Um Método para Geração de Testes Baseado em Máquina Finita de Estado Estendida Combinando Técnicas de Teste Caixa Preta

Selma Bássiga Sabião

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Método para Geração de Testes Baseado em Máquina Finita de Estado Estendida Combinando Técnicas de Teste Caixa Preta

Selma Bássiga Sabião[1]

Maio de 1999

Banca Examinadora:

- Profa. Dra. Eliane Martins (Orientadora)
- Profa. Dra. Ana Maria de Alencar Price (UFRGS)
- Prof. Dr. Ricardo O. Anido (UNICAMP)
- Prof. Dr. João Meidanis (UNICAMP) (Suplente)

[1]Trabalho finaciado pelo CNPq

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA CENTRAL DA UNICAMP

CM-00125693-7

Sabião, Selma Bássiga
Sa13m Um método para geração de testes baseado em máquina finita de estado estendida combinando técnicas de teste caixa preta / Selma Bássiga Sabião. -- Campinas, SP : [s.n.], 1999.

Orientador: Eliane Martins.
Dissertação (mestrado) - Universidade Estadual de Campinas , Instituto de Computação.

1. Software -Testes. 2. Redes do computação - Protocolos - Testes. 3. Engenharia de software. 4. Prolog (Linguagem de programação de computador). I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um Método para Geração de Testes Baseado em Máquina Finita de Estado Estendida Combinando Técnicas de Teste Caixa Preta

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Selma Bássiga Sabião e aprovada pela Banca Examinadora.

Campinas, 24 de Maio de 1999.

Eliane Martins

Profa. Dra. 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

Dissertação defendida e aprovada em 11 de janeiro de 1999, pela Banca Examinadora composta pelos Professores Doutores:

Porfa. Dra. Ana Maria Alencar Price
UFRGS

Prof. Dr. Ricardo de Oliveira Anido
IC - UNICAMP

Profa. Dra. Eliane Martins
IC - UNICAMP

A meus pais Antonio e Leonora,
pelo apoio, carinho e dedicação

"Quem perde seus bens perde muito.
Quem perde um amigo perde mais.
Mais quem perde sua coragem perde tudo".

(Miguel de Cervantes)

Agradecimentos

A Deus, pela vontade e coragem, pelas alegrias e pelos grandes amigos que conquistei nesse período da minha vida.

Aos meus pais Antônio e Leonora, e à minha querida vozinha Lúcia, pela inestimável ajuda e força, nesse, e em todos os momentos da minha vida.

À minha orientadora, Eliane Martins, que depositou sua confiança em meu trabalho e que verdadeiramente me orientou.

Ao Márcio Roberto Stefani (colego), que me ajudou, incentivou e acreditou em minha capacidade (até mais que eu mesma). Sem ele teria sido muito mais difícil.

Às minhas grandes amigas (as Marquinhas): Amanda, Cristina, Karina e Patrícia, que sempre me ajudaram e permitiram momentos de grandes alegrias em todo esse período e, espero, em toda minha vida.

Aos Brows Rogério e Ralph, pela amizade e alegrias.

Aos amigos, que de uma forma ou outra, estiveram sempre presentes: Alessandra, Alessandro, Adilson (Gandhi), André, Cris Campos, Cristina Célia, Cleidson, Delano, Edicezar, Emerson (Gandhi), Freud, Gisele, Guilherme Albuquerque, Janne, Laura, Luciano, Lucas, Marcelo Marcos, Marcos André, Marcos Renato, Mário, Marília, Pascual, Rodrigo (Pavão), Roberto Façanha e Vaguinho.

Às minhas amigas distantes, que mesmo assim, sempre estiveram presentes: Andréia Dal, Adriana, Leliane e Valéria.

Aos professores Ana Maria de Alencar Price e Ricardo Anido, por serem membros da banca examinadora.

Ao INPE, pela disponibilidade em me ajudar no desenvolvimento desse trabalho. Em especial, agradeço a Ana Ambrósio, pelos trabalhos em conjunto, opiniões e dicas, que ajudaram e enriqueceram meu trabalho.

Ao CNPq, pelo apoio financeiro.

E, finalmente, a todas as pessoas que acreditam e lutam por um ensino de qualidade.

Resumo

Com o aumento da demanda por sistemas confiáveis, cresce a necessidade de métodos e ferramentas que possibilitem a validação desses sistemas. Uma forma de validação desses sistemas é através de técnicas de teste.

Nesta dissertação é proposto um método para geração de testes a partir de Máquinas Finitas de Estados Estendidas (MFEE). Este método tem por objetivo a geração de testes englobando aspectos de controle e dados de protocolos de comunicação. Para isso, diversas técnicas de teste caixa preta são combinadas, possibilitando a geração de testes para cobrir todos os aspectos do protocolo.

Este método parte da especificação em MFEE e a transforma em uma especificação de teste. A partir dessa especificação, e de um conjunto de restrições especificadas pelo usuário, seqüências de teste são geradas através de técnicas de teste caixa preta.

Para validar o método proposto, foi implementada uma ferramenta denominada CONDADO. Essa ferramenta implementa três técnicas de teste caixa preta: testes de transição de estados, testes de sintaxe e testes de domínio. Além disso, a CONDADO implementa mecanismos que possibilitam a geração de testes seletivos para cobrir funcionalidades específicas do protocolo através do uso de restrições.

Um experimento foi realizado usando uma implementação do protocolo de solo bordo do satélite SACI-1 do Intituto Nacional de Pesquisas Espaciais (INPE).

Abstract

The demand for reliable systems is increasing and because of that the construction of methods and tools have became more and more important. Specially for the validation through testing techniques.

In this work, a test generation method based on Extend Finite State Machine (EFSM) has been developed. The goal of this method is to generate tests which combine both control and data in the field of communication protocols. Several black box testing techniques are combined by allowing test generation to cover all the protocol aspects.

The method transforms the EFSM into a test specification. From this specification and a set of restrictions supplied by the user, tests sequences are generated through black box testing techniques.

The method was validate by implementing a tool called CONDADO. This tool implements three black box testing techniques: state transition tests, syntax tests and domain tests. CONDADO also implements a mechanism that makes possible the use of restrictions to test specific protocol functionalities.

The tool is intend to be used in the tests of communication systems developed by National Institute for Space Research of Brazil.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A atividade de teste de *software* é um elemento crítico da garantia de qualidade de *software* e representa a última revisão de especificação, projeto e codificação [Pre95].

Com a crescente demanda de *software* para sistemas que envolvem riscos (econômico e de vidas humanas), cresce a necessidade de uma atividade de teste cuidadosa e bem planejada.

A atividade de teste consiste em executar um programa com a intenção de descobrir falhas[1] [Pre95]. Se o comportamento do programa é diferente do esperado na presença de um certo conjunto de dados de entradas (conjunto de testes), significa que o programa tem falhas. Entretanto, se o programa apresenta um comportamento esperado não se pode afirmar que o programa está correto. Daí a importância de selecionar um conjunto adequado de testes.

As principais etapas do processo de teste são: geração, execução e análise dos resultados dos testes. Este trabalho se concentra na primeira etapa do processo de teste, que tem por objetivo a geração de um conjunto de casos de teste, formados por *dados de teste* (entradas) e pelas *saídas esperadas*. Os casos de teste são derivados a partir de um *modelo base*, que pode ser uma especificação do *software*, o próprio código, ou falhas cometidas durante o desenvolvimento.

Os métodos de seleção de casos de teste são classificados de acordo com o *modelo base* usado para os testes. Dessa forma, os métodos de seleção podem ser classificados em: testes baseados na especificação, testes baseados na implementação e testes baseados em falhas. Os **testes baseados na especificação**, também chamados de testes **caixa preta** têm por objetivo determinar se os aspectos requeridos foram implementados. Nos **testes baseados na implementação**, também chamados de testes **caixa branca**, a seleção dos

[1] Uma **falha** (*fault*) é a causa direta de um erro. Um **erro** (*error*) é a parte do estado do sistema que pode levar a um **defeito** (*failure*). Não existe ainda um consenso quanto à terminologia a ser usada em português; os termos usados aqui são baseados em [LLF87]

testes é baseada na informação obtida a partir do código do programa. Estes testes visam mostrar que as várias partes do código podem ser exercitadas sem violar a especificação. Os **testes baseados em falhas** têm como modelo base as falhas cometidas durante o desenvolvimento e visam mostrar que essas falhas não foram cometidas na implementação.

Como o domínio de entrada normalmente é muito grande, testes exaustivos não são possíveis. Dessa forma, é necessário selecionar um subconjunto relativamente pequeno dentro de um domínio representativo de entrada. Para isso, diversos métodos para geração de teste são propostos.

Existem diversos métodos de geração de teste. Esse trabalho concentra-se na geração de testes para protocolos de comunicação. Protocolos são regras que governam a comunicação entre diferentes componentes dentro de um sistema distribuído [BP94]. Para organizar a complexidade dessas regras, elas são usualmente divididas em uma hierarquia de várias camadas, como exemplificado pelo modelo OSI (ver capítulo 2).

A especificação de um protocolo pode levar a várias implementações de *software* ou *hardware*. É muito importante testar uma implementação contra a especificação a fim de garantir a compatibilidade com outras implementações do mesmo protocolo. Esse tipo de teste é chamado de **teste de conformidade**.

Uma das principais características dos testes de conformidade é que eles são do tipo **caixa preta**, pois são baseados na especificação do protocolo. Isso implica em uma especificação precisa do protocolo, que é a base para derivação dos casos de teste e análise dos resultados [BP94].

Existem diversas formas de especificação formal para protocolos de comunicação. Dentre elas destacam-se as Máquinas Finitas de Estados (MFE), Máquinas Finitas de Estados Estendidas (MFEE), Redes de Petri, ASN.1, gramáticas formais e linguagens de programação de alto nível.

Com o trabalho de padronização do modelo OSI, surgiu a necessidade de padronizar a forma de especificação dos protocolos e serviços da ISO. Desse trabalho, três linguagens para especificação de protocolos, denominadas de **técnicas de descrição formal** (FDT), foram criadas: Estelle, LOTOS e SDL. A linguagem LOTOS é baseada em álgebra de processos enquanto que Estelle e SDL são baseadas em MFEE.

De forma geral, a especificação de um protocolo de comunicação consiste em duas partes: controle e dados. Na década passada, muitos métodos de geração de teste se concentraram na parte de controle da especificação [BDA96]. A maioria desses métodos partem de uma especificação na forma de MFEs.

As MFEs são usadas para especificar a parte de controle do protocolo, ou seja, especificam estados e transições. Para especificar predicados, ações e a parte de dados dos parâmetros das interações do protocolo, normalmente são usadas MFEEs e FDTs baseadas em MFEE, tais como Estelle e SDL.

Os métodos para geração de testes cobrindo controle e dados dos protocolos baseados em MFEE normalmente tratam esses aspectos separadamente, ou não levam em conta a parte de dados do protocolo.

Um problema existente na geração de teste para MFEE é o problema da executabilidade [BDAR97]. A geração de casos de teste não executáveis ocorre devido à existência de predicados e ações que podem afetar o fluxo de controle do protocolo. Para tratar esse problema alguns métodos propõem alternativas como: expandir a MFEE a fim de eliminar os predicados [CS87], análise do fluxo de dados [HLJ95], análise dos ciclos existentes na MFEE [BDAR97], dentre outras.

1.1 Motivação

A necessidade de se obter um ambiente de teste englobando todas as suas atividades, levou à elaboração do ATIFS[2]. Esse trabalho se concentrou no estudo e na criação de uma ferramenta para geração de teste a partir de MFEEs.

A geração de teste a partir de MFEE apresenta muitos problemas que vêm sendo tratados pelos métodos mais atuais encontrados na literatura. Esses métodos, normalmente, usam técnicas de teste **caixa branca** para cobrir os aspectos dos dados do protocolo, como por exemplo, técnicas de teste de fluxo de dados.

Em [Pos96], o autor apresenta um estudo mostrando que é possível testar todos os aspectos de um sistema usando a combinação de várias técnicas de teste **caixa preta** (ver capítulo 4).

A partir desse estudo e dos diversos métodos para geração de casos de teste presentes na literatura, percebeu-se a necessidade de um método que possibilitasse a geração de casos de teste a partir de técnicas de teste caixa preta englobando todos os aspectos do protocolo.

1.2 Objetivos

O objetivo desse trabalho foi a elaboração de um método e a sua implementação para geração de teste englobando todos os aspectos do protocolo. A implementação desse método permitiu a sua validação e a obtenção de medidas de tempo e cobertura.

Apesar de todas as vantagens oferecidas pelas linguagens de especificação formal, a maior barreira encontrada em seu uso foi a dificuldade em entender e usar tais linguagens [Pos96]. Uma pessoa que não está familiarizada com lógica booleana, teoria dos

[2]Ambiente integrado de Teste baseado em Injeção de Falhas por Software

conjuntos e cálculo de predicados tem dificuldades com muitas das linguagens de especificação.

Dessa forma, outro objetivo desse trabalho foi a elaboração e construção de uma linguagem para especificação do protocolo que não exigisse um esforço muito grande por parte do usuário para entendê-la e usá-la.

1.3 Organização da Dissertação

Inicialmente, o capítulo 2 apresenta os principais conceitos na área de teste e na área de protocolos de comunicação. É apresentado uma visão geral das técnicas de teste que serão abordadas nesse trabalho. Como esse trabalho foi desenvolvido com base no padrão definido pela ISO, são apresentados também os conceitos relacionados a teste de protocolos dentro desse padrão.

O capítulo 3 apresenta a definição formal de MFE e MFEE. Em seguida, são apresentados os principais trabalhos relacionados que, de alguma forma, influenciaram o desenvolvimento desse trabalho.

A partir de estudos dos métodos existentes para geração de casos de teste, um método de geração de teste para protocolos de comunicação é proposto no capítulo 4. Este método é a base para a ferramenta de geração de teste implementada.

O capítulo 5 apresenta a ferramenta de geração de teste implementada, tendo como base o método proposto no capítulo 4. Para cobrir o método de geração de teste, uma linguagem para especificação de protocolos é proposta. A partir dessa linguagem é construída uma especificação de teste cobrindo a parte de controle e dados do protocolo. Dessa forma, técnicas de teste caixa preta são aplicadas à essa especificação de teste para obtenção dos casos de teste.

Após a descrição da ferramenta e das suas funcionalidades, o capítulo 6 descreve as saídas geradas e os resultados de alguns experimentos realizados. Inicialmente, uma descrição do formato dos casos de teste gerados é apresentada. Nesse capítulo são apresentados também os resultados de um experimento realizado usando uma implementação de um protocolo. Este experimento foi feito com o objetivo de validar a ferramenta implementada e também a linguagem para especificação do protocolo, usada para especificação do protocolo.

Finalmente, o capítulo 7 apresenta as conclusões desse trabalho. Nesse capítulo são abordadas as contribuições, limitações e os trabalhos futuros, que poderão ser realizados a partir desse trabalho.

Capítulo 2

Fundamentos de Testes

Este capítulo tem por objetivo introduzir os principais conceitos relacionados com testes de *software*. E, em particular, apresentar os conceitos relacionados com teste de protocolos de comunicação.

O processo de teste é composto de três atividades principais: geração, execução e análise dos resultados. Este trabalho se concentra na primeira atividade do processo de teste. Mais especificamente, na geração de casos de teste para protocolos de comunicação.

Com a padronização do modelo de referência OSI pela ISO, surgiu a necessidade de padronização dos testes a serem aplicados às implementações de protocolos. Dessa forma, a norma ISO 9646 foi elaborada com o objetivo de padronizar a área de teste de protocolos, mais especificamente, os testes de conformidade.

Como este trabalho foi desenvolvido com base no padrão definido pela ISO, faz-se necessário apresentar os conceitos relacionados a teste de protocolos dentro desse padrão.

Este capítulo está organizado da seguinte forma: inicialmente é feita uma introdução dos conceitos relacionados a testes. A partir desses conceitos são apresentadas as técnicas de teste usadas pelos trabalhos relacionados. Como este trabalho está relacionado com testes de protocolos, os principais conceitos envolvidos com protocolos e testes para protocolos de comunicação são apresentados. Esses conceitos incluem: modelo de referência OSI, formas de especificação, testes de conformidade e as arquiteturas de teste.

2.1 Conceitos Básicos

A garantia da qualidade de um *software* compreende várias atividades. Dentre elas, está a atividade de teste [Pre95].

A seguir são apresentadas as principais características envolvidas com a atividade de teste.

2.1.1 Objetivos

O objetivo da atividade de teste é encontrar falhas. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar uma falha ainda não descoberta [Pre95]. Se o comportamento do programa é diferente do esperado, significa que o programa tem falhas. Entretanto, se todos os casos de testes são aplicados à implementação e nenhuma falha é encontrada, não se pode afirmar que a implementação está correta.

Os resultados dos testes também podem ser usados para se obter medidas da confiabilidade e alguma indicação da qualidade do *software*. Porém, os testes não podem mostrar a ausência de falhas, sendo que esta atividade é executada através de técnicas como prova de correção de programa, embora já existam técnicas de teste que provem a ausência de certos tipos de erros no *software* [Mar98].

2.1.2 Atividades do Processo de Teste

Os testes podem ser divididos em três atividades principais: **geração dos casos de teste**, **execução dos testes** e **análise dos resultados**. A figura 2.1, adaptada de [Pos96], mostra esse conjunto de atividades.

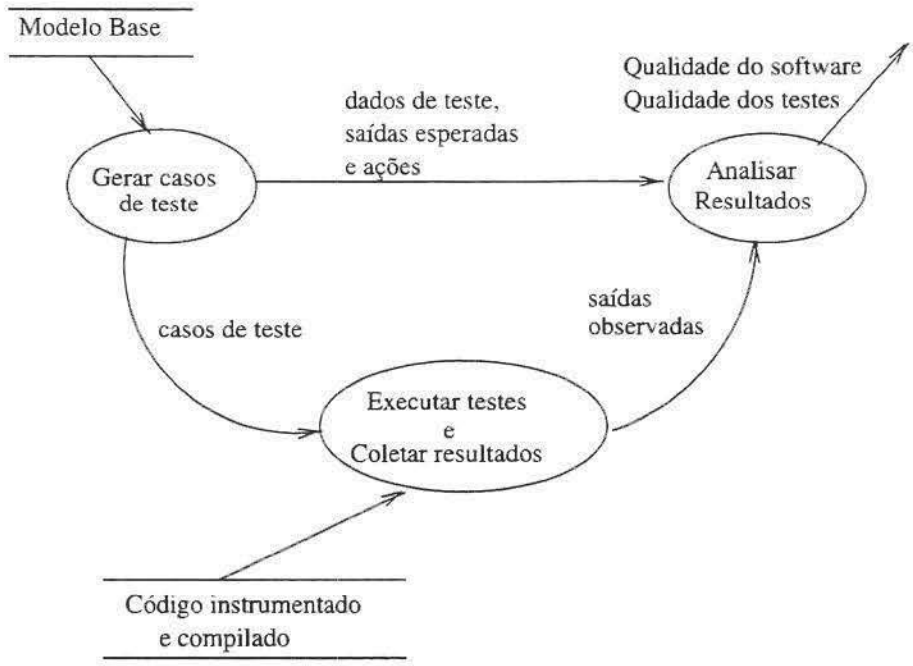

Figura 2.1: O processo de teste.

O processo de teste inicia com a seleção de casos de testes formados pelos *dados de teste* (entradas) e pelas *saídas esperadas*. Os casos de teste são derivados a partir de um *modelo*

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. Tendo esse conjunto de casos de teste, a atividade de execução dos casos de testes é iniciada. Para a realização dessa atividade deve ser construído um ambiente que permita executar o *software* com os dados de teste selecionados e coletar as saídas observadas. A partir das saídas observadas durante a execução dos casos de teste, a atividade de análise dos resultados é realizada. Essa atividade compreende tanto a avaliação da qualidade do *software* quanto a da qualidade dos testes aplicados [Mar98].

A seguir, cada uma dessas atividades será apresentada com mais detalhes.

Geração de Casos de Testes

A principal dificuldade na geração dos casos de teste é selecionar um conjunto de teste que encontre o maior número possível de falhas. Como a geração de todos os casos de teste é inviável, seja pela quantidade que é infinitamente grande, seja pelas próprias restrições de tempo e recursos, é necessário encontrar formas de selecionar os melhores casos de teste (casos de teste que têm maior probabilidade de encontrar falhas).

A fim de determinar uma forma de selecionar um conjunto de casos de teste, são criados os **critérios de teste**. Um critério de teste determina um conjunto finito do modelo que deve ser exercitado durante os testes. No caso de protocolos de comunicação, se o modelo base usado é uma especificação na forma de máquina de estados, um exemplo de critério de teste seria: *todas as transições devem ser exercitadas pelo menos uma vez.* Dessa forma, pode-se dizer que um critério é uma condição sobre o modelo base, que pode ser a implementação ou a especificação, e os casos de testes devem satisfazê-lo.

A atividade de geração de casos de teste envolve a seleção de um conjunto de casos de teste usando um determinado critério. Como o objetivo desse trabalho é a geração de testes para protocolos de comunicação, serão apresentados no capítulo 3 os métodos de geração de casos de testes para protocolos de comunicação mais relevantes para este trabalho.

Execução dos Testes

Para executar os casos de teste é necessário ativar o *software* e aplicar os casos de teste como entrada. Quando as entradas são processadas o *software* produz as saídas, que devem ser coletadas para serem comparadas com as saídas esperadas.

Como o número de testes aplicados a um *software* é muito grande e estes devem ser repetidos várias vezes, aplicá-los manualmente, além de não ser prático é propenso a erros. Daí a necessidade de automação dessa atividade.

Análise dos Resultados

Esta atividade envolve a análise do comportamento observado do *software* em resposta à execução de cada caso de teste. Um mecanismo denominado **oráculo** é responsável por analisar os resultados obtidos e produzir uma sentença denominada **veredicto**. Um veredicto informa se um teste produziu saídas corretas conforme a especificação do *software* [Ste97].

Os possíveis valores para um veredicto podem ser [BD97]:

- **Falhou**: o comportamento observado está em contradição com o especificado;
- **Passou**: o comportamento observado está conforme ao especificado; e
- **Inconclusivo**: o propósito do teste não foi coberto durante a execução do teste, provavelmente devido a implementação sob teste ter escolhido uma iteração permitida, mas não esperada.

Determinar quais saídas estão corretas para um determinado caso de teste constitui o **problema do oráculo**. É um problema porque nem sempre é possível ter uma referência confiável para dizer se o resultado de um teste está correto ou não. Várias soluções já foram propostas para esse problema e um estudo mais amplo pode ser encontrado em [Ezu95] e [Ste97].

2.1.3 Classificação dos Testes

Os métodos de seleção de casos de teste podem ser classificados de acordo com o modelo base usado para os testes. Dessa forma, os testes podem ser classificados como: testes baseados na especificação (**testes caixa preta**), testes baseados na implementação (**testes caixa branca**) e testes baseados em erros [Mar98].

Os métodos de geração de testes estão associados a critérios. Os critérios dependem dos aspectos da especificação (ou implementação) que serão considerados. Em uma especificação pode-se considerar as funções ou os dados sendo transformados pelas funções do programa. Em uma implementação, aspectos importantes a serem testados podem ser as instruções, o fluxo de controle ou o fluxo de dados.

Em protocolos de comunicação, os testes são do tipo caixa preta. Entretanto, com o uso de técnicas de descrição formal para sua especificação, tornou-se possível adotar critérios de teste caixa branca para serem aplicados à especificação do protocolo, desde que esta seja descrita usando uma notação adequada. Dessa forma, para facilitar o entendimento dos trabalhos relacionados e do método proposto nesse trabalho, serão apresentadas algumas técnicas de teste de caixa branca e caixa preta que, de alguma forma, contribuíram para a criação dos métodos de teste para protocolos apresentados nas próximas seções.

2.1.4 Noção de Cobertura

As condições que devem ser satisfeitas durante os testes são chamadas de *critérios de adeqüabilidade* ou simplesmente critérios de teste.

Um critério de adequabilidade C determina um conjunto finito S_c de elementos do modelo que devem ser exercitados durante os testes [The91] *apud* [Mar98]. Assim se o modelo usado para os testes é um modelo do programa, um critério C poderia ser: todas as instruções do programa devem ser exercitadas pelo menos uma vez; S_c seria o conjunto de todas as instruções do programa.

Critérios de adeqüabilidade são úteis para avaliar a 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_c exercitados)/(número total de elementos de S_c). A cobertura pode ser definida antes, para indicar quantos casos de teste devem ser gerados, ou depois, para obter a cobertura atingida pelos testes.

2.2 Testes Baseados na Implementação

Testes baseados na implementação, também chamados de testes *caixa branca*, consideram como modelo base a estrutura interna da implementação do *software*. Procuram caracterizar um conjunto de componentes elementares do *software* que devem ser exercitados. Esses testes são classificados nas seguintes classes de critérios [LM88]:

- **critérios orientados à estrutura**[1]: neste caso são considerados os componentes estruturais do programa, como por exemplo, comandos e dados.
- **critérios orientados ao fluxo**: são consideradas as seqüências de comandos a serem executadas ou a definição-uso dos dados.
- **critérios orientados ao estado**: o objetivo desses critérios é exercitar os diferentes estados do programa. O estado de um programa é determinado pelos valores de suas variáveis em um momento específico da execução.

Esses critérios são subdivididos para considerar componentes mais específicos de um programa. Um estudo sobre todos esses critérios pode ser encontrado em [LM88] e [Mar98].

Antes de discutir os critérios de fluxo de dados, uma definição sobre grafo de fluxo de controle e grafo de fluxo de dados faz-se necessária.

[1] O termo adotado por [LM88] é "**critérios orientados a objetos**" mas, para não ser confundido com os testes de sistemas orientados a objetos, adotou-se o termo usado em [MD92].

2.2.1 Definições de Grafo de Fluxo de Controle

Um **grafo de fluxo de controle** é um grafo finito, direcionado e conectado. Esse grafo define o modelo abstrato do programa representando seu fluxo de controle [LM88].

Definição 2.1 *Um grafo* $G = (N, A, n_0, n_f)$ *é um* **grafo de fluxo de controle**, *onde*

N *é um conjunto finito dos nós;*

A *é um conjunto finito das arestas;*

n_0 *é o nó inicial;*

n_f *é o nó final;*

Um **nó** representa um comando ou um bloco, que é uma seqüência de comandos sem nenhum comando de desvio. Cada **aresta** representa um fluxo de controle entre dois comandos ou blocos. Portanto, uma aresta (n_i, n_j) significa que o comando (ou bloco) representado por n_i é executado antes do comando (ou bloco) representado por n_j. Dessa forma, pode-se dizer que n_i precede n_j e que n_j é o sucessor de n_i.

Através do grafo de fluxo de controle, conceitos importantes podem ser definidos a seguir.

2.2.2 Definições de Grafo de Fluxo de Dados

Grafo de fluxo de dados são usados para derivar geração de testes baseados no fluxo de dados. Esses grafos podem ser considerados como uma extensão dos grafos de fluxo de controle definidos acima.

Grafos de fluxo de dados representam as relações entre os pontos do programa onde as variáveis são definidas e o ponto onde elas são usadas.

As ocorrências de variáveis em um programa podem ser uma **definição de variável** ou **uso de variável**. 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. Um uso de variável ocorre quando a referência a esta variável não estiver definida. Distingui-se dois tipos de uso: **c-uso**, quando a variável é usada em uma computação e **p-uso** quando a variável é usada em um predicado [VMJC97].

Antes de apresentar os critérios baseados em grafo de fluxo de dados, algumas definições são necessárias:

Definição 2.2 *Um* **caminho** *é uma seqüência finita de nós* $(n_1, n_2, \ldots, n_k), k \geq 2$, *tal que existe um arco de* n_1 *a* n_{i+1} *para* $i = 1, 2, \ldots, k-1$. *Um caminho é um* **caminho simples** *se todos os nós que formam esse caminho, exceto o primeiro e o último, são distintos; se todos os nós são distintos diz-se que o caminho é um* **caminho livre de laço**.

Definição 2.3 *Um caminho* $(n_i, \ldots, (n_j, n_k)), i < j, k < j$, *ou seja, um caminho do nó* n_i *ao nó* n_j *ou à aresta* (n_j, n_k), *é um* **caminho livre de definição** *com relação à variável* x *se e somente se nenhuma definição de* x *é feita entre os nós* n_{i+1} *e* n_j.

Definição 2.4 *Uma definição de uma variável* x *é uma* **definição global** *se é a última definição de* x *no bloco e existe um caminho livre de definição com respeito a* x *entre o bloco em que houve a definição de* x *e um bloco contendo um c-uso ou p-uso de* x.

