica de verificação de modelos LTL é fortemente baseada na de conversão de fórmulas LTL em reconhecedores de linguagens infinitas, como o autômato de Büchi [Büc62]. Por basear-se na negação da fórmula, o procedimento de verificação da propriedade contra o modelo produz autômato vazio. Desta maneira, para a obtenção de exemplos, a especificação da propriedade da fórmula deve basear-se em sua negação. A obtenção de exemplos baseia-se então, na violação da propriedade inicialmente especificada. Os testes a serem originados devem, desta maneira, ser negativos [Bin03]. Assim, a aplicação da técnica deve passar por diferentes experimentos, uma vez que os tipos de testes a serem gerados possuem diferentes enfoques.

Capítulo 4

Implementação de um Protótipo: Adaptação sobre o Veritas

A adaptação de um verificador de modelos para a implementação de uma ferramenta de geração de objetivos de teste, baseada na técnica proposta neste documento, deve ser realizada por meio da modificação de seus algoritmos de busca. O objetivo destes algoritmos, que originalmente é o de procurar por somente uma prova ou contra-prova da fórmula fornecida através de caminhos do modelo, passa a ser o de procurar um maior número de caminhos relacionados à propriedade (i.e. exemplos e contra-exemplos), como mencionado no Capítulo 3. Diferentemente do algoritmo original, que visa uma busca mínima por um espaço de estados, a adaptação algorítmica tem como objetivo alcançar maior exaustão na busca sobre o mesmo.

A construção de um protótipo baseado na adaptação do Veritas é abordada neste capítulo. Os algoritmos do Veritas são apresentados, bem como sua adaptação e a estratégia utilizada para a busca por maior número de caminhos do modelo. A implementação dos algoritmos apresentados no Capítulo 3 não é abordada, dado que a apresentação do referido capítulo é suficiente para a definição de sua implementação.

4.0.1 O Verificador de Modelos Veritas

O Veritas [Rod04; RGdF+04] é um verificador de modelos CTL baseado em espaços de estados RPOO [Gue02; dAGdFG04]. RPOO é uma linguagem de modelagem que adiciona

às redes de petri [Jen92] características da orientação a objetos. Tais características fornecem vantagens tais como a possibilidade de modelar sistemas com características de concorrência e paralelismo e modularizar os modelos. Um modelo RPOO consiste de um conjunto de classes e suas respectivas redes de Petri. O espaço de estados de um modelo RPOO pode ser gerado por uma ferramenta chamada *JMobile* [Sil05], a qual, por meio de simulações sobre o modelo, determina o conjunto completo dos estados em um arquivo de saída com formato compatível com o Veritas.

A arquitetura do Veritas é mostrada através da Figura 4.1 [Rod04]. O Veritas é composto por 4 módulos:

- O módulo de análise de espaço de estados é o módulo responsável por garantir que o espaço de estados fornecido ao Veritas possui a sintaxe correta de acordo com a gramática definida.
- O módulo de análise de especificações CTL é o módulo responsável por garantir que a fórmula fornecida ao Veritas possui a sintaxe correta de acordo com a gramática definida.
- O módulo de controle é o responsável pela interpretação das solicitações do usuário através da interface e coordenar a interação entre os módulos.
- O módulo de verificação compõe o principal módulo, o qual contém os algoritmos de busca sobre o espaço de estados que serão modificados para a realização da implementação da técnica proposta neste trabalho. Este módulo efetua a verificação da satisfação da fórmula em relação aos estados do modelo, devendo prover prova ou contra-prova por intermédio de caminhos extraídos do espaço de estados.

O módulo de verificação é o módulo sobre o qual as adaptações sobre o verificador foram realizadas. A seguir são apresentados os principais procedimentos e algoritmos e suas respectivas adaptações.

4.1 Módulo de Verificação

O Veritas implementa os seus algoritmos de busca em profundidade com a linguagem funcional ML [RS95]. Estes algoritmos baseiam-se na abordagem definida em [Hel97;

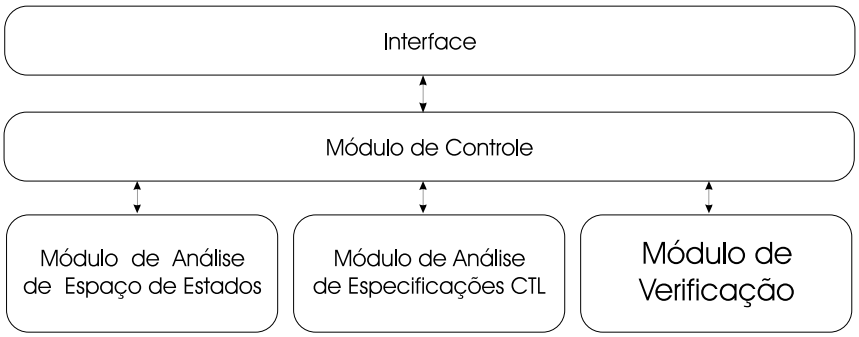


Figura 4.1: Arquitetura do Veritas

VL93], que facilita a extração de exemplos e contra-exemplos durante a verificação do modelo. Nesta abordagem, cada tipo de conectivo (e.g. EU, EG) define a forma de pesquisa sobre o espaço de estados, que deve ser iniciada em um estado específico. Esta estratégia permite que subfórmulas não sejam verificadas sobre todo o modelo, como a definida em [CGP99], minimizando a pesquisa realizada. Sua verificação pode ser realizada com base em estados encontrados após certa profundidade alcançada na busca, como por exemplo, na verificação de outras subfórmulas. Esta definição de verificação (i.e. em estados específicos) define um exemplo ou contra-exemplo durante a verificação ao passo em que o algoritmo evolui na busca.

Os algoritmos de busca implementados pelo Veritas fundamentam-se basicamente nos conectivos EU, EG e EX. Outras fórmulas com quantificadores existenciais e universais são compostas por fórmulas baseadas nestes conectivos, com o auxílio dos conectivos mínimos NOT e $\wedge$. Para realizar a implementação da técnica, apenas os algoritmos dos conectivos EU, EG e EX sofreram adaptações, sendo apresentados a seguir.

4.1.1 Algoritmo EU

O algoritmo 3 mostra o procedimento de busca definido pelo Veritas para o conectivo EU. O procedimento, denominado *check_EU*, recebe como argumento a fórmula f, o estado a ser verificado s e o caminho percorrido π. A fórmula f, por basear-se no conectivo EU, possui dois argumentos, ou subfórmulas, que denominaremos p e q, para o primeiro e segundo, respectivamente.

Inicialmente, o algoritmo realiza a verificação sobre a existência de avaliação prévia

sobre o estado s em questão. Caso o estado tenha sido previamente visitado e sua avaliação em relação à fórmula tenha sido verdadeira, a pesquisa deve ser finalizada (linhas 2,3 e 4, respectivamente). A variável *continue* (linha 4) é utilizada para o controle da condição de parada do algoritmo.

Algoritmo 3 Algoritmo de busca relativo ao conectivo EU

```
1:  proc check_EU(f, s, π)
2:  if marked(f, s) then
3:     if info(f, s) then
4:        continue = false
5:     end if
6:  else
7:     mark(f, s)
8:     if check(arg(f, 2),s,π) then
9:        setInfo(f, s, true)
10:       insert(s, π)
11:       continue = false
12:    else
13:       if check(arg(f, 1),s,π) then
14:          for all t ∈ sucessors(s) ∧ continue do
15:             check_EU(f, t, π)
16:          end for
17:          if ¬ continue then
18:             insert(s,π)
19:             setInfo(f, s, true)
20:          end if
21:       end if
22:    end if
23: end if
```

O segundo argumento da fórmula deve ser verificado em relação ao estado (linha 8). O procedimento *check* deve identificar o tipo de fórmula que recebe como argumento, neste caso q, devendo invocar o método de verificação correto correspondente. Neste caso, q pode ser uma proposição atômica, que deve ter sua validade verificada apenas no estado corrente

(s), ou pode ser uma fórmula qualquer, definindo o aninhamento da fórmula que deve ser verificada por recursão.

Caso a avaliação de q seja positiva em s, a pesquisa é interrompida (linha 11), uma vez que a condição de parada de $EU(p, q)$ é a satisfação de q, inicialmente, ou após uma seqüência de estados que satisfazem p. O estado s é adicionado ao caminho π (linha 9) e o resultado, utilizado pelo verificador em etapa posterior, é armazenado (linha 10). Esta última informação (função *info*) é utilizada pelo verificador para definir o resultado da verificação.

Caso q não seja verdadeira, p é verificada (linha 13). Caso a avaliação de p seja verdadeira, a busca deve continuar pelos estados sucessores de s, através de recursão (linha 15). Após a verificação dos estados sucessores, caso a avaliação tenha sido positiva para algum caminho alcançado pela recursão (linha 17), s é adicionado ao início do caminho (linha 18) e a fórmula é avaliada como verdadeira (linha 19).

4.1.2 Algoritmo EG

O algoritmo 4 mostra o procedimento de busca definido pelo Veritas para o conectivo EG. O procedimento, denominado *check_EG*, recebe como argumento a fórmula f, o estado a ser verificado s e o caminho percorrido π. Caso o estado tenha sido previamente visitado e sua avaliação em relação à fórmula tenha sido verdadeira, a pesquisa deve ser finalizada (linhas 2,3 e 4, respectivamente). O estado s é então, adicionado ao final do caminho que será definido como exemplo da fórmula (linha 5). Neste algoritmo também, o controle da condição de parada do algoritmo é realizado com o auxílio da variável *continue*.

Caso um estado s seja visitado pela primeira vez, verifica-se a validade da subfórmula de f em relação a s (linha 9). Em caso positivo, há a suposição de que s seja um estado integrante de um caminho que prova EG. O estado s é então definido como válido para a fórmula (linha 10). Para cada estado t sucessor de s, é realizada uma chamada recursiva a *check_EG* (linhas 11 a 13). O valor falso para a variável *continue* implica na validade de f no estado t e, portanto, em s. A suposição de que s possa integrar um caminho que comprove f é então, comprovada, sendo s adicionada ao início do caminho π (linhas 14 e 15).

Caso a avaliação dos estados sucessores de s não tenha satisfeito f, definindo a variável *continue* como verdadeira, a suposição de que s seja um estado integrante de um caminho que comprove f é falha. Desta forma, a informação sobre f em s é definida como falsa

Algoritmo 4 Algoritmo de busca relativo ao conectivo EG

```
1:  proc check_EG(f, s, π)
2:  if marked(f, s) then
3:     if info(f, s) then
4:        continue = false
5:        insert(s, π)
6:     end if
7:  else
8:     mark(f, s)
9:     if check(arg(f, 1),s,π) then
10:       setInfo(f, s, true)
11:       for all t ∈ sucessors(s) ∧ continue do
12:          check_EG(f, t, π)
13:       end for
14:       if ¬ continue then
15:          insert(s,π)
16:       else
17:          setInfo(f, s, false)
18:       end if
19:    end if
20: end if
```

(linha 17).

4.1.3 Algoritmo EX

O algoritmo 5 mostra o procedimento de busca definido pelo Veritas para o conectivo EX. O procedimento, denominado *check_EX*, recebe como argumento a fórmula f, o estado a ser verificado s e o caminho percorrido π. O algoritmo realiza a busca pelos sucessores de s entre as linhas 3 e 8. Caso algum sucessor satisfaça a subfórmula de f, a fórmula está provada. A avaliação do estado s é definida como positiva (linha 5) e este estado é adicionado ao caminho π (linha 6). O estado sucessor t deve ser adicionado pelo procedimento $check$ (linha 4), que é responsável pela verificação da subfórmula de f em relação a t.

Algoritmo 5 Algoritmo de busca relativo ao conectivo EX

```
1: proc check_EX(f, s, π)
2: mark(f, s)
3: for all t ∈ sucessors(s) do
4:    if check(arg(f, 1), t, π) then
5:       setInfo(f, s, true)
6:       insert(s,π)
7:    end if
8: end for
```

4.2 Adaptação dos Algoritmos

Para a concretização das adaptações sobre os algoritmos do Veritas foi definida uma estratégia para a extração dos caminhos sobre o modelo. A seguir são apresentadas a estratégia definida e as adaptações sobre os algoritmos do Veritas, apresentados anteriormente.

