Integrando Testes de Domínios aos Testes Baseados em Máquinas Finitas de Estados Estendidas

Flávio Rogério Uber

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Integrando Testes de Domínios aos Testes Baseados em Máquinas Finitas de Estados Estendidas

Flávio Rogério Uber[1]

Outubro de 2001

Banca Examinadora:

- Eliane Martins (Orientadora)
- Profa. Sandra C. P. F. Fabbri
  Departamento de Computação (UFSCar)
- Prof. Ricardo O. Anido
  Instituto de Computação (IC-UNICAMP)
- Prof. Edmundo M. Madeira (Suplente)
  Instituto de Computação (IC-UNICAMP)

[1]Financiado pela CAPES

Integrando Testes de Domínios aos Testes Baseados em Máquinas Finitas de Estados Estendidas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Flávio Rogério Uber e aprovada pela Banca Examinadora.

Campinas, 19 de Novembro de 2001.

Eliane Martins

Eliane Martins (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 19 de novembro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Profª. Drª. Sandra Camargo Pinto Ferraz Fabbri
UFSCar

Prof. Dr. Ricardo de Oliveira Anido
IC - UNICAMP

Profª. Drª. Eliane Martins
IC - UNICAMP

Agradecimentos

Agradeço a Deus que me dá forças para conquistar meus objetivos e sempre guia meus passos.

Todo o resto da minha vida será pouco para agradecer meus pais, Alcides e Sueli. O apoio emocional e principalmente financeiro nesses anos é algo que somente pais podem fazer pelos filhos. Que Deus os proteja e traga-lhes muita felicidade. Obrigado!

Aos meus irmãos, Ana Paula e Renato que também sofreram indiretamente com todas as dificuldades que surgiram nesse tempo e merecem toda paz e felicidade que Deus possa lhes proporcionar.

Um agradecimento especial à Simone, que soube enfrentar os momentos em que a saudade se fez necessária e ainda teve forças para enxugar as lágrimas e usar as palavras certas de incentivo. Obrigado.

É claro que mesmo com todo esse apoio, essa dissertação não seria possível sem os companheiros de república: Danillo, Gerson, Luciano, Sandro e Júnior. Mesmo com os desentendimentos, nossa união sempre foi muito forte e tenho certeza que foi fundamental para que cada um conseguisse seu objetivo. O meu muito obrigado e que Deus nos mantenha sempre próximos e traga-lhes muita saúde e felicidade.

Outros amigos também participaram dessa conquista. A todos os companheiros do IC a minha gratidão e os votos que tenham muito sucesso em todos os momentos.

Um agradecimento especial à minha orientadora, Eliane Martins, que foi mais que uma orientadora. Soube compreender as dificuldades, principalmente nesse último ano e acreditou que seria possível. Obrigado pelos conselhos que são e ainda serão muito mais fundamentais na minha carreira.

Resumo

As máquinas finitas de estados estendidas (MFEE) são muito utilizadas para a especificação de protocolos. Por essa razão existem muitas técnicas baseadas nesse modelo, usadas nos testes de conformidade, os quais servem para determinar se uma determinada implementação satisfaz a sua especificação. Com o objetivo de dar suporte à geração de testes a partir de MFEE foi construída no Instituto de Computação da UNICAMP a ferramenta CONDADO. Até então essa ferramenta implementava os métodos de testes de transição, para cobrir a parte de controle. Com relação à parte de dados (referente ao formato das interações bem como aos valores dos parâmetros dessas interações) eram utilizados os testes de sintaxe e os testes baseados em classes de equivalência.

Este trabalho implementa uma extensão à CONDADO visando melhorar a estratégia de geração de dados de teste por parte desta ferramenta. Em vez dos testes baseados em classes de equivalência são usados testes de domínios que consideram os predicados associados às transições da MFEE . Com isso foi reduzido o número de casos de teste correspondentes a caminhos não executáveis. Acredita-se ainda na melhora do potencial para detecção de falhas com dados de teste próximos aos limites do domínio, o que deverá ser analisado futuramente.

Abstract

Extended Finite State Machine (EFSM) are much utilized in protocol specification. For testing it there are many techniques, used in conformance testing. Conformance testing of communication protocols aims at verifying that the external behaviour of a protocol implementation complies with th protocol specification.

To test EFSM based specifications was developed CONDADO tool, in Computing Institute (UNICAMP). This tool combines different specification-based test methods: transition testing for the control part of a protocol and syntax and equivalence partitioning fot the data part.

This work develops a CONDADO extension, intending to improve data generation. Equivalence partitioning will be substituted by domain testing to consider the predicates associated with transitions. With domain testing will be decreased the number of case tests corresponding to non-executable paths.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Nos dias atuais, a atividade de teste de software é tratada como um elemento crítico de qualidade de um sistema e, como tal, deve ser desenvolvida com critérios e objetivos bem definidos, de forma a reduzir custos e aumentar a confiabilidade do produto final.

Sendo assim, é importante selecionar adequadamente o método de seleção de casos de teste. Os métodos são classificados em duas categorias principais: os testes baseados na especificação (testes caixa preta) e os testes baseados na implementação (testes caixa branca) (veja Capítulo 2).

Tendo o modelo base (uma especificação ou implementação) o processo de teste é dividido em etapas: geração, execução e análise dos resultados. Este trabalho está concentrado na etapa de geração de casos de teste, que constituem dos dados de teste (entradas) e as saídas esperadas.

É importante ressaltar que a aplicação de testes visa descobrir falhas em um programa. Assim, quando a execução de um programa apresenta o comportamento esperado, não se pode garantir que não existam falhas e sim, que o conjunto de testes aplicados foi ruim (veja Capítulo 2), ou ainda, que os casos de teste foram gerados de maneira inadequada.

1.1 Motivação

Este trabalho tem como ponto de partida a ferramenta CONDADO, um trabalho desenvolvido no Instituto de Computação da UNICAMP para testes em protocolos de comunicação [Sab98]. Este considera a máquina finita de estados estendida (MFEE) como a forma de especificação usada para protocolos, já que é a forma mais comum de fazê-la [BP94]. Desta forma, utilizando-se exclusivamente de técnicas caixa preta, foi implementado um

método para testes em MFEE.

Foram implementadas as técnicas de teste de transição de estados (para a parte de controle do protocolo) e os testes de sintaxe e partição de equivalência (para a parte de dados do protocolo). Contudo, o teste de partição de equivalência se mostrou pouco eficiente, já que não considera os predicados da máquina e portanto não faz nenhuma análise acerca da executabilidade do caminho referente a um caso de teste gerado.

1.2 Objetivos

Desta forma a geração de testes para a parte de controle de protocolo já havia sido desenvolvida, restando portanto, uma geração de testes mais eficiente para a parte de dados, fundamental para análise dos predicados associados às transições da máquina.

Para isso o objetivo é a utilização dos testes de domínios, uma técnica originalmente proposta para aplicação em programas mas que pode ser adaptada para máquinas finitas de estados estendidas. Assim sendo, o desafio deste trabalho é escolher uma dentre as técnicas de teste de domínios e adaptá-la para ser aplicada na geração de parâmetros que satisfaçam os predicados associados às transições.

1.3 Organização da Dissertação

O capítulo 2 apresenta conceitos relacionados a teste de software para compreensão do trabalho e do seu contexto, ou seja, quais são os critérios a serem observados na aplicação de testes e as técnicas mais relacionadas com o trabalho.

O capítulo 3 trata dos testes baseados em máquinas finitas de estados estendidas, incluindo definições, abordagens para teste e as dificuldades relacionadas a esse tipo de teste. Serão apresentados também alguns trabalhos que se propõe a resolver o mesmo tipo de problema, destacando os méritos e restrições de cada um.

O capítulo 4 apresenta os testes de domínios, algumas definições e um exemplo que ilustra a aplicação da estratégia. Foram descritas também as estratégias estudadas, ressaltando a técnica escolhida e as justificativas para esta escolha.

O capítulo 5 trata da estratégia implementada. Em primeiro lugar é apresentada a ferramenta CONDADO com o objetivo de situar o contexto do trabalho e na sequência é mostrado um exemplo para ilustrar a aplicação da técnica e o resultado obtido.

O capítulo 6 faz uma avaliação da estratégia apresentada no capítulo 5 mostrando quais os ínidices obtidos após algumas aplicações da estratégia.

Por fim o capítulo 7 traz algumas conclusões obtidas após a realização do trabalho, acerca de sua implementação, dos resultados obtidos, e ainda, dos trabalhos que poderão ser desenvolvidos como sequência deste.

Capítulo 2

Teste de Software

2.1 Introdução

Desde a construção dos primeiros softwares, sob encomenda e de pequeno porte, até os dias atuais, com sistemas distribuídos, técnicas avançadas e usuários exigentes, ocorreram muitas mudanças de filosofia na especificação de um software, devido à crescente preocupação com a qualidade deste. Essa preocupação inclui o desenvolvimento da área de teste de software, que cresce constantemente pelo fato de grande parte do custo do processo de desenvolvimento de um software estar relacionado com essa etapa. Segundo Beizer [Bei90] as atividades de teste consomem no mínimo metade do trabalho gasto na produção de um programa.

Teste é o processo de executar um programa com a intenção de encontrar falhas. Se um caso de teste ocasiona um defeito em um programa, o caso de teste obteve sucesso, caso contrário ele fracassou [Mye79]. Como é difícil prevenir falhas, a atividade de teste se concentra em descobrir falhas. Saber que um programa está incorreto não implica conhecer a falha. Diferentes falhas podem se manifestar de modo semelhante [Bei90].

É importante para a compreensão do restante do texto, uma definição dos termos falha, erro e defeito. Nesse trabalho será utilizado o padrão IEEE, cuja terminologia em português foi definida por [LF87]. Desta forma, quando um programador comete um engano, por exemplo na utilização de um operador aritmético, esse engano é a causa da introdução de uma *falha*(*fault*) em instruções do programa. Quando essa instrução é executada, propagando essa falha para o restante da execução, tem-se de um *erro*(*error*). E finalmente, quando um resultado errado, decorrente de um erro, é apresentado ao usuário, ocorre um *defeito*(*failure*).

As técnicas de teste de software não demonstram a ausência de falhas, ou seja, não se pode determinar a total correção de um software através da aplicação de testes. Os testes são projetados para a descoberta de falhas, se bem projetados e bem sucedidos. Pressman [Pre95], considera que a detecção de falhas graves indica a falta de qualidade e/ou confiabilidade do software. Se todas as falhas detectadas são facilmente corrigidas, indica que, ou a qualidade do software é aceitável ou os testes são inadequados e finalmente, se nenhuma falha é encontrada os testes não alcançaram o resultado desejado, ou seja, outra técnica deveria ter sido aplicada. Segundo o paradoxo do pesticida de Beizer [Bei90], se for mostrado que o programa tem boa qualidade, isto é, já foi submetido a um número razoável de testes atendendo critérios adequados, as falhas ainda existentes no programa podem não ser mais detectáveis por testes. Outra técnica de verificação deve então ser utilizada.

Com a descoberta de falhas através da aplicação de testes, pretende-se identificar se as funções estão obtendo respostas que estão de acordo com a especificação, ou seja, se os requisitos estão sendo cumpridos.

Nesse capítulo, será apresentada a terminologia utilizada para teste de software e o processo de teste, ou seja, como são aplicados e em que momento da construção de um sistema. Em seguida serão definidas as modalidades de testes, e onde se situa o trabalho no contexto de testes.

2.2 Atividades de Teste

As fases de teste são definidas em relação às fases de desenvolvimento do software em questão. Quando um módulo do sistema é construído realiza-se o teste de unidade, com o intuito de verificar se este satisfaz à sua especificação. No momento de compor os subsistemas conforme a arquitetura do sistema, são realizados os testes de integração com o intuito de encontrar falhas de interfaceamento entre os módulos e subsistemas. Os testes de validação visam determinar se o software satisfaz aos requisitos especificados na fase de análise e os testes de sistema visam exercitar o sistema como um todo, incorporando todos os componentes (hardware e software) para determinar se o sistema completo satisfaz à sua especificação.