Definição 2.5 *Uma definição de uma variável* x *é uma* **definição local** *se não é global e existe um c-uso de* x *no bloco em que ela foi definida.*

Definição 2.6 *Um c-uso de uma variável* x *é um* **uso global** *se não existe uma definição de* x *no mesmo bloco, precedendo o c-uso; em outros termos, o c-uso de* x *é afetado por sua definição em um outro bloco. Do contrário o s-uso é local. Quanto ao p-uso, ele é sempre global, pois está associado às arestas que saem de um nó predicado.*

Definição 2.7 *Um caminho* $(n_i, \ldots, (n_j, n_k)), i < j, k < j$, *ou seja, um caminho do nó* n_i *ao nó* n_j *ou à aresta* (n_j, n_k), *é um* **du-caminho** *com relação à variável* x *se* n_i *contém uma definição global de* x *e: (1) ou* n_k *tem um c-uso e* x *e* $(n_i, \ldots, (n_j, n_k))$ *é um caminho livre de definição com respeito a* x; *ou (2)* (n_j, n_k) *tem um p-uso de* x, $(n_i, \ldots, (n_j, n_k))$ *é um caminho livre de definição com respeito a* x *e* $(n_i, \ldots, n_j)$ *é um caminho livre de laço.*

Um grafo de fluxo de dados, também chamado de *grafo def-use*, é um grafo de fluxo de controle estendido, onde cada nó e algumas arestas são marcados por informações adicionais. Cada nó n_i é associado com um conjuntos DEF(n_i) e C-USO(n_i), e cada aresta (n_i, n_j) com o conjunto P-USO(n_i, n_j).

Definição 2.8 DEF(n_i) *é o conjunto de variáveis para as quais o nó* n_i *contém uma definição global.*

Definição 2.9 C-USO(n_i) *é o conjunto de variáveis para as quais o nó* n_i *contém um c-uso global.*

Definição 2.10 P-USO(n_i, n_j) *é o conjunto de variáveis para as quais a aresta* $(n_i, n_j))$ *contém um p-uso.*

Definição 2.11 DCU(x, n_i) *é o conjunto de todos os nós* n_j *tal que* $x \in$ DEF(n_i) *e* $x \in$ C-USE(n_j) *e existe um caminho livre de definição com respeito a* x *entre* n_i *e* n_j.

Definição 2.12 DPU(x, n_i) *é o conjunto de todas as arestas* (n_j, n_k) *tal que* $x \in$ DEF(n_i) *e* $x \in$ P-USE$(n_j, n_k))$ *e existe um caminho livre de definição com respeito a* x *entre* n_i *e* (n_j, n_k).

Com essa base conceitual, a próxima seção irá apresentar os critérios baseados no fluxo de dados.

2.2.3 Critérios Baseados no Fluxo de Dados

Há diferentes abordagens para os critérios baseados em fluxo de dados. Serão apresentados aqui somente a família de critérios de Rapps e Weyuker [RW82], [RW85]. Uma apresentação sobre os demais critérios pode ser encontrada em [VMJC97].

O objetivo dos critérios baseados no fluxo de dados é detectar **anomalias de fluxo de dados**, tais como: variáveis definidas e não usadas ou uso de variáveis indefinidas. Para isso a seleção de caminhos de teste é feita com base na associação entre definições e usos de variáveis dentro do programa [Mar98].

Essas anomalias podem ser detectadas através de diferentes formas de análise do programa. A **análise estática** é a análise feita no código fonte sem a sua execução. A **análise dinâmica** é feita enquanto o programa está em execução e é baseada em valores intermediários que resultam da execução do programa. Análise estática detecta erros sintáticos no código fonte, enquanto que erros do tipo "divisão por zero" são detectados pela análise dinâmica [Bei90].

Muitos compiladores implementam a análise estática. Dessa forma, erros como uso de uma variável indefinida podem ser detectados na compilação do programa.

Os principais critérios baseados em fluxo de dados definidos por Rapps e Weyuker são [RW82] e [RW85]:

- **todas-definições**: requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se for por um **c-uso** ou **p-uso**;
- **todos-usos**: requer que todas as associações entre uma definição e subsequentes **c-usos** e **p-usos** sejam exercitados pelos casos de teste;
- **todos du-caminhos**: requer que toda associação entre uma definição de variável e subsequentes **c-usos** ou **p-usos** dessa variável sejam exercitados por todos os caminhos livres de definição, e livres de laço que cubram essa associação.

Da mesma forma que em testes baseados no fluxo de controle, algumas associações **def-uso** também não podem ser exercitadas, pois nenhum caso de teste é capaz de fazê-lo, devido à existência de caminhos não executáveis. Um estudo mais detalhado sobre esses critérios, bem como suas definições formais podem ser encontradas em [LM88].

2.3 Testes Baseados na Especificação

Para gerar testes a partir da especificação, é necessário que esta seja **testável**. Uma especificação é dita testável quando ela satisfaz aos seguintes requisitos: (*i*) não é ambígua, onde cada termo da especificação tem somente uma definição; (*ii*) consistente, onde cada termo é usado somente de uma forma; (*iii*) completa, com todas as informações necessárias e suficientes para os testes. Com esses requisitos, é possível identificar com maior precisão os dados de entrada para os testes, bem como os resultados esperados [Mar98].

Da mesma forma que nos testes baseados na implementação, existem diversos critérios para seleção de testes a partir da especificação. Esses critérios variam de acordo com a informação usada na especificação, podendo ser baseados nas ações (ou funções), nos dados, nos predicados, nos eventos ou nos estados [Pos96]. Para uma abordagem completa dos testes baseados na especificação deve-se consultar [Bei95].

Como muitos protocolos são especificados através de MFE e MFEE, critérios de teste baseados na transição de estados são freqüentemente usados. Outros critérios como teste de sintaxe e teste funcional são também empregados por alguns métodos de geração de teste de protocolos. Um distinção deve ser feita em relação ao teste funcional, normalmente é encontrado na literatura referenciando aos testes caixa branca, mas aqui diz respeito a uma técnica específica de teste caixa preta. A seguir serão apresentados brevemente esses critérios.

2.3.1 Teste de Transição de Estados

Modelos baseados em transição de estado são úteis para representar aspectos dinâmicos de um sistema. Existem várias notações para modelar tais sistemas: máquinas de estado, redes de Petri, *statecharts* e as técnicas de descrição formal (ver seção 2.4.2). Testes baseados em MFE são a base para os testes de transição de estados. Uma definição formal de MFE será dada na seção 3.1.1.

A estratégia para teste de transição de estados é semelhante à usada para teste de caminhos em grafos de fluxo de controle. Como é impraticável percorrer todos os caminhos em uma grafo de fluxo de controle, é também impraticável percorrer todas as combinações de transições em uma MFE.

O critério de teste de transição de estado requer que cada transição seja executada pelo

menos uma vez. Existem diversos métodos para derivação de testes a partir de MFEs. O mais simples é o método de varredura de transição (método T), que tem por objetivo percorrer a máquina de forma a satisfazer o critério dado [Mar98]. Essa e outras técnicas de teste de transição de estados serão apresentadas em mais detalhes na seção 3.2.

As falhas que podem ser encontradas pelos testes de transição de estados são [Mar95a]:

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.

Como a cobertura de um grafo de fluxo de controle não garante a cobertura de todos os tipos de falha, o mesmo ocorre com os testes de transição de estados. Esses testes não conseguem detectar falhas como: (*i*) falhas nas ações que sejam independentes do modelo, ou seja, a transição está correta, mas houve falha na implementação da ação; (*ii*) falhas de interação entre as ações; neste caso as ações são executadas corretamente, mas a execução de uma ação em uma transição afeta a execução de uma ação em outra transição; (*iii*) a existência de estados ou transições extras [Mar95a].

Teste de transição de estados cobre somente os aspectos de controle, referentes à ordem temporal dos eventos no sistema. Os aspectos de dados, referentes aos parâmetros das entradas e ao formato das mesmas, não são cobertos, pois esses aspectos não são representados pelo modelo de MFE. Uma solução é combinar os testes de transição de estados com outros métodos de teste como por exemplo, teste de sintaxe, que tratam dos aspectos de dados.

2.3.2 Teste de Sintaxe

Algumas aplicações recebem dados de entrada com uma sintaxe muito bem definida. Existem diversas formas de representar essa sintaxe, como a ASN.1, que é usada para representar os dados transmitidos e recebidos pelos protocolos de comunicação, e a BNF[2], uma das notações mais conhecidas para descrever linguagens de programação.

A especificação dos dados pode ser usada para a seleção de casos de teste a fim de criar estruturas válidas e inválidas. A sintaxe pode ser representada por um **grafo de sintaxe** e os testes podem ser gerados a partir desse grafo.

[2]Backus-Naur Form

Os testes de sintaxe procuram encontrar os seguintes tipos de anomalias no tratamento dos dados de entrada [Bei95]:

- **Sintaxe incompleta**: comandos válidos não são aceitos pelo programa;
- **Sintaxe inconsistente**: comandos válidos são interpretados de forma errada;
- **Aceitação errônea**: comandos incorretos são aceitos pelo programa;
- **Sintaxe ou semântica confusa**: erros no *parser* leva à rejeição de valores; por exemplo, em um programa que deve aceitar números inteiros de 0 a 999, 005 é aceito enquanto que 5 não é aceito.
- **Erros léxicos ou alfabéticos**: um exemplo é o caso em que cadeias de caracteres devem vir delimitadas por aspas, mas como tratar cadeias que contêm aspas?

O procedimento de teste de sintaxe é muito simples e, por isso, fácil de ser automatizado. Tomando como base o grafo de sintaxe, pode-se elaborar um conjunto de testes válidos simplesmente percorrendo os caminhos através do grafo. Na prática, não é possível testar todos os caminhos do grafo de sintaxe e, portanto, algumas alternativas para simplificar os testes podem ser usadas. Um maior detalhamento desses testes pode ser encontrado em [Bei90] e [Bei95].

Após gerar testes para as condições válidas, é importante derivar testes para as inválidas. Para isso, situações de erro podem ser obtidas através de algumas alterações no grafo de sintaxe como, por exemplo, substituir um nó por outro, suprimir um nó ou um arco de cada vez, ou mesmo, criar arestas que não existam.

2.3.3 Teste Funcional

Teste funcional vê o programa como uma coleção integrada de funções e seleciona dados de teste a fim de verificar se o programa implementa corretamente essas funções [How80].

O princípio do teste funcional considera que um programa calcula uma ou mais funções e, para testar o programa, cada função deve ser executada por um conjunto de casos de teste pelo menos uma vez.

Inicialmente são identificadas as funções existentes no programa. A identificação é realizada tendo como base a especificação. Caso esta especificação não esteja disponível, as funções são identificadas pela análise do próprio código do programa. Nesse caso, é necessário ter um grande conhecimento de como esse programa funciona para identificar corretamente essas funções.

Após a identificação das funções, são escolhidos valores para os parâmetros de entradas para cada função identificada. [How80] apresenta algumas regras para a escolha dos valores de entrada para os testes. Essas regras variam dependendo do tipo do parâmetro.

Considere, por exemplo, que uma determinada entrada para a função que está sendo testada seja do tipo inteiro com o intervalo $[L, S]$. Para testar o caso normal, os seguintes valores são escolhidos: os valores dos limites L e S e um valor no interior desse limite. Usando essa idéia para um parâmetro inteiro com intervalo $[-1, 38]$ os seguintes valores poderiam ser escolhidos: -1, 15 e 38.

Em um programa com n variáveis de entrada (ou saída), cada uma com k valores para teste, permitem k^n combinações possíveis de valores de teste. Para evitar explosão combinatorial, testes funcionais não requerem a construção de testes envolvendo todas as combinações possíveis. Entretanto, como determinados erros só são descobertos com uma certa combinação de valores para as variáveis de entrada ou saída, métodos para identificar pequenos subconjuntos de variáveis funcionalmente relacionadas são propostos [How80].

Uma das vantagens dos testes funcionais é que eles forçam o teste do programa sobre funcionalidades importantes e também sobre valores especiais. Em muitos casos, esses são os tipos de valores necessários para forçar a manifestação de uma falha associada a caminho parcialmente correto.

2.4 Testes de Protocolos de Comunicação

Protocolos de comunicação são as regras que governam a comunicação entre diferentes componentes dentro de um sistema distribuído [BP94].

A necessidade de padronizar essas regras levou à elaboração, pela ISO[3], de um modelo que viesse sintetizar o funcionamento dos protocolos de comunicação. Com essa padronização, surgiu a necessidade de criar métodos e ferramentas para verificar se as implementações desses padrões estavam de acordo com a sua especificação, garantindo a compatibilidade entre as várias implementações de um mesmo protocolo.

As próximas seções apresentam os conceitos relacionados ao modelo de referência OSI, bem como aos testes de protocolos dentro do padrão ISO.

2.4.1 Modelo de Referência OSI

Para sintetizar, de modo abstrato, o funcionamento das redes de comunicação, a ISO elaborou o modelo de referência para interconexão de sistemas abertos denominado **Modelo OSI**[4]. Este modelo tem como objetivo estruturar a rede como um conjunto de **camadas**

[3] *International Organization for Standardization*

[4] *Open Systems Interconnection*

hierárquicas.

Cada camada (ou nível) é construída utilizando as funções e serviços oferecidos pelas camadas inferiores. Deve ser pensada como um programa ou processo, implementado em *hardware* ou *software*, que se comunica com o processo correspondente na outra máquina [SLC95]. As regras que governam a comunicação de um nível n qualquer são chamadas de **protocolo** do nível n [Tan96].

Os elementos ativos de uma camada são chamados de **entidades**. Entidades no mesmo nível em máquinas diferentes são chamadas **entidades pares**. A entidade na camada n utiliza os serviços oferecidos pela camada n - 1 e implementa serviços usados pela camada $n + 1$. Nesse caso, a camada n é chamada de **fornecedor do serviço** e a camada $n + 1$ é chamada de **usuário do serviço** [SLC95].

O serviço oferecido por um fornecedor é acessado por um usuário através de um **ponto de acesso ao serviço** (SAP[5]). A figura 2.2 mostra o sistema de comunicação do ponto de vista de dois usuários. Os serviços oferecidos por uma camada à outra é especificado por um conjunto de **primitivas de serviço** (ASPs[6]), que são trocadas através dos SAPs.

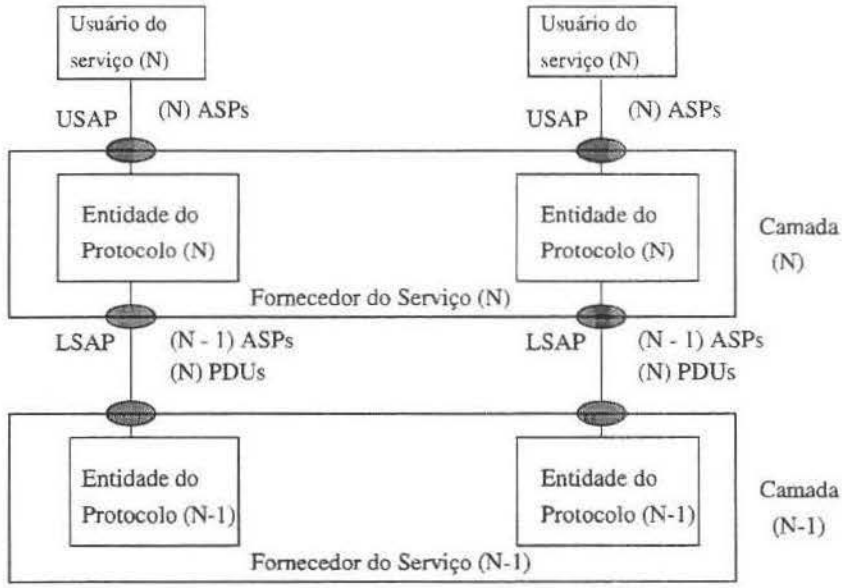

Figura 2.2: Serviço de Comunicação

A definição dos requisitos de comportamento para uma entidade de protocolo é chamada de **especificação do protocolo** [BP94]. Essa especificação envolve as interações nos **pontos de acesso ao serviço superior** (USAP) e **inferior** (LSAP), além de identificar diferentes tipos de **unidades de dados de protocolos** (PDUs[7] ou mensagens), que

[5]Service Access Point
[6]Abstract Service Primitives
[7]Protocol Data Unit

são codificadas e trocadas entre as entidades de protocolo através de serviços da camada inferior $(n-1)$.

A especificação do protocolo pode ser feita de várias maneiras, informal ou formalmente. A próxima seção apresenta as formas de especificação de protocolos, suas vantagens e desvantagens.

2.4.2 Formas de Especificação dos Protocolos de Comunicação

Como apresentado na seção anterior, os requisitos de comportamento do protocolo devem ser especificados. Um protocolo pode ser especificado informal ou formalmente. Uma especificação informal como a linguagem natural, pode até facilitar a leitura e o entendimento de quem deseja ter uma visão geral sobre o trabalho especificado, mas ela pode ser imprecisa e ambígüa. Além disso, torna-se difícil verificar se uma especificação informal está completa e correta [BP94]. Tal especificação pode tornar a implementação do protocolo mais difícil e mais propensa a erros, podendo levar a uma incompatibilidade com outras implementações do mesmo padrão de protocolo.

Devido a esses fatores, a especificação formal faz-se necessária para a representação das funções de um protocolo. Para especificar formalmente um protocolo, diferentes descrições formais foram propostas. Dentre elas destacam-se as Máquinas Finitas de Estados (MFE), Máquinas Finitas de Estados Estendidas (MFEE), Redes de Petri, gramáticas formais, além das linguagens de programação de alto nível.

Com o trabalho de padronização do modelo OSI, surgiu a necessidade de padronizar a forma de especificação dos protocolos e serviços da ISO. Desse trabalho, três linguagens para especificação de protocolos, denominadas de **técnicas de descrição formal** (FDT[8]), foram criadas: Estelle[9], LOTOS[10] e SDL[11]. A linguagem LOTOS é baseada em álgebra de processos enquanto que Estelle e SDL são baseadas em MFEE. A ISO padronizou também uma linguagem para especificação das unidades de dados do protocolo. Essa linguagem é denominada de ASN.1[12].

Testes de protocolos são baseados na especificação. Isso implica que uma especificação de referência deve ser oferecida, pois ela será a base para a derivação dos casos de teste e análise dos resultados.

[8] *Formal Description Techiniques*
[9] *Extended State Transition Language*
[10] *Language of Temporal Ordering Specification*
[11] *Specification and Description Language*
[12] Abstract Syntax Notation One

2.4.3 Teste de Conformidade

No contexto ISO, **testes de conformidade** são usados para verificar se a implementação do protocolo está conforme ao padrão especificado pela ISO. Esses testes têm por objetivo garantir a compatibilidade e a interoperabilidade entre as diversas implementações de um mesmo padrão do protocolo.

De forma geral, o termo "teste de conformidade" é usado para designar um conjunto de testes aplicados à implementação de um protocolo a partir de uma especificação, que não precisa ser um padrão. Nesse caso, testes de conformidade são usados com o objetivo de verificar a conformidade da implementação contra sua especificação. Nesse trabalho o termo **teste de conformidade** será usado considerando esse significado mais amplo.

Testes gerados a partir da especificação são denominados testes do tipo **caixa preta**, onde a implementação é dada como uma caixa preta, e somente seu comportamento observável pode ser testado contra o comportamento da especificação. Uma das vantagens desses testes é que eles podem ser realizados por terceiros sem a necessidade de fazer acesso ao código fonte.

Durante os testes de conformidade, sinais são enviados (entradas) e recebidos (saídas) da implementação. Os sinais da **implementação sob teste** (IUT[13]) são comparados aos sinais da especificação. As entradas fornecidas e as saídas esperadas são descritas em um conjunto de **casos de teste**.

Cada caso de teste tem um propósito de teste como, por exemplo, verificar a habilidade de uma IUT abrir uma conexão. Casos de teste podem ser agrupados, formando assim um **grupo de teste**. Grupos de teste são usados para oferecer uma ordenação lógica dos casos de teste. Um grupo de teste, por sua vez, pode ser unido a outros grupos de teste, formando um cenário de teste (*test suite*). Os casos de teste podem ser decompostos em **passos de teste**, que são formados por **eventos de teste**, que compreendem um conjunto de interações de entradas a serem fornecidas à IUT. Além das interações de entradas, um caso de teste pode incluir informações para analisar as interações de saída, recebidas da IUT em resposta a uma interação de entrada [ISO88b].

Limitações práticas impossibilitam que uma implementação seja testada exaustivamente, além das restrições econômicas, que podem limitar os testes ainda mais. Portanto, a ISO IS9646-1 recomenda que quatro tipos de teste sejam aplicados à uma implementação [ISO88b], [Ray87]:

- **testes de interconexão básica**: detectam casos graves de não conformidade como, por exemplo, quando uma IUT não consegue se conectar com outra IUT ou se as principais características do padrão do protocolo não foram implementadas;

[13]Implementation Under Test

- **testes de capacidade**: verificam se a capacidade observável de uma IUT está de acordo com os requisitos de conformidade estática do protocolo;
- **testes de comportamento**: provêem testes tão completos quanto possíveis das necessidades dos requisitos de conformidade dinâmica especificados pelo padrão do protocolo, dentro das aptidões da IUT;
- **testes de resolução de conformidade**: fornecem diagnósticos mais definitivos possíveis para resolver se uma implementação satisfaz requisitos particulares. Esses testes não são padronizados.

Além dos testes de conformidade, a ISO propõe outros tipos de testes: **testes de interoperabilidade**, que verificam se duas IUTs conseguem interagir; **testes de desempenho**, que medem as características de desempenho de uma IUT, tais como *throughput* e tempo de resposta sob várias condições; e os **testes de robustez**, que determinam quão bem uma IUT se recupera de situações de erro.

Para descrever, de forma precisa, os casos de teste, a ISO desenvolveu uma notação denominada TTCN[14] [PM92]. Esta notação foi projetada para representar todos os atributos de um *test suite* especificados pela norma 9646 [ISO88b]. Como o nome sugere, TTCN consiste na combinação de dois tipos de notação: uma na forma de árvore, usada para descrever o comportamento dinâmico; e uma na forma de componente tabular, usado para simplificar a representação de todos os elementos estáticos (tipos de dados, formatos de PDUs e ASPs, e veredictos associados com eventos de teste). Uma apresentação detalhada de TTCN pode ser encontrada em [PM92].

2.4.4 Arquiteturas de Teste

Implementações de protocolos podem ser testadas considerando uma entidade como uma única camada ou uma entidade com múltiplas camadas. A IUT deve ser estimulada pelas camadas inferior e superior, e as suas reações devem ser observadas.

O estímulo à IUT é feito pelo envio ou recebimento de primitivas de serviço por uma entidade denominada **testador**. Essas primitivas são também chamadas de **primitivas de serviço abstratas** (ASPs). Os SAPs usados pelo testador com este objetivo são chamados de **pontos de controle e observação** (PCO[15]) [Lin90].

Um testador pode ser funcionalmente dividido em dois testadores, que são chamados de **testador inferior** (LT) e **testador superior** (UT). O LT fornece, durante a execução dos testes, controle e observação no PCO inferior, envia e recebe primitivas (N - 1 ASPs) e mensagens (N PDUs). O UT fornece controle e observação no PCO superior da IUT.

[14] *Tree and Tabular Combined Notation*
[15] *Point of Control and Observation*

A ISO 9646 distingue as arquiteturas de testes em *locais* e *externas* [ISO88a]. Na arquitetura de teste local, a comunicação entre a IUT e o Testador Inferior (LT) e Superior (UT) é feita através dos Pontos de Acesso ao Serviço Inferior (LSAP) e Superior (USAP). Nessa arquitetura, os testadores são sincronizados por um módulo de coordenação de testes. Os testadores, o módulo de coordenação de teste e a IUT residem no mesmo sistema. A figura 2.3 apresenta uma visão geral da arquitetura de teste local, retirada do padrão ISO 9646-1 [ISO88a].

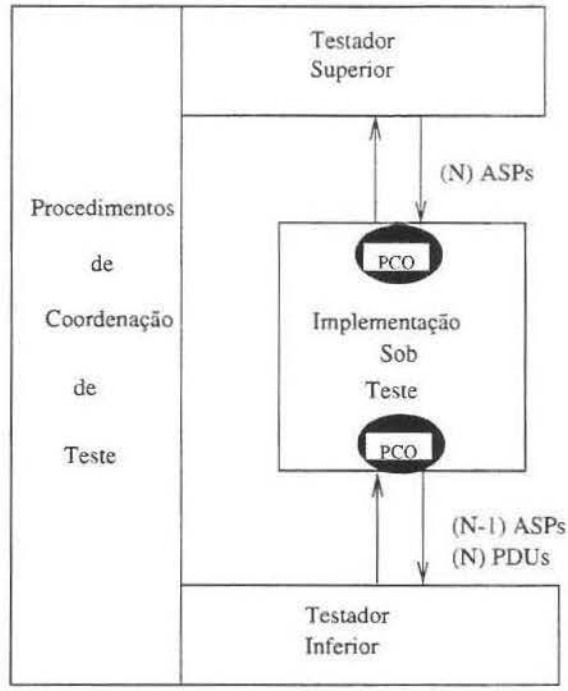

Figura 2.3: Arquitetura de teste local

Uma arquitetura de teste externa pode ser *distribuída*, *coordenada* ou *remota*. Nesse tipo de arquitetura, o testador inferior (LT) reside no sistema de teste (TS), em um computador distinto de onde reside a IUT, chamado de sistema em teste (SUT). A figura 2.4 apresenta uma arquitetura de teste distribuída, onde o sistema de teste e o sistema em teste são conectados através de um rede de comunicação [BP94].

A diferença entre a arquitetura de teste coordenada e uma distribuída é que a arquitetura coordenada possui um protocolo que oferece coordenação e sincronização de ações de testes de ambos os testadores (UT e LT) à distância.

A arquitetura de teste remota exclui o SAP superior do processo de teste, pois em algumas situações práticas, a IUT está embutida em um sistema complexo de forma que o testador superior não pode ser incluído dentro do sistema em teste.

A vantagem das arquiteturas remota e distribuída é que o sistema de teste reside em

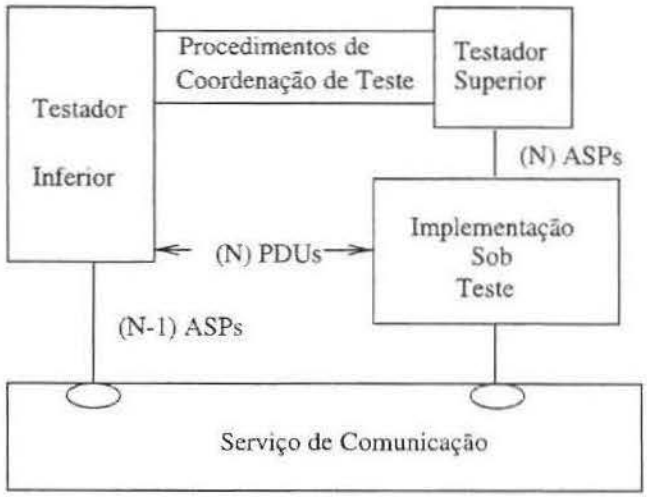

Figura 2.4: Arquitetura de teste distribuída

um computador separado do sistema em teste, possibilitando a comunicação à distância com vários sistemas em teste. Cada arquitetura é adequada para uma situação de teste diferente, sendo que estas arquiteturas têm diferentes impactos sobre a testabilidade da IUT [BP94]. Um estudo sobre essas arquiteturas e o padrão ISO 9646 em geral pode ser encontrado em [ISO88a], [ISO88b] e [Ray87].

2.5 Resumo

Este capítulo apresentou os fundamentos da atividade de teste de maneira geral e, em especial, os fundamentos dos testes de protocolos de comunicação.

O processo de teste é composto de três atividades principais: geração, execução e análise dos resultados. Este trabalho se concentra na primeira atividade do processo de teste. Mais especificamente, na geração de casos de teste para protocolos de comunicação.

Os métodos de seleção de casos de teste podem ser classificados de acordo com o modelo base usado para os testes. Dessa forma, os testes podem ser classificados como: testes baseados na especificação (**testes caixa preta**), testes baseados na implementação (**testes caixa branca**) e testes baseados em falhas (ver seções 2.2 e 2.3).

Protocolos de comunicação são regras que governam a comunicação entre os diferentes componentes de um sistema distribuído. Para organizar a complexidade dessas regras, a ISO definiu um modelo de referência denominado de **Modelo OSI**. Esse modelo tem por objetivo estruturar a rede como um conjunto de camadas hierárquicas. A seção 2.4.1 apresenta os principais conceitos envolvidos com este modelo.

Existem várias técnicas de especificação formal para protocolos de comunicação. Três

linguagens foram padronizadas pela ISO: Estelle, Lotos e SDL. Uma especificação pode levar a várias implementações diferentes de um protocolo. É muito importante testar essas implementações contra a especificação para garantir a compatibilidade entre as implementações de um mesmo protocolo. Esse tipo de teste é chamado de **teste de conformidade**.

Normalmente, em protocolos de comunicação, os teste são gerados a partir da especificação. Estes testes são chamados de testes caixa preta. Um caso de teste define uma seqüência finita de interações de entrada para ser aplicada a uma IUT. Esta seqüência normalmente inclui as saídas esperadas para a atividade de análise de resultados. Os casos de teste gerados podem ser especificados através de TTCN, uma notação para especificação de casos de teste padronizada pela ISO.