4.2.1 Estratégia de Extração de Caminhos

A adaptação realizada sobre os algoritmos de busca do Veritas tem como objetivo a exaustão sobre o modelo. Tem-se como justificativa para esta exaustão a necessidade de coletar, como informações, todas as transições relacionadas às propriedades que darão origem aos objetivos de teste para que os mesmos correspondam às respectivas propriedades. Considerando esta necessidade, definimos uma estratégia para a extração dos caminhos relacionados às propriedades CTL em relação aos modelos fornecidos.

Baseando-se na representação simplificada definida para os caminhos no Capítulo 3, a busca de um verificador de modelos deve ser finalizada ao encontrar o estado final de um caminho correspondente à fórmula. Um estado final de um exemplo ou contra-exemplo corresponde ao estado que define o fim de um caminho que satisfaz, ou viola, a fórmula, podendo ser apresentado pelo verificador de modelos como prova ou contra-prova da fórmula verificada, respectivamente.

Com o intuito de atingir a exaustão, o mecanismo mais simples e direto para a estratégia pode basear-se no alcance de todos os estados definidos como estados finais para exemplos ou contra-exemplos a serem extraídos do modelo. Permite-se assim, a formação de diversos

caminhos que possuam estados finais em comum. No entanto, parte destes caminhos pode possuir conjuntos equivalentes de transições relevantes, gerando volume de informação redundante e, assim, desnecessário.

Desta forma, a estratégia definida determina a extração de apenas um caminho para cada estado final, ou seja, apenas o primeiro caminho encontrado para um estado final deve ser extraído. Entretanto, caminhos extras devem ser obtidos em caso de diferenças em transições classificadas como relevantes, de maneira a manter a exaustão sobre a coleta de informações do modelo e evitando a perda de informações importantes para a geração de objetivos de teste.

Para uma melhor ilustração, a Figura 4.2 mostra a obtenção dos exemplos sobre um grafo a partir da busca em profundidade, realizada pelo Veritas. Os estados representados pelos círculos mais espessos representam os estados finais de caminhos que satisfazem uma determinada fórmula CTL.

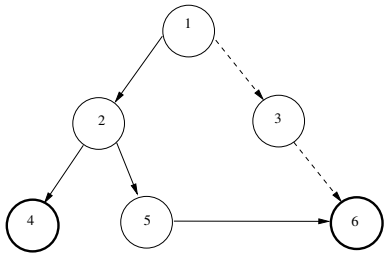


Figura 4.2: Esquema de busca em profundidade

A busca por um caminho deve ter como base os estados finais (no exemplo, os estados 4 e 6). As setas contínuas compõem os caminhos a serem extraídos, juntamente com os estados conectados por estas. As setas tracejadas representam, juntamente com os estados conectados pelas mesmas, caminho que, por possuir algum estado (neste caso o estado 6) já incluído em outro exemplo, não será extraído, durante a busca. Caso haja transição relevante não previamente selecionada pelos caminhos de setas contínuas, estes caminhos devem ser extraídos.

4.2.2 Rotulação dos Estados

A rotulação dos estados é um passo importante para a implementação da técnica de geração dos objetivos de teste. Como apresentado no Capítulo 3, cada estado de um caminho deve

receber como rótulo, o rótulo da proposição atômica (integrante da fórmula) à qual satisfaz, sendo que o último estado da cadeia deve ser rotulado apenas como estado final. Este estado final pode ser *accept*, para o caso em que o caminho extraído componha um exemplo, ou *refuse*, para o caso em que o caminho extraído componha um contra-exemplo.

Os novos algoritmos criados podem realizar a rotulação dos estados ao passo em que realizam a busca pelos caminhos relacionados à propriedade a ser especificada como objetivo de teste. Ao passo em que a relação de satisfação entre estados e fórmula vai sendo definida, os rótulos são atribuídos aos estados, que são adicionados ao caminho sob extração.

4.2.3 Adaptação do Algoritmo EU

A adaptação mostrada pelo algoritmo 6 permite a verificação de fórmulas EU aninhadas, ou seja, para as quais os argumentos p e q sejam proposições atômicas ou subfórmulas. Nesta adaptação, a classificação das transições do modelo é efetuada ao passo em que realiza a busca e extração dos caminhos, que são utilizados para obtenção de informações sobre predecessão entre transições no algoritmo de síntese de objetivos de teste.

Sua condição de parada é a cobertura de todos os estados finais sem que percorra caminhos com auto-laços (linha 2). Caminhos que contêm estados finais previamente cobertos serão extraídos caso haja transições não contidas em outros caminhos extraídos anteriormente.

O controle realizado sobre a extração de caminhos com estados finais previamente cobertos é apresentado entre as linhas 25 e 28. A função *info* passa a ser utilizada apenas para controle destes estados, deixando de ser acionada na avaliação de estados p. Assim, caso o estado visitado seja um estado final e a transição em questão não tenha sido previamente coletada (linha 25), o caminho é extraído (linhas 26 e 27).

Caso a avaliação de q seja positiva em relação ao estado s (linha 4), sendo q uma proposição atômica (função *isAtomicProposition*), seu resultado é armazenado (linha 5), e s é adicionado com rótulo $accept$ ao caminho π (linha 7), que deve ser armazenado como exemplo (linha 8). A transição que ocorre entre s e seu antecessor é armazenada no conjunto de transições relevantes L com a informação sobre o estado resultante, que neste caso é o de aceitação (representado por *acc*, linha 6). O estado antecessor é representado pelo rótulo da subfórmula que o satisfaz, definida pela função *arg*.

Algoritmo 6 Algoritmo EU Adaptado

```
1: proc check_EU(f, s, π)
2: if ¬marked(f, s) then
3:    mark(f, s)
4:    if check(arg(f, 2),s,π) ∧ isAtomicProposition(arg(f, 2)) then
5:       setInfo(f, s, true)
6:       add(getTransition(last(π), s), arg(f, 2), acc, L)
7:       insert(s, accept, π)
8:       save(π,examples)
9:    else
10:      if check(arg(f, 1),s,π) then
11:         if isAtomicProposition(arg(f, 2)) then
12:            add(getTransition(last(π), s), arg(f, 1), arg(f, 1), N)
13:            insert(s,arg(f, 1),π)
14:         end if
15:         for all t ∈ sucessors(s) do
16:            check_EU(f, t, π)
17:         end for
18:      else
19:         add(getTransition(last(π), s), arg(f, 1), ref, L)
20:         insert(s, refuse, π)
21:         save(π,counter − examples)
22:      end if
23:   end if
24: else
25:   if add(getTransition(last(π), s), acc, L) ∧ info(f, s) then
26:      insert(s, accept, π)
27:      save(π,examples)
28:   end if
29: end if
```

Caso q não seja verdadeira, a proposição p é verificada (linha 10). Caso p seja uma proposição atômica e sua avaliação seja verdadeira, a transição que ocorre entre s e seu ante-

cessor é armazenada no conjunto de transições irrelevantes N (linha 12). Para este conjunto, a informação de estados armazenada reflete a representação de auto-laços de rótulo "*". A busca continua pelos estados sucessores de s (linha 14).

A avaliação negativa de p em s define um contra-exemplo, que deve ser extraído (linhas 19 e 20). A transição que ocorre entre s e seu antecessor é armazenada no conjunto de transições relevantes L, definindo o estado de rejeição (*ref*) como resultante (linha 19).

4.2.4 Adaptação do Algoritmo EG

Assim como a adaptação do algoritmo EU, a adaptação de EG, mostrada pelo algoritmo 7, permite a verificação de fórmulas EG aninhadas, ou seja, para as quais seu argumento seja proposição atômica ou subfórmula. A classificação das transições do modelo é efetuada ao passo em que realiza a busca e extração dos caminhos, que são utilizados para obtenção de informações sobre predecessão entre transições no algoritmo de síntese de objetivos de teste.

Uma vez que o algoritmo de busca baseado em EG procura por auto-laços para a definição de um caminho, a primeira análise realizada pelo algoritmo é a visita prévia sobre o estado s. Caso este estado, recebido como argumento, tenha sido previamente visitado e sua avaliação em relação à fórmula tenha sido verdadeira (linhas 2 e 3, respectivamente), conclui-se um caminho a ser extraído. A variável *continue* (linha 4) é utilizada para realizar o controle sobre o sucesso na busca pelo caminho. A busca pelos sucessores do estado definido como final deve cessar. Caso o estado s não tenha sido adicionado ao caminho a ser definido como exemplo, deve ser adicionado (linhas 5 a 10).

Caso um estado s seja visitado pela primeira vez, verifica-se a validade da subfórmula de f em relação a s (linha 14). Em caso positivo, há a suposição de que s seja um estado integrante de um caminho que prova EG, devendo assim, ser avaliado como válido para a fórmula (linha 15). Caso a subfórmula de f seja uma proposição atômica (linha 16), s deve ser adicionado ao caminho (linha 18) e a transição entre este e seu antecessor é adicionada ao conjunto de transições irrelevantes (linha 17).

Para cada estado t sucessor de s, é realizada uma chamada recursiva a *check_EG* (linhas 20 a 21). Esta recursão deve parar em caso de o caminho já ter sido encontrado (linhas 4). A variável *continue* volta a ter valor positivo para que o algoritmo prossiga normalmente a busca (linha 26).

Algoritmo 7 Algoritmo EG Adaptado

```
1: proc check_EG(f, s, π)
2: if marked(f, s) then
3:    if info(f, s) then
4:       continue = false
5:       if s ∉ final then
6:          add(getTransition(last(π), s), arg(f, 1), acc, L)
7:          insert(s, accept, π)
8:          save(π,examples)
9:          insert(s, final)
10:      end if
11:   end if
12: else
13:   mark(f, s)
14:   if check(arg(f, 1),s,π) then
15:      setInfo(f, s, true)
16:      if isAtomicProposition(arg(f, 1)) then
17:         add(getTransition(last(π), s), arg(f, 1), arg(f, 1), N)
18:         insert(s,arg(f, 1),π)
19:      end if
20:      for all t ∈ sucessors(s) ∧ continue do
21:         check_EG(f, t, π)
22:      end for
23:      if continue then
24:         setInfo(f, s, false)
25:      else
26:         continue = true
27:      end if
28:   else
29:      add(getTransition(last(π), s), arg(f, 1), ref, L)
30:      insert(s, refuse, π)
31:      save(π,counter − examples)
32:   end if
33: end if
```

O valor verdadeiro para a variável *continue* após a busca pelos sucessores de s implica que a suposição de que s pertença a um exemplo falhou (linhas 23 e 24). Assim, s volta a receber avaliação negativa em relação à fórmula.

A definição de contra-exemplos dá-se para cada violação da fórmula. Caso a avaliação da validade da subfórmula de f em s (linha 14) tenha sido comprovada como falsa (linha 28), um contra-exemplo é gerado. A transição que ocorre entre o seu antecessor é armazenada no conjunto de transições relevantes L(linha 29). O estado é adicionado ao caminho com o rótulo *refuse* (linha 30) e o contra-exemplo é salvo (linha 31).

4.2.5 Adaptação do Algoritmo EX

A adaptação mostrada pelo algoritmo 8 mostra o procedimento de busca e extração de caminhos para o conectivo EX. O procedimento, denominado *check_EX*, recebe como argumento a fórmula f, o estado a ser verificado s e o caminho percorrido π. O algoritmo realiza a busca pelos sucessores de s entre as linhas 6 e 18.

O estado s é previamente inserido ao caminho π, juntamente com a transição entre o mesmo e seu antecessor. Caso π esteja vazio (linha 3), s é inserido com um rótulo qualquer, definido aqui como 'a' (linha 4).

A subfórmula de f é avaliada em cada sucessor t de s. Caso esta subfórmula seja avaliada como positiva (linha 7) e seja uma proposição atômica (linha 9), a transição que ocorre s, definido como $last(\pi)$, e t é armazenada no conjunto de transições relevantes L com a informação sobre o estado resultante, que neste caso é o de aceitação (representado por *acc*, linha 11). Um exemplo é gerado (linha 12).

Caso esta subfórmula seja avaliada como negativa (linha 14) a transição que ocorre s, definido como $last(\pi)$, e t é armazenada no conjunto de transições relevantes L com a informação sobre o estado resultante, que neste caso é o de rejeição (representado por *ref*, linha 16). Um contra-exemplo é gerado (linha 17).