A figura 2.1, adaptada de [Pos96] mostra a sequência de atividades do processo de teste. O processo é iniciado com a seleção de casos de teste, formado por dados (entradas) de teste e pelas saídas esperadas. Os casos de teste são derivados a partir de um modelo de base, que pode ser uma especificação, o próprio código ou uma representação

das falhas na especificação ou no código, decorrentes de erros cometidos ao longo do desenvolvimento. Nessa fase, o grande problema encontrado é a impossibilidade de se testar todas as entradas possíveis de um programa. Assim, deve-se escolher um critério que, para o problema em questão, tenha a maior probabilidade de encontrar o maior número possível de falhas. Isso é um desafio para a atividade de teste, uma vez que é difícil se determinar essa probabilidade quando se está projetando os casos de teste. Trata-se de um problema intratável, uma vez que não existem recursos computacionais para isso, e indecidível, já que não existe um algoritmo capaz de decidir se um programa termina para todas as entradas fornecidas. Em geral, não se pode testar exaustivamente um programa, ou seja, testar cada combinação possível de valores de entrada, portanto um programa sempre conterá falhas residuais.

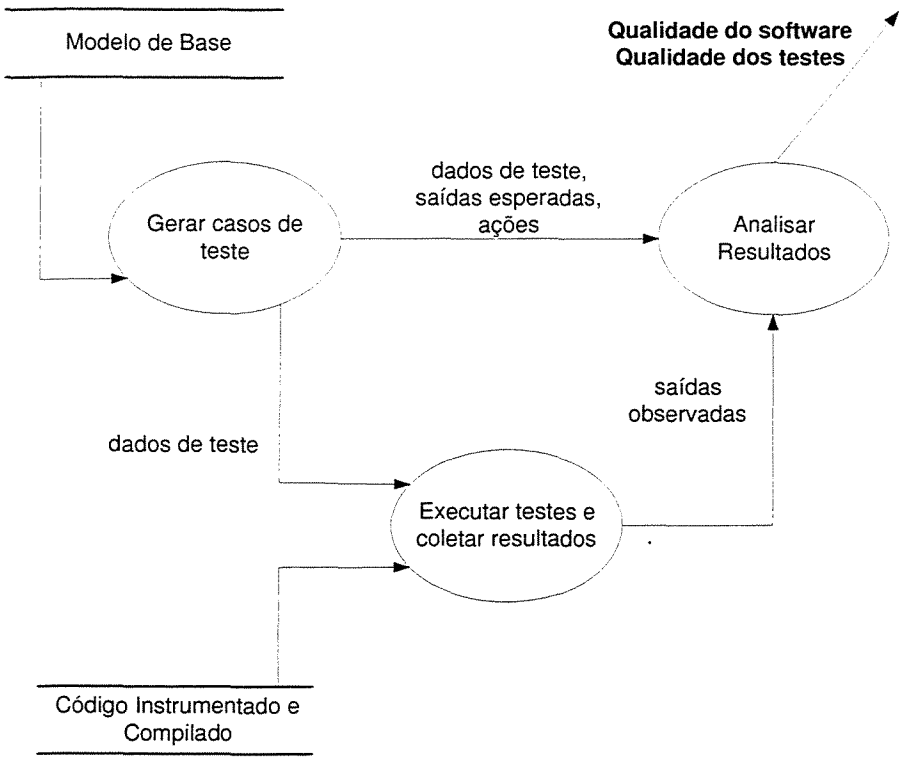

Figura 2.1: Processo de Teste

Em seguida é realizada a execução dos testes. Para isso deve ser construído um ambiente que permita executar o software com os dados de teste selecionados e coletar as saídas observadas. Nessa fase também existem algumas dificuldades. Uma delas é a questão da

testabilidade. Entende-se por testabilidade, o quanto um sistema ou componente facilita o estabelecimento de critérios de teste (veja seção 2.3). E também o quanto ele facilita o desempenho de testes para determinar se esses critérios foram encontrados. Além disso, o número de casos de teste aplicados a um software é muito grande e deve ser repetido várias vezes, ao menos a cada alteração. Portanto, aplicá-los manualmente, além de não ser prático é altamente propenso a erros, daí a necessidade de automatização dos testes.

Por fim, é realizada a análise dos resultados, que compreende tanto a avaliação da qualidade do software quanto da qualidade dos testes aplicados. Avaliar a qualidade do teste implica em determinar sua eficiência em encontrar falhas no software, e o quanto do modelo de base os casos de teste permitiram exercitar. Para essa fase é necessária a existência de um oráculo que possa analisar o resultado obtido. Um oráculo, nas atividades de teste, é um programa, processo ou pessoa que especifica a saída esperada de um conjunto de testes de acordo com o objeto a ser testado. Um oráculo humano, além de propenso a erros, pode não conseguir obter manualmente o resultado de muitos programas, e portanto não pode saber se o resultado obtido está correto. Um oráculo automatizado é obtido a partir da especificação, e a dificuldade é garantir que a espeficicação seja confiável.

2.3 Critério de Adequabilidade e Cobertura

As condições que devem ser satisfeitas quando os testes forem aplicados são chamadas de critérios de adequabilidade, ou simplesmente critérios de teste.

Um critério de adequabilidade C determina um conjunto finito S de elementos do modelo que devem ser exercitados durante os testes, denominados elementos requeridos ou requisitos de teste. Assim, por exemplo, se o modelo usado para o teste é um modelo do programa, um critério C poderia ser: todas as instruções devem ser exercitadas pelo menos uma vez; S seria o conjunto de todas as instruções de um programa.

Critérios de adequabilidade são úteis antes e depois da aplicação dos testes. Antes da aplicação dos testes, são utilizados para se determinar o que deve ser exercitado, ou seja, os requisitos de teste. Depois da aplicação dos testes têm-se uma medida da qualidade dos testes. Esta qualidade pode ser medida pela cobertura dos testes com relação ao critério C, que é normalmente dada na forma de porcentagem: (número de elementos de S exercitados)/(número total dos elementos de S). Desta forma, o valor da cobertura, pode ser definido antes, para indicar quantos casos de teste devem ser gerados, ou depois, para obter a cobertura atingida pelos testes.

2.4 Critérios de Teste de Software

Existem duas modalidades principais de teste de software: os testes caixa preta, que se preocupam com os requisitos funcionais do software, ou seja, preocupam-se em determinar se todos os requisitos foram implementados, e os testes caixa branca que se baseiam no exame dos detalhes procedimentais, ou seja, testam, entre outos itens, conjuntos de condições e laços de um programa.

2.4.1 Testes Caixa Preta

Os testes caixa preta, também chamados de testes funcionais, preocupam-se com a análise dos valores de entrada e saída de um software, ou seja, se a entrada é aceita de acordo com a especificação e a saída é produzida adequadamente, sem se preocupar com detalhes de implementação.

Existem vários critérios ou métodos para seleção de casos de teste, variando de acordo com a informação representada pela especificação. Alguns exemplos que podem ser citados e que estão mais diretamente relacionados a esse trabalho são:

- **Testes de partição de equivalência**: O domínio de entrada é particionado em classes, e assume-se que as entradas pertencentes a uma determinada classe são equivalentes, ou seja, se uma entrada é capaz de encontrar uma determinada falha, assume-se que todos os outros casos de teste pertencentes à mesma classe são capazes de encontrar a mesma falha. Esse tipo de teste é realizado em duas etapas: na primeira, identificam-se as classes de equivalência válidas e inválidas. Na segunda, definem-se os casos de teste para cada uma das classes.

- **Testes de sintaxe**: Em algumas aplicações, como compiladores e protocolos de comunicação, os dados de entrada possuem uma sintaxe muito bem definida, e os testes de sintaxe são usados para se obter casos de teste a partir das notações utilizadas para esses dados (ASN.1, BNF, expressões regulares). Com a utilização dos testes de sintaxe podem ser encontradas três tipos de ações incorretas: o não reconhecimento de uma cadeia de caracteres correta; a aceitação de uma cadeia incorreta, ou ainda a ocorrência de falhas na aceitação ou rejeição de uma cadeia.

- **Máquinas Finitas de Estado (MFE)**: É um modelo que serve para mostrar a variação de estados de um sistema ao longo do tempo e pode ser definida como uma máquina hipotética que pode estar em um único estado, dentre um conjunto

finito de estados possíveis, em um determinado momento [Dav78]. A máquina pode mudar de estado em resposta a uma entrada (ou evento), podendo produzir uma saída. Tanto o novo estado quanto a saída são funções unicamente do estado atual e do evento ocorrido. É o modelo no qual está baseado o trabalho e portanto será tratado com mais detalhes no Capítulo 3.

2.4.2 Testes Caixa Branca

Os testes caixa branca, também chamados de testes estruturais, consideram a estrutura de implementação para a escolha dos casos de teste. Grande parte dos critérios existentes para teste caixa branca representam o programa na forma de um grafo de controle. O grafo de controle é definido por G= (N, A, n_0, n_f), sendo N o conjunto de nós, que representam um comando ou bloco de comandos, A representa o conjunto de arestas, que representa o fluxo de controle entre dois comandos ou blocos, n_0 representa o nó inicial e n_f representa o nó final. A existência de um único nó inicial e um único nó final não significa que não possa haver mais de um ponto de entrada ou de saída no programa, basta que todos os pontos de entrada estejam ligados a n_0 e todos os pontos de saída estejam ligados a n_f.

Uma extensão dos grafos de fluxo de controle, são os grafos de fluxo de dados que representam a relações entre os pontos do programa em que as variáveis são definidas e o ponto onde elas são usadas.

Uma definição de variável ocorre quando um valor é atribuído a uma variável. Em geral, a definição ocorre se a variável está do lado esquerdo de uma atribuição, em um comando de entrada ou ainda como parâmetro de saída em chamada de procedimentos [VMJC97]. Um uso de variável ocorre quando uma variável já definida é referenciada. São dois tipos de uso: c-uso, quando a variável é utilizada em uma computação, e p-uso, quando a variável é usada em um predicado. O objetivo é testar dependências entre cada definição de uma variável e seu(s) uso(s) subsequente(s).

2.4.3 Testes Baseados em Restrições

A abordagem de testes baseados em restrições foi proposta inicialmente para programas mutantes [DO91]. Um programa mutante tem alterações sintáticas em relação ao original. Quando esses programas apresentam resultados diferentes do programa original o mutante é considerado morto.

A proposta original era representar as condições sob as quais programas mutantes são mortos, ou seja, as restrições algébricas que os dados devem satisfazer. Por exemplo, uma variável x deve ter seu valor compreendido entre 10 e 100. Estabelecidas estas restrições podem-se gerar automaticamente os dados para satisfazê-las.

Se fosse feita uma generalização desse conceito, todas as técnicas que impõem restrições algébricas sobre os testes e geram dados automaticamente para satisfazer essas restrições poderiam ser considerados testes baseados em restrições.

O princípio é que, impondo condições necessárias sobre os dados para que algum estado do programa seja alcançado, podem-se gerar automaticamente esses dados com maior probabilidade que o estado desejado seja alcançado. Em uma máquina de estados, antes que um determinado estado seja alcançado, na maioria das vezes é necessário que, nos estados anteriores, os dados tenham satisfeito algumas condições, como por exemplo, ser maior ou menor que uma variável de estado, ou então igual a um valor pré-definido e assim por diante. Enfim, é necessário que os dados tenham atendido algumas restrições.

Esse trabalho tem como objetivo justamente considerar essas condições, que em uma máquina são descritas através dos predicados, para que um determinado caminho dentro da máquina de estados seja exercitado. Com algumas adaptações, que serão vistas nos próximos capítulos, engloba o conceito de testes baseados em restrições.

2.5 Teste de Protocolos

Devido à natureza reativa dos protocolos de comunicação e à necessidade de se especificar a ordem em que as interações ocorrem, as máquinas finitas de estados são muito utilizadas para a especificação destes. Nesse trabalho, o foco é a geração de dados de teste para protocolos de comunicação especificados através de máquinas finitas de estados estendidas. Dessa forma, nesta seção serão apresentadas algumas características de testes de protocolos e nos próximos capítulos serão discutidos os aspectos relacionados às máquinas de estados.

Protocolos de comunicação são regras que governam a comunicação entre diferentes componentes dentro de um sistema distribuído [BD97]. As implementações de um protocolo são testadas de encontro à especificação dele para garantir a compatibilidade entre uma implementação e a especificação. A verificação desta condição é conhecida como teste de conformidade. Uma implementação I está em conformidade com uma especificação E, se para cada sequência de entrada para a qual E é definida, I produz a mesma sequência de saída conforme for definida por E [PvB96].