Para que implementações de protocolos possam ser testadas, a ISO define uma série de arquiteturas. Essas arquiteturas estão divididas em arquiteturas locais e externas. A seção 2.4.4 apresenta uma breve descrição dessas arquiteturas.

Este trabalho se concentra na geração de casos de teste para protocolos de comunicação a partir de MFEE. Os conceitos apresentados nesse capítulos servirão como base para esse trabalho. A definição formal de MFEE e os trabalhos relacionados são apresentados no próximo capítulo.

Capítulo 3

Testes Baseados em Máquinas de Estados

O uso de linguagem natural para especificação de sistemas leva a ambigüidades e são difíceis para verificar sua completude e corretude. Para resolver esses problemas surgiram as várias formas de especificação, como apresentadas no capítulo anterior. Dentre essas formas de especificação, as que se destacam são: as FDTs (Estelle, Lotos e SDL), as Máquinas Finitas de Estado (MFE) e as Máquinas Finitas de Estado Estendidas (MFEE). Neste trabalho, são mostrados as que partem da especificação na forma de MFE e MFEE.

De uma forma geral, máquinas de estados são largamente usadas para especificação e teste de protocolos de comunicação. Em muitos casos, especificações feitas através de Estelle ou SDL são transformadas em MFE e, dessa forma, testes já conhecidos para MFE são aplicados a esses protocolos.

Neste capítulo serão apresentados os conceitos e trabalhos relacionados a máquinas de estados. Inicialmente, uma definição formal de MFE e MFEE é apresentada. Em seguida, são apresentados os principais métodos para geração de testes baseados em MFE e MFEE. Existem muitos trabalhos para geração de testes de protocolo. Entretanto, serão descritos somente os que contribuíram e influenciaram a elaboração desse trabalho.

3.1 Definições de Máquinas de Estados

Devido a natureza reativa dos protocolos de comunicação, as técnicas de descrição formal que permitem definir a ordem com que as interações ocorrem são freqüentemente usadas. Como máquinas de estado possibilitam essa definição, muitos trabalhos em teste de protocolos são baseados em MFE e MFEE [BP94].

A especificação de um sistema pode ser dividido em duas partes: a parte de controle e a parte de dados. Especificar a parte de controle consiste em representar os estados, as

entradas e as saídas decorrentes da mudança de estado. Na parte de dados são descritos os tipos de dados trocados entre os protocolos e o formato desses dados.

A representação da parte de controle normalmente é feita através de MFE. A parte de dados normalmente é representada por uma linguagem de alto nível, como por exemplo, ASN.1.

Uma Máquina Finita de Estados (MFE) é um modelo que serve para mostrar a variação de estados de um sistema ao longo do tempo. A máquina pode mudar de um estado em resposta a uma entrada (ou evento), podendo produzir uma saída. As mudanças de estado são denominadas de **transições** e as saídas são produzidas por **ações** [Mar98].

Uma máquina de estado pode ser representada graficamente por um diagrama de transição de estados, que é um dígrafo onde os nós representam os estados e as arestas, as transições. As arestas são rotuladas pelos eventos que causam a transição e pela ação que é chamada em resposta ao evento. 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. A seguir são apresentadas as definições formais de MFE e MFEE.

3.1.1 Definição Formal de MFE

Uma MFE M é formalmente representada como uma 5-tupla $< S, s_0, I, O, g >$, onde:

S é um conjunto não vazio de estados;

s_0 representa o estado inicial;

I é um conjunto finito de entradas;

O é um conjunto finito de saídas;

g é a função de transição de estado; é definida como $g : S \times I \rightarrow S \times O$;

A função g representa o comportamento da MFE. A expressão $g(s_m, i) = (s_n, o)$ representa a transição do estado s_m para o estado s_n em resposta à ocorrência do evento i, produzindo a saída o [HUK95].

A figura 3.1 apresenta um exemplo de uma MFE. Neste exemplo, o conjunto dos estados é igual a $S = \{S0, S1, S2, S3\}$, sendo $S0$ o estado inicial da MFE. O conjunto de entradas é igual a $I = \{a, b, c, d, e, f\}$ e as saídas são $O = \{x, y, z\}$.

3.1.2 Propriedades

Para que testes de transição de estados possam ser gerados de forma semi-automática é necessário que a MFE satisfaça algumas das seguintes propriedades:

- **Mealy**: uma Máquina de *Mealy* é um modelo abstrato consistindo de um número finito de estados, entradas e saídas associadas às entradas.

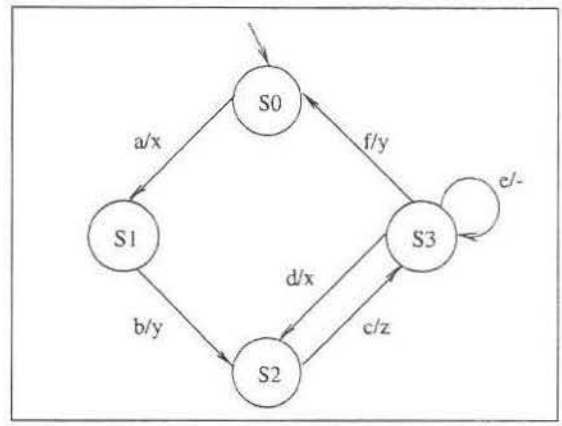

Figura 3.1: Exemplo de uma MFE

- **Determinística**: uma MFE M é determinística quando $|g(s,i)| = 1$ para todo $s \in S$ e para todo $i \in I$, ou seja, existe somente uma função de transição em um determinado estado associada a uma entrada. Se existir mais de uma transição partindo de um estado s_i com a mesma entrada i, diz-se que M é **não determinística**.

- **Completa**: uma MFE M é completa (ou completamente especificada) se, para cada estado $s_i \in S$, existe uma função de transição g associada a cada símbolo de entrada $i \in I$. Se, em um determinado estado não existe uma função de transição g associada a uma entrada i, diz-se que M é **parcialmente especificada** ou **incompleta**.

- **Fortemente conexa**: uma MFE é fortemente conexa quando, para cada par de estados (s_i, s_j), existe um caminho conectando s_i a s_j. Um caminho é uma seqüência finita de arestas adjacentes. Em muitos casos é necessário somente voltar ao estado inicial a partir de qualquer estado. Quando isso ocorre diz-se que a MFE é **inicialmente conexa**.

- **Mínima**: uma MFE é mínima quando não possui estados equivalentes. Dois estados s_i e s_j são equivalentes quando toda seqüência de entradas começando de s_i produz a mesma seqüência de saídas quando começando em s_j.

Vários métodos de geração de seqüências de teste baseados em MFE foram desenvolvidos. Isso se deve ao fato de que as MFEs são largamente usadas para representação de protocolos de comunicação, e porque técnicas de descrição formal, como Estelle e SDL, são baseadas em MFE. Conseqüentemente, sua transformação se dá quase diretamente.

A maioria dos métodos de geração de testes são criados para as MFEs que satisfaçam todas as propriedades definidas acima. Contudo, muitas especificações de protocolos não satisfazem essas propriedades [BP94]. Alguns métodos são apresentados em [BP94] para gerar testes a partir de MFE parcialmente especificadas e não determinísticas.

Um modelo de MFE é muito restrito para definir os aspectos de controle e dados de um protocolo. Para suprir essa necessidade, os modelos de Máquinas Finitas de Estados Estendidas (MFEE) são freqüentemente usados para especificar tanto os aspectos de controle quanto de dados [BP94].

3.1.3 Definição Formal de MFEE

As MFEEs estendem as MFEs através da inclusão de variáveis de contexto, predicados e ações. Uma transição de uma MFEE não é só caracterizada pelo estado origem, estado destino e interação (ou evento) de entrada, como em uma MFE, mas também por predicados e ações.

Uma MFEE é formalmente representada como uma 8-tupla $< S, s_0, I, O, V, P, A, g >$, onde [HLJ95] e [RDT95]:

S é um conjunto não vazio de estados;

s_0 representa 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 estado definida como
$g : S \times I \times P(V) \rightarrow S \times O \times A(V))$;

Uma transição pode ser representada como $s_i \xrightarrow{T} s_j$, onde s_i é o estado origem da transição T e s_j seu estado destino. Uma transição pode ser dividida em duas partes: 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 as variáveis bem como os parâmetros das entradas. A parte da ação pode conter eventos de saídas e um número de comandos envolvendo as variáveis. Uma transição é disparada quando a parte da condição é satisfeita. Quando 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.

A figura 3.2 apresenta um exemplo de uma MFEE. Neste exemplo, o conjunto dos estados é igual a $S = \{S0, S1, S2\}$, sendo $S0$ o estado inicial da MFE. O conjunto de entradas é igual a $I = \{a1, a2, a3\}$ e as saídas são $O = \{b1, b2, b3\}$. Esta MFEE possui

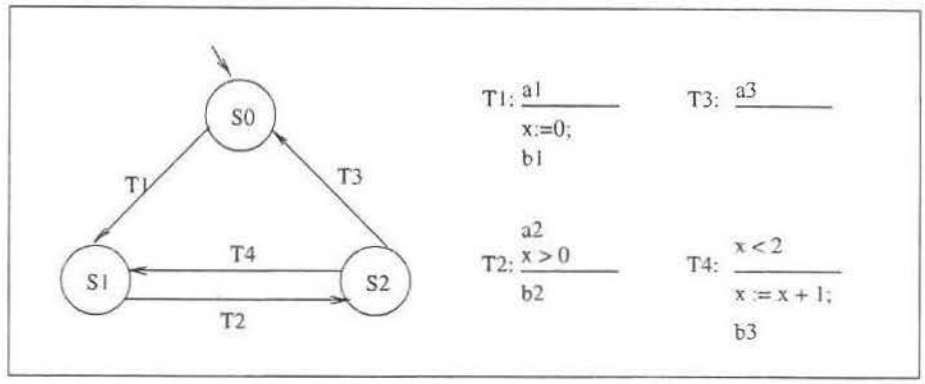

Figura 3.2: Exemplo de uma MFEE

uma variável, x, que pertence a uma ação na transição $T1$ e $T4$ e a um predicado da transição $T2$.

É possível definir **transições espontâneas** em MFEE. Essas transições não dependem de nenhuma entrada e estão associadas aos predicados. Quando o predicado for verdadeiro a transição é disparada. Um exemplo comum do uso desse tipo de transição é a especificação da condição de *time out*. Uma variável incrementa o tempo em que a máquina permanece em um determinado estado. Quando o valor dessa variável for igual ao tempo máximo, o predicado é habilitado e, conseqüentemente, a transição associada é executada. Esse tipo de transição é um dos problemas para geração de teste, pois, como os testes gerados são do tipo caixa preta, não há um controle sobre os valores das variáveis internas.

Nas próximas seções serão apresentados os métodos e trabalhos relacionados à geração de teste de protocolos baseados em MFE e MFEE. Muitos outros trabalhos foram realizados, mas não serão apresentados pois fogem do escopo dessa dissertação.

3.2 Geração de Testes Baseados em MFE

Os métodos de geração de teste baseados em MFE normalmente tratam o fluxo de controle do protocolo. Na década passada muitos métodos para geração de testes baseados no fluxo de controle foram propostos.

Os métodos que tratam a parte de controle do protocolo são baseados nas técnicas de testes de transições de estado apresentado na seção 2.3.1. Estes testes são também denominados de teste de fluxo de controle.

Técnicas de teste de transição de estados têm como objetivo detectar falhas de transição (ou saída) e de transferência. Uma transição com uma saída incorreta possui uma falha de saída. Esse tipo de falha é mais fácil de ser observada. Uma transição onde o

próximo estado é escolhido incorretamente, possui uma falha de transferência. Dado que os estados não são diretamente observados, essas falhas são relativamente mais difíceis de serem detectadas [CVI90].

Vários métodos de geração de teste foram propostos. Alguns deles serão apresentados nas próximas seções.

3.2.1 Testes Baseados em MFEs Completas e Determinísticas

Os métodos para geração de teste a partir de MFEs completamente especificadas e determinísticas são divididos em: métodos de transição de estados e métodos de identificação de estados. Métodos de identificação de estados possuem mecanismos para identificar o próximo estado que será alcançado depois da execução de uma determinada transição. Esses mecanismos permitem identificar falhas de transferência. Os métodos de geração de teste que possuem esses mecanismos são: método U, D e W.

A seguir é apresentada uma breve descrição de cada um desses métodos. Um estudo comparativo desses métodos pode ser encontrado em [SL89].

Método T

O método T (*transition tour*) é relativamente simples comparado aos outros métodos de geração de teste baseados em MFE. Este método assume que a MFE é mínima e fortemente conexa. Os primeiros estudos afirmavam a necessidade da máquina ser completamente especificada, mas testes realizados em [SL89] mostraram que é possível gerar testes para máquinas parcialmente especificadas.

O método T gera caminhos na máquina até que todas as transições tenham sido visitadas pelo menos uma vez [SL89]. Um **caminho** é uma seqüência de transições que parte do estado inicial, passando pelos estados da MFE e voltando ao estado inicial.

Esse método detecta falhas de transição (ou de saída), mas não detecta falhas de transferência [SL89], pois não possui nenhum mecanismo para identificação de estados.

Otimizações desse método foram feitas usando teoria dos grafos. A idéia é encontrar um caminho Euleriano, que é uma seqüência de transições que começa e termina no mesmo estado e que contém todas as transições da MFE exatamente uma vez [BDA96].

Método U

O método U assume que a MFE é mínima, fortemente conexa e completamente especificada. Esse método encontra uma seqüência única de entrada e saída (UIO) para cada estado a máquina. Uma seqüência UIO para um estado de uma máquina M é um comportamento de entrada/saída que não é exibido por nenhum outro estado de M [SL89].

Os testes gerados por esse método cobrem todas as transições e verificam as seqüências UIO para cada estado. Dessa forma, esse método é capaz de detectar falhas de transição e de transferência.

Em [SL89] experimentos foram realizados em MFE parcialmente especificadas e foi possível gerar seqüências UIO para essas máquinas.

Método D

Esse método gera seqüências DS (*Distinguishing Sequences*). Essas seqüências são definidas como um conjunto de entradas que gera um conjunto de saídas diferentes para cada estado da MFE. As propriedades que a MFE deve satisfazer para aplicar esse método são: mínima, fortemente conexa e completamente especificada [SL89].

A desvantagem desse método é que poucas MFEs possuem seqüências DS, e quando essas seqüências existem elas podem ser muito longas, tornando o uso desse método bastante limitado [Maz95].

Método W

Como no método D, este método assume que a MFE é mínima, fortemente conexa e completamente especificada. Para MFEs que não possuem seqüências DS, o método W define seqüências DS parciais. Cada uma dessas seqüências distingue um estado específico da MFE de um subconjunto dos estados restantes, ao invés de distinguir esse estado de todos os estados da MFE.

O conjunto completo dessas seqüências de entrada para a MFE é chamado de conjunto característico W da MFE. Esse conjunto consiste de seqüências de entrada tal que os últimos símbolos da saída observados a partir da aplicação dessa seqüência (em uma ordem especificada) são diferentes para cada estado da MFE.

3.2.2 Testes Baseados em MFEs Parcialmente Especificadas

Muitos protocolos não são completamente especificados, pois em sistemas de comunicação real, nem todas as seqüências de interações são possíveis [BP94]. Uma característica das MFEs parcialmente especificadas é que não existe transição definida para todas as entradas da MFE. Essas transições não precisam ser testadas, contudo diferentes interpretações para transições indefinidas têm impacto nos testes [BP94]:

- **Transições definidas implicitamente**: nesse caso são adotadas algumas **suposições de completude**. Essa suposição é baseada no fato de que toda implementação pode ser representada por uma máquina completa que nunca recusa uma entrada. Essas transições implícitas são implementadas como transições para o mesmo estado

com saída nula, ou como transições que vão para um estado de erro. A convenção usada na implementação deve ser conhecida para os propósitos de teste. Com isso, MFE parciais são substituídas por MFE completas quase equivalentes, evitando o problema de derivação de testes para máquinas parciais.

- **Transições indefinidas por** *default*: nesse caso, a MFE parcial é interpretada como completa e testes são gerados para as transições não implementadas, denominadas de *don't care*. Essas transições podem ser testadas para determinar a reação da IUT a essas entradas inesperadas. Esses testes são denominados de teste de robustez.

- **Transições proibidas**: em alguns casos, o comportamento do sistema não é completamente especificado porque seu ambiente nunca executará certas seqüências. Nesse caso, um teste não pode submeter certas entradas para determinados estados da MFE parcial. Antes de considerar certas transições como proibidas, essas devem ser analisadas com muito cuidado para que casos de teste que possam revelar falhas no sistema não sejam descartados.

Construir um conjunto de casos de teste para máquinas parcialmente especificadas é complicado pelo fato de que a minimalidade dessa MFE não pode ser verificada. Se a máquina não é mínima, então alguns de seus estados podem não ser distingüíveis tanto na especificação quanto na implementação. Assim a abordagem de identificação de transição, usada pelo métodos de geração de teste a partir MFE completas, não são mais aplicáveis à essas máquinas [BP94].

Vários métodos têm sido propostos para testar MFE parcialmente especificadas, como por exemplo o método HSI, que será apresentado a seguir.

Método HSI

O método HSI *(Harmonized State Identification Set)* é baseado no método W. Este método usa subconjuntos do conjunto W gerado para identificação de estados. Conjuntos HSI são tuplas de conjuntos $D_0, D_1, \ldots, D_{n-1}$, onde D_i é um conjunto de prefixos de seqüências em W e n é o número de estados da MFE [TPB96].

Para encontrar um HSI mínimo foram propostas soluções baseadas em heurísticas. Um estudo sobre a aplicabilidade desse método pode ser encontrado em [TPB96].

3.2.3 Testes Baseados em MFEs Não Determinísticas

Devido a sua natureza, sistemas não determinísticos são mais difíceis de serem analisados do que os determinísticos. Contudo, constantes estudos são realizados nessa área para geração de teste para MFEs não determinísticas [BP94].

Para implementações de protocolos não determinísticos, um mesmo caso de teste pode levar a diferentes possibilidades de reação da implementação. Dessa forma, esse mesmo caso de teste poderia ser executado mais de uma vez, até que todas as possíveis saídas sejam obtidas.

3.3 Geração de Testes Baseados em MFEE

Como mencionado anteriormente, MFE é muito restrita para definir todos os aspectos de um protocolo. Portanto, MFEEs são freqüentemente usadas. Através de MFEE é possível especificar variáveis e os parâmetros das interações de entrada/saída. Além dos predicados e ações associadas às transições, como definido na seção 3.1.3.

A notação usada para descrever os predicados e as ações das transições normalmente é emprestada das linguagens de programação de alto nível [BP94]. Portanto, os métodos de geração de teste para MFEE são muitas vezes baseados em técnicas de testes caixa branca, apresentadas em 2.2.

3.3.1 Dificuldades

Uma das questões bastante discutida na geração de testes partindo de MFEE é a geração de testes envolvendo predicados e ações. Mais especificamente quando as ações podem incluir laços. A questão de decidir quais parâmetros de entrada poderiam ser usados para executar uma determinada transição torna-se indecidível [BP94]. Essa questão é chamada de **problema da executabilidade**, onde o problema está em encontrar caminhos na MFEE onde todas as transições pertencentes a estes caminho são executáveis. Uma transição é **executável** se em um determinado estado S, os parâmetros da interação de entrada e os valores das variáveis associadas à transição são tais que o predicado habilitador da transição é verdadeiro [RDT95].

Para tratar do problema da executabilidade vários trabalhos criaram métodos que incluem heurísticas que tendem a solucionar, mesmo que em parte, este problema [RDT95], [BDAR97] e [HLJ95].

Os métodos de geração de testes a partir de MFEEs podem ser classificados em várias abordagens, dependendo da forma com que a MFEE é especificada e também os critérios de teste que se deseja adotar para tratar tanto a parte de controle quanto a parte de dados do protocolo. A seguir uma classificação e definição para essas abordagens é apresentada.

3.3.2 Abordagens de Teste para MFEEs

As abordagens usadas para geração de teste baseados em MFEE são muito variadas. Para facilitar o entendimento dessas abordagens, Bochmann propõe a seguinte classificação [BD97]: (*i*) separar fluxo de controle e fluxo de dados; (*ii*) transformar a especificação na forma normal; e (*iii*) expandir a MFEE (*unfolding*). A partir do estudo dos trabalhos relacionados à geração de teste para MFEE, acrescentou-se mais uma abordagem: (*iv*) transformar a especificação em gramática.

A seguir é dada uma descrição de cada uma dessas abordagens:

1. **Separar Fluxo de Controle e Fluxo de Dados:**

   Nessa abordagem os aspectos controle e dados são separados da especificação, e testes são gerados para cobrir esses aspectos. O fluxo de controle é representado por uma MFE, onde técnicas de teste de transição de estados são aplicadas à essa MFE.

   Critérios baseados em análise de fluxo de dados envolvem os parâmetros das primitivas de entrada e saída e as variáveis de contexto. Selecionando-se casos de teste apropriados envolvendo transições apropriadas, é possível satisfazer os critérios de testes orientados ao fluxo de dados, como por exemplo, todas as definições ou todos os usos [BP94]. Uma descrição desses critérios é dada na seção 2.2.

2. **Transformar a Especificação na Forma Normal:**

   As ações associadas às transições podem conter comandos que influenciam o fluxo de controle do protocolo, tais como, instruções condicionais (*if, case*) e instruções de repetição (*while, repeat, for*). Para simplificar a geração de seqüências de teste, Sarikaya [SBC87] propôs um método para transformar a especificação em uma equivalente contendo somente as chamadas **transições na forma normal**.

   Uma transição na forma normal não contêm instruções que influenciam o fluxo de controle do protocolo. Dessa forma, a geração de teste para cobrir a parte de controle da especificação pode ser feita usando técnicas de testes de transição de estados.

   O método de transformação de uma especificação em uma forma normal equivalente usa a idéia de criar uma nova transição para cada caminho distinto dentro de uma ação. Se, por exemplo, uma ação em uma dada transição inclui um comando *if*, essa transição deve ser substituída por duas outras transições, uma caso o predicado seja verdadeiro e outra para o caso do predicado ser falso [BD97].

   Esse tipo de abordagem é normalmente usada para especificação baseada em Estelle, pois essa linguagem possui comandos, como por exemplo *if, case*, que permite representar aspectos de controle através das ações.

3. **Expandir a MFFE** (*Unfolding*) :

   Uma MFEE pode ser vista como uma notação compacta de uma MFE. Dessa forma, é possível transformar uma MFEE em uma MFE equivalente, expandindo os valores das variáveis e parâmetros associadas às transições [BP94]. Essa abordagem é chamada de *unfolding*.

   A principal razão para o uso de *unfolding* é que os métodos de teste de transição de estados podem ser aplicados à especificação. Um dos problemas que ocorre com essa solução é que, dependendo da MFEE, sua expansão pode levar a uma explosão de estados.

   A transformação de uma MFEE em uma MFE equivalente compreende a eliminação das 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. Essas variáveis são removidas para que a habilitação de uma transição dependa somente do estado corrente e da entrada. Dessa forma, as variáveis não influenciarão o fluxo de controle do protocolo.

   Nem sempre é possível aplicar *unfolding* em uma MFEE. Se todas as variáveis que influenciam o fluxo de controle do protocolo têm um domínio finito, então uma MFE equivalente existe. Por outro lado, se alguma variável não tiver um domínio finito, então o conjunto de estados resultantes é infinito, não sendo possível transformar essa MFEE em uma MFE equivalente [HUK95].

4. **Transformar a Especificação em Gramática**:

   O uso de gramática de atributos para especificar e gerar casos de teste foi inicialmente proposto por Duncan e Hutchison em [DH81] para testes de *software*. Nesse trabalho uma sintaxe em BNF foi proposta para especificar a gramática de teste. Baseado nessa idéia, um método de geração de testes para protocolos de comunicação usando gramática de atributos foi proposto em [UP83].

   A transformação da especificação em gramática de atributos permite representar todos os aspectos do protocolo (controle, dados, predicados e ações) usando uma única notação. A partir dessa gramática, técnicas de teste baseadas em gramática podem ser aplicadas. Com a aplicação dessas técnicas é possível gerar casos de teste cobrindo aspectos de controle e dados, além do formato das interações de entrada e saída do protocolo.

3.4 Trabalhos Relacionados

Nessa seção será apresentada brevemente os trabalhos de geração de teste para MFE e MFEE. A tabela 3.1 apresenta um quadro demonstrativo contendo a forma de especificação, a abordagem e o método usado pelos métodos discutidos nesse capítulo.

Trabalho	Forma de Especificação	Abordagem	Método
[Maz95]	Estelle		Método U
[TPB96]	MFE		Método HSI
[SBC87]	Estelle	Forma Normal	Teste Funcional
[Ura87]	Estelle	Forma Normal	Análise de Fluxo de Dados Estática
[UY91]	Estelle	Forma Normal	Teste de fluxo de dados
[CS87]	MFEE	*Unfolding*	Testes de Transição de Estados
[HUK95]	Estelle	Forma Normal e *Unfolding*	Testes de Transição de Estados
[HLJ95]	MFEE		Testes de Fluxo de Dados
[RDT95]	MFEE		Testes Combinando Fluxo de Controle e Fluxo de Dados
[BDAR97]	MFEE		Testes Combinando Fluxo de Controle e Fluxo de Dados
[VJL+94]	Estelle e ASN.1		Testes de Fluxo de Controle e Fluxo de Dados baseado em Restrições
[BKKW89]	MFEE e ASN.1	Separar Fluxo de Controle e Fluxo de Dados	Testes de Transição de Estados e Testes de Dados.
[CVI90]	Estelle	Forma Normal	Separação dos aspectos de fluxo de controle e dados.
[UP83]	MFEE	Transformar a especificação em gramática	Testes baseados em Gramática

Tabela 3.1: Quadro demonstrativo dos métodos de geração de teste estudados

Como já foi colocado nas seções anteriores, métodos baseados em MFE tratam somente a parte de controle do protocolo. Dessa forma, tanto o método proposto por [Maz95], quanto a ferramenta TAG(*Test Automatic Generation*) [TPB96] se preocupam somente com a parte de controle do protocolo. A diferença entre esses métodos é que a TAG gera testes para MFEs parcialmente especificadas.

A TAG permite a geração de casos de teste seletivos e completos. Os testes seletivos são gerados para obter testes que cobrem uma determinada transição. Nos testes completos, seqüências de teste são geradas para todas as transições da MFE.

Formas diferentes de abordagens são usadas para tratar a MFEE. A partir dessas abordagens diferentes métodos de teste são propostos. Em [SBC87], testes são gerados para cobrir as funções do protocolo. Esse método usa a técnica de teste funcional descrita na seção 2.3. Uma das limitações desse método é que ele requer um grande esforço para identificar as funções e seus relacionamentos, em caso de protocolos mais complexos [UY91].

Alguns trabalhos partem para técnicas de testes de fluxo de dados do protocolo. Em [Ura87] seqüências de teste são geradas através de análise de fluxo de dados estática. Esse trabalho não cobre o fluxo de controle protocolo e, conforme [UY91], as seqüências de teste resultantes provêem somente uma cobertura marginal dos aspectos de fluxo de dados do protocolo.

Em [UY91], o método proposto em [Ura87] é melhorado para possibilitar uma melhor cobertura dos aspectos de fluxo de dados, mas ainda não considera o fluxo de controle do protocolo.

Para levar em consideração as necessidades do usuário, em [VJL+94] é apresentado um ambiente para geração de teste com o objetivo de cobrir os aspectos de controle e dados do protocolo. O método é baseado no mecanismo de restrições, que permite ao usuário especificar quais testes devem ser gerados a partir de um conjunto de entradas, estados e transições.

Os métodos apresentados em [BKKW89] e [CVI90] separam os aspectos de controle e dados da especificação e cobrem esses aspectos separadamente. Uma das características importantes apresentadas por [BKKW89] é que este trabalho apresenta um conjunto de ferramentas, denominado de *RNL Conformance Kit*, que permite o uso de diversas técnicas de testes diferentes para geração dos testes.

Com a preocupação de gerar casos de teste executáveis, surgiram vários trabalhos. Em [HLJ95], um método é proposto para manipular a executabilidade dos casos de teste usando análise de fluxo de dados. Um dos problemas desse método é que pode levar a uma explosão de estados, além de não considerar os aspectos de fluxo de controle da MFEE.

Em [RDT95], aspectos de fluxo de dados e controle são considerados pelo uso de um critério denominado CIUS para identificação dos estados. Esse critério gera uma única seqüência de entradas independente do contexto (condições das transições). Contudo, nem todas as MFEE possuem uma seqüência única para todos os estados.

Um outro método com a preocupação de gerar testes executáveis cobrindo dados e controle é apresentado em [BDAR97]. Neste trabalho é apresentado um método para manipular a executabilidade. Contudo, para especificações com laços ilimitados, esse método não é capaz de tornar um caminho executável.