4.3 Considerações Finais

Neste capítulo apresentamos o verificador Veritas, seus algoritmos de busca referentes aos conectivos EU, EG e EX, bem como as adaptações realizadas sobre os mesmos para a pro-

Algoritmo 8 Algoritmo de busca relativo ao conectivo EX

```
1: proc check_EX(f, s, π)
2: mark(f, s)
3: if length(π) == 0 then
4:    insert(s, a, π)
5: end if
6: for all t ∈ sucessors(s) do
7:    if check(arg(f, 1), t, π) then
8:       setInfo(f, s, true)
9:       if isAtomicProposition(arg(f, 1)) then
10:         insert(t, accept, π)
11:         add(getTransition(last(π), t), last(π), acc, L)
12:         save(π,examples)
13:      end if
14:   else
15:      insert(t, refuse, π)
16:      add(getTransition(last(π), t), last(π), ref, L)
17:      save(π,counter − examples)
18:   end if
19: end for
```

dução de um protótipo de ferramenta capaz de obter as informações necessárias à geração de objetivos de teste.

Esta adaptação possibilita para a extração de maior número de exemplos e contra-exemplos do modelo. Uma estratégia para a realização da coleta destes caminhos foi definida para guiar a implementação da adaptação sobre os algoritmos. Esta estratégia permite a eliminação de caminhos cujas transições já tenham sido coletadas anteriormente, possibilitando um menor volume de dados na análise.

Apesar de ser realizada sobre o Veritas, a adaptação pode ser realizada sobre qualquer outro verificador de modelos baseado em CTL (e.g. [Mcm93]), considerando a forma de definição de exemplo e/ou contra-exemplo provida pelo verificador. A estratégia definida para a extração de caminhos pode ser realizada de maneira diferente, uma vez que os procedimentos de geração de objetivos de teste apresentados no Capítulo 3 são independentes

da maneira como os caminhos são coletados. Entretanto, apesar de haver possibilidade de utilização de estratégias diferentes, como por exemplo a definição de estratégias baseadas na busca em largura, o objetivo destas estratégias deve ser o mesmo, o de alcançar a exaustão sobre o modelo.

Capítulo 5

Estudo de Caso: Geração de Objetivos de Teste para o Protocolo IP Móvel

Para demonstrar o funcionamento dos algoritmos propostos para a geração de objetivos de testes utilizamos o protocolo IP Móvel [Per02] como estudo de caso. A geração dos objetivos de teste dá-se através do protótipo implementado com a adaptação do verificador de modelos Veritas [RGdF+04], apresentado no Capítulo 4. O modelo formal do protocolo IP Móvel [RGdF+04] utilizado é definido com a linguagem RPOO, suportada pelo Veritas. Como o objetivo deste capítulo é apresentar a geração de objetivos de teste por meio da especificação de propriedades desejáveis à uma implementação do protocolo, o seu modelo RPOO não será apresentado, sendo este, apresentado informalmente, com o auxílio de um diagrama de classes UML [RGdF+04].

Após a apresentação do protocolo e seu modelo abstrato, serão apresentadas as propriedades a serem utilizadas como objetivos de teste e suas respectivas fórmulas CTL utilizadas no processo de geração. Os objetivos de teste gerados com base em tais fórmulas são apresentados ao passo em que as propriedades CTL são delineadas. Em seguida, casos de teste referentes a estas propriedades são gerados por intermédio da ferramenta TGV, previamente apresentada no Capítulo 2.

Para viabilizar a utilização da ferramenta TGV a partir da especificação RPOO, implementamos uma ferramenta para conversão do espaço de estados RPOO, utilizado pelo Veritas, para o formato utilizado pelo TGV. O espaço de estados RPOO convertido possui 1086 estados e 2502 transições, cujos rótulos foram classificadas explicitamente em ações de en-

trada, saída e internas para utilização do TGV. Para concluir o estudo de caso, uma análise dos casos de teste gerados pelo TGV e uma comparação entre o caminhos extraídos do modelo do IP Móvel e os casos de teste gerados são realizadas.

5.1 IP Móvel

Os protocolos de roteamento da Internet não operam com a possibilidade de migração de seus nodos de forma dinâmica, não prevendo alterações de endereços IP. Esta migração de nodos na rede surgiu com a conexão de dispositivos móveis (e.g. telefones celulares) à Internet. Isto ocorre devido à característica de mobilidade destes dispositivos, que, por conta do limite de alcance dos sinais emitidos podem necessitar da mudança de roteador para mantê-los conectados à rede, recebendo assim, novo endereço IP sob o domínio do novo roteador. O roteador original, ao perder a conexão com o dispositivo, perde a capacidade de localizá-lo e conseqüentemente transmitir-lhe pacotes. Há assim, um rompimento das conexões previamente estabelecidas entre *hosts* (denominados *correspondent nodes*) e o dispositivo, causando a perda dos pacotes enviados a este, uma vez que esses pacotes estarão destinados ao endereço IP original.

O protocolo IP móvel foi proposto com o intuito de resolver o problema de roteamento surgido. O protocolo visa manter a conexão dos dispositivos móveis, denominados *mobile nodes*, à rede, de modo que as migrações ocorridas sejam transparentes às aplicações. Ou seja, mesmo após migração, a comunicação com um dispositivo deve ocorrer como se este estivesse em sua rede de origem.

Para prover a transparência nas migrações, o protocolo propõe como solução que um dispositivo móvel possua dois endereços IP, o endereço original, denominado *home address*, e um endereço que será atribuído a cada migração, denominado *care-of-address* (COA). O *home address* é obtido na rede de origem, denominada *home network*, enquanto o COA é obtido na rede estrangeira para a qual o dispositivo esteja migrando, sendo chamada de *foreign network*. Assim, enquanto está em uma rede estrangeira, o dispositivo deve receber os pacotes através do roteador desta, chamado *foreign agent*. Este roteador, por sua vez, recebe os pacotes destinados ao dispositivo através do roteador da rede de origem, chamado *home agent*.

Os pacotes enviados por um *host* ao dispositivo são destinados ao endereço *home address*, que ao chegar ao *home agent* é encapsulado dentro de outro pacote cujo endereço de destino é o COA. Este enlace criado entre o *home agent* e o *foreign agent* tem o efeito de um túnel, sendo o processo de encapsulamento e reenvio do pacote ao *foreign agent* chamado de tunelamento.

O processo de migração de um *mobile node* consiste na descoberta de um novo endereço, COA, ao conectar-se à rede estrangeira e posterior registro do COA junto ao *home agent*. Este registro é intermediado pelo *foreign agent*, que encaminha a mensagem informando o COA ao *home agent*, que por sua vez, atualiza os registros e confirma tal atualização.

A Figura 5.1 [RGdF+04] mostra o esquema de funcionamento do protocolo. O envio de uma mensagem por um *host* (CN) ao dispositivo (MN) dá-se através de seu endereço original. O *home agent* (HA) deve encaminha-la ao dispositivo diretamente, caso este encontre-se na rede de origem. Caso o dispositivo encontre-se em uma rede estrangeira, o processo de tunelamento é realizado, no qual o *home agent* encapsula os pacotes destinados ao dispositivo dentro de um outro pacote cujo endereço de destino é o COA obtido pela rede estrangeira. O pacote encaminhado à Internet chega ao *foreign agent* (FA), que o encaminha ao dispositivo.

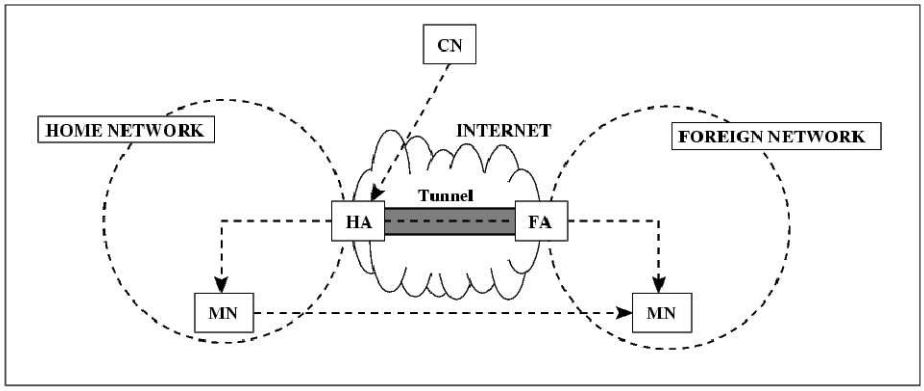


Figura 5.1: Esquema de funcionamento do IP Móvel

5.1.1 O Modelo do IP Móvel

O comportamento do protocolo descrito na Seção 5.1 é especificado através do modelo mostrado na Figura 5.2. Com base no foco do suporte à mobilidade provida pelo protocolo, abstraindo questões como a de segurança e do processo de descobrimento de endereço, que

é realizado com base no protocolo *ICMP Router Discovery Messages* [Dee91], o modelo produzido é composto por cinco entidades.

O dispositivo móvel é representado por *MobileNode* está relacionado ao agente de mobilidade, representado pela entidade *MobilityAgent*. Por sua vez, o agente de mobilidade, que é a entidade responsável pelo endereçamento correto dos pacotes aos nodos móveis, é especializado por duas outras entidades, o agente local, representado pela entidade *HomeAgent*, e o agente estrangeiro, representado pela entidade *ForeignAgent*. As conexões entre os agentes de mobilidade e o dispositivo móvel são efetuadas através de um meio de comunicação, que por sua vez, é representado pela entidade *Medium*. As entidades *Internet* e *CorrespondentNode* representam abstrações de conjuntos de roteadores da *Internet* e nodos estáticos que enviam mensagens aos dispositivos móveis do sistema.

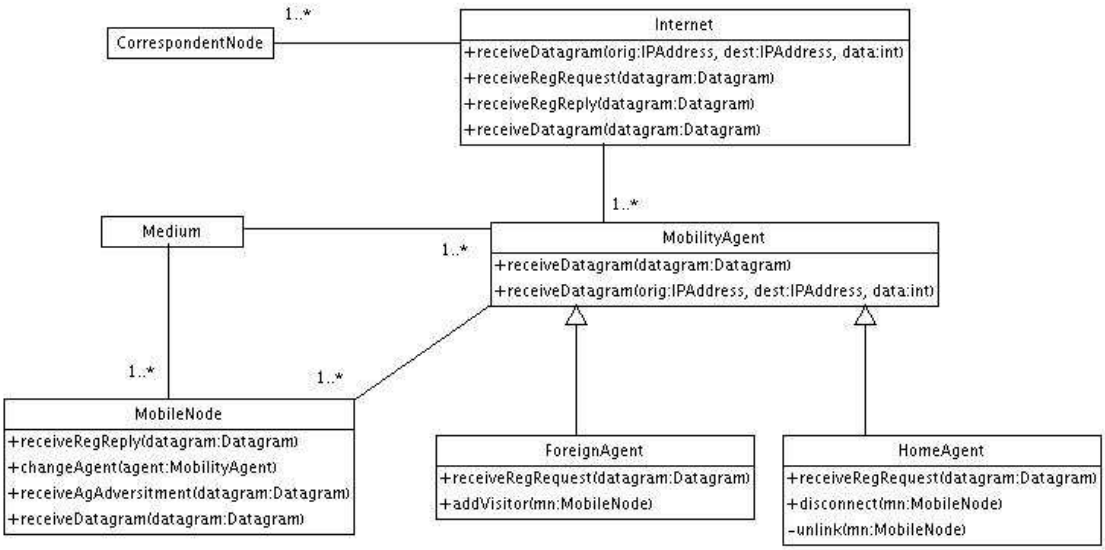


Figura 5.2: Diagrama de classes do modelo do IP Móvel

5.2 Definição de Propriedades e Geração dos Objetivos de Teste

O estudo de caso realizado sobre o IP Móvel foi definido com base em cinco propriedades a serem testadas sobre uma implementação. Sobre estas propriedades as fórmulas CTL cor-

respondentes foram especificadas e submetidas ao protótipo construído sobre a adaptação do Veritas.