Portanto, de acordo com os testes de conformidade, os protocolos são tratados como caixa preta, conforme descrito na seção 2.4, onde a implementação é dada como uma caixa preta e somente seu comportamento observável pode ser testado de encontro ao comportamento da especificação.

Durante os testes de conformidade, dados são enviados e recebidos da implementação. Os dados de saída da implementação sob teste são comparados aos dados da especificação. As entradas fornecidas e as saídas esperadas são descritas em um conjunto de casos de teste.

Como existem limitações para se testar exaustivamente uma implementação, recomenda-se que sejam aplicados quatro tipos de teste [Ray87]:

- **testes de interconexão básica:** detectam casos graves de não conformidade como, por exemplo, quando uma implementação sob teste não consegue se conectar com outra, ou se as principais características do padrão do protocolo não foram implementadas.
- **testes de capacidade:** verificam se a capacidade observável de uma implementação está de acordo com os requisitos de conformidade estática do protocolo.
- **testes de comportamento:** fornecem testes tão completos quanto possíveis das necessidades dos requisitos especificados pelo padrão do protocolo, em tempo de execução, dentro das aptidões da implementação.
- **testes de resolução de conformidade:** fornecem diagnósticos mais definitivos possíveis para resolver se uma implementação satisfaz requisitos particulares. Esses testes não são padronizados.

2.6 Considerações Finais

Este capítulo apresentou conceitos de teste de software necessários para a compreensão dos demais capítulos. Foi feita uma introdução das técnicas que serão tratadas, no sentido de classificá-las com relação à teoria de teste de softaware. As técnicas mais envolvidas no desenvolvimento do trabalho serão tratadas com detalhes nos próximos capítulos.

Capítulo 3

Testes Baseados em Máquinas Finitas de Estados Estendidas

3.1 Introdução

As máquinas finitas de estados estendidas (MFEE) são muito utilizadas para especificação de sistemas reativos que permitem definir a ordem com que as interações ocorrem. Essa característica é verificada, por exemplo, em protocolos de comunicação e outros sistemas de tempo real. Dessa forma, este trabalho está inserido no contexto de testes de MFEE com o objetivo de gerar casos de testes para protocolos de comunicação.

As máquinas finitas de estados estendidas estendem o conceito das máquinas finitas de estados (MFE) com a inclusão de predicados e ações relacionados às variáveis da máquina.

Dessa forma, neste capítulo serão tratadas definições de MFE e MFEE. Na sequência serão apresentadas as abordagens para teste em MFEE, quais as dificuldades encontradas nesse tipo de teste incluindo a questão da executabilidade e os trabalhos existentes que se propõe a tratar esse tipo de dificuldade.

3.2 Definições

3.2.1 Máquinas Finitas de Estados (MFE)

Uma máquina finita de estados (MFE) é um modelo que serve para mostrar a variação de estados de um sistema ao longo do tempo. É uma máquina hipotética que pode estar em um único estado em um dado momento. Em resposta a uma entrada, a máquina gera, ou

não, uma saída e muda de estado [Dav78]. A máquina possui um estado inicial e um ou mais estados finais. Normalmente, em protocolos de comunicação, o estado final é igual ao estado inicial.

Uma MFE pode ser definida de maneira formal como sendo uma 5-tupla <S,s_0,I,O,g> onde:

S é um conjunto não vazio de estados;

s_0 é o estado inicial;

I é um conjunto finito de entradas;

O é um conjunto finito de saídas;

g é a função de transição de estados definida como: g: S x I $\rightarrow$ S x O.

As primeiras estratégias de geração de testes baseados em MFE normalmente tratavam apenas o fluxo de controle do protocolo. Os métodos utilizados para isso são baseados em técnicas de teste de transição de estados.

O teste de transição de estados requer que cada transição seja exercitada pelo menos uma vez [Bei90]. As falhas que podem ser encontradas são:

1. Uma transição errada foi selecionada.
2. A transição tem falhas, que podem ser de transferência (o próximo estado escolhido é errado) ou de saída (a ação errada foi executada).
3. Uma saída errada foi gerada para uma determinada transição.
4. Faltam transições na implementação. Com isso não se pode tratar um evento em um determinado estado especificado pela MFE.

No entanto, como as MFEs não representam todos os aspectos do protocolo como predicados e ações relacionadas às transições, torna-se difícil realizar testes para o fluxo de dados do protocolo. Para representar essas informações, as máquinas finitas de estados estendidas MFEEs são muito utilizadas, e portanto são necessárias técnicas para realizar testes nesse tipo de modelo.

3.2.2 Máquinas Finitas de Estados Estendidas (MFEE)

Em relação às MFEs, as Máquinas Finitas de Estados Estendidas (MFEE) representam também variáveis de contexto, ações relacionadas a essas variáveis e predicados que operam sobre essas variáveis. Assim sendo, uma transição pode ser dividida em duas partes:

a parte da condição e a parte da ação. A parte da condição contém um evento de entrada e/ou um predicado. O predicado pode ser definido como uma expressão booleana que pode envolver variáveis bem como os parâmetros de entrada. A parte da ação pode conter eventos de saída e um número de comandos envolvendo as variáveis.

Uma transição é disparada quando a parte da condição é satisfeita. Quado uma transição T é disparada, a ação correspondente a essa transição é executada e a MFEE muda para o estado destino. Uma ação define as saídas produzidas e pode alterar os valores das variáveis associadas à transição.

É possível ainda a existência de transições espontâneas em uma MFEE. Essas transições não dependem de nenhuma entrada e estão associadas a um predicado. Quando o predicado é verdadeiro a transição é disparada. Essa situação é muito comum para representar a existência de *time out*, ou seja, a indicação de que a máquina deve mudar de estado quando um tempo máximo é atingido.

Formalmente, uma MFEE pode ser definida como uma 8-tupla <S, s_0 , I, O, V, P, A, g> onde:

S é um conjunto não vazio de estados;

s_0 é o estado inicial;

I é um conjunto finito de entradas;

O é um conjunto finito de saídas;

V é um conjunto de variáveis;

P é o conjunto de predicados que operam sobre as variáveis;

A é o conjunto de ações relacionadas às variáveis;

g é a função de transição de estados definida como: g: S x I x P(V) $\rightarrow$ S x O x A(V).

Nas próximas seções serão discutidas as abordagens para teste em MFEE e as dificuldades relacionadas a esse tipo de teste.

3.3 Geração de teste para MFEE

Como as MFEE introduzem em sua notação aspectos relacionados aos dados de uma máquina, as técnicas de transição de estados usadas em MFE tornam-se insuficientes para testar as máquinas estendidas. Nesta seção serão apresentadas algumas abordagens para teste de máquinas finitas de estados estendidas, diferentes da abordagem utilizada na CONDADO, e em seguida os problemas relacionados com esse tipo de teste.

3.3.1 Abordagens para teste em MFEE

Conforme descrito anteriormente, os testes baseados em MFEE necessitam portanto cobrir tanto o fluxo de controle quanto o fluxo de dados. Os diferentes estudos existentes procedem nesse caso de diferentes formas [BD97]:

1. Separar fluxo de controle e fluxo de dados: Nesta abordagem, os aspectos controle e dados são separados na especificação. O fluxo de controle é representado por uma MFE, onde técnicas de teste de transição de estados são aplicadas. Através de critérios de testes orientados ao fluxo de dados, (como por exemplo todos-DU-caminhos, onde o objetivo é testar todo caminho livre de laços entre uma definição global de uma variável e todos os seus usos [VMJC97]), testam-se os parâmetros das primitivas de entrada e saída e as variáveis de contexto [BDAR97].

2. Transformar a especificação na forma normal: Uma especificação contendo apenas as chamadas transições na forma normal não contém, em suas ações, instruções que influenciem o fluxo de controle do protocolo, tais como instruções condicionais (*if*, *case*) e instruções de repetição (*while*, *repeat*). Para transformar uma especificação em uma equivalente na forma normal cria-se uma nova transição para cada caminho distinto dentro de uma ação[UY91]. Estando na forma normal, podem ser aplicadas técnicas de transição de estados.

3. Expandir a MFEE: Uma MFEE pode ser vista como uma notação compactada de uma MFE. A transformação pode então ser feita eliminando-se as variáveis de contexto através da criação de um conjunto de novos estados e novas transições. Os novos estados são formados pela combinação dos estados da MFEE com os valores das variáveis de contexto. Assim, a habilitação de uma transição dependerá somente do estado corrente e da entrada. Essa abordagem é chamada "desdobramento" (*unfolding*). Sua utilização pode levar a uma explosão de estados, o que dificulta sua manipulação. Além disso, quando uma variável não tem um domínio finito não é possível aplicar o desdobramento a uma MFEE [CS87].

4. Transformar a especificação em gramática: A transformação da especificação em gramática permite representar todos os aspectos do protocolo (controle, dados, predicados e ações) usando uma única notação. A partir disso, técnicas de teste baseadas em sintaxe podem ser aplicados [UP83]. O problema desta abordagem é a necessidade de um trabalho extra no sentido de construir esta notação para representação do protocolo.

Para a ferramenta CONDADO, foi utilizada uma abordagem sugerida por Poston [Pos96], que combina testes dirigidos aos eventos e testes dirigidos aos dados, apenas com a utilização de técnicas caixa preta.

Os testes dirigidos aos dados se concentram em encontrar falhas de manipulação de valores de dados como números, listas ou strings e falhas na manipulação de estruturas de dados como arrays, registros ou arquivos. Este tipo de teste é dividido em cinco categorias: análise de limite, de partição, de domínio, de lista e de sintaxe [Pos96]. As categorias de testes dirigidos a dados implementadas pela CONDADO são análise de partição e análise de sintaxe descritas na seção 2.4.1.

3.3.2 Dificuldades

Embora uma MFEE tenha mais elementos para representar um protocolo, isso torna a atividade de teste mais complexa. Essa complexidade surge justamente pela dificuldade de se levar em conta os predicados.

Em um programa, os predicados muitas vezes torna impossível determinar dados de entrada para executar uma determinada instrução, ocosionando o que se chama de **problema da executabilidade**. Segundo [Ver97] um caminho é não executável se não existir um conjunto de valores para as variáveis de entrada, parâmetros e variáveis globais que causem a sua execução.

Esse conceito pode ser utilizado em uma MFEE, já que as transições da máquina são executáveis. Uma transição é executável se em um determinado caminho S, os parâmetros da interação de entrada e os valores das variáveis associadas à cada transição **t** são tais que o predicado habilitador de **t** é verdadeiro.

Dessa forma, o problema da executabilidade é bastante pertinente nos testes baseados em MFEE, e esse trabalho visa diminuir seus efeitos no método implementado pela CONDADO (veja capítulo 5).

3.3.3 Trabalhos Correlatos

Por se tratar de um problema intrínseco aos testes de MFEE, vários trabalhos já foram propostos no intuito de oferecer métodos que minimizem o problema da executabilidade e possam gerar casos de teste mais eficazes. Alguns desses trabalhos serão apresentados abaixo.

Huang [HLJ95] apresenta um método para gerar e manipular a executabilidade dos casos de teste usando análise de fluxo de dados. A partir das entradas definidas pelo usuário, verifica-se os estados que são alcançavéis através de um análise de alcançabilidade. Com essa informação é possível concluir se um caminho é executável ou não com as

um caminho na máquina em questão. Quando o algoritmo falha, o caso de teste correspondente a esse caminho é descartado, reduzindo assim, o número de casos de teste correspondentes a caminhos não executáveis. Diferentemente das estratégias citadas, os caminhos já foram gerados pela CONDADO, o que se pretende é tentar gerar dados para torná-lo executável.

3.4 Considerações Finais

Este capítulo tratou os testes de máquinas finitas de estados estendidas (MFEE). Maior destaque foi dado ao problema da executabilidade, inerente a esse tipo de teste, e presente na CONDADO. Foram apresentados alguns trabalhos desenvolvidos no sentido de minimizar o problema da executabilidade nos testes de MFEE. Os próximos capítulos apresentarão o teste de domínio e como ele será aplicado para minimizar este problema na CONDADO.