Outra forma de tratar controle e dados e o problema da executabilidade é através do uso de *unfolding*. Os trabalhos apresentados em [CS87] e [HUK95] usam esta abordagem e de maneiras distintas criam métodos para expandir a MFEE, e dessa forma, geram testes usando técnicas de testes de transição de estados. Uma característica importante é que o trabalho apresentado em [CS87] trata dos ciclos da especificação através de expressões regulares.

Uma outra forma de gerar testes para MFEE é a partir de uma gramática. Em [UP83], um método de geração de teste é proposto. Nesse trabalho a especificação é transformada em gramática, e a partir dessa gramática testes são gerados através de técnicas de teste baseadas em gramática, tais como teste de sintaxe ou teste de domínio.

3.5 Resumo

O objetivo desse capítulo não é apresentar todos os trabalhos existentes em teste de protocolos, mas sim, colocar as principais características, vantagens e limitações dos trabalhos que influenciaram o método proposto nessa dissertação.

Para um entendimento dos conceitos usados, inicialmente foi apresentado as definições de MFE e MFEE. Em seguida, os critérios de testes usados pelos métodos de geração de teste para protocolos de comunicação foram discutidos.

Com esses conceitos em mente, a geração de teste baseada em MFE e MFEE é apresentada.

Para testes baseados em MFEE várias abordagens podem ser adotadas: separar fluxo de controle e fluxo de dados, transformar a especificação na forma normal, expandir a MFEE e transformar a especificação em gramática. A partir dessas abordagens técnicas de testes são aplicadas. Para tratar os aspectos de dados da MFEE, muitos dos métodos estudados partem para técnicas de teste caixa branca.

No próximo capítulo é apresentado o método proposto nesse trabalho, que visa aplicar técnicas de teste caixa preta para cobrir os diferentes aspectos de uma MFEE.

Capítulo 4

Método Proposto

A partir de estudos dos diversos métodos existentes para geração de teste, e da necessidade de obter um método que proporcionasse uma cobertura de todos os aspectos do protocolo, um método para geração de teste foi proposto.

Esse método parte de estudos que mostram que é possível testar todos os aspectos da especificação combinado diversas técnicas de teste caixa preta. Partindo desse princípio, este capítulo apresenta o método proposto para geração de teste. Inicialmente, são apresentadas as motivações que levaram à elaboração desse método. Em seguida, é descrito o método propriamente dito, com um detalhamento dos passos que o compõe.

4.1 Motivação

Vários métodos para geração de teste a partir de MFE e MFEE já foram propostos na literatura. Dentre esses métodos, muitos já foram implementados em ferramentas automatizadas, como apresentado na seção 3.4. Esses métodos normalmente usam técnicas de teste caixa branca para cobrir aspectos de dados do protocolo, tais como técnicas de teste de fluxo de dados.

A partir do estudo desses métodos e tendo como objetivo gerar testes para MFEE, surgiu a necessidade de um método de geração de teste cobrindo os aspectos de controle e dados do protocolo.

O método proposto nesse trabalho parte de estudos apresentados por Poston [Pos96], onde o autor mostra que é possível testar todos os requisitos da especificação de um *software* através de estratégias de teste caixa preta.

Esse método combina diversas técnicas de teste caixa preta para cobrir todos os aspectos do protocolo. A representação de todos esses aspectos é feita através de uma única notação, facilitando a aplicação de diferentes técnicas de teste.

As próximas seções mostram o método proposto. Quais os requisitos da especificação

que devem ser satisfeitos e quais as técnicas de teste caixa preta que devem ser aplicadas para cobrir todos os aspectos do protocolo.

4.2 Visão Geral do Método Proposto

O método proposto nesse trabalho é composto de três passos principais: (1) especificar os requisitos do protocolo através de uma MFEE; (2) transformar a especificação em um formato conveniente, denominado especificação de teste, que engloba dados e controle do protocolo; e (3) a partir dessa especificação de teste, gerar seqüências de testes através de técnicas de teste caixa preta, para cobrir os diferentes tipos de falhas do protocolo.

Uma visão geral do método proposto é apresentado na figura 4.1. A seguir são discutidos cada um dos passos que compõe este método. Em primeiro lugar são apresentados os requisitos que devem ser satisfeitos na especificação do protocolo. Tendo a especificação do protocolo, esta é transformada em uma especificação de teste, a fim de permitir a aplicação de diferentes técnicas de teste. Essas técnicas são apresentadas na seção 4.5.

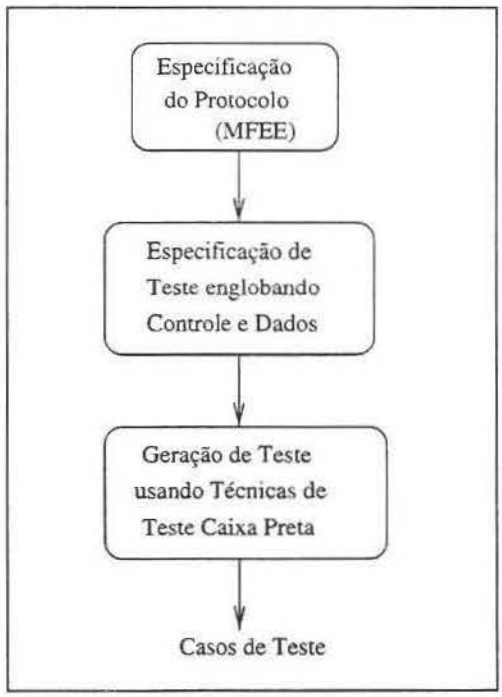

Figura 4.1: Visão Geral do Método Proposto

4.3 Requisitos da Especificação do Protocolo

O primeiro passo do método proposto é fornecer a especificação do protocolo de comunicação a ser testado. Essa especificação é dada através de uma MFEE.

As formas com que especificações em MFEE podem ser obtidas variam, desde do uso de tabelas até a criação de linguagens compiladas, que permitem ao usuário especificar a MFEE usando um determinado formato. Como Estelle e SDL são baseadas em máquinas de estados, elas podem ser usadas para essa especificação. Dessa forma, uma notação para representar a MFEE deve ser elaborada.

Tendo definido a linguagem de especificação a ser usada, alguns requisitos devem ser levados em conta para obter uma especificação do protocolo que possibilite a aplicação e cobertura de diferentes técnicas de teste caixa preta.

Inicialmente, a especificação deve ser dada na **forma normal**, como descrito em [SBC87], e apresentado na seção 3.3.2. Esse requisito é importante para garantir que toda a parte de controle da MFEE esteja representada explicitamente pelas transições entre os estados. Dessa forma, garante-se que não haja nenhuma informação de controle embutida nas ações. A importância desse requisito está na facilidade de aplicação das técnicas de teste para cobrir a parte de controle do protocolo. Assim, técnicas de teste de transição de estados podem ser aplicadas, não necessitando analisar as ações para cobrir a parte de controle do protocolo.

Outros requisitos importantes são: máquina de *Mealy*, mínima e inicialmente conexa. A propriedade de minimalidade é necessária para evitar a geração de casos de teste redundantes. Quanto à propriedade de inicialmente conexa, esta faz-se necessária para que técnicas de teste de transição de estados sejam aplicadas, pois esta propriedade garante que é possível retornar ao estado inicial a partir de qualquer estado. Não é necessário que a máquina seja completa, pois através do método usado é possível gerar testes para máquinas parcial ou completamente especificadas.

4.4 Especificação de Teste

O protocolo deve ser especificado através de uma notação conveniente, para que técnicas de testes caixa preta possam ser aplicadas. Essa especificação é denominada **especificação de teste**.

Alguns métodos utilizam grafos de fluxo de dados e fluxo de controle para derivação dos testes. Outros métodos usam gramáticas para representação de todos os aspectos do protocolo, como em [UP83]. O objetivo desse método é obter uma especificação genérica, onde testes possam ser derivados para cobrir todos os aspectos do protocolo.

Como diversas técnicas de teste caixa preta podem ser aplicadas, a especificação de

teste deve prover uma representação conveniente, que tenha informação necessária para possibilitar a aplicação de técnicas de teste. Por exemplo, para aplicar testes de domínio, é necessário que a especificação de teste tenha informações sobre os domínios dos dados a serem testados.

4.5 Técnicas de Teste Caixa Preta

O método proposto nesse trabalho usa diversas técnicas de teste caixa preta para cobrir para cobrir todos os aspectos do *software* [Pos96]. Os aspectos que devem ser testados a fim de cobrir todo o sistema em teste são:

- ações que o *software* pode realizar;
- os dados que as ações processam;
- expressões lógicas que irão limitar como as ações serão realizadas;
- eventos que sincronizam as ações do sistema com as ações externas;
- estados no qual a ação é permitida;

Cada um desses pontos corresponde a uma estratégia de teste, conforme apresentado na tabela 4.1. Uma especificação completa conterá definições para as ações, dados, parte lógica, eventos e estados. Gerar testes cobrindo todos esses aspectos, usando as estratégias de teste específicas, possibilitará cobrir todos os requisitos especificados. Se alguma parte do sistema não for coberto por essas técnicas de teste, diz-se que ações extras foram implementadas, que não constam da especificação. Nesse caso, deve-se considerar duas possibilidades: (*i*) essas ações são desnecessárias; ou (*ii*) ocorreu falha na especificação do sistema, onde ações não estão especificadas.

As próximas seções apresentam uma descrição geral de cada uma das técnicas de teste apresentadas na tabela 4.1.

4.5.1 Testes de Ações

No testes de ações (ou testes funcionais) é necessário criar pelo menos um caso de teste para cada ação ou função especificada. Deve-se criar casos de teste válidos e inválidos. Um caso de teste válido contém o nome e um valor válido para toda entrada da função e pelo menos uma saída esperada ou "efeito colateral". Um caso de teste inválido contém o nome e um valor inválido para pelo menos uma das funções que o caso de teste exercita e o nome de pelo menos uma saída [Pos96].

Técnicas de Teste Caixa Preta	Grupos de Falhas do Software					
	Falta de Ações	Ações incorretas causadas por erro no				Ações Extras
		Processamento de dados	Manipulação lógica	Tempo de eventos	Seqüência de estados	
Testes de Ações	√					
Testes de Dados		√				
Testes Lógicos			√			
Testes de Eventos				√		
Testes de Estados					√	
Código não exercitado						√

Tabela 4.1: Cobertura das Técnicas de Teste Caixa Preta

Técnicas para gerar testes de ações normalmente são baseadas em [How80], onde estratégias de teste funcionais são apresentadas. Na seção 2.3.3 é apresentado uma visão geral de teste funcional.

4.5.2 Testes de Dados

A especificação de um *software* deve conter uma descrição da interface oferecida pelo mesmo. Nesta descrição são identificadas as entradas, saídas e o espaço de valores possíveis que estas entradas e saídas podem ter, denominado **domínio**. Esta categoria de testes tem por objetivo a busca de falhas na manipulação de dados [Mar98].

As falhas na manipulação de dados podem ser classificados em duas categorias: (*i*) falhas de manipulação dos valores de dados primitivos, tais como números, listas ou cadeias de caracteres e, (*ii*) falhas de manipulação das estruturas de dados tais como registros, vetores ou arquivos.

Existem várias técnicas de teste de dados, as principais são: teste de partição de equivalência, domínio, valores limites e sintaxe. Teste de sintaxe são usados para encontrar falhas de manipulação de estruturas, enquanto que as outras técnicas se preocupam com falhas na manipulação dos valores dos dados primitivos.

A seção 2.3 apresenta uma visão geral dos testes de sintaxe e de domínio. Um estudo mais detalhado sobre essas técnicas de teste de dados pode ser encontrado em [Bei95].

4.5.3 Testes Lógicos

Testes lógicos (ou testes de predicados) têm por objetivo determinar se a lógica indicada na especificação foi corretamente implementada. Duas formas bastante usadas para derivação desses testes são: tabelas de decisão e grafos causa-efeito [Pos96].

Na especificação, a lógica pode ser expressa de quatro formas: (1) lógica baseada em tempo ou lógica temporal; (2) lógica aritmética, na qual equações aritméticas estão incluídas dentro das ações lógicas; (3) cálculo de predicados; e (4) lógica booleana. Muitos métodos são baseados em lógica booleana, onde os valores verdadeiro e falso são escolhidos para exercitar os predicados constantes da especificação. Uma descrição dos testes lógicos pode ser encontrado em [Pos96] e [Bei95].

4.5.4 Testes de Eventos

A ocorrência de um evento causa uma ação no sistema. Considere, por exemplo, um sistema que está realizando seu processamento corrente e chega uma mensagem de um sistema remoto. Com esse evento, o sistema corrente irá tomar uma determinada ação, como o tratamento dessa mensagem.

Técnicas de teste dirigidas a eventos geram casos de teste para exercitar todo evento pelo menos uma vez. Um conjunto de casos de teste que exercita muitos eventos diferentes é melhor para encontrar erros. Tais conjuntos de casos de teste são bem aplicados em sistemas de tempo real, orientados a objetos e sistemas que usam interface gráfica.

Normalmente, esses testes são combinados com testes de estados. Essas duas técnicas algumas vezes são impossíveis de serem separadas na prática [Pos96].

4.5.5 Testes de Estados

Testes de transição de estados são bastante usados em protocolos de comunicação. Diversos métodos foram propostos a fim de gerar testes para modelos baseados em transição de estados. Uma definição destes testes é apresentada na seção 2.3 e um estudo sobre esses métodos foi apresentado no capítulo 3.

4.6 Resumo

Neste capítulo, um método de geração de teste para protocolos de comunicação a partir de MFEE é proposto. O objetivo é usar um conjunto de técnicas de teste caixa preta para cobrir todos os aspectos do protocolo. Outra característica importante é usar uma única representação para todos os aspectos do protocolo, facilitando a aplicação de diferentes técnicas de teste.

O método apresentado é composto de três passos principais: (1) especificar o protocolo através de uma MFEE; (2) transformar a MFEE em uma especificação de de teste; e (3) a partir da especificação de teste, aplicar técnicas de teste caixa preta.

Com o objetivo de validar este método uma ferramenta foi implementada. Esta ferramenta será apresentada em detalhes no próximo capítulo.

Capítulo 5

CONDADO: Uma Ferramenta para Geração de Teste

Com o objetivo de validar o método proposto no capítulo 4, foi implementada uma ferramenta para geração de teste combinando controle e dados, denominada CONDADO. Este capítulo apresenta os principais componentes dessa ferramenta.

Para cobrir o método de geração de teste, uma linguagem para especificação de protocolo é descrita. A partir dessa linguagem é construída uma especificação de teste cobrindo as partes de controle e dados do protocolo. Dessa forma, técnicas de teste caixa preta são aplicadas a essa especificação para obtenção dos casos de teste.

A CONDADO combina três técnicas de teste **caixa preta**: testes de transição de estados, testes de sintaxe e testes de domínio. Dessa forma, a CONDADO possibilita a geração de teste cobrindo a parte de controle e os dados dos parâmetros de interações do protocolo.

Inicialmente, nesse capítulo, é apresentada uma visão geral da CONDADO. Em seguida, o ambiente ATIFS, no qual a ferramenta está inserida. A explicação da linguagem para especificação de protocolo e os componentes da CONDADO é auxiliada por um exemplo de uma especificação em MFEE. É apresentada também a forma da especificação de teste implementada pela CONDADO, e os métodos de geração de teste, finalizando com as vantagens e limitações dessa ferramenta.

5.1 Modelo Geral da ferramenta CONDADO

A partir do método proposto no capítulo 4, foi implementada uma ferramenta para geração de testes combinando controle e dados de protocolos de comunicação. Esta ferramenta, denominada CONDADO [SM98], realiza os passos 2 e 3 do método proposto: transformar a especificação do protocolo em uma especificação de teste, e gerar casos de teste usando

técnicas de teste caixa preta, respectivamente.

Inicialmente, são levantados os requisitos do protocolo, que podem ser representados por uma MFEE. Essa tarefa equivale ao primeiro passo do método proposto. Para especificar o protocolo, uma Linguagem para Especificação de Protocolo, denominada LEP, é proposta.

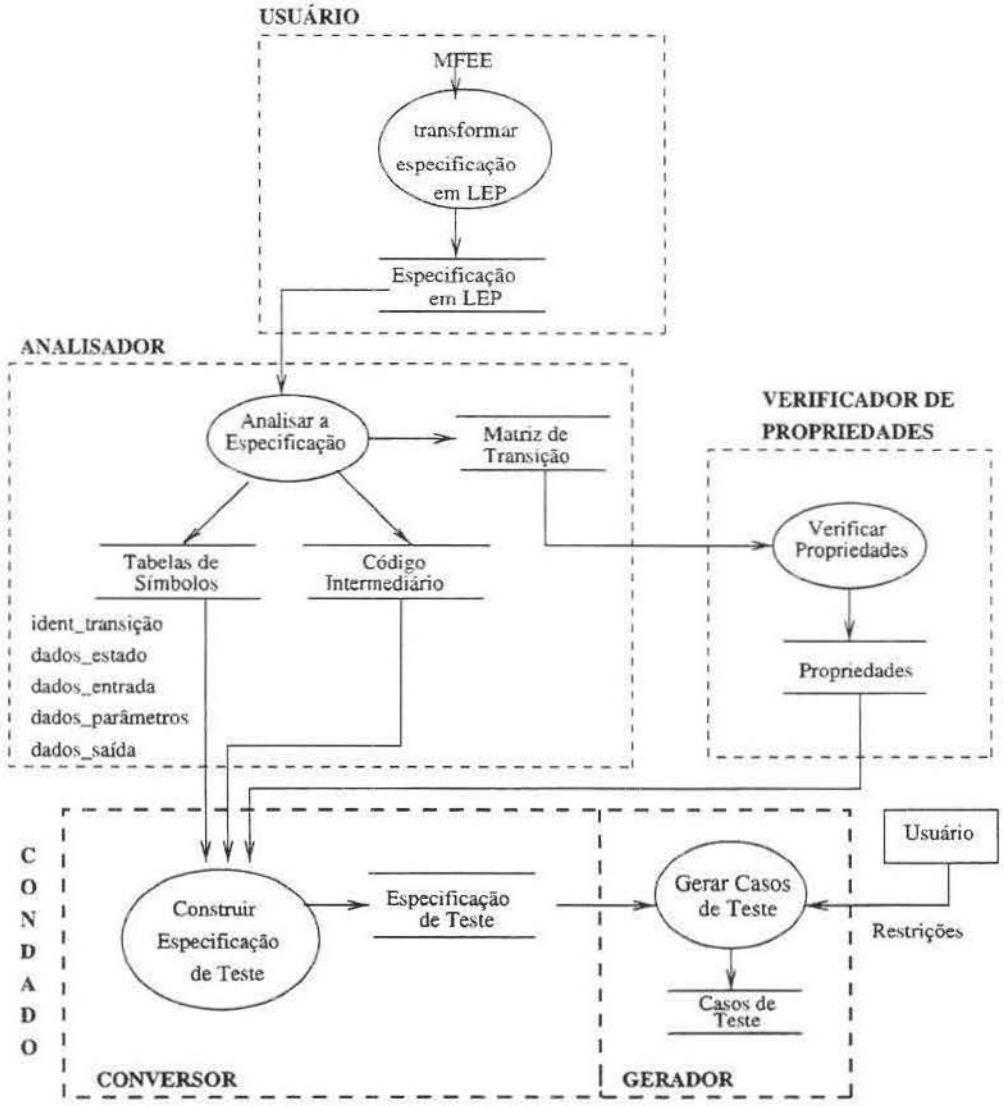

Figura 5.1: Visão Geral do Método de Geração de Teste

Uma visão geral dos procedimentos a serem realizados durante a geração de teste é dada na figura 5.1. Esta figura apresenta cinco componentes principais do método de geração de testes: **usuário**, **analisador**, **verificador de propriedades**, **conversor** e **gerador**. Os componentes conversor e gerador fazem parte da CONDADO.

A transformação da especificação em uma Linguagem para Especificação de Protocolos, denominada LEP, não é automatizada, sendo realizada manualmente pelo usuário. A partir dessa especificação, o **analisador** monta as tabelas de símbolos e o código intermediário da especificação, além de gerar uma matriz de transição contendo a parte de controle do protocolo. Com essa matriz de transição, o **verificador de propriedades** analisa a especificação a fim de detectar quais propriedades que esta possui.

As tabelas de símbolos criadas pelo **analisador** são equivalentes àquelas criadas por um compilador. Para cada palavra (seqüência de caracteres separados por espaço em branco) que o analisador lê da especificação em LEP, é gerado um *token* e uma entrada na tabela de símbolos, caso essa palavra ainda não tenha ocorrido na especificação. O *token* irá representar essa palavra no código intermediário. O formato do código intermediário pode ser encontrado no apêndice C.

O **analisador** e o **verificador de propriedades** oferecem serviços a CONDADO para que ela possa obter as informações contidas na tabela de símbolos, código intermediário e as propriedades da especificação. Uma descrição desses componentes, bem como dos seus serviços, pode ser encontrada no apêndice C. Maiores detalhes sobre o verificador de propriedades pode ser encontrado em [Bue].

A CONDADO é responsável em converter a especificação do protocolo numa especificação de teste, e gerar casos de teste usando técnicas de teste caixa preta. A partir dos serviços oferecidos pelo analisador e o verificador de propriedades, a CONDADO realiza a geração de casos de teste, com base em seus dois componentes principais:

- **Conversor**: a partir da especificação em LEP, obtida através dos serviços do analisador, e tendo satisfeito os requisitos da especificação, este módulo transforma dados e transições em uma especificação de teste. O formato dessa especificação é baseado em *Cláusulas de Horn*, que são interpretadas por Prolog [CM87];

- **Gerador**: através de consultas à especificação de teste, construída pelo conversor, e de restrições, fornecidas pelo usuário, um conjunto de técnicas de teste caixa preta são aplicadas. Seqüências de testes são geradas englobando controle e dados. Essas técnicas são implementadas em Prolog.

Dessa forma, A CONDADO permite a geração de casos de teste completos ou seletivos. Nos casos de teste seletivos, o usuário especifica propósitos de teste específicos, como por exemplo: quais transições, ou o número de vezes de execução de uma ou mais transições. As restrições são usadas pelo usuário para permitir a geração dos casos de teste seletivos. Através do uso de restrições é possível gerar testes cobrindo funcionalidades distintas do protocolo. Outro fator importante é a redução dos casos de teste gerados, facilitando a execução dos mesmos. Caso nenhuma restrição seja especificada pelo usuário, todos os casos de teste serão gerados.

Para um melhor entendimento dos componentes da CONDADO que serão descritos nesse capítulo, na seção 5.3 será apresentado um exemplo de protocolo especificado em MFEE. Este exemplo será usado para a apresentação da linguagem LEP e da sua transformação em uma especificação de teste

Para obter detalhes sobre a especificação e implementação da CONDADO, o apêndice C deve ser consultado.

A CONDADO faz parte de um ambiente de teste denominado ATIFS. Para situar esta ferramenta dentro do processo de teste, a próxima seção apresenta as principais características do ambiente ATIFS.

5.2 Principais Características do ATIFS

ATIFS é um Ambiente[1] integrado de Testes baseado em Injeção de Falhas por Software [Mar95b]. Este ambiente engloba as principais atividades do processo de teste: geração, execução e análise dos resultados. A CONDADO faz parte da primeira atividade desse processo, a geração dos casos de teste.

Este ambiente provê uma série de facilidades que permitem a comunicação entre seus vários subsistemas, além de uma interface uniforme para o usuário. O ATIFS é composto dos seguintes subsistemas:

- **Desenvolvimento dos Testes**: este subsistema compreende as funções de geração de testes e definição das falhas. A ferramenta CONDADO é responsável pela função de geração dos testes. A definição das falhas tem por objetivo auxiliar o usuário na definição das classes de falhas a serem consideradas para injeção, seus atributos e o método para injetá-las.

- **Geração do *Script***: este subsistema permite que o usuário descreva como a seqüência de teste deve ser realizada. As principais informações contidas no *script* são: os casos de teste a serem aplicados e os dados que eles utilizam, as classes de falhas a serem injetadas, o momento da injeção, a duração do experimento, entre outras informações.

- **Controle de Execução**: os objetivos desse subsistema são: controlar a execução dos testes e monitorar o sistema alvo. O controle da execução é responsável pela ativação e interação com o sistema em teste, de acordo com o *script* gerado. A função de monitorar o sistema alvo é responsável por coletar os dados observados durante os testes.

[1] Na verdade, trata-se de um grupo de ferramentas voltadas para os testes

- **Tratamento dos Resultados**: consiste no tratamento dos dados coletados durante os testes e compreende as seguintes funções: análise do comportamento observado e obtenção de estatísticas.

Além desses subsistemas, o ATIFS provê mais duas camadas: uma para gerenciamento de objetos de configuração e outra para o gerenciamento e armazenamento dos dados. Dessa forma, esse ambiente oferece um suporte à realização de todas as atividades do processo de teste de forma integrada.

5.3 Exemplo de uma Especificação na Forma de MFEE

Conforme apresentado no capítulo 3, uma MFEE pode ser definida como uma Máquina Finita de Estados (MFE) com variáveis de contexto, predicados e ações. A figura 5.2 mostra um exemplo, retirado de [HLJ95], da especificação de um protocolo através de MFEE.

Cada membro da entrada é representado da seguinte forma: `?U.SENDrequest`, onde `?` representa uma entrada, e `U` indica que esta entrada `SENDrequest` chega à IUT pela camada superior. Cada membro da saída é representado da seguinte forma: `!L.CR`, onde `!` indica que se trata de uma saída e `L` é usado para indicar que a IUT está enviando a saída `CR` para a camada inferior.

Uma ação pode modificar o valor de uma variável ou um campo de uma primitiva. Por exemplo, na Figura 5.2, na transição `T3` é realizada uma ação onde as variáveis `number` e `counter` são inicializadas. Em `T4` a ação produz uma saída, `DT`, com parâmetro `SDU[number]`, além de inicializar o temporizador e atualizar variável `number`.

Um predicado é utilizado como uma guarda, agindo como controlador das transições da MFEE. Por exemplo, a transição `T7` possui o seguinte predicado: `number = number_of_segment`, onde `number` representa uma variável e `number_of_segment` um campo da entrada `DATArequest`. Assim, a transição `T7` só é disparada quando a IUT receber um `ACK` da interface inferior e se esse predicado for verdadeiro.

5.4 Linguagem para Especificação de Protocolo

Para especificar o protocolo na forma de MFEE foi desenvolvida uma Linguagem para Especificação de Protocolo, denominada LEP. Existem diversas formas para representar uma MFEE, tais como Estelle e SDL. A escolha de uma nova linguagem para especificação de protocolo se deve à necessidade de obter uma notação única para representar controle,

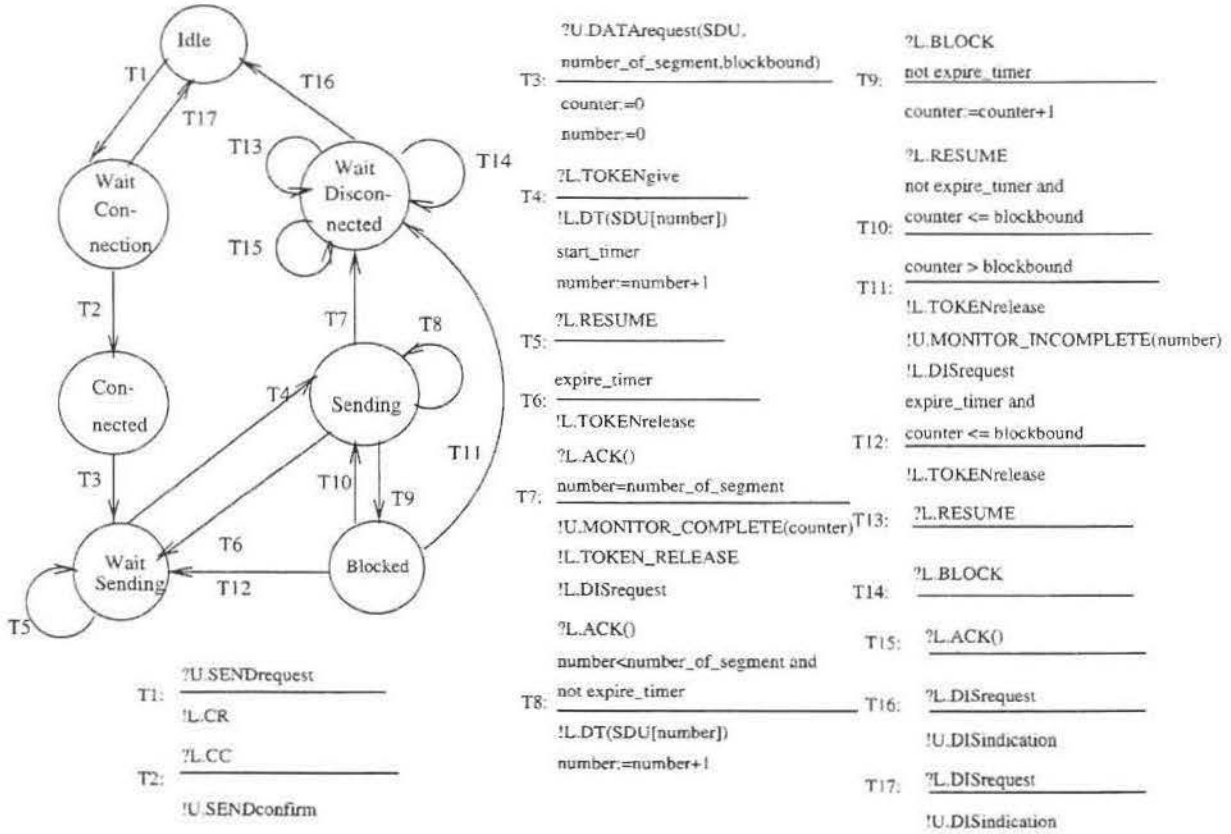

Figura 5.2: Protocolo especificado através de MFEE

dados e o formato das interações do protocolo de comunicação. Além disso, LEP oferece uma notação simples possibilitando um rápido aprendizado do usuário.