5.2.1 Propriedade 1

Como um primeiro objetivo de teste, desejamos verificar a conformidade entre implementação e modelo para os casos em que as mensagens são enviadas de um nodo estático ao dispositivo móvel, que encontra-se em sua rede original. Desejamos que neste caso, enquanto o dispositivo se encontre em sua *home network*, este deve receber os pacotes a ele destinados via *home agent*. Assim, definimos como proposições p e q, "O *mobile node* migrou para uma *foreign network*"e "O *home agent* entrega as mensagens ao *mobile node*", respectivamente. A seguinte propriedade CTL é especificada:

$$EU(NOT(p), q) \tag{5.1}$$

Apesar de o requisito expressado em relação à entrega de mensagens implicar em uma fórmula mais complexa com quantificador universal, decidimos expressá-la desta maneira para tornar mais simples a análise e demonstração da técnica. Assim, conseguimos extrair exemplos em que não migrando, o *mobile node* recebe mensagens diretamente do *home agent*, e, contra-exemplos com a violação de $NOT(p)$, quando o *mobile node* migra. Denominamos tal propriedade P1. A Figura 5.3 mostra o grafo que contém os exemplos e contra-exemplos relativos à propriedade P1. O número de exemplos e contra-exemplos extraídos do modelo foram 7 e 29, respectivamente, contendo um total de 64 estados e 63 transições. Apesar de a estratégia de extração definida acima ter como objetivo diminuir a massa de dados redundantes, a cobertura de caminhos para esta propriedade coincidiu com a quase totalidade dos mesmos. Além da restrição sobre o número de caminhos obtidos pela estratégia, restrições sobre a fórmula contribuíram para sua redução. Como exemplo, foi definido que as mensagens enviadas ao *mobile node* deveriam ser de um único tipo específico, diminuindo o número de exemplos relacionados.

A análise sobre os exemplos e contra-exemplos da Figura 5.3 permitiu a extração das ações especificadas para as transições entre os estados. A análise algorítmica definiu as ações a serem abstraídas e as que comporiam o objetivo de teste. Os rótulos definidos sobre

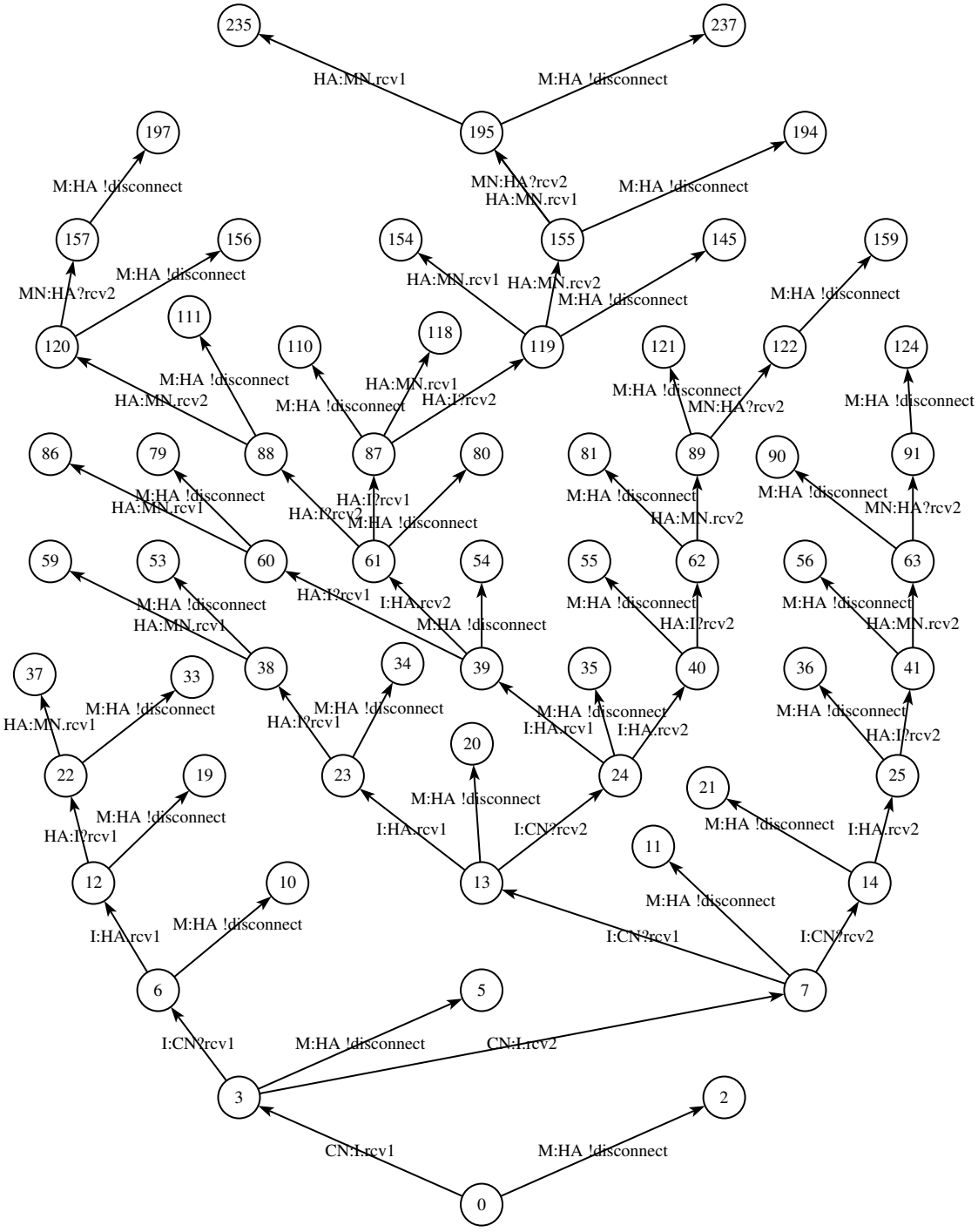


Figura 5.3: Grafo composto por exemplos e contra-exemplos da propriedade P1

os estados, de acordo com a representação simplificada definida no Capítulo 3, foram 3: $NOT(p)$, $accept$ e $refuse$. Os estados $NOT(p)$ são os estados que satisfazem apenas o primeiro argumento da fórmula, definido pela a proposição $NOT(p)$.

A análise efetuada permitiu concluir que apenas uma ação específica é necessária para levar de um estado $NOT(p)$ a um estado $accept$, representando o envio de uma mensagem do *home agent* ao *mobile node*. Esta transição foi representada nos grafos pelo rótulo "*HA:MN.rcv1*", assim, $L_{NOT(p)\_accept} = \{$*HA:MN.rcv1*$\}$. Novamente, para levar de um estado p a um estado $refuse$, somente uma transição foi detectada no processo. Esta transição corresponde à migração do *mobile node*, sendo representada através do rótulo "*M:HA!disconnect*", assim, $L_{NOT(p)\_refuse} = \{$*M:HA!disconnect*$\}$. A transição ao estado $refuse$, representando a migração do *mobile node* não é de interesse do objetivo de teste, portanto não deve gerar casos de teste. Com base nas informações obtidas pela análise dos caminhos extraídos do modelo, o LTS correspondente ao objetivo de teste da fórmula 5.1 foi definido, sendo mostrado na Figura 5.4.

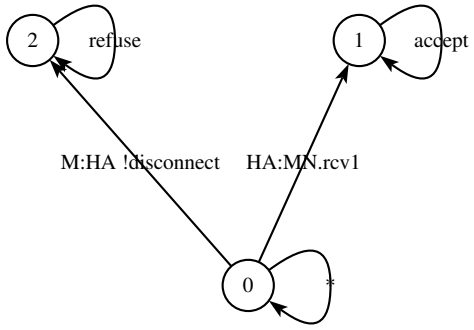


Figura 5.4: Objetivo de teste da propriedade P1 (fórmula 5.1)

Após a geração do objetivo de teste, o conjunto completo dos casos de teste (CTG) foi gerado através do TGV. O CTG gerado contém 65 estados e 82 transições, abrangendo os estados e transições contidos nos exemplos e contra-exemplos. Como o número de caminhos extraídos atingiu quase a totalidade dos caminhos relacionados à propriedade existentes no modelo (representados pelo CTG), a diferença entre o número de estados destes e do CTG foi pequena. Porém, a combinação de estados para a composição de casos de teste definida pelo CTG possibilita números maiores dos mesmos, podendo chegar a infinitos casos de teste (e.g. em caso de laços no modelo). A apresentação do grafo do CTG produzido pelo TGV é de legibilidade inviável para este documento.

Um caso de teste gerado pelo TGV é mostrado na Figura 5.5. O caso de teste da figura contém 36 estados e 38 transições. A execução do caso de teste deve levar em consideração o veredicto definido no rótulo de alguns estados. A exibição da propriedade por uma implementação deve alcançar estados rotulados por *PASS* (e.g. estado 35), indicando que a implementação a exibiu. Situações previstas no modelo, porém não objetivadas pelo caso de teste, como por exemplo, a migração do *mobile node* antes de receber uma mensagem do *home agent*, não devem influenciar nos resultados do caso de teste, atingindo estados rotulados com *INCONCLUSIVE* (e.g. estados 31 e 32). Situações não previstas pelo caso de teste devem levar a veredictos que apontem a falha da implementação. Tais situações não são representadas no grafo apresentado, sendo alcançadas após a violação da relação *ioco*, na qual a implementação realiza operação não definida pelo modelo.

Com base na figura, nota-se que um caso de teste não é composto por um caminho de execução simples contido no modelo. Em verdade, é composto por vários deles. Uma questão a ser analisada com base neste caso de teste diz respeito a este fator, bem como ao número de estados de aceitação (i.e. estados com rótulos *PASS*) do mesmo. O caso de teste, projetado para sistemas reativos com características de não-determinismo, deve prever situações em que o não-determinismo inerente ao sistema possibilite interações diversificadas, possibilitando atingir a propriedade desejada de variadas maneiras, ou até impedindo o alcance da mesma durante algumas interações. Desta maneira, deve possibilitar a realização dos testes de maneira flexível, com possibilidade de análises completas.

Este fator é forte argumento contra a utilização de técnicas de conversão direta de exemplos e contra-exemplos extraídos de modelos por intermédio de verificadores de modelos [PEAM98; GH99; JCE96] em casos de teste. Um exemplo pode representar uma seqüência de interação com uma implementação, porém, provê pouca flexibilidade para a análise de sua execução. O êxito na execução de uma determinada seqüência de ações sobre uma IUT pode ser ínfimo em relação a sistemas não-deterministas, fato que pode inviabilizar os testes, seja pela incapacidade de executar os casos de teste, ou pela impossibilidade de obter veredictos corretos sobre a sua execução.