Capítulo 4

Teste de Domínios

4.1 Introdução

A estratégia de Teste de Domínios original, proposta por White e Cohen [WC80], é uma forma de encontrar dados de entrada capazes de testar um caminho de um programa, satisfazendo algumas condições nos comandos de fluxo de controle (*if*, *while* e etc). Portanto, os Testes de Domínios são classificados, em sua origem, como teste caixa branca.

Beizer [Bei90] cita a possibilidade de se aplicar os Testes de Domínios como uma técnica caixa preta, mas não apresenta uma estratégia para tal. Foram estudadas várias técnicas e escolheu-se aquela que melhor poderia ser adaptada para testar caminhos de uma MFEE, ou seja, que pudesse ser aplicada como uma técnica caixa preta.

Neste capítulo, primeiramente, serão apresentadas as definições de alguns termos importantes para a compreensão da estratégia e, em seguida, serão tratadas as estratégias de Teste de Domínios estudadas, com ênfase na estratégia de Hajnal e Forgács, que foi escolhida para ser adaptada ao problema em questão.

4.2 Termos Importantes

Na apresentação das técnicas de Teste de Domínios, é necessário o conhecimento de alguns termos próprios desse tipo de teste. Os conceitos mais importantes serão apresentados segundo a definição de [Bei95]:

Variável de Entrada: Valor numérico apresentado ao sistema sob teste para ser processado. Em geral, o sistema processa **n** variáveis de entrada.

Vetor de entrada: Conjunto de valores das **n** variáveis de entrada, sendo um valor

para cada variável. Chamado também de vetor de teste ou ponto de teste.

Espaço de Entrada: Espaço n-dimensional no qual o vetor de entrada pode ser definido, onde **n** é o número de variáveis de entrada.

Domínio: Um subconjunto do espaço de entrada sobre o qual o processamento é definido. Deseja-se saber se é possível determinar se um vetor de entrada está ou não dentro de um domínio.

Caminho: Para teste caixa branca é definido como sequência de instruções de um programa que são processadas a cada execução. Para MFEE, um caminho é uma sequência de transições.

Limites do Domínio: Definido na forma de desigualdades algébricas.

Desigualdades Limite: Expressão algébrica sobre as variáveis de entrada que define que pontos do espaço de entrada pertencem ao domínio de interesse. Por exemplo, a desigualdade $\mathbf{x \geq 7}$ define que os pontos no domínio de interesse incluem os valores iguais ou maiores que 7 para a variável x.

Limite Fechado: Um limite de um domínio é fechado se os pontos sobre o limite estão incluídos no domínio de interesse. Por exemplo, em $\mathbf{x \geq 7}$, o limite é fechado porque o valor 7 está incluído no domínio de interesse.

Limite Aberto: Um limite de domínio é aberto se os pontos sobre o limite não pertencem ao domínio de interesse, eles pertencem, portanto, ao **domínio adjacente**. Assim, em uma desigualdade definida como $\mathbf{7 \leq x < 44}$, o domínio de interesse é fechado em **7** e aberto em **44**, já que o ponto x=44 não pertence ao domínio de interesse.

Correção Coincidente: Limitação inerente aos testes de domínios. Refere-se ao fato de que dados de teste podem fazer com que um caminho errado seja seguido, mas coincidentemente produzam saídas correspondentes ao caminho correto.

Caminho Perdido: Outra limitação comum. Trata do fato de um predicado requerido não aparecer no programa a ser testado.

4.3 Estratégias para Teste de Domínios

Nesta seção serão apresentadas algumas estratégias de Teste de Domínios estudadas com o intuito de se encontrar uma técnica que melhor se adaptasse ao problema em questão: gerar dados que permitissem testar caminhos em uma máquina finita de estados estendida.

As estratégias estudadas foram: a estratégia original, proposta por White e Cohen [WC80]; a estratégia proposta por Clarke, Hassell e Richardson [CHR82] a estratégia, denominada simplificada, para testes de domínios proposta por Jeng e Weyuker [JW94];

e finalmente a técnica de Hajnal e Forgács [HF98] que melhor se adaptou às necessidades deste trabalho.

4.3.1 Estratégia de White e Cohen

A estratégia de White e Cohen [WC80] parte da definição que uma falha de domínio ocorre quando, a partir de um dado de entrada específico, um caminho errado é seguido devido a uma falha no fluxo de controle de um programa.

Comandos de fluxo de controle em um programa particionam o espaço de entrada em um conjunto de domínios mutuamente exclusivos, cada um deles correspondente a um caminho do programa. O domínio é constituído de dados que fazem com que o caminho correspondente seja executado.

White e Cohen impõem uma restrição sobre a linguagem de programação utilizada: a ocorrência de cadeias de caracteres e a chamada de sub-rotinas impossibilitam a aplicação dessa estratégia. No entanto, eles admitem serem características importantes de uma linguagem e que portanto precisam ser consideradas.

As variáveis são divididas em três classes: as variáveis de entrada, que aparecem em um comando de leitura; as variáveis de saída, que aparecem em um comando de escrita; e as variáveis de programa que não se enquadram em nenhuma das classes anteriores.

Os predicados são definidos com o uso de operadores relacionais: $<, >, =, \leq, \geq, \neq$ sendo que os predicados formados por mais de um operador relacional são denominados **predicados compostos**.

Para aplicação da técnica é necessário que o espaço de entrada seja contínuo e não discreto, já que é necessário a escolha de pontos ON e OFF. Um ponto ON está, segundo White e Cohen, sobre a borda do domínio de interesse e um ponto OFF está a uma pequena distância da borda. Devem ser escolhidos dois pontos ON e um OFF, sendo que a projeção do ponto OFF na borda deve estar entre os dois pontos ON. A Figura 4.1 mostra a identificação de domínio de interesse e domínio adjacente, e os pontos ON e OFF.

Para bordas fechadas os pontos ON pertencem ao domínio de interesse e o ponto OFF pertence a um domínio adjacente. Para bordas abertas, os pontos ON pertencem a um domínio adjacente e o ponto OFF pertence ao domínio de interesse.

Considerando esses pontos como um caso de teste, se qualquer um deles levar a uma saída incorreta, então existe um erro na borda, ou seja, os limites desse domínio foram implementados de forma incorreta.

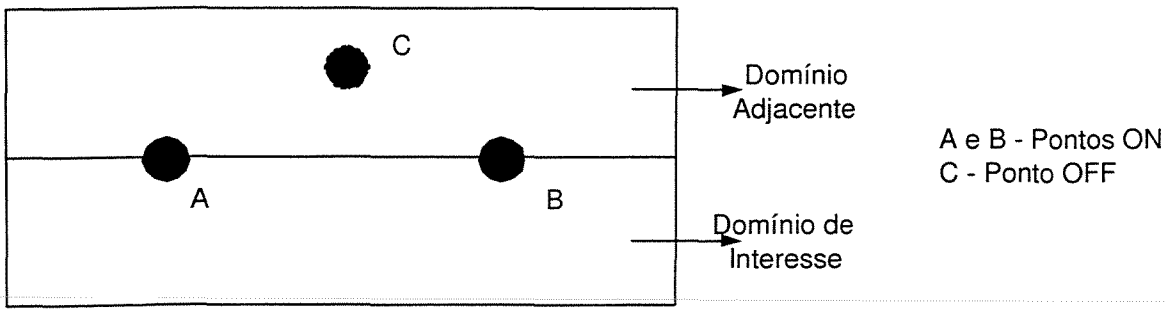

Figura 4.1: Pontos ON e OFF

A Figura 4.2 mostra um exemplo de aplicação da técnica, adaptada de [WC80]. Considerando que I e J sejam as variáveis de entrada. A variável I tem seus valores compreendidos entre -3 e 4, enquanto J tem seus valores compreendidos entre -2 e 6. Assim, tem-se o espaço de entrada definido pela figura 4.2(a). Considere o seguinte trecho de código:

Read I, J
C = I + 2*J - 1;
if C > 6
then D = C - I; (1)
else D = C + I - J + 2; (2)
if D = C + 2
then E = I; (3)
else F = J; (4)

O primeiro predicado (*C>6*) é definido em termos de uma variável de programa (C) e precisa ser definido em termos de variáveis de entrada. Como C=I+2*J-1, facilmente pode ser feita a substituição para que I+2*J-1>6. Atribuindo o menor valor possível para I (-3), tem-se -3+2*J-1>6, ou seja, J>5. Em seguida é atribuído o maior valor possível para I (4) e tem-se que 4+2*J-1>6, ou seja, J>1.5. Dessa forma, tem-se a reta ligando os pontos (-3,5) e (4,1.5) mostrado na figura 4.2(b). Assim, quando se deseja obter valores para I e J que tornem o predicado *C>6* verdadeiro basta escolher valores que estão acima da reta. Consequentemente os valores abaixo da reta tornam o predicado falso.

O segundo predicado é mais complexo, já que o valor de D depende do caminho seguido no predicado anterior. Assim, tendo seguido o caminho (1), D=C-I, enquanto o caminho (2) provoca D=C+I-J+2. Desta forma, considerando o espaço acima da reta na figura 4.2(b) C-I=C+2, e portanto para esse caso I=-2, ocasionando a divisão representada

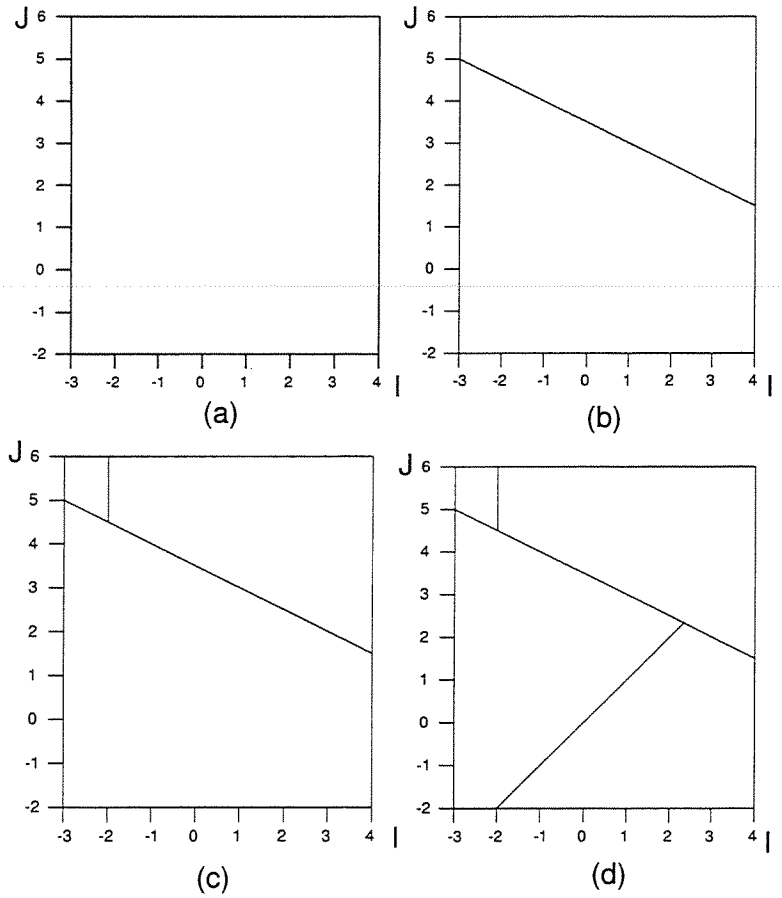

Figura 4.2: Exemplo de Aplicação de Testes de Domínios

pela figura 4.2(c). Da mesma forma, considerando o espaço abaixo da reta, tem-se C+I-J+2=C+2, que resolvido proporciona I=J, sendo obtida a figura 4.2(d). Assim, para quatro caminhos possíveis no programa tem-se quatro domínios facilitando a tarefa de se escolher os valores de entrada.

4.3.2 Estratégia de Clarke et. al.

O trabalho de Clarke et. al. é baseado na estratégia de White e Cohen, com o intuito de tentar eliminar as restrições impostas pela estratégia original.

A principal diferença se refere à correção coincidente. A solução para este problema,

proposta por Clarke et. al. é que, ao se testar os pontos ON de um domínio, caso sejam obtidos resultados corretos, deve-se verificar o domínio adjacente. Em caso de igualdade no resultado investiga-se a possibilidade da borda pertencer a este domínio adjacente, ou seja, é necessário testar mais pontos entre o limite dos dois domínios para se ter uma conclusão sobre a borda.