LEP é baseada na linguagem utilizada pela ferramenta TAG [TPB96], especialmente no que diz respeito à especificação da parte de controle. Para representação dos dados foi tomada como base a notação ASN.1 [NV92], que é usada para a descrição das estruturas de dados usadas nas interações dos protocolos. LEP é composta de cinco partes principais: (1) definição das variáveis, (2) definição dos estados, (3) definição das entradas e saídas, (4) definição dos dados e (5) descrição das transições do protocolo. A gramática formal para a LEP é dada no apêndice A.

5.4.1 Definição das Variáveis

Para a definição de variáveis, utiliza-se a palavra reservada `VARIABLES` seguida do nome das variáveis e de seus tipos. Há seis tipos definidos na LEP: `integer`, `boolean`, `real`, `bitstring`, `octetstring` e `enumerated`, conforme apêndice A. Considere a variável `number` do protocolo apresentado na Figura 5.2, essa variável é do tipo `integer` e ela

é declarada da seguinte forma em LEP:

```
VARIABLES:
     integer number;
```

5.4.2 Definição dos Estados

Para a declaração dos estados, utiliza-se a palavra reservada `STATES` seguida do nome de cada estado. O estado inicial é identificado pelo símbolo `#`. Por exemplo, para representar dois estados da máquina, sendo que `idle` é o estado inicial, usa-se a seguinte notação:

```
STATES:
    #idle;
     waitconec;
```

5.4.3 Definição das Entradas e Saídas

As entradas e saídas devem ser especificadas separadamente. As entradas são precedidas pela palavra reservada `INPUTS`, enquanto que as saídas, pela palavra reservada `OUTPUTS`. Por exemplo:

```
INPUTS:
     SENDrequest;
     DISrequest;
     ...
OUTPUTS:
     CR;
     DISrequest;
     ...
```

Em alguns casos, uma interação pode ser tanto de entrada como de saída. Nesse caso ela deve ser discriminada na entrada e na saída. Por exemplo, a interação `DISrequest` da figura 5.2 aparece como saída nas transições `T7` e `T11`, e como entrada nas transições `T16` e `T17`.

5.4.4 Definição dos Dados

Para a definição do tipo e formato das informações trocadas pelos protocolos, LEP usa um subconjunto dos tipos de ASN.1: `integer`, `real`, `boolean`, `bitstring`, `octetstring` e `enumerated`. O tipo `integer` corresponde ao conjunto de valores inteiros positivos e

negativos. O tipo `real` corresponde ao conjunto dos valores reais positivos e negativos. O tamanho máximo permitido para os números inteiros e reais não são estipulados pela LEP, sendo impostos pela implementação do protocolo. O tipo `boolean` pode assumir valores `TRUE` e `FALSE`. O tipo `bitstring` é usado para representar um conjunto de *bits* de um tamanho especificado. O tipo `octetstring`, é usado para representar qualquer seqüência de caracteres ASCII delimitados por aspas (") de um tamanho especificado.

Além desses tipos de dados, é possível representar estruturas de dados através dos seguintes tipos estruturados: `SEQUENCE`, `SET`, `SEQUENCE OF`, `SET OF` e `CHOICE`.

Os tipos `SEQUENCE` e `SET` são similares, ambos são usados para agrupar uma coleção de tipos, incluindo tipos simples e estruturados. A diferença entre eles está na ordenação desses tipos. No tipo `SEQUENCE`, a ordem em que os tipos são colocados é importante, enquanto que no tipo `SET` a ordem não importa. Os tipos `SEQUENCE OF` e `SET OF` são similares ao tipo *array* de algumas linguagens de programação. Eles implicam em uma seqüência de valores com um tamanho especificado, que pode ser ordenada (`SEQUENCE OF`) ou não ordenada (`SET OF`). O tipo `CHOICE` permite escolher um tipo dentre um conjunto de tipos definidos.

Para exemplificar a representação dos dados, considere os tipos de dados pertencentes à estrutura `SEQUENCE`, onde o dado `DATArequest` é formado pelos campos `SDU`, `number_of_segment` e `blockbound`. A descrição dos campos das primitivas começa com a palavra reservada `DATA` seguida da estrutura de dados de cada primitiva. Por exemplo:

```
DATA:
DATArequest :: = SEQUENCE {
                 SDU                octetstring    5,
                 number_of_segment  enumerated     2 | 4 | 8,
                 blockbound         integer        3 .. 15};
```

Na verdade, `SDU` é uma unidade de dados da camada de sessão do protocolo. Para fins de exemplificação, será considerado como um conjunto de caracteres de tamanho 5, mas poderia ser definido com `SEQUENCE OF`. Para representar valores não seqüenciais, como valores de dados aleatórios, cadeia de caracteres, entre outros, usa-se o tipo `enumerated`. Por exemplo, `number_of_segment` pode ser 2, 4 ou 8, conforme exemplificado acima. Para valores seqüenciais, como em `blockbound` que pode receber um valor do intervalo de 3 a 15, usa-se a notação `3 .. 15`.

Ao especificar o tipo estruturado `SEQUENCE OF` ou `SET OF` é necessário especificar seu tamanho máximo. Considere que o tipo `SEQUENCE OF` seja usado para representar um conjunto de cadastros de funcionários de uma empresa, contendo as seguintes informações: nome, endereço e salário. Supondo que essa empresa possa ter no máximo 10 funcionários, a especificação desse cadastro é feita da seguinte forma em LEP:

```
Tpcadastro :: = SEQUENCE OF [10] {
                nome          octetstring   20,
                endereco      octetstring   30,
                salario       real          110,33 .. 1044,87};
```

5.4.5 Descrição das Transições

Após as declarações são especificadas as transições, que descrevem a parte dinâmica do protocolo. A descrição das transições inicia com a palavra reservada TRANSITIONS, seguida da especificação de cada transição pertencente à máquina. Tomando como exemplo a transição T1, sua representação em LEP é da seguinte forma:

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

Onde o símbolo * precede a identificação da transição, nesse caso, t1. O símbolo >, antes do nome do estado, é usado para indicar que idle é o estado origem. A entrada é especificada como: ?U.SENDrequest, onde ?U indica que o dado SENDrequest chegou da camada superior. A saída é representada por !L.CR, onde !L indica que CR será enviado para a camada inferior. Finalmente, o símbolo < indica que o estado waitconec é o estado destino da transição.

O exemplo anterior representa uma transição simples que não possui predicados, ações nem dados relacionados à entrada. Considere agora exemplos das transições T3 e T12 que contêm esses elementos:

```
*t3
>connected
?U.DATArequest(SDU, number_of_segment, blockbound)
{number := 0} {counter := 0}
<waitsend;

*t12
>blocked [expire_timer] [counter <= blockbound]
!L.TOKENrelease
<waitsend;
```

A transição T3 possui duas ações, nas quais as variáveis number e counter são inicializadas. As ações são especificadas entre { }. Nessa transição, como há uma primitiva, seus campos devem ser discriminados conforme a especificação acima da primitiva DATArequest. A transição T12 possui dois predicados associados. Um deles verifica se o tempo de acesso terminou, e o segundo verifica se a variável counter é menor ou igual ao dado de entrada blockbound. Os predicados são representados entre [ ].

LEP também permite o uso de comentários, que devem ser precedidos por // no início de cada linha. O fim da especificação é dado pela palavra reservada END.

Uma descrição completa em LEP da MFEE da figura 5.2 é apresentada no apêndice B.

5.5 Especificação de Teste

Conforme o método apresentado no capítulo 4, a partir da especificação em MFEE, uma especificação de teste deve ser construída. Alguns trabalhos usam grafos de fluxo de dados e controle para esse fim, a partir dos quais aplicam-se técnicas de teste caixa branca.

Na CONDADO, a notação escolhida para a especificação de teste foram as *Cláusulas de Horn*. Essa representação é interpretada por Prolog. Dessa forma, a especificação de teste pode ser diretamente consultada pelo **gerador**, que é implementado em Prolog.

O componente responsável pela transformação de LEP em uma especificação de teste é o **conversor**, conforme mostrado na figura 5.1. Antes de demonstrar este componente e suas funcionalidades, faz-se necessário apresentar as principais características do Prolog.

5.5.1 Uso de Prolog para Especificação e Geração de Teste

A linguagem de programação PROLOG[2] teve origem na área de Inteligência Artificial com o objetivo de *Prova Automática de Teoremas* [Bit96]. Com o tempo Prolog ganhou popularidade em outras áreas, como por exemplo, na área de Engenharia de Software.

Especificamente na área de teste de *software*, vários trabalhos são encontrados na literatura usando Prolog para auxiliar principalmente na especificação e geração de teste [Den91], [PSSS85], [FSFT87], [HS91]. Prolog é usado também para especificação e geração de teste para protocolos de comunicação [Sou85], [UP83].

Como linguagem para especificação, Prolog oferece uma série de vantagens [Den91]:

- Muitas notações usadas para especificação formal, dentre elas grafos de fluxo de dados, gramáticas formais e máquinas de estado, têm uma implementação direta como programas lógicos;
- Indeterminismos podem ser especificados naturalmente usando programação lógica;

[2] PROgramming in LOGic

- Apesar de existir um padrão para a linguagem, existem interpretadores Prolog disponíveis para PCs e estações e são compatíveis;
- Especificações em Prolog são executáveis, possibilitando a geração automática de casos de teste pela execução da especificação.

Como especificações em Prolog podem ser executadas, é possível gerar testes diretamente a partir da especificação dada em *Cláusulas de Horn*, não necessitando de um *parser* para interpretar tal especificação, como nas linguagens de especificação existentes [PSSS85].

Prolog oferece também um mecanismo de *backtracking*, que facilita na geração dos casos de testes, pois permite a geração de todas as combinações existentes na especificação [PSSS85].

Além disso, Prolog é particularmente adequado para resolver problemas que envolvem objetos e relacionamentos entre estes. Tais problemas podem ser ilustrados pelas máquinas de estados, onde os estados correspondem aos objetos e as transições correspondem às relações [Bra86].

Com essas vantagens e facilidades oferecidas pelo Prolog, a construção de uma ferramenta para validar um determinado método se dá de forma mais rápida. Em [Ham95], um estudo é apresentado sobre o uso do Prolog como linguagem para prototipação de ferramentas de teste de *software*. Esse trabalho apresenta várias vantagens, enfatizando o fato de que o estilo declarativo de programação Prolog possibilita a implementação de um método de teste. Além disso, o tempo gasto para a construção de uma ferramenta em Prolog é muito menor que em outras linguagens de programação convencional como C, C++ e Pascal.

O uso de *cláusulas de Horn* para especificação de teste foi escolhida perante as várias facilidades acima apresentadas. Como o gerador é implementado em Prolog, a especificação de teste é consultada de forma direta.

Para possibilitar que leitores não acostumados com a linguagem de programação Prolog possam entender o formato da especificação de teste, a próxima seção apresenta as principais características dessa linguagem.

5.5.2 Linguagem de Programação Prolog

Prolog é uma linguagem interpretada composta de *Cláusulas de Horn* [CM87]. As cláusulas podem ser **fatos** ou **regras**. Um fato consiste de uma afirmação sobre objetos. Por exemplo, podemos dizer que *Antonio é do sexo masculino.* Trata-se, portanto, de uma informação sobre o objeto Antonio. Em Prolog os fatos são representados da seguinte forma:

```
sexo(antonio, masculino).
progenitor(antonio, maria).
```

Um conjunto de fatos forma uma base de dados (ou base de conhecimento). No fato acima, Antonio está escrito com letra minúscula porque, em Prolog, todo símbolo iniciado por letra maiúscula é uma constante ou variável.

As regras são usadas para dizer que um fato depende de um grupo de outros fatos. Uma regra é formada por uma **cabeça** e um **corpo**. A cabeça da regra aparece antes do símbolo :- que é seguido pelo corpo. O corpo de uma cláusula pode ser composto de uma ou mais condições, onde a vírgula tem o significado lógico *e*, e o ponto-e-vírgula o significado *ou*. Os fatos que compõem o corpo da regra são chamados de **objetivos** e, para que a regra seja verdadeira, o conjunto de objetivos precisa ser satisfeito. Por exemplo:

```
pai (X,Y) :- progenitor(X, Y), sexo(X, masculino).
```

Neste caso X e Y são **variáveis**. Essa regra é interpretada da seguinte forma: *X é pai de Y se X é progenitor de Y e X é do sexo masculino.* A partir desses fatos e regras é possível fazer consultas para obter informações sobre a base de conhecimento. Por exemplo, com as informações acima podemos perguntar se Antonio é pai de Maria, o que Prolog responderá após uma procura em suas cláusulas:

```
?- pai(antonio,maria).
   Yes.
```

O escopo de uma variável é a cláusula onde ela está sendo usada. Portanto, usar o mesmo nome de uma variável em duas cláusulas diferentes significa duas variáveis diferentes.

Um mecanismo importante do Prolog é o *backtracking*, que consiste em satisfazer objetivos novamente quando uma regra ou um fato não é satisfeito; ou mesmo para tentar encontrar soluções alternativas a uma pergunta feita à base de fatos do Prolog. Para exemplificar o *backtracking* considere o seguinte conjunto de fatos:

```
pai(antonio, maria).
pai(eduardo, ana).
pai(ricardo, sandra).
```

Com esta base de fatos, pode-se perguntar de quem determinada pessoa é pai e as seguintes respostas podem ser encontradas para essa pergunta:

```
?- pai(X,Y).
X=antonio, Y=maria;
X=eduardo, Y=ana;
X=ricardo, Y=sandra
```

Existem várias respostas a essa consulta. Ao encontrar o primeiro fato em que as variáveis X e Y "casam", ou seja, que satisfaça a consulta, Prolog devolve a solução encontrada. Para que outras soluções possam ser encontradas é necessário pedir ao Prolog para satisfazer novamente os objetivos. Dessa forma, outra solução será procurada para essa questão. O ponto-e-vírgula após cada resposta representa o pedido de *backtracking*, ou seja, pedir ao Prolog para verificar se existem outras soluções para esta questão em sua base de fatos.

Com essas definições, é possível apresentar os componentes da CONDADO. Um estudo introdutório para auxiliar no aprendizado da linguagem Prolog é encontrado em [CM87]. Um estudo mais detalhado é encontrado em [Bra86], e técnicas para melhorar e otimizar a programação em Prolog são apresentadas em [Kee90].

5.5.3 Construção da Especificação de Teste

A construção da especificação de teste é de responsabilidade do módulo **conversor** (ver figura 5.1). A partir da especificação do protocolo, o **conversor** transforma os dados e transições da MFEE, representada pela LEP, em fatos e regras.

O primeiro passo do **conversor** é a geração de um fato que identifica o estado inicial do protocolo. Usando o exemplo da figura 5.2, o estado inicial da MFEE, que é `idle`, será representado da seguinte forma em Prolog:

```
inicial(idle).
```

Em seguida, são geradas as regras para as transições do protocolo. Considere a transição `t1`, que é representada da seguinte forma em LEP:

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

Essa transição é representada da seguinte forma em Prolog:

```
trans(idle, t1, waitconec,L0,Ln) :-
      receiveu('SENDrequest',L0,L1),
      transmitl('CR',L1,Ln).
```

A transição é representada por uma regra. A cabeça da regra especifica a parte de controle do protocolo, enquanto que o corpo representa as interações de entrada e saída da transição. Esta regra é interpretada da seguinte forma: *a transição* `t1` *com estado origem* `idle` *e estado destino* `waitconec` *será disparada se a entrada* `SENDrequest` *chegar à IUT pela camada superior* (`receiveu`) *e a IUT enviar a saída* `CR` *à interface inferior* (`transmitl`). As variáveis `L0,L1,Ln` associadas à regra, são usadas para armazenar os casos de teste, ou seja, uma seqüência finita de interações de entrada com suas saídas correspondentes.

A transição `t1` descrita acima não tem dados associados à entrada. No exemplo da figura 5.2, a transição `t3` tem como entrada a primitiva `DATArequest` que possui três parâmetros. Esta transição é representada da seguinte forma em LEP:

```
*t3
>connected
  ?U.DATArequest(SDU, number_of_segment, blockbound)
  {number := 0} {counter := 0}
<waitsend;
```

A CONDADO trata somente as partes de controle e dados[3], não havendo, até então, um tratamento automatizado para os predicados. Dessa forma, os predicados e ações não serão transformados para regras ou fatos. Assim, a transição `t3` será representada da seguinte forma em Prolog:

```
trans(connected,t3,waitsending,L0,Ln):-
      receiveu('DATArequest',L0,L1),
      sequence([[i,octetstring,5],[c,number_of_segment,3],
                [i,integer,3,15]],L1,L2),
      transmit(L2,Ln).
```

Nesse caso, como a primitiva `DATAsequest` possui parâmetros associados é necessário representá-los em Prolog, a fim de que os testes possam ser gerados englobando esses dados. O objetivo `sequence` representa o tipo estruturado desses parâmetros, e seus componentes representam as informações dos dados propriamente dito.

Cada parâmetro é separado por colchetes. Os parâmetros dessa primitiva estão representados em LEP na seção 5.4.4. O primeiro deles, `SDU`, é representado em Prolog como `[i,octetstring,5]`, onde `i` especifica que esse parâmetro é um intervalo de valores do tipo `octetsting` com tamanho 5. O último parâmetro representa `blockbound`,

[3]Testes de dados realizados pela CONDADO compreendem a aplicação de técnicas de testes caixa preta, como teste de sintaxe e de domínio.

um dado inteiro que pode assumir qualquer valor no intervalo de `3..15`. O campo `number_of_segment` é uma coleção de valores, representado por `c`, com três elementos. Para representar os possíveis valores desse campo e, possibilitar a geração de teste, são criados fatos em Prolog. Para o parâmetro `number_of_segment` que pode ter o valor `2|4|8`, sua representação será a seguinte:

```
dados(number_of_segment,0,'2').
dados(number_of_segment,1,'4').
dados(number_of_segment,2,'8').
```

O primeiro campo do fato `dados`, `number_of_segment` especifica o nome do parâmetro. O segundo é um contador associado a esse parâmetro e, por último, é especificado um dos valores que esse parâmetro pode assumir.

A transição `t3`, representada acima, não possui saída. Nesse caso, um fato `transmit` com dois parâmetros é criado para especificar que esta transição não possui saída.

Os tipos estruturados `SET` e `CHOICE` são representados da mesma forma que o tipo `SEQUENCE`, exemplificado acima. A representação dos tipos `SET OF` e `SEQUENCE OF` é feita da mesma forma que dos tipos anteriores, a única diferença é a inclusão de um campo para especificar seu tamanho. Supondo, por exemplo, que o tipo estruturado da primitiva `DATArequest` seja do tipo `SEQUENCE OF` de tamanho 2. Sua representação em Prolog é dada da seguinte forma:

```
trans(connected,t3,waitsending,L0,Ln):-
      receiveu('DATArequest',L0,L1),
      sequenceof([[i,octetstring,5],[c,number_of_segment,3],
                  [i,integer,3,15]],2,L1,L2),
      transmit(L2,Ln).
```

É necessário especificar também quando o estado destino de uma transição é o estado final da especificação. No exemplo da figura 5.2, o estado inicial é igual ao seu estado final, o que ocorre na maioria dos protocolos de comunicação. Para representar a transição `t16` com estado origem igual a `waitdisconnected` e estado destino `idle` (estado final da MFEE), a seguinte regra é criada:

```
trans(waitdisconnected,t16,fim,L0,Ln):-
      receivel('DISrequest',L0,L1),
      transmitu('DISindication',L1,Ln).
```

Nesse caso, a diferença está na cabeça da regra, que em vez de especificar `idle` como estado destino, é trocado pela palavra `fim`.

Tendo toda a especificação do protocolo representada com *Cláusulas de Horn*, o próximo passo é a geração de casos de teste. A próxima seção apresenta como esta representação é usada na geração dos testes.

5.6 Geração de Casos de Teste

A geração de casos de teste é realizada pelo módulo denominado de **gerador**, conforme representado na figura 5.1. Este módulo representa o terceiro e último passo do método apresentado no capítulo 4. O **gerador** implementa um conjunto de técnicas de teste caixa preta em Prolog, para a geração dos casos de teste. A partir de consultas às regras e fatos construídas pelo **conversor**, o gerador é executado gerando um conjunto de casos de teste cobrindo os aspectos de controle e dados do protocolo de comunicação.

As técnicas implementadas até agora pelo gerador são: testes de transição de estados, teste de sintaxe e teste de domínio.

De forma geral, esse conjunto de técnicas é combinado para gerar as seqüências de teste cobrindo controle e dados. Percursos são realizados na especificação e, a cada transição encontrada com dados associados, técnicas de testes de sintaxe e de domínio são aplicadas.

As próximas seções descrevem as dificuldades encontradas para implementação dessas técnicas em Prolog e as heurísticas que foram usadas para contornar essas dificuldades.

5.6.1 Dificuldades Encontradas

O fluxo de controle do protocolo é coberto por uma técnica de teste de transição de estado baseada no método T (*transition tour*) [SL89], apresentado na seção 3.2.

A partir do estado inicial, percursos (ou caminhos) são realizados na MFEE a fim de cobrir todas as suas transições, sempre voltando ao estado inicial. Esses caminhos (seqüências de transições) são gerados com um algoritmo de busca em profundidade.

Uma facilidade encontrada na implementação desse algoritmo em Prolog foi o mecanismo de *backtracking* existente na linguagem. A vantagem no uso desse mecanismo é que, ao gerar um caminho, Prolog tenta satisfazer novamente as regras a fim de encontrar novos caminhos na MFEE.

Com esse mecanismo, o algoritmo de busca em profundidade é facilmente implementado. Entretanto, uma limitação foi encontrada no uso desse mecanismo: os ciclos existentes na máquina. Um ciclo pode ser definido como uma ou mais transições $t_1, t_2, .., t_k$, tal que o estado destino da transição t_k é igual ao estado origem de t_1 [BDAR97]. Ao tentar encontrar novos caminhos em uma máquina com ciclos, usando mecanismo de *backtracking*, Prolog repete o mesmo caminho. Como um ciclo é uma recursão, o mecanismo de Prolog entende que este é um novo caminho.

Este problema, muitas vezes chamado de problema da recursão [Den91], é normalmente tratado com o uso de heurísticas. Vários autores apresentam diferentes heurísticas para resolver esse caso [Den91], [Kee90] e [Bra86]. A seguir será apresentada a solução adotada nesse trabalho para tratar o problema dos ciclos.

5.6.2 Solução Adotada para Detectar os Ciclos

Para possibilitar a geração de casos de teste, fez-se necessário detectar e controlar os ciclos existentes na especificação. A primeira idéia na geração de teste foi detectar todos os ciclos antes de executar o gerador e, dessa forma, fornecer guardas para que transições pertencentes aos ciclos fossem geradas de maneira controlada. Mas, como o número de ciclos cresce exponencialmente à medida em que a especificação aumenta, um algoritmo para detectar todos os ciclos terá complexidade exponencial. Isso ocasionaria uma demora muito grande na obtenção dos casos de teste, visto que protocolos de comunicação possuem especificações complexas e o número de ciclos pode ser muito grande.

Para solucionar esse problema foi usada uma classificação de arestas feita na aplicação do método de busca em profundidade. O algoritmo de busca em profundidade é apresentado na próxima seção. Em seguida é mostrada a solução encontrada para detecção e controle dos ciclos.

Algoritmo de Busca em Profundidade

O algoritmo de busca em profundidade descrito nessa seção foi retirado de [Szw84] e é usado pelo gerador para produzir os casos de teste para a parte de controle do protocolo.

Para entender o algoritmo, uma definição de grafos faz-se necessária. Um dígrafo (ou grafo direcionado) $D(V, A)$ é um conjunto finito não vazio V de vértices e um conjunto A de arestas de pares ordenados de vértices distintos. Portanto, num dígrafo, cada aresta (v, w) possui uma única direção de v para w [Szw84].

A busca em profundidade é realizada em um grafo conexo. Dessa forma, todo vértice é alcançado na busca. A busca começa pelo vértice inicial denominado de *raiz*. A partir desse vértice, caminhos são construídos. O algoritmo de busca em profundidade é apresentado a seguir [Szw84]:

```
dados digrafo D(V,A)
procedimento P(v)
  marcar v;
  colocar v na pilha Q;
  para w pertencente a A(v) efetuar
    visitar (v,w);
```

```
    se w nao marcado entao P(w)
  retirar v de Q
desmarcar todos os vertices
definir uma pilha Q
definir uma raiz s pertencente a V
P(s)
```

Este algoritmo apresenta o caso geral, onde é possível gerar uma busca em profundidade partindo de qualquer vértice. No caso da especificação de protocolos de comunicação, a raiz será sempre o estado inicial. A partir desse estado o algoritmo acima é aplicado, retornando todos os caminhos que começam e terminam no estado inicial.

Através da busca em profundidade é possível fazer uma classificação das arestas do grafo. Considere a visita a uma aresta (v, w). Seja v alcançado antes de w na busca. Se w se encontrava desmarcado no momento da visita, então (v, w) é chamada **aresta da árvore**, caso contrário (w marcado) é denominado **aresta de avanço**. Seja agora, w alcançado antes de v na busca. Se $w \in Q$ no momento da visita, então (v, w) é denominado de **aresta de retorno**, caso contrário ($w \notin Q$), a aresta (v, w) é denominada **aresta de cruzamento**. Desse modo, a busca em profundidade em um dígrafo divide o seu conjunto de arestas em quatro subconjuntos distintos [Szw84].

Para exemplificar a classificação de arestas descritas acima, a figura 5.3 (a) apresenta um dígrafo $D(V, A)$, e a figura 5.3 (b) apresenta uma busca em profundidade nesse dígrafo com a classificação das arestas considerando como raiz o estado 1.

Tratamento dos Ciclos

O problema da detecção de ciclos deve ser resolvido, não só porque o **gerador** pode não terminar sua execução ao encontrar um ciclo na máquina, mas também porque os ciclos em protocolos de comunicação exercem um papel importante na geração de caminhos executáveis. Um caminho é dito executável, quando todas as transições da MFEE pertencentes a esse caminho são executáveis 3.3.

Há caminhos na máquina que podem não ser executáveis devido a existência de predicados. Como a CONDADO não está tratando os predicados, algumas vezes a geração de teste pode levar a caminhos não executáveis.

Em muitos casos, os ciclos influenciam diretamente a geração de caminhos executáveis [BDAR97]. Um exemplo pode ser encontrado na figura 5.2. A seqüência de transições `t1,t2,t3,t4,t9,t11`, começando pelo estado inicial, só é executável quando `counter > blockb`
Supondo que `blockbound` tenha valor 2 na transição `t3`, a seqüência de transições acima não seria executável. Para torná-la executável é necessário que o ciclo `t9, t10`, que incrementa `counter`, seja executado duas vezes antes da transição `t11`. Para a execução de `t11`

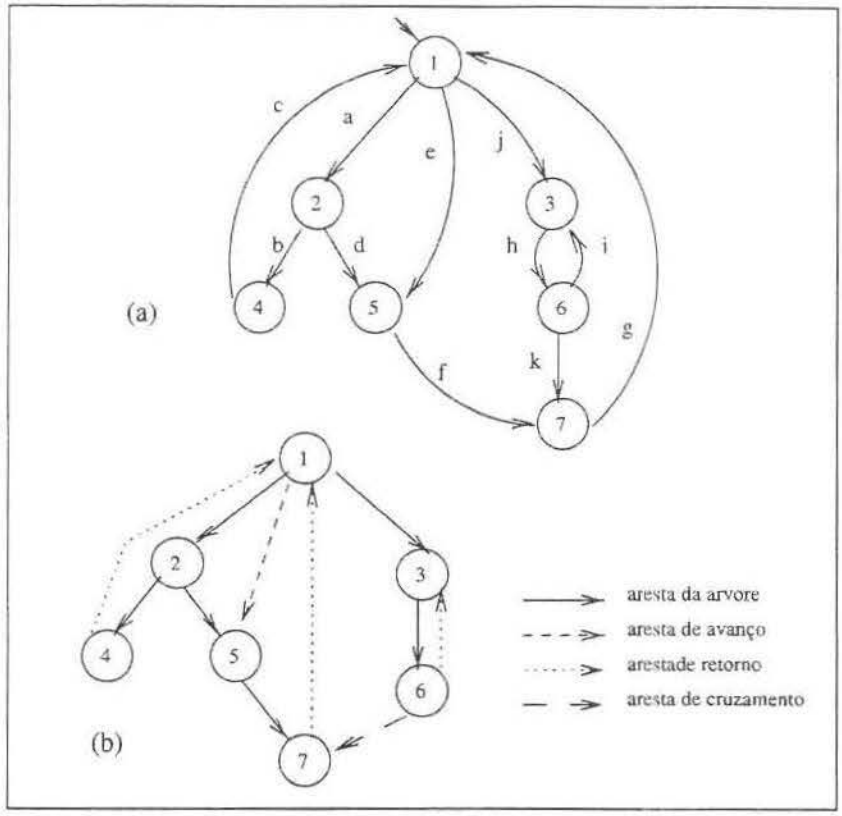

Figura 5.3: Exemplo de um grafo (a) e a busca em profundidade com a classificação das arestas (b)

com `blockbound` igual 2, a seqüência de transições deve ser: `t1,t2,t3,t4,t9,t10,t9, t10, t9,t11,t16`.