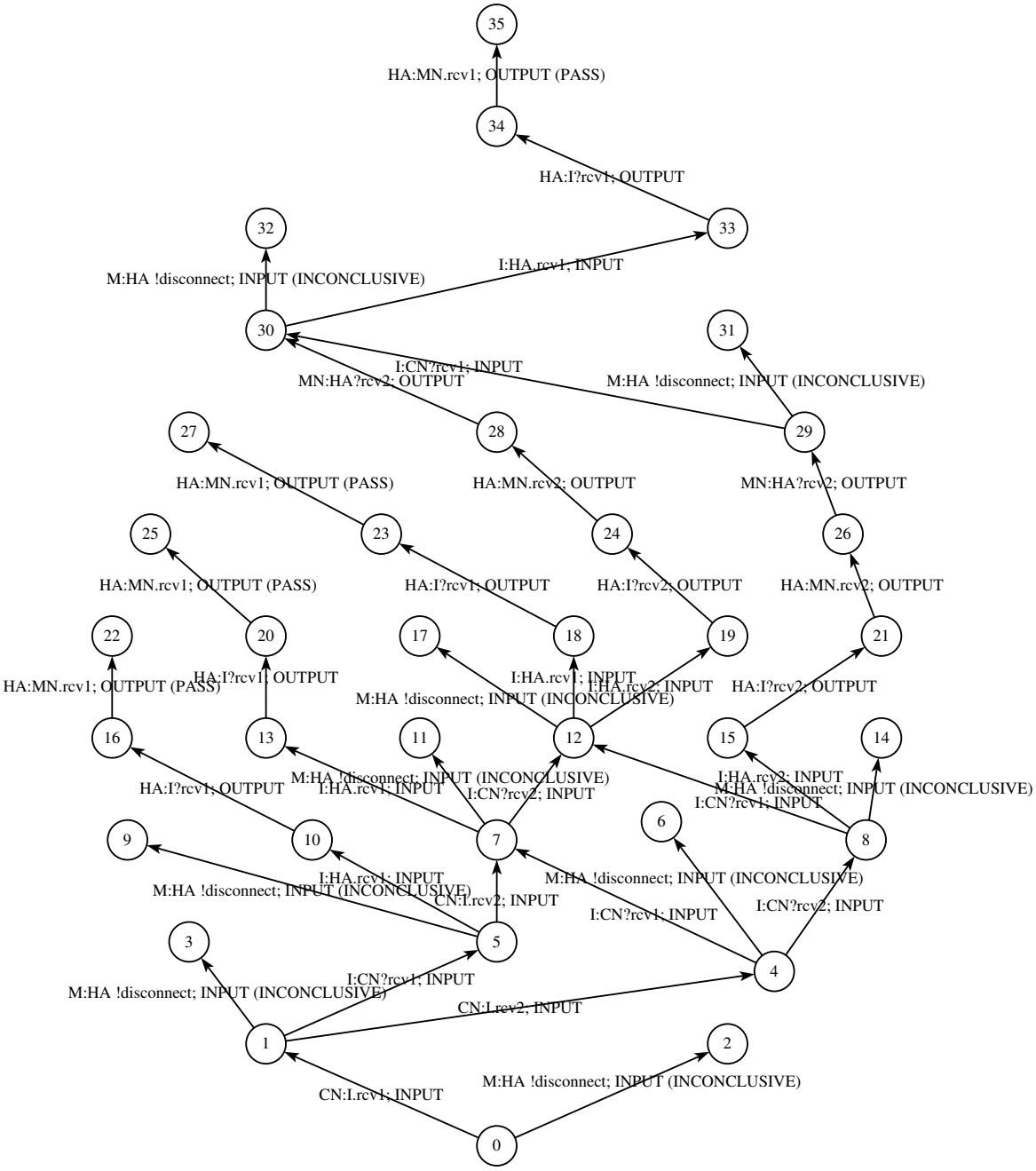


Figura 5.5: Caso de teste referente à propriedade P1

5.2.2 Propriedade 2

O modelo definido para o IP Móvel prevê a troca de mensagens entre um nodo estático, *correspondent node*, e um dispositivo móvel, *mobile node*. As mensagens enviadas por um *correspondent node* devem ser encaminhadas à *Internet* para que cheguem então a um agente de mobilidade, por um *home agent* ou um *foreign agent*, para que seja entregue ao *mobile*

node. Como uma segunda propriedade a ser testada, definimos o envio de mensagens pelo *correspondent node* através da *Internet*. Assim, definimos que sempre haverá mensagens enviadas pelo *correspondent node* pela *Internet*.

Nesta propriedade desejamos mostrar a utilização de fórmulas baseadas no quantificador universal que expressem ocorrências futuras, i.e. através do operador AF. A proposição atômica correspondente ao requisito exposto acima é definida como "*correspondent node* envia mensagem". Denominamos tal proposição como p. A propriedade CTL correspondente é definida pela fórmula 5.2.

$$AF(p) \tag{5.2}$$

Uma fórmula baseada em quantificador universal não provê possibilidade de extração de contra-exemplos, provendo apenas exemplos relacionados à fórmula. Esta fórmula possibilitou a extração de 12 exemplos que contêm 24 estados e 23 transições. Como o envio de mensagens por um *correspondent node* é a base do modelo utilizado no estudo de caso, os envios de mensagens ocorrem precocemente no modelo. Desta maneira, o número de exemplos obtidos, assim como o número de estados destes é pequeno.

Como a implementação do protótipo tem como base os operadores EG e EU. A fórmula definida pela equação 5.2 é traduzida para a sua equivalente baseada no operador EG. Assim, a fórmula passa a ser tratada pelo protótipo como: $NOT(EG(NOT(p)))$. A análise sobre os exemplos obtidos permitiu a extração das ações especificadas para as transições entre os estados. A análise algorítmica definiu as ações a serem abstraídas e as que comporiam o objetivo de teste. Os rótulos definidos sobre os estados dos caminhos foram 2: $NOT(p)$ e $accept$.

A análise efetuada permitiu concluir que apenas uma ação específica é necessária para levar de um estado $NOT(p)$ a um estado $accept$, representando o envio de uma mensagem do *correspondent node* através da *Internet*. Esta transição foi representada nos grafos pelo rótulo "*CN:I.rcv1*". Assim, $L_{NOT(p)\_accept} = \{$*CN:I.rcv1*$\}$. Com base nas informações obtidas pela análise, o LTS correspondente ao objetivo de teste da fórmula 5.2 foi definido, sendo mostrado na Figura 5.6.

O CTG gerado, mostrado na Figura 5.7, contém 24 estados e 23 transições, abrangendo os estados e transições contidos nos exemplos. Como o número de caminhos extraídos atin-

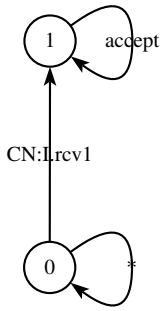


Figura 5.6: Objetivo de teste da propriedade P2 (fórmula 5.2)

giu a totalidade dos caminhos correspondentes à propriedade no modelo, o CTG conteve a mesma quantidade de estados e transições.

Um caso de teste gerado pelo TGV é mostrado na Figura 5.8. O caso de teste da figura contém 18 estados e 17 transições. O caso de teste constitui um subconjunto dos estados e transições contidos no CTG. A execução de tal caso de teste deve levar em consideração o veredicto definido no rótulo de alguns estados. A exibição da propriedade por uma implementação deve alcançar estados rotulados por *PASS* (e.g. estado 17), indicando que a implementação exibiu tal propriedade. Para este caso de teste não foram definidos veredictos inconclusivos, pois todos os caminhos do modelo levam ao envio de mensagem por um *correspondent node*, não ocorrendo situações que não atendam à propriedade especificada.

5.2.3 Propriedade 3

Uma vez verificada e testada a Propriedade 2, em que o *correspondent node* sempre envia uma mensagem encaminhada pela *Internet*, desejamos testar a entrega destas mensagens ao *mobile node* pelo *home agent*. Entretanto, desejamos testar apenas as situações em que as mensagens são sempre entregues pelo *home agent* e são consumidas imediatamente depois pelo *mobile node*, sem que este migre.

Nesta propriedade desejamos mostrar a utilização de fórmulas aninhadas baseadas no quantificador existencial que expressem ocorrências globais e recorrentes, i.e. através do aninhamento dos operadores EG e EF. Três proposições atômicas correspondentes ao requisito exposto acima são definidas. A primeira, denominada p, como "*home agent* envia a mensagem ao *mobile node*". A segunda, denominada q, definida como "O *mobile node* recebe a mensagem". A terceira, denominada r, é definida como "O *mobile node* está em

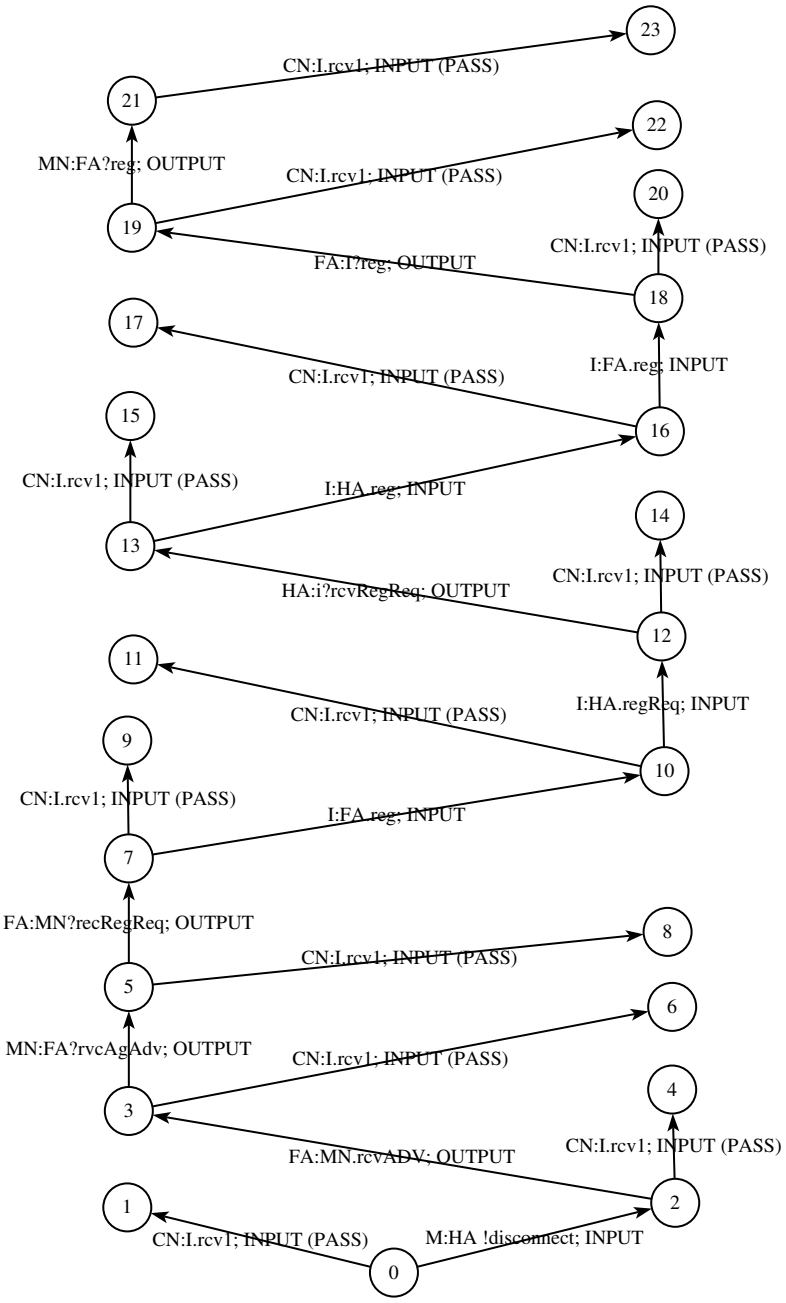


Figura 5.7: CTG da propriedade P2

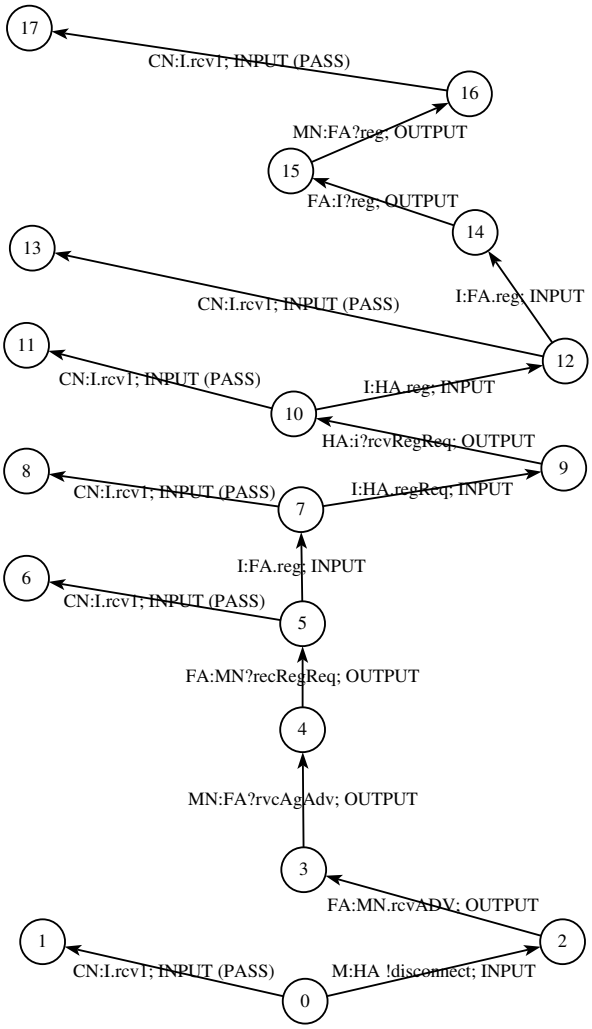


Figura 5.8: Caso de teste referente à propriedade P2

sua rede local". A propriedade CTL correspondente é definida pela fórmula 5.3.

$$EG(EF((r \wedge p) \longrightarrow EF(q))) \tag{5.3}$$

Entre os 20 caminhos obtidos a partir desta fórmula, houve um total de 33 estados e 32 transições. A análise sobre estes caminhos permitiu a extração das ações especificadas para as transições entre os estados. A análise algorítmica definiu as ações a serem abstraídas e as que comporiam o objetivo de teste. Os rótulos definidos sobre os estados dos caminhos foram 4: ϕ, ψ, $accept$ e $refuse$. O modelo define que inicialmente o *mobile node* encontra-se em sua rede local, portanto a proposição r é inicialmente satisfeita. Desta forma, o rótulo do estado ϕ corresponde à satisfação da proposição r e da negação de p, ou seja, $\phi \equiv r \wedge NOT(p)$. O rótulo ψ corresponde à satisfação das proposições p e r, ou seja, $\psi \equiv r \wedge p$.