White e Cohen consideram que o espaço de entrada deve ser contínuo para que o ponto ON possa ser escolhido sobre a borda. Se esse fosse discreto, o limite do domínio poderia não ter nenhum ponto que pertencesse ao espaço de entrada. No entanto Clarke et. al. mostram que pontos ON situados na região da borda apresentam respostas idênticas àquelas obtidas quanto os pontos são situados exatamente sobre a borda, o que abre a possibilidade de se ter espaços de entrada discretos.

Por fim, outra difereça significativa é com relação a escolha dos pontos ON e OFF. Ao invés de se escolher n pontos ON para um espaço n-dimensional, ou seja, a quantidade de pontos ON depender do número de variáveis de entrada, Clarke et. al. propõem que sejam escolhidos V pontos ON, onde V é o número de vértices do domínio de interesse e V pontos OFF. Assim, por exemplo, o domínio representado no canto inferior direito da figura 4.2(d) teria 4 pontos ON, um em cada vértice do domínio, e 4 pontos OFF, cada um próximo a um vértice.

4.3.3 Estratégia de Jeng e Weyuker

Jeng desenvolveu um método no intuito de simplificar a aplicação dos testes de domínios. Para isso a primeira conclusão foi com relação aos pontos ON e OFF. Para Jeng, basta que sejam escolhidos um ponto ON e um ponto OFF para que a estratégia tenha resultados semelhantes aos obtidos por outras técnicas.

Elimina definitivamente a necessidade de se escolher os pontos ON exatamente sobre a borda, permitindo assim que se tenha um espaço de entrada discreto.

Jeng também considera pouco eficiente a estratégia de Clarke et. al. para solução da correção coincidente, o que faz com que esta estratégia esteja sujeita a esse problema, ou seja, parte-se do princípio que esta situação não ocorre.

Outra limitação da técnica é a necessidade de se ter um ponto de entrada I_0 que pertença ao domínio de interesse, o que nem sempre é possível de ser obtido em se tratando de teste caixa preta.

Dessa forma, a partir desse ponto I_0 é gerado o ponto ON e OFF para o caminho correspondente. Utilizando-se do ponto OFF, ou seja, outro caminho no programa, são

gerados mais caminhos e é feita a análise de domínios para cada um deles.

4.3.4 Estratégia de Hajnal e Forgács

Hajnal e Fogács propuseram um algoritmo que segue a mesma linha de Jeng. Esse algoritmo parte de um único valor de entrada I_0 que atravessa um caminho conhecido P, e obtém os pontos ON e OFF para esse caminho (se possível), e os demais caminhos, com seus pontos ON e OFF. Numa primeira etapa o valor I_0 é variado exponencialmente até que este ultrapasse os limites do domínio, obtendo assim um ponto ON e outro OFF. Em uma segunda etapa, esses valores são refinados até que eles se aproximem o máximo possível do limite (borda) do domínio.

Isso é feito baseado na definição de Critério de Teste de Bordas [HF98], definido para um predicado arbitrário p:

1. Esse critério é satisfeito para desigualdades se cada borda é testada por dois pontos (ON e OFF) tais que:
   (a) Os pontos devem seguir o mesmo caminho prefixo associado a essa borda, ou seja, o caminho percorrido até se chegar ao predicado p deve ser o mesmo para os dois pontos.
   (b) Se para um dos dois pontos a saída do predicado p é verdadeira, então para o outro ponto a saída de p é falsa.
   (c) Os pontos devem ser tão próximos quanto possível.

2. O mesmo critério é satisfeito para igualdade e diferença, se cada borda é testada por três pontos (OFF-ON-OFF) tais que:
   (a) Os pontos devem seguir o mesmo caminho prefixo associado a essa borda.
   (b) Se para o ponto ON (pontos OFF) a saída do predicado p é verdadeira, então para os pontos OFF (ponto ON) a saída de p é falsa.
   (c) Os pontos OFF devem ser tão próximos quanto possível do ponto ON.
   (d) Os pontos OFF devem estar fora da borda.

Outra característica do método de Hajnal e Forgács é a consideração dos predicados compostos. Considere um predicado que contém quatro expressões relacionais A, B, C e

D, tal como o predicado p = (A AND B) OR (C AND D). Todas as expressões devem ser testadas separadamente. Consideremos a expressão A. Se (C AND D) tiver valor constante e falso e B for verdadeiro, então A -> p (A implica p), ou seja, a expressão A determinará se o predicado p é verdadeiro ou não. Assim, a expressão A pode ser trabalhada em relação à sentença (a) do item 1 descrita acima. O mesmo pode ser feito para as demais expressões.

Com relação ao algoritmo, como dito anteriormente, o valor inicial da variável é variado exponencialmente até ultrapassar os limites do domínio. Isso é feito baseando-se no método de minimização de função [Kor90]. Se aplicarmos um valor de teste t a um predicado p, devemos construir uma função cujo valor é positivo para t. Uma busca exploratória varia esse valor t da seguinte forma: em primeiro lugar localizamos a melhor direção, ou seja, determinamos se é necessário incrementar ou decrementar o valor t para que a função diminua seu valor. Então a variação é feita exponencialmente até que o valor da função torne-se negativo, que se refere ao ponto fora do domínio (ponto OFF). Em seguida o ponto ON (valor inicial da variável) e o ponto OFF obtido, são refinados de modo a satisfazer à sentença (c) do item 2.

As características de cada estratégia está sumarizada na tabela 4.1. O objetivo é ressaltar como cada uma trata os diferentes tipos de predicados, quais são suas limitações e como ela trata o problema da correção coincidente.

De um modo geral, cada uma delas tem a preocupação de eliminar restrições na aplicação de testes de domínios e algumas delas foram resolvidas de uma estratégia para outra. Outra preocupação foi com relação à quantidade de pontos ON e OFF, ou seja, buscou-se sempre métodos menos custosos para determinação desses pontos, e ainda, tentar tornar mais simples a sua utlização.

4.4 Considerações Finais

Este capítulo apresentou o teste de domínios como a estratégia a ser utilizada para minimizar o problema da executabilidade na CONDADO. Foram apresentadas as técnicas de teste de domínios estudadas e um exemplo para ilustrar a aplicação desse tipo de teste em códigos de programas.

Para a escolha da técnica a ser utilizada, além das limitações existentes na técnica, foi considerada a facilidade de adaptação, que seria necessária para a sua aplicação em MFEE. Seguindo este princípio, o algoritmo de Hajnal e Forgács se mostrou mais adequado, já que além de não possuir a maioria das limitações das primeiras estratégias ainda se mostrou

Tabela 4.1: Estratégias de Teste de Domínios

Proposta	Desigualdade	Igualdade / diferença	Limitações	Espaço de entrada	Correção coincidente
White e Cohen	Requer N[1] pontos ON e um ponto OFF	Requer N pontos ON e dois pontos OFF	Não prevê elementos de entrada *array* e nem a ocorrência de predicados não lineares[2]	Apenas espaço de entrada contínuo	Considera a não ocorrência
Clarke, Hassel e Richardson	Requer V[3] pontos ON e V pontos OFF		Apenas predicados lineares	Pode-se adaptar a estratégia para espaços discretos	Propõe uma possível solução para correção coincidente
Jeng e Weyuker	Requer um ponto ON e um ponto OFF	Requer um ponto ON e dois pontos OFF	Dificuldade de Implementação e necessidade de um valor I_0	Considera espaços discretos, já que o ponto ON não precisa estar sobre a borda	Considera a não ocorrência
Hajnal e Forgács	Requer um ponto ON e um ponto OFF	Requer um ponto ON e dois pontos OFF	Necessidade de um valor I_0	Considera espaços discretos	Considera a não ocorrência

[1] Dimensão do espaço [2] Pred. linear possui variáveis com potência=1 [3] Núm. de vértices da borda

mais adequado para ser adaptado. Este algoritmo prevê a execução do programa, e como uma MFEE é um modelo executável é possível a adaptação.

Outra vantagem, em relação ao algoritmo de Jeng e Weyuker, é a possibilidade de suprimir a etapa de obtenção de todos os caminhos da máquina, que é feita em um segundo passo. Como a CONDADO já fornece os caminhos, essa tarefa não é necessária para o problema em questão. Já na estratégia de Jeng, tudo é feito simultaneamente, ocasionando um custo desnecessário, ou de execução, no caso da obtenção dos caminhos que não serão utilizados, ou de implementação, no caso de adaptar o algoritmo de modo a suprimir esta etapa.

Capítulo 5

Estratégia Implementada

5.1 Introdução

Depois de apresentados os testes baseados em MFEE e os testes de domínios, esta seção tratará da estratégia adotada para reunir essas duas abordagens como forma de gerar dados para os parâmetros das interações de entrada e eliminar os casos de teste correspondentes a caminhos não executáveis.

Conforme dito anteriormente, a estratégia de Hajnal e Forgács foi escolhida para ser adaptada aos testes de máquinas de estados. Neste capítulo será mostrado como foi feita essa adaptação e dois exemplos que a ilustre. Antes disso, será apresentada a ferramenta CONDADO para situar a atuação dos testes de domínios na estratégia implementada por ela.

5.2 Ferramenta CONDADO

A ferramenta CONDADO [Sab98] foi desenvolvida com a intenção de implementar uma estratégia que proporcionasse uma cobertura dos aspectos de controle (relacionado às transições) e dados (relacionado às variáveis e parâmetros) de um protocolo de comunicação a partir de técnicas de testes caixa preta, com a utilização dos testes de transição de estados, dos testes de sintaxe e dos testes de partição de equivalência.

A Figura 5.1 mostra como se dá a utilização da CONDADO. A partir de uma especificação em forma de MFEE, esta é transformada para a Linguagem de Especificação de Protocolos (LEP) que serve de entrada para um analisador. Esse analisador converte a máquina em LEP para uma forma intermediária, que serve de entrada para a CONDA-

DO. A ferramenta então converte o código intermediário para cláusulas Prolog. O uso do Prolog se deve, entre outras vantagens, ao seu mecanismo de *backtracking*, que facilita a geração dos casos de teste, pois permite a geração de todas as combinações existentes na especificação.

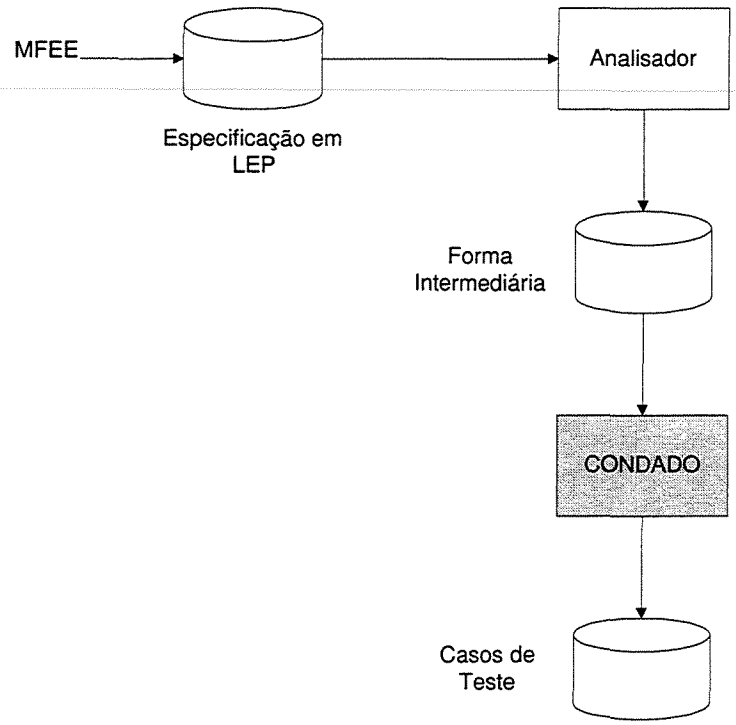

Figura 5.1: Utilização da CONDADO

O usuário pode impor algumas restrições à CONDADO. Essas restrições se referem, por exemplo, às transições que serão testadas, o que é muito útil para testar funcionalidades específicas do protocolo. Outra possível restrição é quanto ao número de vezes que um ciclo será executado.

Estas restrições são a única forma de que o usuário dispõe para tentar garantir a executabilidade de um determinado caminho. Caso o usuário não se preocupe com isso, corre-se o risco da ferramenta gerar casos de teste correspondentes a caminhos não executáveis, como será mostrado a seguir.