A solução para o problema da executabilidade não é resolvido somente com o tratamento dos ciclos presentes na máquina, outros fatores também influenciam na geração de caminhos não executáveis. Na CONDADO, ainda não foram implementadas técnicas específicas de geração de teste que tratam o problema da executabilidade. Assim, a geração controlada de um certo número de vezes de cada ciclo pode amenizar o problema da geração de casos de teste não executáveis.

Para solucionar o problema dos ciclos e possibilitar a geração de casos de teste de maneira controlada, foi usada como base a classificação de arestas dada no algoritmo de busca em profundidade descrito na seção anterior.

Observou-se que, no algoritmo de busca, uma **aresta de retorno** do grafo caracteriza o fechamento de um ciclo. Dessa forma, toda vez que uma aresta de retorno é encontrada, ela é colocada em uma lista de arestas de retorno, e só poderá ser gerada novamente, no mesmo caso de teste, se o usuário da CONDADO solicitar, através do uso das restrições,

que serão explicadas na seção 5.6.6.

No exemplo da figura 5.2, a aplicação do algoritmo de busca em profundidade detectou as seguintes arestas de retorno: `t5`, `t6`, `t8`, `t10`, `t12`, `t13`, `t14`, `t15`, `t16` e `t17`. As arestas de retorno que levam ao estado inicial são consideradas como transições finais da busca. Neste caso, quando esta aresta for encontrada, o algoritmo implementado pelo gerador considera que mais um caso de teste foi gerado, passando para a geração do próximo caso de teste, se existir.

Através dessa classificação de arestas, técnicas de teste foram implementadas no gerador. A seguir serão apresentadas essas técnicas e quais tipos de falhas que cada uma delas cobre.

5.6.3 Técnica de Teste de Transição de Estados Implementada pelo Gerador

Testes de transição de estados têm por objetivo encontrar falhas de transições e de transferências, dentre outras, conforme apresentado na seção 2.3.1.

A técnica de teste de transição de estados implementada na CONDADO é baseada no método T (*transition tour*) [SL89]. Como no método T, a técnica de teste de transição de estados implementada nesse trabalho detecta falhas de transições, como por exemplo, uma ação errada foi executada gerando uma saída incorreta. Falhas de transferência não são detectadas, pois não são implementados mecanismos para identificação de estados.

Na técnica de teste implementada pela CONDADO, testes são gerados para todas as combinações existentes das transições especificadas. Dessa forma, o número de casos de teste gerados será maior que os gerados pelo método T. De certa forma, a probabilidade de detectar maior número de falhas aumenta, mas muitos casos de teste equivalentes podem ser gerados, além de aumentar o número de casos de testes não executáveis.

Para exemplificar os casos de teste que podem ser gerados por essa técnica, considere o grafo da figura 5.3 (a). Para este grafo os seguintes casos de testes são gerados usando a técnica de teste de transição de estados implementada no **gerador**:

```
abc
adfg
efg
jhihkg
jhkg
```

Uma facilidade que a CONDADO apresenta é a geração controlada de um ciclo repetidamente. Com isso, o número de casos de teste gerados tende a permanecer igual ao número de casos de teste gerados sem exercitar os ciclos, o que irá aumentar é o caso de

teste que possui as transições pertencentes ao ciclo em questão. Para exemplificar, considere o grafo da figura 5.3, supondo que se deseja gerar todos os casos de teste passando 3 vezes pelo ciclo existente entre os estados 3 e 6 do grafo. Dessa forma, os seguintes casos de teste serão gerados:

```
abc
adfg
efg
jhihihihkg
jhkg
```

Como este método gera os casos de teste cobrindo a combinação de todas as transições, o número de casos de teste pode ser muito alto, dependendo da especificação. Isso ocorre em especificações onde existem muitos ciclos interligados ou "aninhados". Ciclos "aninhados" são ciclos que contêm outros ciclos. Nesse caso, há uma combinação muito grande entre essas transições; conseqüentemente, muitos casos de teste são gerados.

Para possibilitar a geração de um número mais significativo de casos de teste e, ao mesmo tempo, que cubram todas as possibilidades (ou combinações) das transições, a CONDADO propõe o uso de restrições. Dessa forma, testes podem ser gerados para partes específicas do protocolo. Os tipos de restrições disponíveis pela CONDADO são apresentados na seção 5.6.6.

5.6.4 Técnica de Teste de Sintaxe Implementada pelo Gerador

Como LEP possui uma sintaxe bem definida para representação dos dados trocados entre os protocolos de comunicação, testes de sintaxe podem ser aplicados. Para que seja possível gerar testes para os parâmetros das interações e para reconhecer o formato dessas interações, o gerador implementa teste de sintaxe juntamente com teste de domínio.

A técnica de teste de sintaxe é responsável por gerar testes válidos e inválidos, a partir de uma sintaxe definida [Bei90]. No caso da CONDADO, serão gerados somente os casos válidos. Os inválidos ficam a cargo do mecanismo de injeção de falhas, conforme explicado na seção 4.5.

A geração de teste de sintaxe é feita no decorrer da geração dos percursos, dos testes de transição de estados. Para cada interação que contém dados associados, o gerador aplica testes de sintaxe a fim de encontrar uma seqüência válida de acordo com a especificação.

Os tipos estruturados definidos em LEP são: `SEQUENCE`, `SET`, `CHOICE`, `SEQUENCE OF` e `SET OF`. Testes de sintaxe são gerados para cada um desses tipos.

No tipo `SEQUENCE`, os parâmetros das interações devem ser gerados na ordem em que foram especificados. Assim, para cada parâmetro que compõe esse tipo são usadas técnicas de teste de domínio.

Para o tipo SET, a ordem da geração dos parâmetros de interação não é importante. Nesse caso, o gerador escolhe, aleatoriamente, qual a ordem em que esses parâmetros serão gerados. Com a ordem definida, testes de domínio são aplicados para obter os valores para cada um dos parâmetros especificados.

No tipo CHOICE, somente um, dentre um conjunto de parâmetros, deve ser escolhido. O gerador escolhe um dos parâmetros aleatoriamente e, para o parâmetro escolhido, técnica de teste de domínio é chamada.

Para os tipos estruturados SET OF e SEQUENCE OF, procede-se da mesma forma que os tipos SET e SEQUENCE, respectivamente. A única diferença é que o procedimento é repetido para o número de vezes especificado para este tipo de dado.

5.6.5 Técnica de Teste de Domínio Implementada pelo Gerador

No caso da CONDADO, testes de domínio são implementados para complementar os testes de sintaxe. Para cada um dos parâmetros das primitivas, testes de domínio são aplicados a fim de devolver um valor para o parâmetro em questão. O valor a ser devolvido é selecionado aleatoriamente dentro do domínio especificado para esse parâmetro.

Para aplicar técnicas de teste de domínio é necessário conhecer os possíveis valores para os parâmetros de entrada. Dessa forma, é possível definir um domínio, e dentro desse domínio, um dado de teste é selecionado.

Há várias estratégias usadas para teste de domínio [Bei90]. Até o momento, a CONDADO aplica a estratégia de gerar um valor aleatório dentro do domínio especificado. Dessa forma, os limites dos parâmetros são tratados sempre como dados.

Considere o exemplo da primitiva DATArequest da figura 5.2. A regra que especifica os dados é dada na seguinte forma:

```
...
 sequence([[i,octetstring,5],[c,number_of_segment,3],
                             [i,integer,3,15]],L1,L2),
...
```

Para selecionar dados de teste para cada um dos parâmetros, é feita uma escolha aleatória dentro do domínio especificado.

O primeiro parâmetro é do tipo octetstring de tamanho 5. A geração de valores para esse campo é feita considerando-se os valores da tabela ASCII. Os números de 32 a 126 representam os caracteres da tabela ASCII, incluindo caracteres especiais. Valores aleatórios são gerados dentro desse intervalo. A única exceção é o número 34, que representa o caracter aspas ("), e será usado como delimitador das cadeias de caracteres geradas.

O último parâmetro é do tipo `integer` com um limite definido. Nesse caso, `3..15` representa uma faixa de valores de 3 a 15. O gerador seleciona um valor aleatório para esse campo dentro do limite especificado.

O parâmetro `number_of_segment` é considerado como uma coleção de valores, especificado pela letra 'c'. Nesse caso, os valores dessa coleção estão descritos como fatos que possuem contadores associados. O gerador escolhe um número aleatório entre 0 e $n-1$, onde n é o número de elementos existentes na coleção. Neste exemplo, $n = 2$. Supondo que o número escolhido seja 1, então no grupo de fatos abaixo o parâmetro 4 da coleção será escolhido.

```
dados(number_of_segment,0,'2').
dados(number_of_segment,1,'4').
dados(number_of_segment,2,'8').
```

Outras estratégias podem ser implementadas para teste de domínio, incrementando a escolha aleatória implementada até agora.

5.6.6 Uso de Restrições

O uso de restrições é uma facilidade muito importante na geração dos casos de teste. O número de caminhos ou percursos em uma máquina de estados pode ser muito grande, e até infinito; especialmente se for levado em conta os ciclos e variações de valores dos parâmetros de interações [VJL+94].

A CONDADO permite a geração de casos de teste levando em consideração uma ou mais restrições. Dessa forma, o usuário da ferramenta pode solicitar a geração de casos de teste para cobrir uma determinada funcionalidade do protocolo. Além disso, pode-se solicitar a geração de um ciclo mais de uma vez, para casos onde os ciclos podem influenciar na executabilidade dos casos de teste.

O uso de restrições implementadas pela CONDADO é baseado no trabalho proposto em [VJL+94]. As seguintes restrições foram implementadas:

- **Restringir as transições**: especificar um subconjunto de transições a serem exercitadas pelos casos de teste. Com essa restrição, o gerador irá fornecer somente os percursos que passam por tais transições, desconsiderando as demais. Esse tipo de restrição é bastante usado para testar funcionalidades distintas do protocolo. Por exemplo, o estabelecimento de conexão, transmissão de dados, dentre outras funcionalidades.

- **Restringir o número de vezes de um ciclo**: permite especificar o número de vezes que se deseja gerar um determinado ciclo. Com o uso dessa restrição, o usuário

pode verificar o número de vezes que um determinado ciclo precisa ser gerado para que a seqüência de teste seja executável.

Essas restrições podem ser combinadas e gerar testes para uma determinada funcionalidade do protocolo. Quando nenhuma restrição é imposta pelo usuário, testes são gerados para cobrir todas as transições, passando uma vez em cada ciclo existente na especificação do protocolo.

Para restringir as transições, é necessário especificar quais transições deverão ser consideradas na geração dos testes. Para isso, o usuário deve usar a identificação da transição, conforme especificada em LEP (ver seção 5.4). Quanto a restringir o número de ciclos, o usuário deverá especificar as transições que compõem esse ciclo, juntamente com o número de vezes que este ciclo deverá ser percorrido.

O capítulo 6 apresenta uma série de exemplos que usam restrições para geração de casos de teste.

5.7 Vantagens e Limitações da CONDADO

A CONDADO é uma ferramenta de geração de teste de protocolos, que engloba a parte de controle e os parâmetros de interações, para MFEE.

Uma das grandes contribuições da CONDADO está na validação do método proposto no capítulo 4, que propõe a combinação de diversas técnicas de teste caixa preta para cobrir todos os aspectos do protocolo. As técnicas implementadas até agora são: teste de transição de estados, teste de sintaxe e teste de domínio. Não é possível ainda comprovar a cobertura de todos os aspectos do protocolo, pois testes de predicados não foram implementados.

Apesar da CONDADO não detectar falhas de transferência, ela detecta falha nos dados e no formato das interações. Isso ocorre porque são implementadas técnicas de teste de sintaxe e domínio combinadas aos testes de transição de estados.

Outra vantagem está no uso de restrições. A TAG [TPB96] também apresenta esta funcionalidade, mas ela permite somente a escolha de uma transição, enquanto que na CONDADO é possível cobrir uma funcionalidade do protocolo, com a escolha de mais de uma transição.

Um dos problemas em gerar testes para MFEE sem considerar predicados é que pode-se gerar caminhos não executáveis. Vários fatores influenciam a executabilidade de uma transição, como os valores das variáveis de contexto, valores dos parâmetros das interações, dentre outros fatores. Na tentativa de contornar esse problema, a CONDADO oferece mecanismos para gerar um número repetido de vezes de cada ciclo. Dessa forma, é possível tratar os casos onde a executabilidade de uma transição depende da geração de um número x de vezes de um determinado ciclo.

Como a CONDADO não garante a executabilidade de todos os casos de teste gerados, uma intervenção do usuário é sempre necessária. Dessa forma, o usuário pode alterar o valor gerado para um dado parâmetro, acrescentar ou retirar uma dada transição a fim de que o caso de teste seja executável. Para possibilitar essa intervenção do usuário, os casos de teste são gerados em um formato de fácil compreensão. Exemplos dos casos de teste gerados a partir da CONDADO são apresentados no capítulo 6.

5.8 Resumo

Este capítulo apresentou os componentes e funcionalidades da ferramenta CONDADO. Foi apresentada também uma linguagem para especificação de protocolos, denominada de LEP.

Baseado no método proposto no capítulo 4, a CONDADO transforma a especificação do protocolo em uma especificação de teste. A partir dessa especificação, técnicas de teste caixa preta são usadas a fim de gerar testes para cobrir toda a especificação do protocolo.

Antes de iniciar o processo de geração de teste, é necessário que a especificação seja transformada em LEP. A seção 5.4 descreve os componentes da LEP e como especificar um protocolo usando essa linguagem. As principais motivações que levaram a criação de uma linguagem para especificação do protocolo foram a simplicidade oferecida e a possibilidade de representar controle, dados e formato das interações usando uma única notação.

Com as informações obtidas da especificação em LEP, a CONDADO inicia a geração dos casos de teste. O primeiro passo é a transformação em uma especificação de teste. A especificação usa a notação na forma de *Cláusulas de Horn*, pois, como o gerador é implementado em Prolog, essa especificação possibilita a consulta pelo gerador de forma direta. O componente responsável pela especificação de teste é o **conversor**.

Tendo a especificação de teste, o próximo passo é a geração dos casos de teste. Esta etapa é de responsabilidade do **gerador**. Conforme o método descrito no capítulo 4 técnicas de teste caixa preta são usadas para derivar os casos de teste. Três técnicas foram implementadas na CONDADO: teste de transição de estados, teste de sintaxe e teste de domínio. Essas técnicas são combinadas possibilitando a geração de teste cobrindo aspectos de controle e dados do protocolo.

A CONDADO oferece também o uso de restrições na geração dos testes. Dessa forma, é possível selecionar funcionalidades do protocolo para serem testadas ou fornecer o número de vezes que se deseja gerar um determinado ciclo. Essas restrições são úteis para limitar o número de testes gerados e também podem ajudar a contornar o problema da geração de casos de teste não executáveis.

Capítulo 6

Geração de Testes Usando a CONDADO

Após a descrição da CONDADO e das suas funcionalidades (capítulo 5), este capítulo descreve as saídas geradas pela CONDADO e os resultados obtidos com a geração de teste para alguns experimentos.

Inicialmente, uma descrição do formato dos casos de teste gerados é apresentada. Com isso, o usuário da CONDADO possui subsídios suficientes para interpretar os resultados obtidos.

Também são apresentados exemplos mostrando a importância do uso de restrições. O uso de restrições para a geração de testes é um recurso da CONDADO para permitir a redução do número de estados e transições a serem exercitadas, reduzindo assim, o número de casos de teste.

Neste capítulo são apresentados também os resultados de um experimento realizado usando uma implementação do protocolo de comunicação da camada de transferência do sistema de telecomando da estação solo do satélite SACI-1 [Car97]. Este experimento foi feito com o objetivo de validar a CONDADO e também, a linguagem LEP, usada para especificar o protocolo a ser testado.

6.1 Formato dos Casos de Teste Gerados

Com a execução da CONDADO é possível obter os casos de teste para uma determinada especificação satisfazendo certas restrições do usuário.

Casos de teste definem um conjunto de seqüências de interações de entrada para serem aplicadas à IUT. Os casos de teste podem incluir também as interações de saída enviadas pela IUT em resposta à uma dada entrada [BD97].

Cada caso de teste é formado por um conjunto de **interações de teste**. Uma interação

de teste é composta por uma entrada e as saídas enviadas em resposta à essa entrada. Protocolos representados através de MFEE incluem as transições espontâneas. Transições estas que não dependem de nenhuma entrada e estão associadas aos predicados (ver seção 3.1.3). Dessa forma, uma interação de teste para uma transição espontânea irá incluir somente as saídas enviadas pela IUT.

Os casos de teste gerados pela CONDADO incluem interações de entrada e saída. Segue um exemplo de um caso de teste gerado pela CONDADO para a máquina de estados da figura 5.2:

```
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"xHn*e",2,14)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,ACK)  recdata(U,MONITOR_COMPLETE)
                 recdata(L,TOKENrelease)  recdata(L,DISrequest)
senddata(L,DISrequest)  recdata(U,DISindication)
```

Nesse caso, `senddata` indica o envio de uma primitiva a IUT. O seu primeiro parâmetro especifica a interface pela qual será enviada a primitiva, superior (U) ou inferior (L). O segundo parâmetro especifica o nome da primitiva. Outros parâmetros podem ser acrescentados e são usados para especificar os valores dos dados associados à interação de entrada.

A saída enviada pela IUT é representada por `recdata`. Seu primeiro parâmetro especifica a interface para a qual a IUT irá enviar a primitiva de saída (inferior (U) ou superior (L)). O segundo e último parâmetro representa o nome da primitiva a ser enviada. Neste caso, os dados associados as interações de saída não são especificados.

Para representar as transições espontâneas e as transições que não geram saída, usa-se `senddata( )` e `recdata( )`, respectivamente. Neste caso, esses comandos não possuem parâmetros.

No caso de teste exemplificado acima a primeira interação de teste é `senddata(U, SENDrequest) recdata(L,CR)`. Essa interação mostra que a entrada `SENDrequest` será enviada a IUT pela interface superior, e que a camada inferior está esperando a saída `CR` da IUT em resposta a essa entrada.

A primitiva `DATArequest` possui dados associados. Os dados associados a essa primitiva são (ver seção 5.4.4):

```
DATArequest :: = SEQUENCE {
                 SDU                 octetstring    5,
```

```
number_of_segment   enumerated      2 | 4 | 8,
blockbound          integer         3 .. 15};
```

Para os dados dessa primitiva, a CONDADO gerou os seguintes valores: `senddata (U,DATArequest,"xHn*e",2,14)`, onde a cadeia de caracteres `xHn*e` delimitada por aspas(") foi gerada para o parâmetro `SDU`. Para `number_of_segment` foi gerado o valor 2 e para `blockbound` foi gerado o número 14.

6.2 Resultados Obtidos com o Uso de Restrições

Como descrito na seção 5.6.6, as restrições assumem um papel importante na geração dos casos de teste. Elas podem diminuir significativamente o número de testes gerados. Além disso, as restrições permitem ao usuário testar partes específicas (ou funcionalidades) do protocolo. As restrições podem ser aplicadas às transições ou aos ciclos.

Para a MFEE exemplificada na figura 5.2, várias seqüências de testes foram geradas usando diferentes restrições. Os resultados obtidos são apresentados na tabela 6.1.

Restrição	Tempo de CPU (em segundos)	Número de Inferências	Número de Casos de Teste
nenhuma	9.97	1.167.096	156
cobrir a transição `t11`	9.70	1.117.614	64
cobrir a transição `t17`	9.39	1.082.990	1
passar 2 vezes pelo ciclo `t4-t6`	11.32	1.308.380	156

Tabela 6.1: Dados coletados na geração de teste com uso de restrições

O experimento apresentado na tabela 6.1 foi feito usando uma estação Sun4m SPARC. Nesta tabela são especificados: a restrição usada para geração dos casos de teste, o tempo de CPU, o número de inferências e o número de casos de teste gerados. O número de inferências é a soma do número total de chamadas e de *redo* que o Prolog realiza na execução do gerador. Uma chamada ocorre quando o Prolog inicia o processo de tentativa de satisfazer um *objetivo*. Um *redo* ocorre quando o Prolog volta para o objetivo já chamado na tentativa de satisfazê-lo novamente [CM87].

Os métodos de geração de teste implementados pela CONDADO garantem a cobertura de todas as transições. Essa cobertura só não é obtida quando os testes gerados são baseados em alguma restrição. Isso ocorre porque nem sempre um caminho que passa pela transição solicitada passa por todas as outras transições da máquina.

Apesar do número de casos de teste diminuir significativamente com o uso de restrições, o tempo e o número de inferências não diminuem na mesma proporção. Isso ocorre porque

o algoritmo gera um caso de teste e só depois de gerado que ele verifica se este caso de teste satisfaz à restrição imposta pelo usuário. Caso não satisfaça, o caso de teste é descartado e o mesmo processo é realizado para o próximo caso de teste gerado, e assim, sucessivamente.

Na tabela 6.1 foram gerados casos de teste usando restrições nos ciclos. Neste caso, o número de inferências e o tempo irão aumentar significativamente, pois em vez de gerar o conjunto de transições `t4-t6` uma vez, essas transições serão geradas duas vezes, no mesmo caso de teste. Isso explica porque o número de casos de teste não aumenta com o uso dessa restrição, diferente do tempo e do número de inferências.

6.3 Experimentos Realizados

Testes foram gerados para a implementação de um protocolo a fim de validar o método de geração de teste implementado pela CONDADO. O protocolo usado é um protocolo de comunicação da camada de transferência do sistema de telecomando da estação solo do satélite SACI–1 [Car97] do Instituto de Nacional de Pesquisas Espaciais (INPE).

Este protocolo está especificado na forma de MFE. Então, antes de iniciar o processo de geração de teste usando a CONDADO, esta especificação foi transformada em LEP. Com isso, houve também a validação da linguagem para especificação de protocolo (LEP).

A especificação desse protocolo é composta por 6 estados, 46 entradas e um total de 233 transições. A figura 6.1 apresenta uma visão geral da MFE desse protocolo. Cada aresta, conectando dois estados, representa uma ou mais transições existentes entre esses estados. O número total de transições entre dois estados é especificado pelo número que acompanha a aresta existente entre esses estados. Por exemplo, do estado `s1` para o estado `s6` existem 13 transições especificadas no protocolo. O estado inicial da MFE é o estado `s6`.

A MFE apresentada na figura 6.1 está expandida. Para expandir essa MFE foi usada a técnica de *unfolding* (seção 3.3.2). Dessa forma, elevou-se o número de transições presentes na máquina. A técnica de *unfolding* foi aplicada para que todos os eventos pudessem ser especificadas através das transições. Em uma MFEE, mais de um evento pode ser especificado em uma transição usando predicados e variáveis de contexto, o que diminui, sensivelmente, o número de estados e transições presentes na máquina.

Como apresentado na seção 3.3.2, uma das limitações no uso de *unfolding* é o aumento do número de estados e/ou transições. Neste protocolo houve um aumento no número de transições. Isso ocasionou a geração de uma grande quantidade de casos de teste, conforme será apresentado na seção 6.3.1. Além disso, a especificação desse experimento é completa, aumentando ainda mais o número de transições da MFE e, em conseqüência, o número de casos de teste gerado.

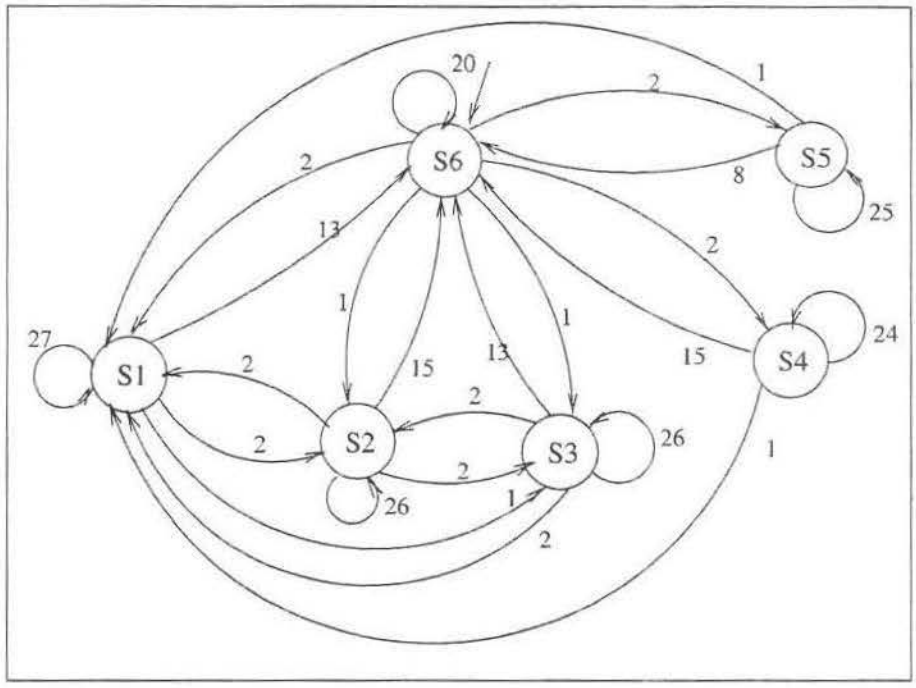

Figura 6.1: Especificação do protocolo de comunicação da estação solo do satélite SACI-1.

A técnica de teste de transição de estados implementada pela CONDADO gera testes cobrindo a combinação de todas as transições. Dessa forma, a existência de ciclos entre duas ou mais transições, que tenham outros ciclos embutidos, pode elevar muito o número de casos de teste gerado. Isso ocorre devido a grande combinação que existe na derivação de testes para ciclos "aninhados". A especificação da figura 6.1 apresenta esse tipo de ciclos. Por exemplo, dentro do ciclo existente entre `s1` e `s3` existem dois outros ciclos: entre `s1` e `s2` e entre `s2` e `s3`.

Para entender o efeito do algoritmo de geração de teste de transição de estados, implementado pela CONDADO, será apresentada a geração de casos de testes para a figura 6.1 com o aumento progressivo das transições. Dessa forma, é possível verificar o quanto aumenta o número de testes gerados à medida que transições que compõem ciclos são acrescentadas à máquina.

6.3.1 Resultados Obtidos na Geração de Teste para o Experimento

Nesta seção será apresentada a geração de teste para a MFE da figura 6.1, com o aumento progressivo das transições que formam os ciclos presentes na máquina.

Os resultados obtidos na geração desses testes têm por objetivo demonstrar como o

algoritmo implementado funciona e, também, mostrar a importância de uma especificação concisa e do uso de restrições para a geração dos casos de teste.

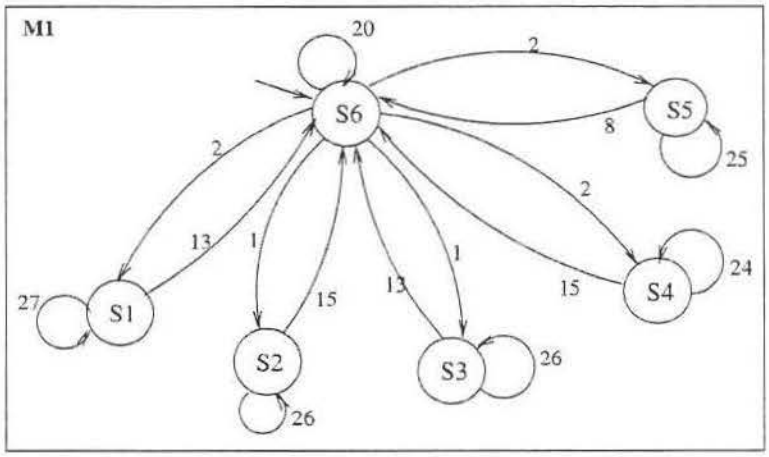

Figura 6.2: MFE da figura 6.1 sem as transições que formam ciclos entre dois ou mais estados

A figura 6.2 apresenta a máquina M1, derivada da MFE apresentada na figura 6.1. Nessa máquina foram retiradas todas as transições que formam ciclos entre estados. Estão presentes somente as transições que formam *self-loops*[1], ou que formam ciclos com o estado inicial (S6). Nesse último caso não existe problemas porque ao retornar para o estado inicial, considera-se que o percurso na MFE terminou e, conseqüentemente, um caso de teste é gerado. A geração de teste foi realizada usando uma estação Sun4m SPARC. Os resultados obtidos na geração de teste para M1 são apresentados na tabela 6.2.