A análise efetuada permitiu concluir que apenas uma ação específica é necessária para levar de um estado ϕ a um estado p, representando o envio de uma mensagem do *home agent* ao *mobile node*. Esta transição foi representada nos grafos pelo rótulo "*HA:MN.rcv1*", sendo, $L_{\phi\_p} = \{$*HA:MN.rcv1*$\}$. Para levar de um estado p a um estado $accept$, representando o recebimento da mensagem pelo *mobile node*, a transição "*MN:HA?rcv1*" foi encontrada, sendo $L_{p\_accept} = \{$*MN:HA?rcv1*$\}$. A violação da proposição r (i.e. migração do dispositivo) leva ao estado $refuse$. A migração do dispositivo pode levar dos estados ϕ e p ao $refuse$, assim, $L_{\phi\_refuse} = \{$*M:HA !disconnect*$\}$ e $L_{p\_refuse} = \{$*M:HA !disconnect*$\}$. O objetivo de teste correspondente é mostrado na Figura 5.9.

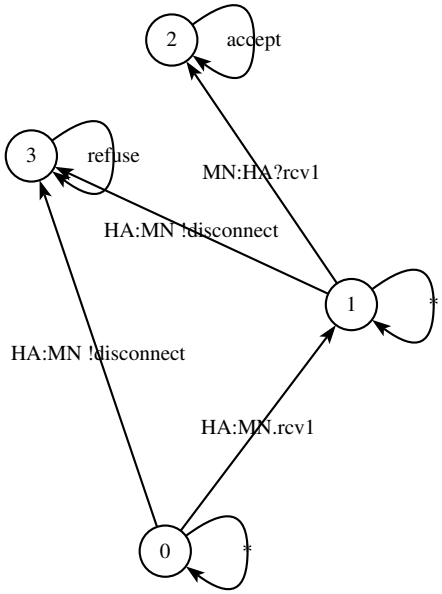


Figura 5.9: Objetivo de teste da propriedade P3 (fórmula 5.3)

O CTG gerado, que não é mostrado neste documento, contém 79 estados e 102 transições. Neste caso, como o número de caminhos extraídos do modelo para realização do processo não atingiu 100% de cobertura dos caminhos referentes à propriedade, houve uma maior diferença nos números de estados e transições. Um caso de teste gerado produziu 40 estados e 42 transições, possuindo também mais estados e transições que os caminhos inicialmente extraídos. A Figura 5.10 apresenta o IOLTS do caso de teste gerado para a propriedade 5.3 com base no objetivo de teste da Figura 5.9.

Baseando-se nas diferenças dos números de estados e transições obtidos com caminhos, CTG e caso de teste, pode-se concluir que a abstração de informações, com a exclusão

de caminhos, não interferiu na geração de um objetivo de teste representativo e preciso. Como definido pela estratégia de extração de caminhos, informações relevantes não devem ser perdidas, sendo descartadas apenas informações redundantes.

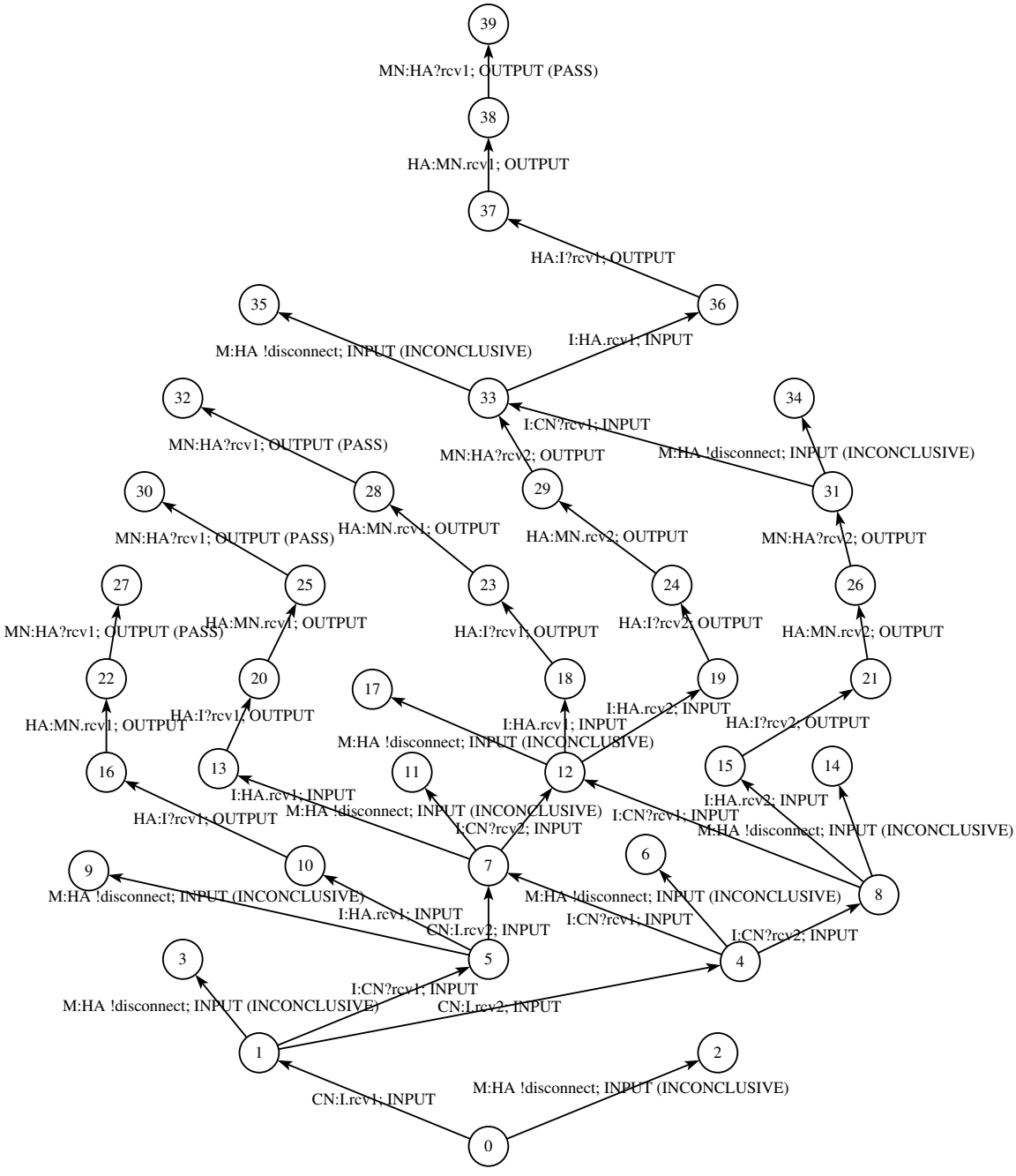


Figura 5.10: Caso de teste referente à propriedade P3

5.2.4 Propriedade 4

Após a migração do *mobile node* deve haver a sua identificação pelo agente da rede estrangeira. Após a identificação, o agente estrangeiro deve notificar o *mobile node* sobre sua nova localização. Assim, um objetivo de teste a ser especificado com base nesta informação é o ato da notificação após a migração. Três proposições atômicas correspondentes a este requisito são definidas. A primeira, denominada p, como "O *mobile node* encontra-se na rede local". A segunda, denominada q, "*home agent* migra". A terceira, definida como r, "O *foreign agent* notifica o *mobile node* sobre sua nova localização". A propriedade CTL correspondente é definida pela fórmula 5.4.

$$EU(p, EU(q, r)) \tag{5.4}$$

Entre os caminhos obtidos a partir desta fórmula, houve um total de 116 estados e 115 transições. A análise algorítmica definiu as ações a serem abstraídas e as que comporiam o objetivo de teste. Os rótulos definidos sobre os estados dos caminhos foram 3: p, q e $accept$. Devido à característica da fórmula, de não haver situações que impeçam a sua satisfação, não foram definidos rótulos com $refuse$.

A análise efetuada permitiu concluir que apenas uma ação específica é necessária para levar de um estado p a um estado q, representando a migração do *mobile node*. Esta transição foi representada nos grafos pelo rótulo "*M:HA !disconnect*", portanto, $L_{p\_q} = \{$*M:HA !disconnect*$\}$. A notificação do *foreign agent* ao *mobile node* representa o alcance do objetivo de teste, devendo levar do estado q ao $accept$. Assim, $refuse$: $L_{q\_accept} = \{$*FA:MN.rcvADV*$\}$. O objetivo de teste resultante é mostrado na Figura 5.11.

O CTG gerado, que não é mostrado neste documento, contém 129 estados e 233 transições. Um caso de teste gerado produziu 60 estados e 65 transições, possuindo mais estados e transições que os caminhos obtidos do modelo. A Figura 5.12 apresenta o IOLTS do caso de teste gerado para a propriedade 5.4 com base no objetivo de teste da Figura 5.11.

5.2.5 Propriedade 5

Caso o *mobile node* migre durante a entrega de uma mensagem pelo *home agent*, esta deve ser encaminhada ao *foreign agent*, que deve realizar o serviço de entrega ao *mobile node*.

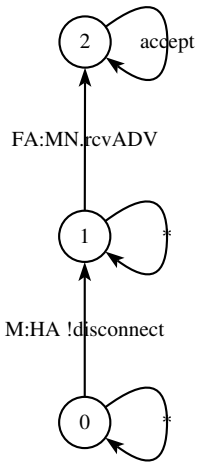


Figura 5.11: Objetivo de teste da propriedade P4 (fórmula 5.4)

Desta maneira, devemos realizar testes com base nas situações em que ao migrar, as mensagens pendentes devem ser entregues pelo *foreign agent*. Três proposições atômicas correspondentes a este requisito são definidas. A primeira, denominada p, como "O *mobile node* encontra-se na rede local". A segunda, denominada q, "*home agent* recebe uma mensagem destinada ao *mobile node*". A terceira, definida como r, "O *mobile node* encontra-se em uma rede estrangeira". A quarta, denominada s, é definida como "O *foreign agent* envia a mensagem ao *mobile node*". A propriedade CTL correspondente é definida pela fórmula 5.5.

$$EU(p, EU(q, r \wedge EF(s))) \tag{5.5}$$

Entre os caminhos obtidos a partir desta fórmula, houve um total de 33 estados e 32 transições. A análise algorítmica definiu as ações a serem abstraídas e as que comporiam o objetivo de teste. Os rótulos definidos sobre os estados dos caminhos foram 5: p, q, r, $accept$ e $refuse$.