Considere a máquina de estados dada pela Figura 5.2 e sua descrição, mostrada na Figura 5.3. A especificação completa da máquina, em LEP é feita em anexo.

Quando é requisitada a geração de todos os casos de teste que exercite as transições

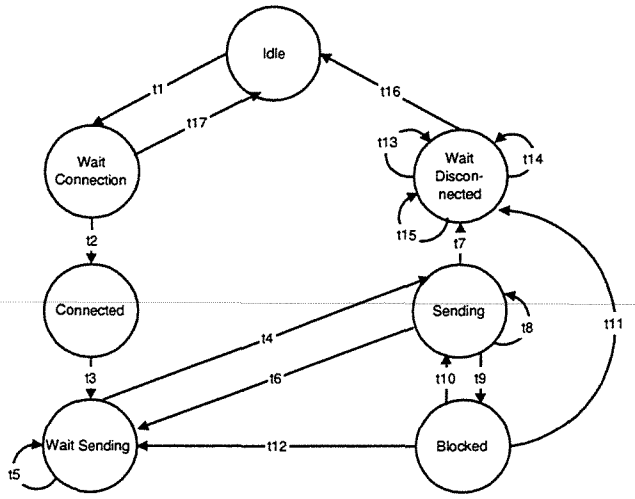

Figura 5.2: Exemplo de uma Máquina Finita de Estados Estendida

t5 e t12, serão obtidos oito. Tomemos como exemplo o caso de teste que exercite as seguintes transições: t1, t2, t3, t4, t6, t4, t9, t10, t9, t12, t5, t4, t7, t13, t14, t15, t16. A única transição, para esse caso de teste, que incrementa a variável *number* é a transição t4. Portanto quando a transição t7 for atingida, t4 terá sido exercitada 3 vezes, e *number* terá valor 3. Para a habilitação de t7, é necessário que *number* seja igual a *number-of-segment*. Logo, trata-se de um caso de teste correspondente a um caminho não executável, já que *number* é uma variável inicializada em 0 e incrementada na transição t4 e *number-of-segment* é uma variável que tem seu domínio de entrada restrito aos valores 2, 4 e 8.

Como atualmente a CONDADO gera valores aleatórios (dentro do domínio de entrada) para as variáveis de entrada, torna-se uma tarefa difícil para o usuário restringir o número de vezes que a transição t4 deve ser exercitada para habilitar a transição t7. Para minimizar este tipo de problema os predicados devem ser considerados para a geração dos casos de teste. Para isso será utilizado o teste de domínios, conforme será mostrado a seguir.

5.3 Estratégia

O algoritmo de Hajnal e Forgács será utilizado, portanto, como um gerador de parâmetros. A sua integração com a CONDADO se dará conforme descrito pela Figura 5.4.

Trans.	Entrada	Predicado	Saída	Ação
t1	U.SENDrequest	-	L.CR	-
t2	L.CC	-	U.SENDconfirm	-
t3	U.DATArequest	-	-	counter:=0 number:=0
t4	L.TOKENgive	-	L.DT	start_timer number:= number+1
t5	L.RESUME	-	-	-
t6	-	expire_timer	L.TOKENrelease	
t7	L.ACK	number=number_of_segment	U.MONITOR_COMPLETE L.TOKEN_RELEASE L.DISrequest	
t8	L.ACK	number<number_of_segment not expire_timer	L.DT	number:= number+1
t9	L.BLOCK	not expire_timer	-	counter:= counter+1
t10	L.RESUME	not expire_timer counter<=blockbound	-	-
t11	-	counter>blockbound	L.TOKENrelease U.MONITOR_INCOMPLETE L.DISrequest	-
t12	-	Expire_timer counter<=blockbound	L.TOKENrelease	-
t13	L.RESUME	-	-	-
t14	L.BLOCK	-	-	-
t15	L.ACK	-	-	-
t16	L.DISrequest	-	U.DISindication	-
t17	L.DISrequest	-	U.DISindication	-

Figura 5.3: Descrição da Máquina Exemplo

Como o algoritmo original foi proposto para aplicação em programas, algumas adaptações foram necessárias.

Além de não existir explicitamente os comandos de decisão, a adaptação mais importante se refere à execução. A geração dos dados requer que o programa seja executado a cada passo para se refinar os valores. Como no problema em questão não existe um programa executável e sim uma máquina de estados foi necessário o desenvolvimento de um simulador para a máquina, ou seja, um procedimento que exercitasse o caminho para o qual se pretende gerar os dados.

Esse procedimento lê, a partir da especificação da máquina, os predicados e ações de um estado. Após testar os predicados e realizar as ações, principalmente aquelas relacionadas com alteração de valores de variáveis, verifica-se qual o estado alcançado e repete-se o processo.

Abaixo é mostrado um primeiro exemplo dos casos de teste gerados pela CONDADO e a atuação do gerador de parâmetros. Se fosse requisitado a CONDADO a geração dos casos de teste que exercitasse as transições t6 e t8, na máquina mostrada pela figura 5.2,

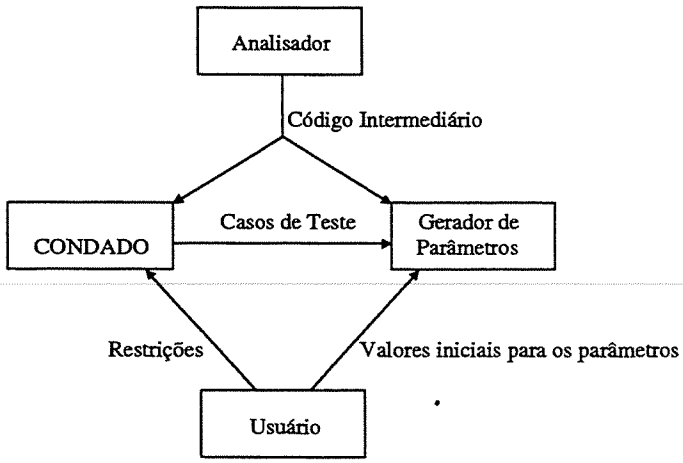

Figura 5.4: Integração com a CONDADO

são gerados 16 casos de teste correspondentes às seguintes sequências de transições:

1. t1,t2,t3,t4,t6,t4,t8,t7,t13,t14,t15,t16
2. t1,t2,t3,t4,t6,t4,t9,t10,t8,t7,t13,t14,t15,t16
3. t1,t2,t3,t4,t6,t4,t9,t10,t9,t12,t4,t8,t7,t13,t14,t15,t16
4. t1,t2,t3,t4,t6,t4,t9,t12,t4,t8,t7,t13,t14,t15,t16
5. t1,t2,t3,t4,t6,t4,t9,t12,t4,t9,t10,t8,t7,t13,t14,t15,t16
6. t1,t2,t3,t4,t8,t6,t4,t7,t13,t14,t15,t16
7. t1,t2,t3,t4,t9,t10,t6,t4,t8,t7,t13,t14,t15,t16
8. t1,t2,t3,t4,t9,t10,t8,t6,t4,t7,t13,t14,t15,t16
9. t1,t2,t3,t4,t9,t10,t8,,t6,t4,t7,t13,t14,t15,t16
10. t1,t2,t3,t4,t9,t10,t9,t12,t4,t6,t4,t7,t13,t14,t15,t16
11. t1,t2,t3,t4,t9,t10,t9,t12,t4,t8,t6,t4,t7,t13,t14,t15,t16
12. t1,t2,t3,t4,t9,t12,t4,t6,t4,t8,t7,t13,t14,t15,t16

13. t1,t2,t3,t4,t9,t12,t4,t6,t4,t9,t10,t8,t7,t13,t14,t15,t16

14. t1,t2,t3,t4,t9,t12,t4,t8,t6,t4,t7,t13,t14,t15,t16

15. t1,t2,t3,t4,t9,t12,t4,t9,t10,t6,t4,t8,t7,t13,t14,t15,t16

16. t1,t2,t3,t4,t9,t12,t4,t9,t10,t6,t4,t8,t7,t13,t14,t15,t16

Tomando como exemplo os casos de teste 1 e 4. Como será visto adiante, o caso 1 corresponde a um caminho não executável (e portanto o algoritmo deve falhar) enquanto para o caso 4 o algoritmo deve conseguir gerar um ponto ON e um ponto OFF.

O formato gerado pela CONDADO leva em conta as entradas e saídas de uma transição e não propriamente o rótulo da transição, assim sendo, as 12 transições exercitadas pelo caso de teste 1, são representadas da seguinte forma:

senddata(U,SENDrequest) recdata(L,CR)
senddata(L,CC) recdata(U,SENDconfirm)
senddata(U,DATArequest,'MlepV',0,8) recdata( )
senddata(L,TOKENgive) recdata(L,DT)
senddata( ) recdata(L,TOKENrelease)
senddata(L,TOKENgive) recdata(L,DT)
senddata(L,ACK) recdata(L,DT)
senddata(L,ACK) recdata(U,MONITOR-COMPLETE) recdata(L,TOKENrelease)
recdata(L,DISrequest)
senddata(L,RESUME) recdata( )
senddata(L,BLOCK) recdata( )
senddata(L,ACK) recdata( )
senddata(L,DISrequest) recdata(U,DISindication)

A transição t3, através da entrada DATArequest recebe os parâmetros de entrada que serão utilizados pelo gerador de parâmetros. Desta forma, como pode ser observado, para o parâmetro *SDU* a CONDADO gerou a sequência de caracteres 'MlepV' enquanto para *blockbound* foi gerado o valor 0 e para *number-of-segment* o valor 8.

A partir desse momento, para utilizar-se do gerador de parâmetros o usuário deve alterar esses valores para que o caminho equivalente a essa sequência de transições possa ser exercitado. O valor de *SDU* não interfere na executabilidade assim como o valor de *blockbound* para esse caminho. No entanto o valor de *number-of-segment* não pode ser 8, já que a transição t4 é exercitada duas vezes e a t8 uma vez, fazendo com que *number* assuma o valor 3. Como *number-of-segment* deve ser igual a *number* para que t7 seja

exercitada, é impossível dentro do domínio de entrada de *number-of-segment*, atribuir um valor que habilite a execução de t7.

Mesmo que o usuário tente executar o algoritmo com o valor original 8 ou outro valor qualquer o algoritmo retornará 'False', já que não será possível encontrar pontos ON e OFF.

Já o caso de teste 4 é gerado pela CONDADO da seguinte forma:

senddata(U,SENDrequest) recdata(L,CR)
senddata(L,CC) recdata(U,SENDconfirm)
senddata(U,DATArequest,'MlepV',0,8) recdata( )
senddata(L,TOKENgive) recdata(L,DT)
senddata( ) recdata(L,TOKENrelease)
senddata(L,TOKENgive) recdata(L,DT)
senddata(L,BLOCK) recdata( )
senddata( ) recdata(L,TOKENrelease)
senddata(L,TOKENgive) recdata(L,DT)
senddata(L,ACK) recdata(L,DT)
senddata(L,ACK) recdata(U,MONITOR-COMPLETE) recdata(L,TOKENrelease)
recdata(L,DISrequest)
senddata(L,RESUME) recdata( )
senddata(L,BLOCK) recdata( )
senddata(L,ACK) recdata( )
senddata(L,DISrequest) recdata(U,DISindication)

Da mesma forma, os valores para *SDU*, *blockbound* e *number-of-segment*, são os mesmos, respectivamente, 'MlepV', 0 e 8. Também para esse caso, *SDU* e *blockbound* não interferem na executabilidade, mas como a transição t4 é exercitada 3 vezes e t8 uma vez, *number* assume valor 4. Assim, para habilitar a transição t7 o valor 4 pode ser atribuído a *number-of-segment* por ser um valor dentro do domínio de entrada.

Tendo o usuário escolhido esse caso de teste para geração dos parâmetros e tendo alterado o valor de *number-of-segment* para 4, de modo que se tenha um valor I_0 que torne o caminho executável, inicia-se a execução do algoritmo.

Algumas definições são necessárias para a compreensão do algoritmo:

1. I_0 é o valor inicial para o parâmetro que o usuário definiu.

2. UNIT é o incremento para uma varíavel dentro do seu espaço de entrada. Assim sendo, se os valores possíveis para uma variável x são v1,v2,v3,...,vk então Vj+UNIT

= Vj+1 e Vj+1 - UNIT = Vj

3. F é uma função definida como E1-E2, se E1-E2 é positivo para I_0 e E2-E1 caso contrário. Assim, se o predicado a ser considerado for a>b, F=a-b.