O número de casos de teste aumenta exponencialmente à medida que as transições que formam ciclos são acrescentadas à MFE. A figura 6.3 apresenta as máquinas M2 a M9. A diferença de uma máquina para a anterior é o acréscimo de uma transição pertencente a um ciclo. A transição que está sendo acrescentada é indicada por uma linha pontilhada. Foram gerados testes para cada uma dessas máquinas. Os resultados obtidos na geração dos testes são apresentados na tabela 6.2.

Os resultados apresentados na tabela 6.2 são obtidos com a geração dos casos de teste sem o uso de restrições. Na próxima seção será mostrado como o uso de restrição pode ser útil na diminuição do número de casos de teste.

6.3.2 Uso de Restrições na Geração dos Casos de Teste

A tabela 6.3 apresenta os resultados da geração de teste usando restrição para algumas máquinas da figura 6.3.

[1] *self-loops* são formados por transições onde o estado origem e o estado destino são iguais.

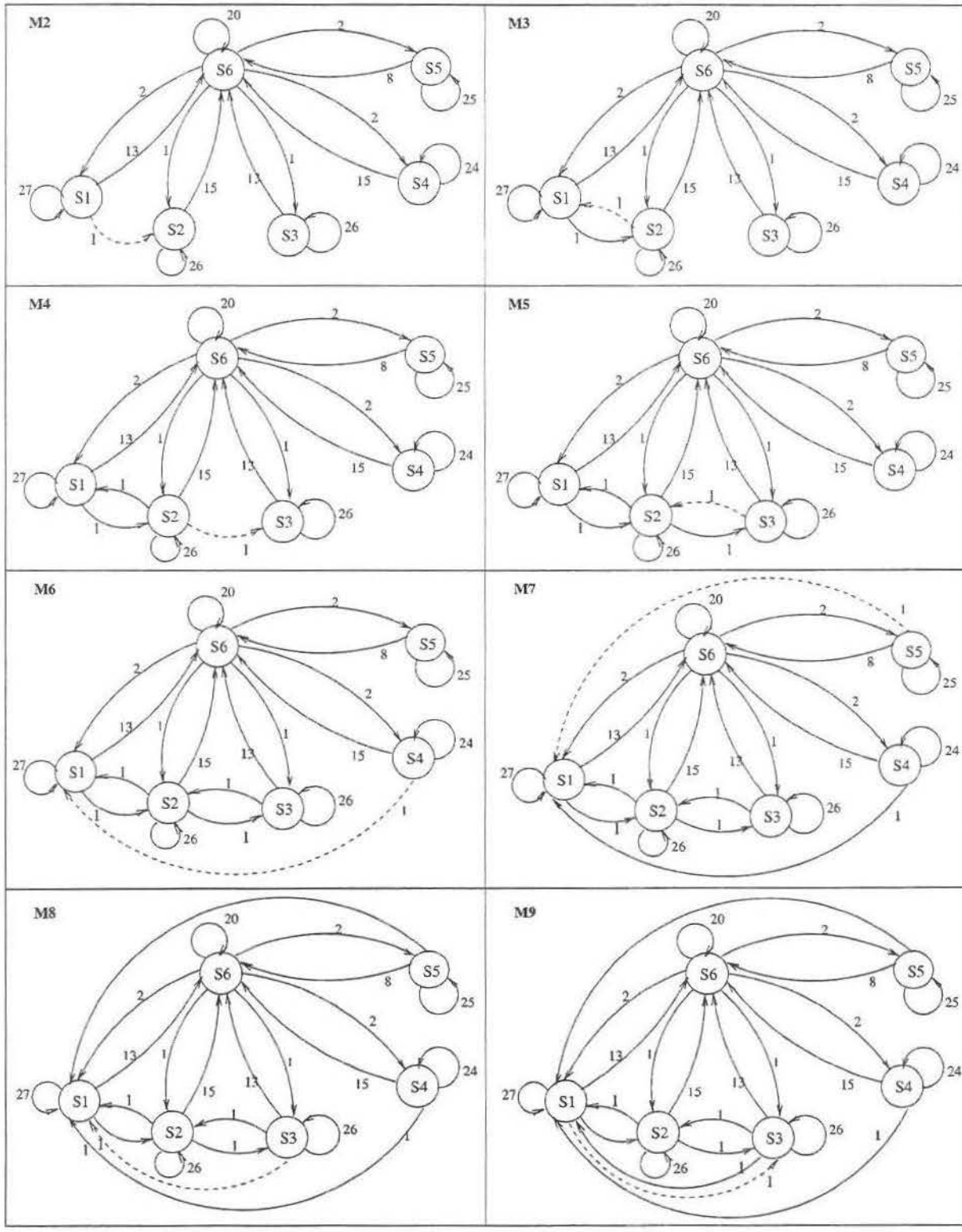

Figura 6.3: Aumento Progressivo das transições para a MFE da figura 6.1

Máquina	Número de Transições	Tempo de CPU (em segundos)	Número de Inferências	Número de Casos de Teste
M1	220	1.74	202.563	324
M2	221	2.21	259.498	406
M3	222	4.85	576.164	689
M4	223	6.64	799.303	923
M5	224	20.90	2.635.606	2.326
M6	225	32.09	4.044.813	3.366
M7	226	44.64	5.511.680	4.406
M8	227	168.38	20.944.831	13.093
M9	228	696.34	86.415.661	51.584

Tabela 6.2: Dados coletados na geração dos testes para MFEs das figuras 6.2 e 6.3

Máquina	Restrição	Tempo de CPU (em segundos)	Número de Inferências	Número de Casos de Teste
M2	transição de `s1` para `s2`	1.59	218.712	82
M3	transição de `s2` para `s1`	4.06	522.495	283
M3	3 vezes o ciclo entre `s1` e `s2`	8.47	1.244.454	818
M7	transição de `s1` para `s2`	37.77	4.715.312	1090

Tabela 6.3: Dados coletados na geração dos testes para MFEs das figuras 6.2 e 6.3 usando restrições

A partir dos resultados obtidos percebe-se que o número de casos de teste diminui sensivelmente. Mas, é preciso deixar claro que depende muito das transições escolhidas para ter essa diferença no número de teste. Muitas vezes, a transição escolhida para a restrição pertence a todos (ou quase todos) os caminhos gerados na MFE. Nesse caso, o número de casos de teste gerados não diminui significativamente comparado ao total de casos de teste para a MFE em questão. Um exemplo típico é a transição `t1` da figura 5.2.

Não foram gerados testes para a especificação completa da MFE representada na figura 6.1. Como pode ser visto na figura 6.1, existe mais de um conjunto de transições que representam o mesmo ciclo. Por exemplo, o ciclo entre `s1` e `s2` é formado por duas transições de `s1` para `s2` e duas de `s2` para `s1`. Dessa forma, o número de combinações aumenta ainda mais.

Considere que na máquina M4 da figura 6.3 seja acrescentada mais uma transição de `s1` para `s2` e mais uma de `s2` para `s3`. Ao gerar testes para essa máquina os seguintes

resultados serão obtidos: tempo de 18.81 segundos, número de inferências igual a 2.632.336 e o número de casos de teste gerados é igual a 2.390. Comparando esse resultado ao da máquina M4, percebe-se que o número de casos de teste gerados, ao aumentar essas duas transições, será mais que o dobro.

6.4 Resumo

Este capítulo apresentou o formato dos casos de teste gerados e os resultados de alguns experimentos obtidos usando a CONDADO.

A seção 6.2 apresenta a importância do uso de restrição na geração dos casos de teste. Alguns exemplos mostram que ocorre uma diminuição significativa no número de casos de teste gerados.

Para validar a CONDADO, testes foram gerados para a especificação de um protocolo real: o protocolo de comunicação da camada de transferência do sistema de telecomando da estação solo do satélite SACI-1 [Car97].

Devido a forma com que o algoritmo para geração de teste foi implementado, em certas máquinas, a combinação de transições pode levar a um grande número de casos de teste gerados. Na seção 6.3.1 testes foram gerados para demonstrar o impacto desse algoritmo perante certos tipos de transições.

Com esses experimentos foi possível validar a CONDADO e, também, a linguagem para especificação de protocolo (LEP). O usuário foi o responsável por transformar a especificação do protocolo de MFE para LEP , não apresentando nenhum problema para a realização dessa tarefa.

Com a geração dos testes usando a CONDADO foi possível observar quais são os tipos de transições que formam os pontos críticos na geração dos testes. Foi possível mostrar também o quanto o uso de restrições pode facilitar o trabalho do usuário para o entendimento e execução dos casos de teste gerados.

Capítulo 7

Conclusão

Nessa dissertação foram apresentados os conceitos relacionados a testes, protocolos de comunicação e testes de protocolos de comunicação. Como o objetivo desse trabalho é a geração de testes a partir de MFEE, foram apresentados, também, os conceitos relacionados com MFE e MFEE, além dos principais trabalhos desenvolvidos nessa área.

A partir dos trabalhos existentes e da necessidade de um método de geração de teste cobrindo todos os aspectos do protocolo, um método para geração de teste foi proposto. Este método foi implementado através de uma ferramenta de geração de casos de teste denominada CONDADO. Resultados obtidos com a geração de teste usando essa ferramenta possibilitaram a sua validação.

Este capítulo apresenta as conclusões obtidas durante a fase de estudo e elaboração desse trabalho. As conclusões estão divididas em: trabalho realizado, dificuldades encontradas para sua realização, contribuições, limitações e as extensões futuras desse trabalho.

7.1 Trabalho Realizado

O objetivo desse trabalho foi a criação e implementação de um método para geração de testes cobrindo os aspectos de controle e dados de um protocolo. Com o estudo dos principais trabalhos na área de teste de protocolo, um método para geração de casos de teste foi proposto no capítulo 4. A partir desse método, uma ferramenta de geração de casos de teste, denominada CONDADO, foi implementada.

Com a implementação dessa ferramenta foi possível mostrar a viabilidade na combinação de diversas técnicas de teste caixa preta. As técnicas implementadas até agora pela CONDADO são: teste de transição de estados, teste de sintaxe e teste de domínio. Não é possível ainda, garantir a cobertura de todos os aspectos do protocolo, pois técnicas de teste levando em consideração os predicados não foram implementadas.

Para possibilitar a geração de teste de uma determinada funcionalidade do protocolo,

o uso de restrições é oferecida pela CONDADO. Essas restrições são úteis para limitar o número de testes gerados e, também, podem ajudar a contornar o problema na geração de casos de teste não executáveis.

Como a geração dos testes é feita a partir da especificação em MFEE, foi definida uma linguagem para especificação do protocolo, denominada LEP. As principais motivações que levaram a criar uma linguagem para especificação do protocolo foram a simplicidade oferecida e a possibilidade de representar controle, dados e formato das interações usando uma única notação.

Testes foram gerados para a implementação de um protocolo a fim de validar o método de geração de teste implementado pela CONDADO. O protocolo usado nesse experimento é o protocolo de comunicação do sistema de telecomando da estação solo do satélite SACI-1 [Car97].

Os experimentos realizados foram úteis para demonstrar o impacto das técnicas de geração de teste implementadas pela CONDADO perante certos tipos de transições. Além disso, foi possível observar quais são os tipos de transições que formam os pontos críticos na geração dos testes e, também, o quanto o uso de restrições pode facilitar o trabalho do usuário para o entendimento e execução dos casos de teste gerados.

7.2 Dificuldades Encontradas

As principais dificuldades foram encontradas na implementação das técnicas de geração de casos de teste, principalmente na técnica de teste de transição de estados. Isso porque o uso do mecanismo de *backtraking* do Prolog facilitaria, em grande parte, a implementação da busca em profundidade, usada para geração dos casos de teste de transição de estados. A dificuldade foi tratar os ciclos existentes na máquina. Ao tentar encontrar novos caminhos em uma máquina com ciclos, usando mecanismo de *backtracking*, Prolog repete o mesmo caminho. Como um ciclo é uma recursão, o mecanismo de Prolog entende que este é um novo caminho.

Foram encontrados na literatura somente algumas soluções para esse problema, mas mesmo assim, tratando problemas muito específicos. Assim, houve a necessidade de estudar esse problema a fim de encontrar uma solução que ajudasse na detecção e tratamento dos ciclos durante a execução da ferramenta.

A idéia inicial era detectar todos os ciclos antes iniciar o processo de geração de teste. Essa idéia foi descartada, pois o número de ciclos em uma máquina de estados pode ser exponencial. Dessa forma, o tempo de execução de um algoritmo para encontrar esses ciclos é de ordem exponencial. Visto que a geração de teste por si já uma tarefa que consome muito tempo, acrescentar o tempo de busca desses ciclos, tornaria o trabalho de geração de teste uma tarefa extremamente demorada.

7.3 Contribuições

Esse trabalho traz diversas contribuições para a área de teste de *software*, mais especificamente para a área de teste de protocolos de comunicação. As principais contribuições são:

1. Estudo e comparação de diversos métodos para geração de casos de teste para protocolos de comunicação a partir de MFE e MFEE;
2. Criação de um método para geração de casos de teste a partir de técnicas de teste caixa preta;
3. Criação de uma linguagem para especificação de protocolo (LEP), possibilitando representar todos os aspectos do protocolo (controle, dados, predicados e ações) usando uma única notação;
4. Implementação de uma ferramenta para validar o método proposto, unindo diversas técnicas de teste caixa preta possibilitando, dessa forma, a detecção de falhas de transferência, de dados e no formato das interações;
5. Obtenção de soluções para detecção e tratamento dos ciclos;
6. Possibilidade de gerar testes cobrindo uma determinada funcionalidade do protocolo, através do uso das restrições;
7. Realização de experimentos usando a especificação de um protocolo real, permitindo, dessa forma, a validação da CONDADO além de oferecer um estudo das transições críticas na geração dos casos de teste.

7.4 Limitações

Um dos problemas em gerar testes para MFEE sem considerar predicados é que casos de teste gerados podem não ser executáveis. Vários fatores influenciam a executabilidade de uma transição, como os valores das variáveis de contexto, valores dos parâmetros das interações, dentre outros fatores. Na tentativa de contornar esse problema, a CONDADO oferece mecanismos para gerar um número repetido de vezes para cada ciclo, através do uso das restrições. Dessa forma, é possível tratar os casos onde a executabilidade de uma transição depende da geração de um número x de vezes para um determinado ciclo.

Como a CONDADO implementa uma combinação de diferentes técnicas de teste e, também a técnica de teste de transição de estados implementada gera uma combinação das transições, o número de casos de teste gerado será muito grande, dependendo da

especificação. O uso das restrições é uma forma que a CONDADO oferece para permitir a geração de testes para partes específicas do protocolo, diminuindo, dessa forma, o número de casos de teste gerados.

7.5 Trabalhos Futuros

Da forma como a CONDADO foi implementada é possível acrescentar novas técnicas de teste sem alterar as técnicas já implementadas.

Como a CONDADO não está tratando os predicados, uma primeira extensão dessa ferramenta é o estudo e implementação das técnicas de teste existentes para geração de teste cobrindo os predicados. Com isso, será possível analisar a cobertura que a combinação de diversas técnicas de teste caixa preta oferece.

Além disso, faz-se necessário um estudo das condições críticas que levam a não executabilidade dos casos de teste gerados.

Outra extensão a esse trabalho é um estudo de viabilidade para implementar outras técnicas de testes de transição de estados, como as técnicas que permitem a identificação de estados. Assim, seria possível realizar uma comparação com a técnica já implementada.

Apêndice A

Gramática Formal para a LEP

A.1 Palavras Reservadas da LEP

- VARIABLES
- STATES
- INPUTS
- OUTPUTS
- DATA
- TRANSITIONS
- END

A.2 *Tokens* da LEP e seus Significados

:	segue palavra reservada
//	precede comentários
=	atribuição
;	encerra uma atribuição
::=	precede um tipo estruturado
{}	delimitam ações
[ ]	delimitam condições e também o tamanho de uma SEQUENCE OF ou SET OF
?	representa a ação de enviar
!	representa a ação de receber
L	segue ? ou ! representa testador inferior

U	segue ? ou ! representa testador superior
>	precede nome do estado origem
<	precede nome do estado destino
#	precede estado inicial
@	precede uma falha
*	precede a identificação da transição

A.3 Descrição Formal da Gramática para a LEP

```
<FSM_espec>        ::= [<variables_defs>]<states_defs><inputs_defs>
                       <outputs_defs>[<data_defs>]<transitions_defs>END'.'
<variables_defs>   ::= VARIABLES':'<variable_def>';'{<variable_def>';' }
<states_defs>      ::= STATES':'<state_def>';'{<state_def>';'}
<inputs_defs>      ::= INPUTS':'<input_def>';'{<input_def>';'}
<outputs_defs>     ::= OUTPUTS':'<output_def>';'{<output_def>';'}
<data_defs>        ::= DATA':'<data_def>';'{<data_def>';'}
<transitions_defs> ::= TRANSITIONS':'<transition_def>';'{<transition_def>';'}
<variable_def>     ::= <variable_name>':'<type>
<state_def>        ::= ['#'] <state_name>
<input_def>        ::= <input_name>
<output_def>       ::= <output_name>
<data_def>         ::= <input_name>'::='<structured_type>'{'
                       <data_name><type><data_value>
                       { ','<data_name><type><data_value>} '}'
<transition_def>   ::= '*'<trans_name>'>'<state_name><transition_cont>'<'<state_name>
<transition_cont>  ::= <input>{<condition>}{<action>}[fault]{<output>} |
                       [<input>]<condition>{<condition>}{<action>}[fault]{<output>}
<structured_type>  ::= SEQUENCE | SET | SEQUENCE OF<size> | SET OF<size> | CHOI(
<input>            ::= '?'('U.'|'L.')<input_name>
<output>           ::= '!'('U.'|'L.')<output_name>
<fault>            ::= '@'<fault_name>
<condition>        ::= '['boolean_exp']'
<action>           ::= '{'program_statements'}'
<data_value>       ::= <value>['..'<value>] | <id>{'|'<id>}
                       // n..m significa n, n+1, .., m-1, m
                       // n|m|k representam valores não seqüênciais
<type>             ::= INTEGER | REAL | OCTETSTRING | BITSTRING
```

| ENNUMERATED
<size> ::= '['<value>']'
<value> ::= digit {digit}
<id> ::= (letter | digit){letter | digit}
<trans_name> ::= <name>
<variable_name> ::= <name>
<state_name> ::= *a* ... *z*{letter | digit}
// nome do estado deve sempre começar com letra minúscula
<input_name> ::= <name>
<output_name> ::= <name>
<data_name> ::= <name>
<fault_name> ::= <name>
<name> ::= letter {letter | digit}

Apêndice B

Geração de Teste Usando a CONDADO

Este apêndice apresenta todos os passos do método de geração de teste implementado pela CONDADO através de um exemplo.

Inicialmente é apresentado a especificação do protocolo usando LEP. A próxima seção apresenta a especificação de teste gerada a partir da especificação em LEP, esta transformação é feita usando o **conversor**. Finalmente, são apresentados alguns casos de testes gerados para esse exemplo usando o **gerador**.

B.1 Especificação em LEP

Para exemplificar o uso da Linguagem para Especificação de Protocolo(LEP), esta seção apresenta a especificação completa do protocolo da figura 5.2.

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

States:
    #idle;
    waitconec;
    connected;
    waitsend;
    sending;
    blocked;
    waitdisc;
```

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

TRANSITIONS:
*t1
>idle
    ?U.SENDrequest
    !L.CR
<waitconec;
*t2
>waitconec
    ?L.CC
```

```
    !U.SENDconfirm
<connected;
*t3
>waitconec
    ?L.DISrequest
    !U.DISindication
<idle;
*t4
>connected
    ?U.DATArequest(SDU, number_of_segment, blockbound)
    {number := 0} {counter := 0}
<waitsend;
*t5
>waitsend
    ?L.RESUME
<waitsend;
*t6
>waitsend
    ?L.TOKENgive {start_timer}
    {number:=number + 1}
    !L.DT(SDU[number])
<sending;
*t7
>sending
    [expire_timer]
    !L.TOKENrelease
<waitsend;
*t8
>sending
    ?L.ACK [number = number_of_segment]
    !U.MONITOR_COMPLETE(counter)
    !L.TOKEN_RELEASE
    !L.DISrequest
<waitdisc;
*t9
>sending
    ?L.ACK [number<number_of_segment] [not expire_timer]
    {number:=number + 1}
```

```
    !L.DT(SDU[number])
<sending;
*t10
>sending
    ?L.BLOCK [not expire_timer]
    {counter:=counter + 1}
<blocked;
*t11
>blocked
    ?L.RESUME [not expire_timer] [counter <= blockbound]
<sending;
*t12
>blocked
    [counter > blockbound]
    !L.TOKENrelease
    !U.MONITOR_INCOMPLETE(number)
    !L.DISrequest
<waitdisc;
*t13
>blocked
    [expire_timer] [counter <= blockbound]
    !L.TOKENrelease
<waitsend;
*t14
>waitdisc
    ?L.RESUME
<waitdisc;
*t15
>waitdisc
    ?L.BLOCK
<waitdisc;
*t16
>waitdisc
    ?L.ACK
<waitdisc;
*t17
>waitdisc
    ?L.DISrequest
```

```
    !U.DISindication
<idle;
END.
```

B.2 Especificação de Teste

Para a especificação em LEP apresentada na seção B.1, a seguinte especificação de teste foi gerada, usando o **conversor** apresentado no capítulo 5:

```
inicial(idle).

trans(idle,t1,waitconnection,L0,Ln):-
        receiveu('SENDrequest',L0,L1),
        transmitl('CR',L1,Ln).

trans(waitconnection,t2,connected,L0,Ln):-
        receivel('CC',L0,L1),
        transmitu('SENDconfirm',L1,Ln).

trans(connected,t3,waitsending,L0,Ln):-
        receiveu('DATArequest',L0,L1),
        sequence([[i,octetstring,5],[c,number_of_segment,3],
                  [i,integer,3,15]],L1,L2),
        transmit(L2,Ln).

trans(waitsending,t4,sending,L0,Ln):-
        receivel('TOKENgive',L0,L1),
        transmitl('DT',L1,Ln).

trans(waitsending,t5,waitsending,L0,Ln):-
        receivel('RESUME',L0,L1),
        transmit(L1,Ln).

trans(sending,t6,waitsending,L0,Ln):-
        receive(L0,L1),
        transmitl('TOKENrelease',L1,Ln).

trans(sending,t7,waitdisconnected,L0,Ln):-
```

```
        receivel('ACK',L0,L1),
        transmitu('MONITOR_COMPLETE',L1,L2),
        transmitl('TOKENrelease',L2,L3),
        transmitl('DISrequest',L3,Ln).

trans(sending,t8,sending,L0,Ln):-
        receivel('ACK',L0,L1),
        transmitl('DT',L1,Ln).

trans(sending,t9,blocked,L0,Ln):-
        receivel('BLOCK',L0,L1),
        transmit(L1,Ln).

trans(blocked,t10,sending,L0,Ln):-
        receivel('RESUME',L0,L1),
        transmit(L1,Ln).

trans(blocked,t11,waitdisconnected,L0,Ln):-
        receive(L0,L1),
        transmitl('TOKENrelease',L1,L2),
        transmitu('MONITOR_INCOMPLETE',L2,L3),
        transmitl('DISrequest',L3,Ln).

trans(blocked,t12,waitsending,L0,Ln):-
        receive(L0,L1),
        transmitl('TOKENrelease',L1,Ln).

trans(waitdisconnected,t13,waitdisconnected,L0,Ln):-
        receivel('RESUME',L0,L1),
        transmit(L1,Ln).

trans(waitdisconnected,t14,waitdisconnected,L0,Ln):-
        receivel('BLOCK',L0,L1),
        transmit(L1,Ln).

trans(waitdisconnected,t15,waitdisconnected,L0,Ln):-
        receivel('ACK',L0,L1),
        transmit(L1,Ln).
```

```
trans(waitdisconnected,t16,fim,L0,Ln):-
        receivel('DISrequest',L0,L1),
        transmitu('DISindication',L1,Ln).

trans(waitconnection,t17,fim,L0,Ln):-
        receivel('DISrequest',L0,L1),
        transmitu('DISindication',L1,Ln).

dados(number_of_segment,0,'2').
dados(number_of_segment,1,'4').
dados(number_of_segment,2,'8').
```

B.3 Casos de Teste Gerados

A partir da especificação de teste apresentada acima, o gerador pode ser executado a fim de gerar as seqüências de teste. Supondo que se deseja gerar testes para a especificação da figura 5.2 com a seguinte restrição: *gerar todos os casos de testes para as transições* `t5` *e* `t12`. Para esta restrição foram gerados 8 casos de teste, num tempo de 9.65 segundos com 1088257 inferências.

Os casos de teste gerados são apresentados a seguir:

```
caso de teste 1:
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata(L,RESUME)  recdata( )
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste 2:
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  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,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste 3:
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata(L,RESUME)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste: 4
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata(L,RESUME)  recdata( )
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste: 5
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata(L,RESUME)  recdata( )
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  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)

caso de teste: 6
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste 7:
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata(L,RESUME)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  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)

caso de teste 8:
senddata(U,SENDrequest)  recdata(L,CR)
senddata(L,CC)  recdata(U,SENDconfirm)
senddata(U,DATArequest,"ws>?}",4,5)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata(L,BLOCK)  recdata( )
senddata( )  recdata(L,TOKENrelease)
senddata(L,RESUME)  recdata( )
senddata(L,TOKENgive)  recdata(L,DT)
senddata( )  recdata(L,TOKENrelease)
senddata(L,TOKENgive)  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)
```

Apêndice C

Descrição do Projeto da CONDADO

Este apêndice apresenta uma descrição detalhada do projeto da ferramenta CONDADO.

Inicialmente, é apresentado o modelo de objetos da ferramenta e um dicionário de dados detalhado, descrevendo cada objeto, atributo e método. O modelo funcional é descrito no capítulo 5 e o modelo dinâmico não faz parte desse documento de projeto, pois o sistema não apresenta um comportamento dinâmico. Neste apêndice é apresentado também uma descrição do **Analisador** e **Verificador de Propriedades**, que fornecem uma série de serviços para a CONDADO.

C.1 Modelo de Objetos

Esta seção apresenta uma descrição do modelo de objetos da CONDADO, um dicionário de dados detalhando as classes, seus métodos e atributos.

C.1.1 Descrição do Modelo de Objetos

A figura C.1 apresenta o modelo de objetos da CONDADO, descrita no capítulo 5. As classes **Analisador** e **Verificador de Propriedades** não fazem parte da CONDADO, mas como fornecem uma série de serviços para a ferramenta, elas estão representadas no modelo.