A análise efetuada permitiu concluir que apenas uma ação específica é necessária para levar de um estado p a um estado q, representando a tentativa de envio de uma mensagem do *home agent* ao *mobile node*. Esta transição foi representada nos grafos pelo rótulo "*HA:MN.rcv1*", assim, $L_{p\_q} = \{$*HA:MN.rcv1*$\}$. A migração do dispositivo antes de o *home agent* receber uma mensagem destinada a ele não é de interesse, levando do estado p a um estado $refuse$. Assim, $L_{p\_refuse} = \{$*M:HA !disconnect*$\}$. Indesejável também, para este objetivo de teste, é o recebimento da mensagem pelo *mobile node* ainda em

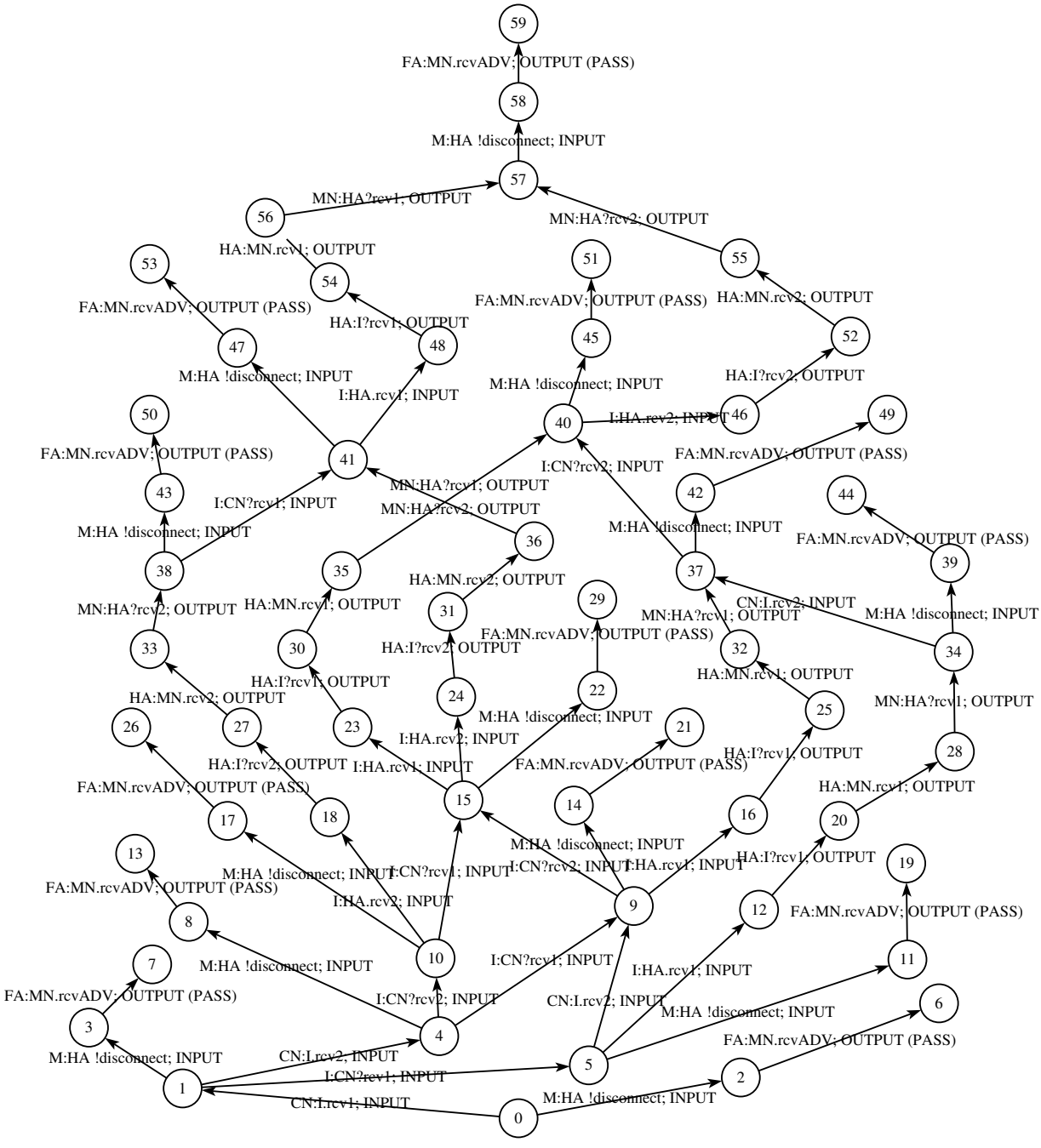


Figura 5.12: Caso de teste referente à propriedade P4

sua rede de origem. Assim, o recebimento da mensagem leva de um estado q ao $refuse$: $L_{q\_refuse} = \{$*MN:HA?rcv1*$\}$.

As situações almejadas com este objetivo de teste implicam na migração do *mobile node* antes da entrega final da mensagem a este. Desta forma, a migração do dispositivo representa a mudança de um estado q a um estado r, sendo, $L_{q\_r} = \{$*M:HA !disconnect*$\}$. Não foram encontradas no modelo, situações em que o *mobile node* retorna à rede de origem antes receber a mensagem que deveria ser entregue pelo *foreign agent*, não havendo, portanto, transições que causem a mudança de um estado r ao $refuse$. Finalmente, o recebimento da mensagem através do *foreign agent* representa o alcance do objetivo de teste, onde, $L_{q\_accept} = \{$*MN:FA?rcv1*$\}$. O objetivo de teste resultante é mostrado na Figura 5.13.

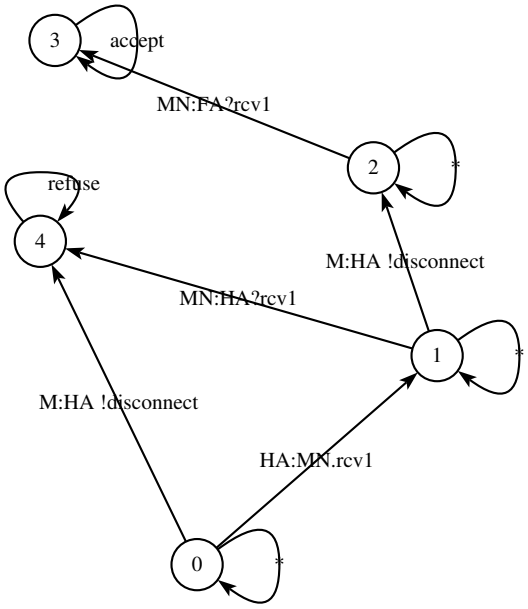


Figura 5.13: Objetivo de teste da propriedade P5 (fórmula 5.5)

O CTG gerado, que não é mostrado neste documento, contém 93 estados e 134 transições. Um caso de teste gerado produziu 51 estados e 56 transições, possuindo mais estados e transições que os caminhos obtidos do modelo. A Figura 5.14 apresenta o IOLTS do caso de teste gerado para a propriedade 5.5 com base no objetivo de teste da Figura 5.13.

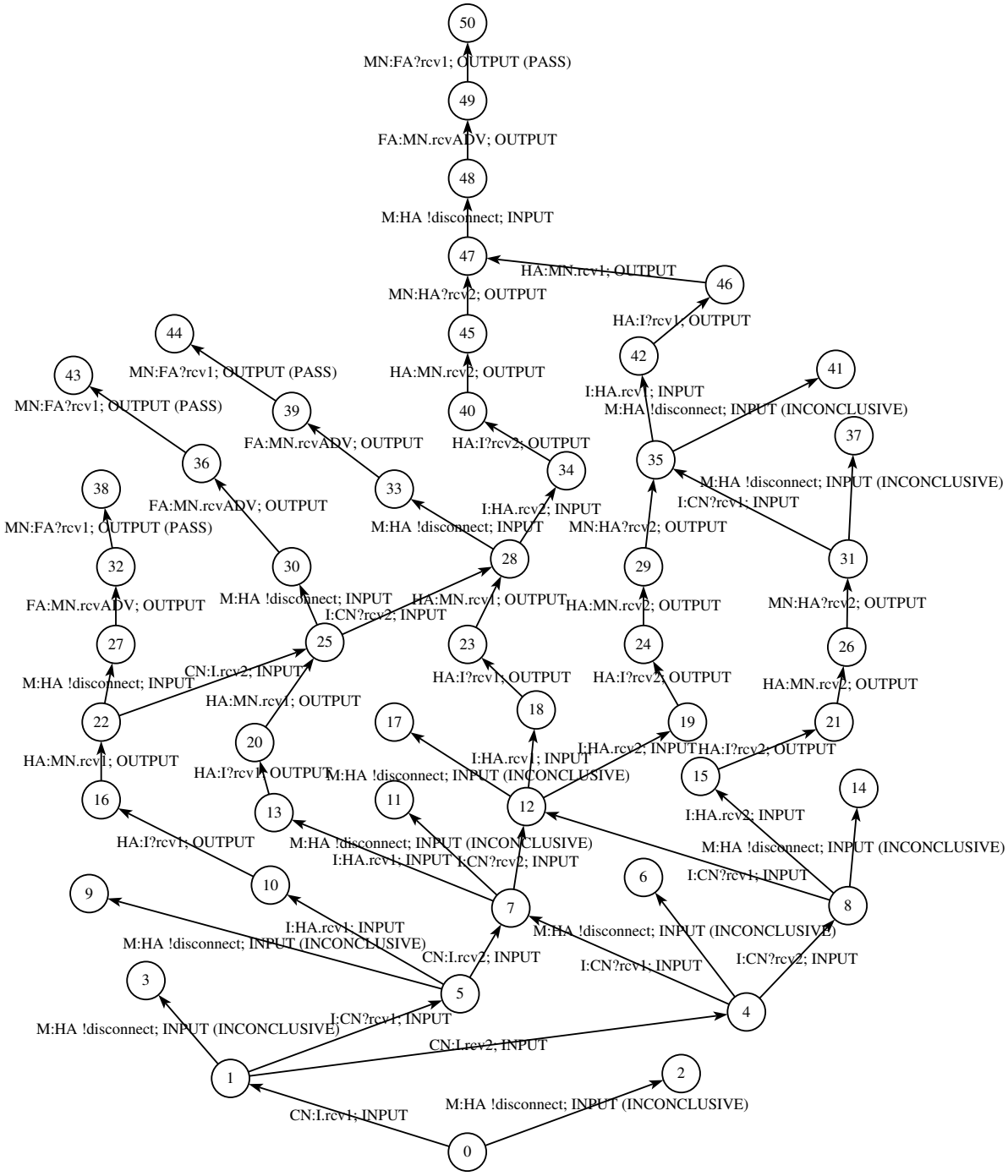


Figura 5.14: Caso de teste referente à propriedade P5

5.3 Considerações Finais

Neste capítulo apresentamos um estudo de caso aplicando a técnica apresentada no Capítulo 3 em conjunto com a implementação de um protótipo (Capítulo 4) a um modelo do protocolo IP Móvel. Foi demonstrada a geração de objetivos de teste com base em propriedades

especificadas por fórmulas CTL e posterior geração de casos de teste com TGV. A análise dos casos de teste e dos CTG's gerados durante o estudo de caso possibilitou a avaliação do funcionamento da técnica de geração de objetivos de teste proposta.

Durante a avaliação dos CTG's pôde-se concluir que a sua cobertura sobre o modelo foi completa, ou seja, atingiu a exaustividade sobre o modelo em relação à propriedade CTL especificada. Em relação à representação simplificada dos caminhos extraídos do modelo e da estratégia de extração, a cobertura de estados finais destes (i.e. exemplos e contra-exemplos) foi de 100%. Entretanto, a cobertura dos CTG's foi além da obtida com a estratégia, que teve como objetivo extrair somente um caminho por estado final, excetuando os que apresentassem transições ainda não coletadas (*cf.* Capítulo 4). Os CTG's alcançaram cobertura total sobre os caminhos relativos às respectivas fórmulas no modelo. Os caminhos extraídos do modelo para a geração dos objetivos de teste compuseram assim, subconjunto dos que foram cobertos pelos casos de teste, definidos pelos CTG's, que compuseram conjunto de casos de teste e-exaustivos em relação ao modelo e às propriedades.

Além de e-exaustivos, os casos de teste gerados pelo TGV são e-efetivos, de acordo com a relação ioco [JJ04]. Garante-se assim, que qualquer implementação não conforme com a especificação será detectada e que, apenas implementações corretas passarão nos testes. No entanto, tal quantidade de casos de teste implica na impossibilidade de sua implementação e execução. Faz-se necessário assim, um mecanismo de seleção dos casos de teste mais adequados para realização dos testes sobre uma IUT real. Este procedimento de seleção de subconjunto dos casos de teste implica em menor rigor na e-exaustividade. Porém, a e-efetividade assegurada aos casos de teste pelo rigoroso processo de geração destes, garante qualidade ao processo de testes, que, apesar de não assegurar correção da implementação, pode aumentar a confiabilidade desta e evita a rejeição de implementações corretas.

O fato de os casos de teste gerados serem e-exaustivos mostra a representatividade do objetivo de teste especificado pela técnica apresentada. A abstração provida pelo objetivo de teste contém informações suficientes para uma eficiente geração de casos de teste que possam corresponder fielmente à fórmula CTL especificada em relação ao modelo.