As Figuras 5.5 e 5.6 mostram o algoritmo de Hajnal e Forgács adaptado ao problema em questão. A execução do algoritmo é iniciada pela função ON-OFF. É gerada a função F com relação a p e I_0. Como o predicado p é *number = number-of-segment*, a função F é *number - number-of-segment* (E1-E2). Como o valor I_0 para *number-of-segment* é 4 e number, no momento da verificação do predicado p, também é 4, então F=0.

Em seguida é chamada a execução da função Distinction. Durante a fase de refinamento dessa função, tenta-se incrementar o valor I_0 até que F torne-se negativa, o que acontece já na primeira iteração, uma vez que incrementando *number-of-segment*, tem-se o valor 5 e F passa a ser -1. F tornando-se negativa significa que foi atribuído um valor para I_0 fora do domínio.

Na fase de refinamento tenta-se aproximar o valor de I_0, que saíra do domínio, para o mais próximo possível da borda, o que para esse caso não acontece, já que o valor 5 é o valor fora do domínio mais próximo da borda.

Por fim esses valores (4 e 5) são considerados como pontos ON e OFF e como o predicado é de igualdade é necessário um segundo ponto OFF, gerado ao final da função ON-OFF.

5.4 Considerações Finais

Este capítulo tratou a forma com que a estratégia de Hajnal e Forgács foi aplicada para geração de parâmetros da CONDADO. O desenvolvimento de um simulador para MFEE foi necessário para a adaptação da técnica. No próximo capítulo serão apresentados resultados preliminares obtidos após a realização de alguns testes e algumas constatações feitas a partir deles.

```
Function ON_OFF_search (input: p, I_o ; output: ON, OFF, ON2/OFF2): boolean

Gerar F com relação a p e I_0 ;
Se Distinction (F, I_0 ; I_in , I_out , x_i , dir , STEP) então
        Halving (F , I_in, I_out, x_i, dir, STEP; I_in, I_out )
        se I_in != 0 e I_out != 0 e o predicado é de igualdade ou diferença então
                Return FALSE;
        senão
                ON= I_in;
                OUT= I_out;
        fim_se
        se a borda é de igualdade ou diferença então
                OFF2 (ON2) = I_in - I_out;
                OFF2(ON2) = OFF2(ON2) + I_in;
                Return TRUE
        senão
                Return FALSE
        fim_se
fim_se
fim_função


Function Distinction (input: F,I_0; output: I_in, I_out, x_i, dir, STEP): boolean

I_curr := I_0;
enquanto ES (I_curr , {conj. de todas as variáveis influenciadoras}, x_i , dir)
        //fase de extensão
        STEP := UNIT_xi ;
        I_next := I_curr ; I_next [x_i] := I_next [x_i] dir STEP;
        Exercia_maquina para I_next;
        enquanto não ocorre violação de restrição para I_next e F(I_nest ) < (I_curr)
                se F(I_next ) é negativo então
                        I_in := I_curr ; I_out := I_next;
                        return true;
                senão
                        STEP := STEP * 2;
                        I_curr := I_next ; I_next [x_i] := I_next [x_i] dir STEP;
                        Exercita_máquina para I_next;
                fim_se;
        fim_enquanto;
        //fase de refinamento
        enquanto ES (I_curr , { x_i }; x_i , dir)
                STEP := STEP / 2;
                I_h := I_curr ; I_h [x_i] := I_h [x_i] dir STEP;
                Exercita_maquina para I_h ;
                Enquanto ocorrer violação da restrição para I_h ou F (I_h ) >= F (I_curr )
                        STEP := STEP / 2;
                        I_h := I_curr ; I_h [x_i] := I_h [x_i] dir STEP;
                        Exercita_maquina para I_h ;
                fim_enquanto;
                se F (I_h ) é negativo então
                        I_in := I_curr ; I_out := I_h ;
                        return true;
                senão
                        I_curr := I_h ;
                fim_se;
```

Figura 5.5: Algoritmo de Hajnal e Forgács

```
function ES (input: F,I, V; output: x_i, dir): boolean

para cada x_i no conjunto das variáveis influenciadoras faça
        x_i = x_i (+/-) UNIT; //sera incrementado e se necessario, decrementado
        exercita_máquina;
        se não houve violação e F foi decrementada então
                selecione variável x_i;
                dir = + ou -;
                return TRUE;
        fim_se
return FALSE;
fim_função


Procedimento Halving (input: F, I_in, I_out, x_i, dir, STEP; output: I_in, I_out)

enquanto (STEP!=UNIT) faça
        STEP=STEP / 2;
        I_h = I_in ;
        I_h [x_i] := I_h [x_i] dir STEP;
        Exercita_máquina para I_h;
        Se F(I_h ) é negativo então //ou 0 no caso de bordas abertas
                I_out = I_h ;
        Senão
                I_in = I_h ;
        fim_se
Fim_enquanto;
Fim_procedimento

Função exercita_máquina( )

Enquanto houver transições não exercitadas faça
        Leia próxima instrução da sequência;
        Leia na especificação os predicados associados;
        Analise os predicados;
        Se predicados são verdadeiros então
                Execute ações;
        Senão
                Return FALSE;
        fim_se
Fim_enquanto;
Return TRUE;
Fim_função
```

Figura 5.6: Algoritmo de Hajnal e Forgács (cont)

Capítulo 6

Avaliação da Estratégia

6.1 Introdução

Após a implementação da ferramenta de apoio à geração de parâmetros a partir dos casos de teste gerados pela CONDADO, foram realizados alguns testes no sentido de analisar o ganho obtido com relação aos caminhos não executáveis de uma máquina.

Esses testes foram feitos utilizando dois exemplos. No primeiro deles (apresentado na seção 6.2) foi utilizada a MFEE representada pela figura 5.2. O segundo exemplo (apresentado na seção 6.3) foi baseado na máquina ilustrada na figura 6.1.

Para os dois exemplos foi realizado o mesmo procedimento. Foi requisitado à CONDADO a geração de testes atendendo algumas restrições quanto às transições que deveriam ser exercitadas. A partir disso, foi analisada a quantidade de casos de teste gerados e quantos desses seriam correspondentes a caminhos executáveis.

6.2 Primeiro Exemplo

Maiores detalhes sobre a máquina dada pela figura 5.2 já foram apresentadas no capítulo 5. A Tabela 6.1 resume a aplicação dos testes para essa máquina, relatando qual a restrição imposta a CONDADO, o número de casos gerados, e a porcentagem de casos de teste descartada se fosse utilizada a geração de parâmetros.

Tabela 6.1: Exemplo 1 - Utilização do Gerador de Parâmetros

Restrições do usuário	Número de casos de teste gerados pela CONDADO	Número de casos de teste correspondentes a caminhos executáveis	% de casos descartados
Trans. t6 e t8	16	11	31,25
Trans. t5	11	4	63,6
Trans. t3 e t8	16	11	31,25
Trans. t10 e t12	77	23	70,13
Trans. t7 e t9	79	29	63,29
Trans. t5 e t9	9	1	88,89
-	156	62	60,26

6.3 Segundo Exemplo

Para o segundo exemplo, foi utilizada uma MFEE adaptada de [vBJ79] que representa parte do protocolo X25. X25 é um protocolo de acesso padrão para uso de circuitos virtuais fornecidos por redes de dados públicas. A figura 6.2 descreve as transições da máquina em termos de predicados e ações associadas. A especificação completa da máquina é dada em anexo.

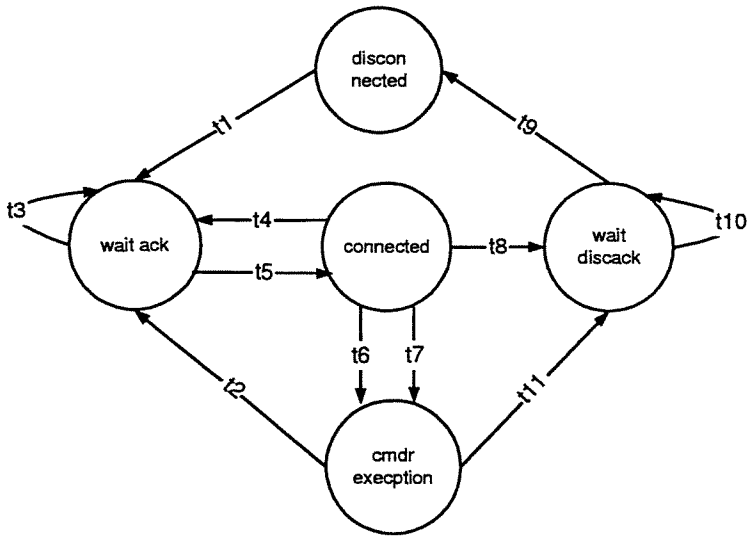

Figura 6.1: Exemplo 2 - Representação da MFEE

Trans.	Entrada	Predicado	Saída	Ação
t1	Highlevel.connect	-	-	errcount:=0
t2	Highlevel.connect	-	-	errcount:=0
t3	-	errcount<maxerrcount	-	errcount:=errcount+1
t4	-	errcount=maxerrcount	-	errcount:=0
t5	-	kind=ua fbit=1	highlevel.reportcmdr	-
t6	-	kind=cmdr	highlevel.reportcmdr	-
t7	-	kind=erroneousframe	highlevel.error	-
t8	highlevel.disconnect	-	-	errcount:=0
t9	-	kind=ua fbit=1	highlevel.reportcmdr	-
t10	-	errcount<maxerrcount	-	errcount:=errcount+1
t11	highlevel.disconnect	-	-	errcount:=0

Figura 6.2: Exemplo 2 - Descrição da MFEE

Dessa forma, a tabela 6.2 apresenta os resultados obtidos para esse exemplo.

Tabela 6.2: Exemplo 2 - Utilização do Gerador de Parâmetros

Restrições do usuário	Número de casos de teste gerados pela CONDADO	Número de casos de teste correspondentes a caminhos executáveis	% de casos descartados
Trans. t4 e t8	5	2	60
Trans. t11	8	6	25
Trans. t2 e t10	4	2	50

6.4 Resultados Obtidos

Com relação aos dois exemplos, pode-se realizar uma análise conjunta, já que o comportamento demonstrado é semelhante em vários aspectos. Pode-se perceber, em ambos os casos, que o ganho obtido varia muito de acordo com o tipo de restrição imposta pelo usuário. No entanto, mesmo nos casos em que menos casos de teste foram descartados, ou seja, aqueles em que são descartados cerca de 30% deles, o ganho também é considerável uma vez que tende-se a economizar 30% do tempo na aplicação dos testes.

Outro fator que pode ser verificado, e isso provavelmente se dará em qualquer tipo de MFEE, é que algumas transições são importantes na executabilidade de um caminho mais do que outras. Ou seja, suponha que um determinado caminho possua uma transição

com o predicado a>b. Se existirem, nesse caminho, transições que alterem o valor de a e/ou de b, essas transições poderão ser decisivas na análise de executabilidade de um caminho. Portanto, ao se fazer restrições à CONDADO sobre as transições que devem ser exercitadas, a inclusão ou não dessas transições faz variar a quantidade de casos de teste correspondentes a caminhos executáveis.

Analisando, tanto a Tabela 6.1 quanto a Tabela 6.2, pode-se verificar que quando o número de casos de teste descartados é menor foi colocado como restrição a necessidade de exercitar uma transição que altera valores de variáveis que são utilizadas em predicados.

Vale lembrar ainda que, além de descartar vários casos de teste que correspondem a caminhos não executáveis, o gerador de parâmetros também gera valores para variáveis de entrada próximos do limite do domínio dessas variáveis o que também é fundamental nesse tipo de teste para se obter maior eficiência em revelar falhas.

6.5 Considerações Finais

Este capítulo apresentou alguns testes aplicados onde pode-se verificar o ganho obtido com relação aos caminhos não executáveis da máquina. Verificou-se que este ganho varia de acordo com as restrições impostas para a geração de casos de teste.

Pode-se observar também que mesmo nos casos em que o número de casos de teste descartados é menor o ganho obtido é também significativo.

Este capítulo apresentou alguns testes aplicados com relação à geração de parâmetros para uma máquina finita de estados estendida. Em primeiro lugar, vale ressaltar que os resultados obtidos para os dois exemplos utilizados são semelhantes, ou seja, em ambos os casos conseguiu-se a eliminação de casos de teste não executáveis.

Verificou-se ainda que este ganho varia de acordo com as restrições impostas para a geração de casos de teste. Pode-se observar também que mesmo nos casos em que o número de casos de teste descartados é menor o ganho obtido também foi significativo.

Capítulo 7

Conclusões

7.1 Introdução

O objetivo principal deste trabalho foi, a partir da ferramenta CONDADO, gerar parâmetros que possam testar de forma mais eficiente uma máquina finita de estados estendida.

Foram implementadas na CONDADO, as técnicas de teste de transição de estados (para a parte de controle do protocolo) e os testes de sintaxe e partição de equivalência (para a parte de dados do protocolo). Contudo, os testes de partição de equivalência se mostraram pouco eficientes, já que não considera os predicados da máquina e portanto não faz nenhuma análise acerca da executabilidade do caminho referente a um caso de teste gerado.

Dessa forma, a geração de testes para a parte de controle de protocolo já havia sido desenvolvida, restando portanto, uma geração de testes mais eficiente para a parte de dados, fundamental para análise dos predicados associados às transições da máquina.

Para isso foi utilizado o teste de domínios, uma técnica originalmente proposta para aplicação em programas mas que pode ser adaptada para máquinas finitas de estados estendidas. Assim sendo, foi escolhida uma dentre as técnicas de teste de domínios para adaptá-la na geração de parâmetros que satisfaçam os predicados associados às transições.

7.2 Análise da Estratégia

Utilizando os testes de domínios para esse problema foi possível algumas conclusões. Primeiramente, serão apresentadas conclusões gerais acerca da eficácia da estratégia. Em seguida serão analisadas algumas limitações da estratégia e por fim serão tratadas algumas

possíveis extensões a esse trabalho.

A primeira análise teve por objetivo avaliar a eficácia dos testes de domínios na redução do problema da executabilidade. Os dados apresentados no Capítulo 6, indicam um ganho com a exclusão de casos de teste correspondentes a caminhos não executáveis. Além disso, obtêm-se dados de teste mais eficientes em revelar falhas à medida que eles se aproximam dos limites do domínio.

A segunda análise se refere à implementação dos testes de domínios. Estudos apontam que apesar de poderosa, é uma técnica de difícil automatização. O que limita sua utilização, para o algoritmo escolhido, é a necessidade de um ponto inicial que possa garantir que o caminho em questão seja executável. Isso nem sempre é uma tarefa simples para o usuário, o que pode desencorajá-lo a utilizar o gerador de parâmetros. Vale lembrar que a utilização do gerador de parâmetros não é obrigatória e portanto o usuário necessitará deste valor inicial apenas se desejar obter dados de teste mais eficientes.

Outra limitação do algoritmo é a impossibilidade de o usuário interferir na geração dos parâmetros. Muitas vezes o valor de um parâmetro de uma interação depende do valor de um parâmetro de uma outra. Suponha um protocolo com abertura de conexão. Primeiramente deve-se abrir a conexão com o hospedeiro X, por exemplo, para em seguida se enviar dados através desta conexão. Da forma como são gerados os testes atualmente, pode ser gerada uma interação de teste para abrir conexão com o hospedeiro X e, em seguida, um teste que envie dados para o hospedeiro Y (dado que Y é um hospedeiro válido). Para casos como este, a ferramenta deveria conhecer tais dependências e permitir a escolha dos valores com base nelas.

7.3 Trabalhos Futuros

Diante disso, algumas extensões podem ser vislumbradas para um futuro próximo. A primeira delas seria melhorar sua funcionalidade permitindo que restrições sejam impostas sobre os parâmetros, de forma a levar em conta as dependências existentes entre eles.

Seria interessante também fazer uma análise do ganho em eficiência obtido pelo fato de os valores escolhidos estarem próximos ao limite do domínio. Mesmo acreditando que os casos de teste sejam dessa forma mais eficientes para descoberta de falhas seria necessário um estudo que confirmasse essa realidade e ainda desse uma medida do quanto esses dados são melhores.

Por fim, um outro trabalho possível é a implementação da segunda parte do algoritmo de Hajnal e Forgács que permite a geração de mais caminhos na máquina a partir do caso

de teste que o usuário escolheu para a geração de parâmetros. Normalmente, quando o usuário faz a escolha por um caso de teste, ele deseja testar alguma(s) funcionalidade(s) relacionada(s) a esse caminho. Dessa forma, gerando outros caminhos a partir desse pode-se ter casos de teste que verifiquem funcionalidades próximas a esta ou ainda casos de teste diferentes que testem a mesma funcionalidade.

Apêndice A

A.1 Especificação em LEP - Figura 5.2

Esta seção apresenta a especificação completa da MFEE ilustrada na figura 5.2 em LEP (Linguagem para Especificação de Protocolo).

```
Variables:
    number: integer;
    counter: integer;

States:
    #idle;
    waitconec;
    connected;
    waitsnd;
    sending;
    blocked;
    waitdisc;

Inputs:
    SENDrequest;
    CC;
    DATArequest;
    TOKENgive;
    RESUME;
    ACK;
    BLOCK;
```

```
    DISrequest;

Outputs:
    CR;
    SENDconfirm;
    DT;
    TOKENrelease;
    MONITOR-COMPLETE;
    TOKEN-RELEASE;
    DISrequest;
    MONITOR-INCOMPLETE;
    DISindication;

DATA:
DATArequest :: = SEQUENCE{
     SDU octetstring 5,
     number-of-segment enumerated 2 — 4 — 8,
     blockbound integer 3 .. 15};

TANSITIONS:
*t1
>idle
    ?U.SENDrequest
    !L.CR
<waitconec;

*t2
>waitconec
    ?L¿CC
    !U.SENDconfirm
<connected;

*t17
>waitconec
    ?L.DISrequest
```

```
    !U.DISindication
<idle;

*t3
>connected
    ?U.DATArequest(SDU,number-of-segment,blockbound)
    number:=0counter:=0
<waitsend;

*t5
>waitsend
?L.RESUME
<waitsend;

*t4
>waitsend
?L.TOKENgive
start-timernumber:=number+1
!L.DT(SDU[number])
<sending;

*t6
>sending
[expire-timer]
!L.TOKENrelease
<waitsend;
```

A.2 Especificação em LEP - Figura 6.1

Esta seção apresenta a especificação completa da MFEE ilustrada na figura 6.1 em LEP (Linguagem para Especificação de Protocolo).

```
Variables:
    errcount: integer;
    timeout: integer;
```

```
States:
    #disconnected;
    waitack;
    connected;
    cmdrexection;
    waitdisack;

Inputs:
    connect;
    disconnect;

Outputs:
    reportcmdr;
    error;

DATA:
    maxerrcount integer 1 .. 10,
    fbit enumerated 0 — 1
    kind octetstring 10;

TANSITIONS:

*t1
>disconnected
    ?highlevel.connect
    errcount:=0
<waitack;

*t2
>cmdrexeption
    ?highlevel.connect
    errcount:=0
<waitack;

*t3
```

```
>waitack
    [errcount < maxerrcount]
    errcount:=errcount+1
<waitack;

*t4
>connected
    [errcount=maxerrcount]
    errcount:=0
<waitack;

*t5
>waitack
    [kind=ua][fbit=1]
    !highlevel.reportcmdr
<connected;

*t6
>connected
    [kind=cmdr]
    !highlevel.reportcmdr
<cmdrexecption;

*t7
>connected
    [kind=erroneousframe]
    !highlevel.error
<cmdrexeption;

*t8
>connected
    ?highlevel.disconnect
    errcount:=0
<waitdisack;
```

```
*t9
>waitdisack
    [kind=ua][fbit=1]
    !highlevel.reportcmdr
<disconnected;

*t10
>waitdisack
    [errcount < maxerrcount]
    errcount:=errcount+1
<waitdisack;

*t11
>cmdrexeption
    ?highlevel.disconnect
    errcount:=0
<waitdisack;
```

Bibliografia

[BD97] G. V. Bochmann and R. Dssouli. Test Development for Distributed Systems: Towards Automation. *XV Simpósio Brasileiro de Redes de Computadores. São Carlos-SP*, Tutorial 4, 1997.

[BDAR97] C. Bourhfir, R. Dssouli, E. M. Aboulhamid, and N. Rico. Automatic Executable Test Case Generation for Extended Finite State Machine Protocols. *http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*, 1997.

[Bei90] Boris Beizer. Software Testing Techniques. *International Thomson Computer Press*, 1990.

[Bei95] Boris Beizer. Black-Box Testing: Techniques for Functional Testing of Software and Systems. *John Wiley*, 1995.

[BP94] G. V. Bochmann and A. Petrenko. Protocol Testing: Review of methods and relevance for software testing. *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–124, 1994.

[CHR82] L. Clarke, J. Hassell, and D. J. Richardson. A Close Look at Domain Testing. *IEEE Trans. on Software Engineering*, SE-8:380–390, July 1982.

[CS87] R. Castanet and R. Sijelmasi. Methods and Semi-Automatic Tools for Preparing Distributed Testing. *IFIP*, 1987.

[Dav78] A. M. Davis. A Comparison of techniques for the specification of external system behavior. *Communication of ACM*, 31(9), September 1978.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based Automatic Test Data Generation. *IEEE Transaction on Software Engineering*, 17(9):900–910, September 1991.

[HF98] A. Hajnal and I. Forgács. An Applicable Test Data Generation Algorithm for Domain Errors. *ISSTA*, pages 63–72, 1998.

[HLJ95] H. Huang, Y. Lin, and M. Jang. An Executable protocol test sequences generation method for efsm-specified protocols. *IWPTS - 8th International Workshop on Protocol Test Systems*, pages 29–44, 1995.

[JW94] B. Jeng and E. J. Weyuker. A Simplified Domain Testing Strategy. *ACM Trans. on Software Enginneering and Methodology*, 3(3):254–270, July 1994.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transaction on Software Engineering*, pages 870–879, 1990.

[LF87] J. C. B. Leite and O. G. Loques Filho. Introdução à Tolerância a Falhas. *2nd SCTF*, 1987.

[Mye79] G. Myers. The Art of Software Testing. *John Wiley*, 1979.

[Pos96] R. M. Poston. Automating Specification-based Software Testing. *IEEE Computer Society Press*, 1996.

[Pre95] R. S. Pressman. Engenharia de Software. *Makron Books*, 1995.

[PvB96] A. Petrenko and G. v. Bochmann. On Fault Coverage of Tests for Finite State Specification. *http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*, 1996.

[Ray87] D. Rayner. OSI Conformance Testing. *Computer Network and ISDN Systems*, pages 79–98, 1987.

[Sab98] S. B. Sabião. Um método para Geração de Testes Baseado em Máquina Finita de Estados Estendida Combinando Técnicas de Teste Caixa Preta. *Tese de Mestrado. IC - UNICAMP. Campinas*, 1998.

[SMF01] S. R. S. Souza, J. C. Maldonado, and S. C. P. F. Fabbri. FCCE: Uma Família de Critérios de Teste para Validação de Sistemas Especificados em Estelle. *XV Simpósio Brasileiro de Engenharia de Software*, pages 256–271, 2001.

[UP83] H. Ural and R. L. Probert. User-guided Test Sequence Generation. *Protocol Specification, Testing and Verification III*, pages 421–436, 1983.

[UY91] H. Ural and B. Yang. A Test Sequence Selection Method for Protocol Testing. *IEEE Transaction On Communications*, 39(4), 1991.

[vBJ79] G. v. Bochmann and T. Joachim. Development and Structure of an X.25 Implementation. *IEEE Transactions on Software Engineering*, SE-5:429–439, September 1979.

[Ver97] S. R. Vergílio. Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes. *Tese de Doutorado. FEEC - UNICAMP*, 1997.

[VMJC97] P. R. Vilela, J. C. Maldonado, M. Jino, and M. Chain. Uma Visão Sobre o Teste Estrutural Baseado em Análise de Fluxo de Dados. *Workshop do Projeto, Validação e Testes de Sistemas de Informação. Águas de Lindóia*, pages 3–14, 1997.

[WC80] L. White and E. I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Trans. On Software Engineering*, SE-6:247–257, May 1980.