Capítulo 6

Conclusão

O trabalho apresentado neste documento propõe uma técnica para a geração automática de objetivos de teste para sistemas reativos com base em técnicas de verificação de modelos. O procedimento base parte da especificação de fórmulas CTL para posterior geração dos objetivos de teste, na forma de LTS, por intermédio de algoritmos adaptados da verificação de modelos.

Como suporte à técnica, foi apresentada a implementação de um protótipo, realizada com base na adaptação sobre os algoritmos do verificador de modelos Veritas. Os algoritmos de busca implementados pelo Veritas, objetos da adaptação, baseiam-se nos conectivos EU, EG e EX, dos quais é possível derivar outros tipos de fórmulas (e.g. baseadas em quantificadores universais). Para a realização da adaptação definiu-se uma estratégia para a coleta de informações, representadas por exemplos e contra-exemplos relativos à propriedade CTL, de forma a obter exaustão sobre o modelo. A estratégia tem como objetivo estabelecer os caminhos do modelo a serem extraídos, de modo a permitir a manipulação de um menor número de informações, com um menor número de caminhos do que o contido no modelo, sem prejuízo para a síntese do objetivo de teste.

Um estudo de caso foi realizado sobre o modelo do protocolo IP Móvel para demonstrar o funcionamento da técnica. As análises comparativas realizadas sobre o volume de dados (e.g. número de estados e transições) obtidos pelos caminhos extraídos do modelo, bem como CTG's e casos de teste gerados pelo TGV, possibilitaram a avaliação da técnica e de seus objetivos de teste. A geração de casos de teste a partir dos objetivos de teste providos pela técnica, seguindo a teoria relacionada, apresentada na Seção 2, pode produzir conjuntos

de casos de teste e-exaustivos e e-efetivos em relação ao modelo e à propriedade especificada.

Propriedades que devem ser representadas por fórmulas compostas e/ou aninhadas são contempladas pela técnica. Possibilita-se assim, a sua aplicação aos padrões de fórmulas mais utilizados [DAC99]. Propriedades de vivacidade (i.e. *liveness*) [Lam02], que geralmente implicam em execuções infinitas (e.g. fórmulas baseadas em EG) são representadas de forma finita pelos objetivos de teste gerados. Entretanto, o processo de teste com a utilização de propriedades definidas na verificação de modelos sofre de restrições em relação ao tamanho dos casos de teste gerados. Execuções infinitas são impraticáveis para teste. Técnicas de geração de casos de teste baseados nesta técnica, ao utilizarem propriedades de vivacidade devem prover mecanismos para a limitação dos casos de teste. Tais mecanismos podem basear-se na limitação do número de iterações (e.g. em caso de auto-laços), do tamanho dos casos de teste gerados, ou até mesmo na especificação de limites de tempo para sua execução.

6.0.1 Contribuições

Considerando o importante papel dos objetivos de teste para o teste orientado a propriedades, a escassez de técnicas e ferramentas de automatização do processo de geração dos mesmos e seu conseqüente impacto nos custos e eficiência do teste, a principal contribuição deste trabalho caracteriza-se com a definição de uma técnica de geração automática de objetivos de teste.

Além da técnica, foram definidos algoritmos a serem implementados por ferramentas, bem como uma estratégia para a coleta e análise de informações sobre o modelo a ser utilizada na adaptação de verificadores de modelos. A estratégia definida possibilita a sua utilização por diferentes verificadores de modelos e diferentes formalismos.

A definição de um procedimento para a utilização de lógica temporal ramificada como formalismo base para a especificação dos objetivos de teste permite maior nível de abstração durante o processo. Fórmulas de lógica temporal são artefatos mais concisos, mais fáceis de especificar e manter do que sistemas de transições rotuladas. Porém, apesar de ser um formalismo mais propício para a descrição de propriedades, exige conhecimento técnico que implica custos com o treinamento de uma equipe.

Uma vez que propriedades utilizadas na verificação de modelos são especificadas por

meio de formalismos de lógica temporal, como CTL, a reutilização destas propriedades para o teste pode ser realizada de forma direta, constituindo um importante mecanismo formal para definição dos objetivos de teste. Possibilita-se ao teste maior rigor, tanto na definição de suas propriedades, quanto na geração dos respectivos casos de teste.

Assim como diversos trabalhos têm se baseado na aplicação de técnicas de verificação de modelos para a geração de casos de teste, como em [JJ04; dVT98], a eficiência alcançada na geração de objetivos de teste mostra a viabilidade de ampla integração das técnicas. Contribui-se assim, com uma maior integração entre equipes de verificação e validação, provendo ligação direta entre as etapas de verificação de modelos e teste. Uma maior integração entre as diferentes equipes de desenvolvimento permite maior comunicação, com conseqüente possibilidade de aumento de produtividade e qualidade do *software*.

6.0.2 Trabalhos Futuros

O desenvolvimento de técnicas de teste com o auxílio de métodos formais tem sido objeto de parte importante das pesquisas da área de verificação e validação. Apesar dos avanços alcançados com o desenvolvimento de técnicas e ferramentas, como as de geração de casos de teste (e.g. [JJ04; dVT98]), diversos trabalhos podem ser desenvolvidos no contexto do trabalho proposto neste documento de modo a contribuir com sua evolução. Dentre eles, podemos citar:

- A aplicação da técnica de geração de objetivos de teste com base em especificações de lógica temporal LTL [CGP99] tem sua motivação na característica complementar de expressividade entre os formalismos CTL e LTL. Propriedades que não podem ser especificadas por meio de fórmulas CTL poderão ser especificadas em LTL, aumentando a capacidade de abrangência e o poder de expressividade na especificação de propriedades para a geração de objetivos de teste.

- Este trabalho baseia-se na representação explícita de espaços de estados, sendo assim, acometido pelas limitações causadas com o problema da explosão de espaços de estados [CGP99], do qual limita-se a verificação de modelos baseada em representações explícitas. Como uma abordagem a este problema, a representação simbólica [Mcm93] tem sido utilizada na geração de casos de teste ([CJRZ02;

FTW05]). Seguindo esta linha de pesquisa, a adaptação da tecnica aqui proposta para a utilização de espaços de estados de representação simbólica pode ser investigada em futuros trabalhos.

- Outras abordagens que têm sido utilizadas para a geração de casos de teste como a exemplo da abordagem de máquinas de estados finitos estendidas [PBG04] e a utilização de reconhecedores de linguagens, como palavras de tamanho infinito (e.g. [Rab72; Büc62]), a exemplo da linha seguida em [FMP04]. Não há, na literatura atual, técnicas de geração de objetivos de teste baseadas nestas abordagens, podendo ser assim, objeto de outros trabalhos futuros.

Bibliografia

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

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT, Vol.2*, pages 99–119, 1991.

[Bin03] R. V. Binder. *Testing object-oriented systems*, volume 1. Addison-wesley, 2003.

[Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings Logic, Methodology and Philosophy of Sciences 1960*, Stanford, CA, 1962. Stanford University Press.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Cho78] T. S. Chow. Test design modelled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[CJRZ02] D. Clarke, T. Jeron, V. Rusu, and E. Zinovieva. STG – A Symbolic Test Generation Tool. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 2002.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[dAGdFG04] F. V. de A. Guerra, J. C. A. de Figueiredo, and D. D. S. Guerrero. Timed extension of an object oriented petri net language. In *Simpósio Brasileiro de Métodos Formais - SBMF 2004*, pages 132–148, Recife - Brasil, Novembro 2004.

[Dee91] S. Deering. Rfc 1256:ICMP router discovery messages, September 1991. Status: Proposed Standard.

[dVT98] R. G. de Vries and J. Tretmans. On-the-fly conformance testing using spin. In E. Najm G. Holzmann and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the Spin Model Checker, ENST 98 S 002, Paris, France. Ecole Nationale Supérieure des Télécommunications*, pages 115–128, November 1998.

[dVT01] R. G. de Vries and J. Tretmans. Towards formal test purposes. In *Proc. 1st International Workshop on Formal Approaches to Testing of Software (FATES), Aalborg, Denmark*, pages 61–76, August 2001.

[EFW01] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of conformance test suites for protocols with verification technology. *Science of Computer Programming*, 29:123–146, 1997.

[FMP04] J. Fernandez, L. Mounier, and C. Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing, Proceedings of FATES*

2003, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163, Montreal, Canada, 2004. Springer.

[FTW05] L. Frantzen, J. Tretmans, and T. A. C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES'04*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[FvBK+91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, 1991.

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

[GH99] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ACM SIGSOFT Software Engineering Notes*, volume 24 of *Software Engineering Notes*, pages 146–162, November 1999.

[Gue02] D. D. S. Guerrero. *Redes de Petri Orientadas a Objetos*. PhD thesis, Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba – Campus II, Campina Grande, Paraíba, Brasil, apr 2002.

[Hel97] K. Heljanko. Model checking the branching time temporal logic CTL. Technical Report A45, Helsinki University of Technology, 1997.

[HLU03] O. Henniger, M. Lu, and H. Ural. Automatic generation of test purposes for testing distributed systems. In *Formal Approaches to Software Testing, Proceedings of FATES'03*, volume 2931 of *Lecture Notes in Computer Science*, pages 178–191. Springer, October 2003.

[HN04] A. Hartman and K. Nagin. 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, 2004. ACM Press.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[HRV+03] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES), Montreal, Canada*, October 2003.

[JCE96] F. Shneider J. Callahan and E. Easterbrook. Automated software testing using model-checking. In *In Proceedings of the SPIN Workshop*, August 1996.

[Jen92] K. Jensen. *Coloured Petri Nets 1: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, Berlin, Alemanha, 1992.

[JJ04] C. Jard and T. Jéron. 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, October 2004.

[Kat99] J-P. Katoen. *Concepts, Algorithms and Tools for Model Checking*. Friedrich-Alexander-Universitaet Erlangen-Nuernberg, 1999.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Mcm93] K. L. Mcmillan. Symbolic model checking. In *Kluwer Academic Publishers*, 1993.

[MS01] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*, volume 1. Addison-wesley, 2001.

[MSM05] P. D. L. Machado, D. A. Silva, and A. C. Mota. Towards property oriented testing. In *Simpósio Brasileiro de Métodos Formais - SBMF 2005*, pages 2–16, Porto Alegre - Brasil, Novembro 2005.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[PBG04] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in efsm testing. *IEEE Trans. Softw. Eng.*, 30(1):29–42, 2004.

[PEAM98] P. E. Black P. E. Ammann and W. Majursky. Using model checking to generate tests from specifications. In IEEE Computer Society, editor, *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, November 1998.

[Per02] C. Perkins. Rfc 3344:IP mobility support for IPv4, aug 2002. Status: Proposed Standard.

[Pnu77] A. Pnueli. The temporal logic of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, 1977.

[Rab72] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA, 1972.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, LNCS 1945, pages 338–357. Springer Verlag, November 2000.

[RGdF+04] C. L. Rodrigues, F. V. Guerra, J. C. A. de Figueiredo, D. D. S. Guerrero, and T. S. Morais. Modeling and verification of mobility issues using object-oriented petri nets. In *Proc. of 3rd International Information and Telecommunication Technologies Symposium (I2TS2004)*, 2004.

[Rod04] C. L. Rodrigues. Verificação de modelos em redes de petri orientadas a objetos. Master's thesis, Coordenação de Pós-Graduação em Informática - COPIN, Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brasil, feb 2004.

[RS95] S. Romanenko and P. Sestoft. Moscow ml owner's manual, 1995.

[SEG+98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink - Putting SDL-based test generation into practice. In *In: Testing of Communicating Systems (Editors: A. Petrenko,N . Yevtuschenko), volume 11, Kluwer Academic Publishers, 1998*, June 1998.

[Sil05] T. M. Silva. Simulação automática e geração de espaço de estados de modelos em redes de petri orientadas a objetos. Master's thesis, Coordenação de Pós-Graduação em Informática - COPIN, Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brasil, september 2005.

[SJPLP99] C. J. Trammell S. J. Prowell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering - Technology and Process*. Addison-Wesley, 1999.

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.

[VL93] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In E. Best, editor, *CONCUR'93: Proc. of the 4th International Conference on Concurrency Theory*, pages 447–461. Springer, Berlin, Heidelberg, 1993.