Um objeto **Conversor** é responsável por gerar a especificação de teste a partir da especificação do protocolo. Essa especificação é armazenada em um objeto **Especificação de Teste**, a partir do qual o objeto **Gerador** produz os casos de testes. Os testes gerados podem ser: testes de transição de estados, de sintaxe e de domínio. Esses testes são armazenados em objetos da classe **Casos de Teste**, podendo ser gerados uma série de

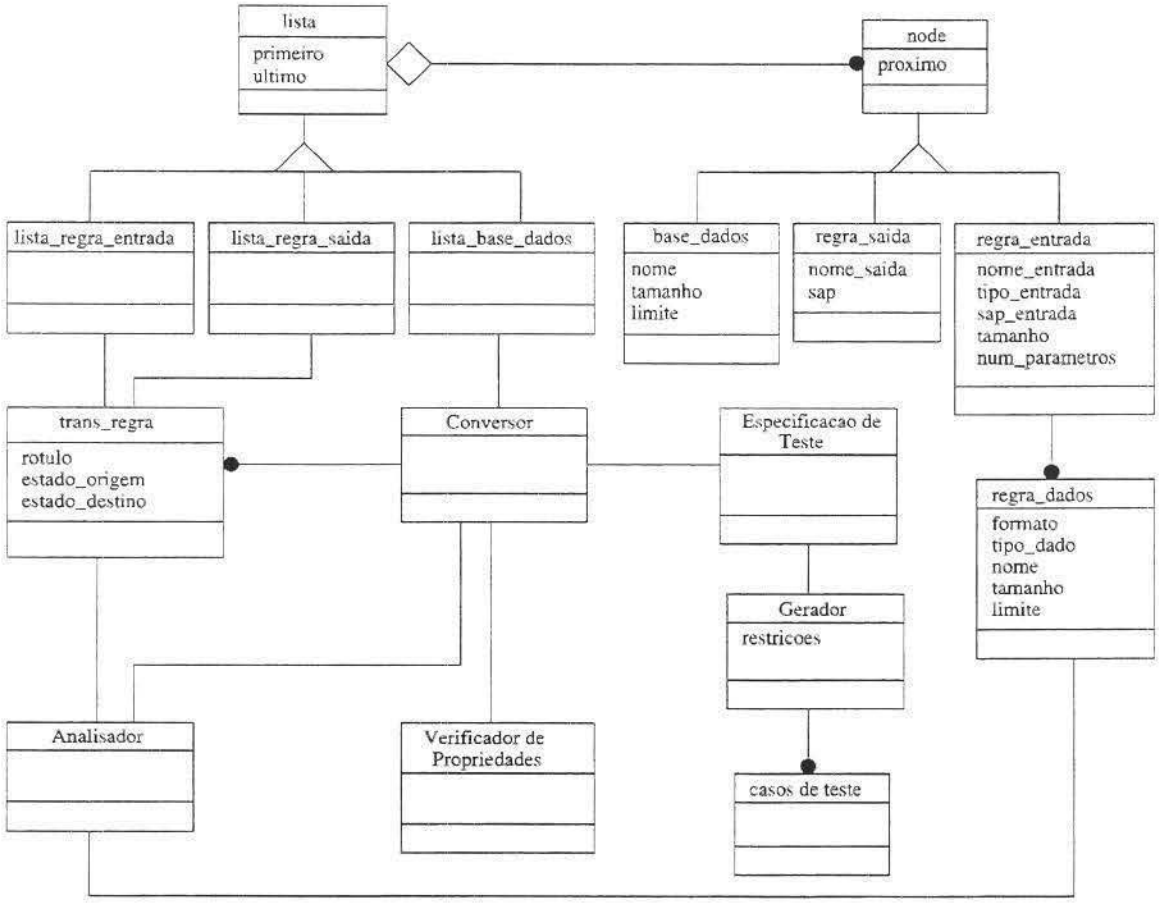

Figura C.1: Modelo de Objetos da CONDADO

testes, variando as restrições a serem aplicadas sobre a especificação de teste. As classes **Especificação de Teste** e **Gerador** foram implementadas em Prolog. As demais foram implementadas em C++.

Os demais objetos do modelo são usados para auxiliar o projeto e implementação das funcionalidades da CONDADO. Na próxima seção será apresentada uma descrição de cada componente do modelo, com uma descrição detalhada dos atributos e métodos de cada uma das classes.

C.1.2 Dicionário de Dados

Esta seção irá detalhar os objetos do modelo apresentado na figura C.1, seus atributos e métodos.

Classe Analisador : classe responsável por fornecer à CONDADO a especificação do protocolo através de uma série de serviços. Esta classe faz parte da ferramenta

denominada **Analisador** que será brevemente descrita na seção C.2.1.

Classe **Verificador de Propriedades** : responsável por fornecer uma série de serviços a CONDADO para verificar quais propriedades que a especificação possui. Esta classe faz parte da ferramenta denominada **Verificador de Propriedades** que será brevemente descrita na seção C.2.2.

Classe **node** : classe auxiliar usada para criar um nó numa estrutura de lista encadeada.
Atributos:
prox: indica o próximo nó da lista.
Operações:
node();

Classe **lista** : classe auxiliar usada para criar uma lista encadeada. Tanto esta classe quanto a classe **node** possuem as funções básicas para criar um nó e uma lista encadeada, e as classes derivadas especializam essas funções para seu propósito.
Atributos:
primeiro: indica o primeiro elemento da lista.
ultimo: indica o último elemento da lista.
Operações:
lista();
*void insere(node *obj_node);* insere um elemento no final da lista.
*node *remove();* apaga o primeiro elemento da lista, devolvendo um ponteiro para este elemento.
*node *obtem_primeiro();* devolve o primeiro elemento da lista sem que ele seja retirado.
enum boolean vazia(); verifica se a lista está vazia.

Classe **regra_entrada** : classe derivada da classe **node**, usada para armazenar as informações de entrada de uma dada transição para construir uma regra em Prolog.
Atributos:
nome_entrada
sap_entrada: especifica o ponto de acesso ao serviço, que pode ser superior (U) ou inferior (L).
tp_entrada: especifica o tipo estruturado da entrada (SET, SEQUENCE, CHOICE, SET OF, SEQUENCE OF), caso a entrada tenha dados relacionados.
tamanho: especifica o tamanho máximo para um vetor quando os tipos estruturados forem iguais a SEQUENCE OF ou CHOICE OF.
num_parametros: determina o número de parâmetros, caso a entrada tenha dados associados.

Operações:
regra_entrada(int token);
regra_entrada();
*void atribui_nome_entrada(char *init_nome);*
*void atribui_sap(char *init_sap);*
void atribui_tp_entrada(enum t_estruturado init_tp_entrada);
void atribui_tamanho(int init_tamanho);
void atribui_num_parametros(int init_num_parametros);
*void atribui_lst_regra_dados(regra_dados *init_lst_regra_dados, int i);*
*char *obtem_nome_entrada();*
*char *obtem_sap();*
enum t_estruturado obtem_tp_entrada();
int obtem_tamanho();
int obtem_num_parametros();
*regra_dados *obtem_lst_regra_dados(int i);*

Classe lista_regra_entrada : classe derivada da classe **lista** que compõe uma lista encadeada de objetos da classe **regra_entrada**, contendo todas as entradas de uma transição.

Operações:
lista_regra_entrada() ;
*void insere(regra_entrada *obj_node);*
*regra_entrada *remove();*
*regra_entrada *obtem_primeiro(); enum boolean vazia();*

Classe regra_dados : objetos desta classe são usados para armazenar temporariamente dados dos parâmetros de entrada.

Atributos:
formato: especifica se o conjunto de valores para este parâmetro é um intervalo de valores, representado pela letra 'i' (por exemplo, 0..15) ou um conjunto de valores aleatórios (por exemplo, 2|5|9), representados por 'c'.
nome: especifica o nome do parâmetro.
tp_dado: especifica seu tipo que pode ser: `integer`, `real`, `octetstring`, `bitstring`, `boolean` ou `ennumerated`.
tamanho: especifica quantos campos terá o atributo limite.
limite: vetor contendo todos os possíveis valores para o parâmetro, ou seus valores limites quando o formato for igual a um intervalo.

Operações:
regra_dados(int token, int i);

regra_dados();
void atribui_formato(char init_formato);
*void atribui_nome(char *init_nome);*
void atribui_tp_dado(enum tipo init_tp_dado);
void atribui_tamanho(int init_tamanho);
*void atribui_limite(int tam, char * *init_limite);*
char obtem_formato();
*char *obtem_nome();*
enum tipo obtem_tp_dado();
int obtem_tamanho();
*char *obtem_limite(int posicao);* devolve um valor do vetor limite especificado pelo parâmetro **posicao**.
char *obtem_limite();* devolve o vetor limite.

Classe **regra_saida** : classe derivada da classe **node**, usada para armazenar as informações de saída de uma dada transição, para construir uma regra em Prolog.
Atributos:
nome_saida
sap: especifica se o ponto de acesso ao serviço é superior (U) ou inferior (L).
Operações:
regra_saida(int token);
regra_saida();
*void atribui_nome_saida(char *init_nome_saida);*
*void atribui_sap(char *init_sap);*
*char *obtem_nome_saida();*
*char *obtem_sap();*

Classe **lista_regra_saida** : classe derivada da classe **lista**. Esta classe compõe uma lista encadeada de objetos da classe **regra_saida**, contendo todas as saídas de uma transição.
Operações:
lista_regra_saida() ;
*void insere(regra_saida *obj_node) lista::insere((node*)obj_node);*
*regra_saida *remove() return (regra_saida*)lista::remove();*
*regra_saida *obtem_primeiro() return(regra_saida*)lista::obtem_primeiro();*
enum boolean vazia() return lista::vazia();

Classe **base_dados** : classe derivada da classe **node**. Um objeto desta classe é usado para armazenar as informações de um parâmetro do tipo coleção. Os objetos dessa classe serão escritos na base de dados do Prolog em forma de fatos.

Atributos:
nome: especifica o nome do parâmetro.
tamanho: especifica o tamanho do atributo **limite**.
limite: contém todos os valores possíveis para o parâmetro.
Operações:
base_dados();
base_dados();
*void atribui_nome(char *init_nome);*
void atribui_tamanho(int init_tamanho);
*void atribui_limite(char * *init_limite);*
*char *obtem_nome();*
int obtem_tamanho();
*char *obtem_limite(int posicao);*

Classe lista_base_dados : classe derivada da classe **lista**. Esta classe compõe uma lista encadeada de objetos da classe **base_dados**, contendo todos as informações dos parâmetros de uma transição que são do tipo coleção.
Operações:
lista_base_dados() ;
*void insere(base_dados *obj_node) lista::insere((node*)obj_node);*
*base_dados *remove() return (base_dados*)lista::remove();*
*base_dados *obtem_primeiro() return(base_dados*)lista::obtem_primeiro();*
enum boolean vazia() return lista::vazia();

Classe trans_regra : Um objeto desta classe é usado para armazenar as informações obtidas de uma transição. Essas informações são obtidas a partir do código intermediário e da tabela de símbolos. Contendo assim, todos os dados necessários para gerar uma regra em Prolog.
Atributos:
rotulo: usado para identificar a transição (`t1`, `t2`, etc).
estado_origem: especifica o estado origem da transição.
flag_origem: se este atributo for igual a 1, então o estado origem da transição é o estado inicial da MFEE.
estado_destino: especifica qual o estado destino da transição.
Operações:
trans_regra();
trans_regra();
void atribui_rotulo(int token);
void atribui_estado_flag_origem(int token);

void atribui_obj_regra_entrada(int token);
void atribui_nulo_lst_entrada();
void atribui_obj_regra_saida(int token);
void atribui_nulo_lst_saida();
void atribui_estado_destino(int token);
*char *obtem_rotulo();*
*char *obtem_estado_origem();*
int obtem_flag_origem();
*regra_entrada *obtem_obj_regra_entrada();*
*regra_saida *obtem_obj_regra_saida();*
*char *obtem_estado_destino();*

Classe **conversor** : Esta classe contém métodos para transformar as transições especificadas em LEP em regras ou fatos em Prolog.

Operações:

conversor();
conversor();
*void atribui_obj_base_dados(base_dados *init_base_dados);*
*base_dados *obtem_obj_base_dados();*
void gera_regras(); : gera uma regra em Prolog para cada transição da MFEE.
void escreve_base_dados(); : escreve fatos em Prolog para cada parâmetro do tipo coleção.
*void escreve_regra(trans_regra *obj_regra);* : escreve a regra gerada pelo método **gera_regras** em um arquivo de saída.
*void pega_tipo_entrada(enum t_estruturado tipo_entrada, char *temp);*
void trata_erro(int num_erro); : trata os erros de especificação da MFEE, como falta de estado origem, falta de estado destino, dentre outros erros.

Classe **especificação de teste** : Objetos desta classe são construídos a partir dos métodos da classe **conversor**. Estes objetos são usados para armazenar a especificação de teste em Prolog.

Classe **gerador** : Esta classe contém métodos que são responsáveis por gerar os casos de teste, levando em consideração as restrições, obtidas através do atributo restrições. Esta classe foi implementada na linguagem Prolog.

Atributos:

restrições: atributo usado para especificar as restrições a serem consideradas na geração dos casos de teste.

Operações:

gera_teste_trans_estado; : gera testes de transição de estados.

gera_teste_sintaxe; : gera testes de sintaxe.
gera_teste_domínio; : gera testes de domínio para os parâmetros das entradas. A estratégia de teste usada é a geração de um valor aleatório dentro do domínio especificado.

Classe casos de teste : Objetos desta classe são construídos a partir dos métodos da classe **gerador**. Contém um conjunto de casos de teste gerados para a especificação de um determinado protocolo.

C.2 Interface com Outras Ferramentas

A CONDADO utiliza os serviços de duas ferramentas para obter as informações sobre a especificação em LEP da MFEE. Dessa forma, faz-se ncessário uma descrição dessas ferramentas e os serviços que elas oferecem para a CONDADO.

C.2.1 Analisador

O analisador oferece uma série de serviços para a CONDADO. Esses serviços têm por objetivo possibilitar a obtenção do código intermediário e das informações contidas na tabela de símbolos. Na verdade, o analisador implementa os passos de análise léxica e sintática de um compilador.

Para que a CONDADO possa obter as informações contidas nas tabelas de símbolos e, também, obter o código intermediário, o analisador oferece um conjunto de serviços. Esses serviços estão descritos na tabela C.1, juntamente com a forma da chamada de cada um deles.

O serviço **obter código intermediário** fornece o código intermediário de uma determinada transição através da variável `transicao`. A variável `posicao` é usada como um contador, iniciando em 0 até o número máximo de transições. O serviço retorna o valor 1 quando o valor associado à variável `posicao` corresponde a uma transição e o valor 0 caso contrário. A chamada desse serviço é apresentada na tabela C.1.

Com o código intermediário da transição, o próximo passo é obter as informações da tabela de símbolos representadas pelos *tokens*. O código intermediário de uma determinada transição é devolvido da seguinte forma pelo serviço **obter código intermediário**:

```
*13 >19 ?87 !21 <56;
```

Nesse caso, a transição possui uma entrada e uma saída, e não possui nem condição e nem ação. O *token* 13, representa a identificação dessa transição, pois como visto na seção anterior, o símbolo * precede a identificação da transição. Para obter essa identificação,

Serviço	Forma da Chamada
Obter código intermediário	`int obtem_cod_intermediario(int posicao, char* transicao);`
Obter identificação da transição	`void obtem_ident_transicao(int token, char *ident_transicao);`
Obter dados estado	`void obtem_dados_estado(int token, char *nome_estado, int &flag);`
Obter dados entrada	`void obtem_dados_entrada(int token, char *nome_entrada,char *sap, enum t_estruturado &tipo_entrada, int &tamanho, int &num_parametros);`
Obter dados parâmetros	`void obtem_dados_parametros(int token, int posicao,char *nome, enum tipo &tipo_dado, char &formato, int &tamanho,char* *limite);`
Obter dados saída	`&void obtem_dados_saida(int token, char *nome_saida, char *sap);`

Tabela C.1: Serviços oferecidos pelo analisador

o serviço **obter identificação da transição** é oferecido pelo analisador. Este serviço tem como entrada o *token* (no exemplo acima, o número 13), e devolve a identificação da transição através da variável `ident_transicao`.

Para obter o estado origem identificado por um *token* no código intermediário que é seguido do símbolo >, o serviço **obter dados estado** é chamado. A partir do *token* que identifica o estado origem, esse serviço devolve o nome do estado e se este estado é o estado inicial. Para identificar que o estado é inicial da especificação a variável `flag` será igual a 1, caso contrário esta variável terá valor 0.

Para obter as informações da interação de entrada, o serviço **obter dados entrada** é oferecido pelo analisador. O *token* associado à entrada segue o símbolo ?, e é usado pelo serviço para devolver as seguintes informações da interação de entrada:

1. **nome da entrada**: essa informação é devolvida pela variável `nome_entrada`;
2. **ponto de acesso ao serviço**: essa informação é obtida pela variável `sap` e especifica se a entrada chegou à IUT pelo **ponto de acesso ao serviço superior** (USAP) ou **inferior** (LSAP);
3. **tipo de dado da entrada**: a variável `tipo_entrada`, será igual a *nenhum* caso a interação de entrada em questão não tenha dados associados. Se a entrada possuir

dados, essa variável conterá o seu tipo, que pode ser: `SEQUENCE`, `SET`, `CHOICE`, `SEQUENCE OF` ou `SET OF`, conforme seção 5.4.4;

4. **tamanho**: essa informação será diferente de 0 quando o tipo estruturado da entrada for igual a `SEQUENCE OF` ou `SET OF`. Neste caso a variável `tamanho` conterá otamanho máximo permitido para os dados dessa interação, conforme descrito na seção 5.4.4;

5. **número de parâmetros**: está relacionado aos dados, devolve em `num_parametros` o número de parâmetros que a interação possui. Caso a interação não tenha dados associados, esta variável será igual a 0.

Caso a interação de entrada tenha dados associados, como a entrada `DATArequest` da figura 5.2, o serviço **obter dados entrada** irá retornar algumas informações sobre esses dados, como por exemplo, o número de parâmetros. A partir desse número, é possível obter informações sobre todos os parâmetros associados à entrada.

O serviço **obter dados parâmetros** devolve, um a um, todas as informações dos parâmetros de um dado associado à uma interação de entrada. Para obter essas informações é necessário fornecer o *token* da entrada e um número associado ao parâmetro que se deseja obter. Esse número é um contador, que inicia com 0 e é incrementado a cada chamada ao serviço até `num_parametros-1`. Este número é especificado pela variável `posicao`.

A cada chamada ao serviço **obter dados parâmetros**, as seguintes informações são devolvidas:

1. **nome do parâmetro**: devolve o nome do parâmetro através da variável `nome`;

2. **tipo do dado**: devolve em `tipo_dado` o tipo que o parâmetro em questão possui. Os tipos de dados podem ser: `integer`, `real`, `boolean`, `bitstring`, `octetstring` e `enumerate`, conforme apresentado na seção 5.4.4;

3. **formato**: este campo especifica se os valores para um dado parâmetro estão dentro de um limite (`0..23`), ou se são informações aleatórias do tipo `enumerate` (`2|4|8`). No primeiro caso, a variável `formato` será igual a `i` (intervalo) e no segundo, igual a `c` (coleção);

4. **tamanho**: esse campo é usado para armazenar o tamanho do vetor que conterá os possíveis valores para o parâmetro. Se o parâmetro for do tipo `enumerate`, a variável `formato` irá conter o número de elementos que formam esse tipo. Por exemplo, para um parâmetro com valores `2|4|8`, a variável `tamanho` será igual a 3. Para parâmetros dos tipos `integer` e `real`, `tamanho` será igual a 2, pois esses tipos possuem limite inferior e superior para serem representados. Para os tipos `bitstring` e

octetstring, tamanho será igual a 1, para especificar o tamanho máximo que esse parâmetro pode ter. Para o tipo boolean, não é necessário armazenar nenhuma informação, dessa forma, tamanho será igual a 0.

5. **limite**: representa um vetor que irá conter as informações dos dados do parâmetro em questão. Veja exemplo na tabela C.2. O tamanho desse vetor é dado pela variável tamanho.

Para exemplificar o serviço **obter dados parâmetros**, considere a interação de entrada DATArequest da figura 5.2. A representação dessa transição em LEP é dada da seguinte forma:

```
DATArequest :: = SEQUENCE {
                 SDU                octetstring    5,
                 number_of_segment  enumerated     2 | 4 | 8,
                 blockbound         integer        3 .. 15};
```

A tabela C.2 apresenta a instanciação das variáveis do serviço **obter dados parâmetros**, para cada um dos parâmetros. Neste exemplo será considerado o valor 87 para o *token* da primitiva DATArequest.

Variávies	Valores das variáveis para os parâmetros da primitiva *DATArequest*						
token	87	87			87		
posição	0	1			2		
nome	SDU	number_of_segment			blockbound		
tipo_dado	octetstring	enumerate			integer		
formato	i	c			i		
tamanho	1	3			2		
limite	5	2	4	8	3	15	

Tabela C.2: Resultado da chamada **obter dados parâmetros** para a primitiva *DATA-request*

A saída é representada pelo símbolo ! seguido do seu respectivo *token*. O serviço **oObter dados saída** devolve o nome da saída em nome_saida, e seu ponto de acesso ao serviço especificado em sap. A variável sap será igual a U se a saída é enviada pelo **ponto de acesso superior** (USAP) ou **inferior** (LSAP).

Como a CONDADO não está tratando os predicados e ações, a interface dos serviços para obter essas informações ainda não está definida.

C.2.2 Verificador de Propriedades

Uma ferramenta denominada de **Verificador de Propriedades** foi desenvolvida com o objetivo de verificar a especificação em LEP e devolver quais as propriedades que esta especificação satisfaz.

Na CONDADO os serviços dessa ferramenta são usados no início da geração dos testes. As propriedades que uma especificação necessita ter para ser considerada pela CONDADO são: *Mealy*, Mínima e Inicialmente Conexa. Essas propriedades fazem parte dos requisitos da especificação determinado pelo método proposto nesse trabalho (4.3).

O verificador de propriedades devolve as propriedades que a especificação satisfaz através uma série de serviços. A interface desses serviços é dada da seguinte forma:

```
int obter_mealy( );
int obter_minima( );
int obter_conexa( );
```

Esses serviços devolvem 1 se a propriedade é satisfeita e 0 caso contrário.

Bibliografia

[BD97] Gregor v. Bochmann and Rachida Dssouli. Test development for distributed systems: Towards automation. In *XV Simpósio Brasileiro de Redes de Computadores*, São Carlos - SP., 1997. Tutorial 4.

[BDA96] C. Bourhfir, R. Dssouli, and E. Aboulhamid. Automatic test generation for efsm-based
systems. *RL:http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html/*, August 1996.

[BDAR97] C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols. *RL:http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html/*, March 1997.

[Bei90] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2nd edition, 1990.

[Bei95] Boris Beizer. *Black-Box Testing: techniques for funcional testing of software and systems.* John Wiley & Sons, Inc., 1995.

[Bit96] Guilherme Bittercourt. *Inteligência Artificial - Ferramentas e Teorias.* Instituto de Computação, UNICAMP, 1996. Trabalho apresentado na Escola de Computação.

[BKKW89] S. P. Burgt, J. Kroon, E. Kwast, and H. J. Wilts. The rnl conformance kit. In *2nd Int. Workshop on Protocol Test Systems*, pages 295–310, Berlim, 1989.

[BP94] Gregor v. Bochmann and Alexandre Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proc. Intern. Symposium on Software Testing and Analysis (ISSTA)*, pages 109–124, Washington-USA, 1994. ACM.

[Bra86] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[Bue97] R. N. Bueno. Desenvolvimento de uma ferramenta para geração de testes e falhas baseados em máquinas de estados. Relatório de Iniciação Científica, 1997.

[Car97] M. J. M. Carvalho. Implementação e testes do protocolo da camada de transferência do sistema de telecomando da estação solo do satélite saci-1. Master's thesis, Universidade Federal do Rio Grande do Norte, Natal, dez. 1997.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 3rd edition, 1987.

[CS87] R. Castanet and R. Sijelmassi. Methods and semi-automatic tools for preparing distributed testing. In *Protocol Specification, Testing and Verification*, pages 177–188. Elsevier Science Publishers B. V., 1987.

[CVI90] Wendy Y. L. Chan, Son T. Vuong, and M. Robert Ito. On test sequence generation for protocols. In *IX Protocol Specification, Testing and Verification*, pages 119–130. Elsevier Science Publishers B. V., 1990.

[Den91] Richard Denney. Test-case generation from prolog-based specifications. *IEEE Software*, march 1991.

[DH81] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proc. of 5th Int. Conf.on Software Engineering*, pages 170–178, New York, March 1981.

[Ezu95] S. Alan Ezust. Tango: The trace analysis generator. Master's thesis, School of Computer Science, McGill University, Montreal, April 1995.

[FSFT87] An Feng, Yuji Sugiyama, Mamoru Fujii, and Koji Torii. Generating practical prolog programs from attribute grammars. In *11o. Compsac: Computer Software and Aplications Conference*, pages 605–612, Tokio, Japan, October 1987.

[Ham95] Dick Hamlet. Implementing prototype testing tools. *Software-Practice and Experience*, 25(4):347–371, April 1995.

[HLJ95] Hung-Ming Huang, Yuan-Chuen Lin, and Ming-Yuhe Jang. An executable protocol test sequences generation method for efsm-specified protocols. In

IWPTS - 8th International Workshop on Protocol Test Systems, pages 29–44, Evry - France, 1995.

[How80] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980.

[HS91] Daniel M. Hoffman and Paul Strooper. Automated module testing in prolog. *IEEE Transations on Software Engineering*, 17(9):934–943, September 1991.

[HUK95] O. Henniger, A. Ulrich, and H. Konig. Transformation of estelle modules aiming at test case generation. In *IWPTS - 8th International Workshop on Protocol Test Systems*, pages 45–60, Evry - France, 1995.

[ISO88a] ISO/IEC/JCT1/SC21. *OSI Conformance Testing Methodology and Framework - Part 2: Abstract Test Suite Specification.* ISO, 2nd dp 9646-1 edition, July 1988.

[ISO88b] ISO/IECJTC1/SC21. *OSI Conformance Testing Methodology and Framework - Part 1: General Concept.* ISO, 2nd dp 9646-1 edition, July 1988.

[Kee90] Richard A. O' Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[Lin90] Richard J. Linn, Jr. Conformance testing for osi protocols. *Computer Networks and ISDN Systems*, pages 203–219, 1990.

[LLF87] Julius C. B. Leite and Orlando G. Loques Filho. Introdução à tolerância a falhas. In *2nd SCTF*, Campinas - SP, 1987.

[LM88] Ursula Linnenkugel and Monika Mullerburg. On the quality of test data sets. In *1o. Seminário sobre a qualidade de software*, pages 347–363, Bruxelas, Abril 1988.

[Mar95a] B. Marick. *The Craft of Software Testing.* Prentice-Hall, 1995.

[Mar95b] Eliane Martins. Um abiente de testes baseados em injeção de falhas por software. Technical Report DCC-95-24, Unicamp, 1995.

[Mar98] Eliane Martins. Teste de software. Notas de curso, UNICAMP, 1998.

[Maz95] Vitório Bruno Mazzola. Utilização da técnica estelle para gerar sequências de teste para protocolos de comunicação. In *13o. Simpósio Brasileiro de Redes de Computadores*, pages 3–22, 1995.

[MD92] Larry J. Morrell and Lionel E. Deimel. Unit analysis and testing. Technical Report SEI-CM-9-2.0, Carnegie Mellon University, June 1992.

[NV92] Gerald Neufeld and Son Vuong. An overview of asn.1. *Computer Networks and ISDN Systems*, pages 393–415, 1992.

[PM92] Robert L. Probert and O. Monkewich. Ttcn: the international notation for specifying tests of communications systems. *Computer Networks and ISDN Systems*, pages 417–438, 1992.

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

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

[PSSS85] Herbert Pesch, Peter Schnupp, Hans Schaller, and Anton Paul Spirk. Test case generation using prolog. In *8th International Conference on Software Engineering*, pages 252–258, London, UK, August 1985.

[Ray87] D. Rayner. Osi conformance testing. *Computer Networks and ISDN Systems*, pages 79–98, 1987.

[RDT95] T. Ramalingom, Anindya Das, and K. Thulasiraman. A unified test casa generation method for the efsm model using context independent unique sequences. In *IWPTS - 8th International Workshop on Protocol Test Systems*, pages 289–305, Evry - France, 1995.

[RW82] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *6th International Conf. on Software Engineering*, pages 272–278, 1982.

[RW85] S Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, SE-11(4):367–375, April 1985.

[SBC87] Behçet Sarikaya, Gregor v. Bochmann, and Eduard Cerny. A test design methodology for protocol testing. *IEEE Transactions on Software Engineering*, SE-13(5):518–531, May 1987.

[SL89] Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *Transactions on Software Engineering*, 15(4):413–426, April 1989.

[SLC95] Luiz Fernando Gomes Soares, Guido Lemos, and Sérgio Colcher. *Redes de Computadores: das LANs, MANs e WANs às redes ATM.* Campus, 2nd. edition, 1995.

[SM98] Selma Bássiga Sabiao and Eliane Martins. Condado: Uma ferramenta para geração de testes de protocolos combinando controle e dados. In *16o. Simpósio Brasileiro de Redes de Computadores*, pages 404–423, Maio 1998.

[Sou85] Wanderley Lopes de Souza. Utilização de prolog para a especificação e validação de serviços e protocolos de comunicação. In *V Congresso da Sociedade Brasileira de Computação*, pages 51–59, Porto Alegre, 1985.

[Ste97] Márcio Roberto Stefani. Análise de traço com geração de diagnóstico para teste de comportamento de uma implementação de protocolo de comunicação em presença de falhas. Master's thesis, UNICAMP, Campinas - SP, 1997.

[Szw84] Jayme Luiz Szwarcfiter. *Grafos e Algoritmos Computacionais.* Editora Campus, 1984.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 3rd. edition, 1996.

[The91] P. Thevenod. Test: Matériel et logiciel. Notas de Curso, 1991.

[TPB96] Q. M. Tan, A. Petrenko, and G. v. Bochmann. A test generation tool for specifications in the form of state machines. *RL:http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html*, 1996.

[UP83] Hasan Ural and Robert L. Probert. User_guided test sequence generation. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing, and Verification III*, pages 421–436, North - Holland, 1983. Elsevier Science Publishers B.V.

[Ura87] Hasan Ural. Test sequence selection based on static data flow analysis. *Computer Communications*, 10(5):234–242, October 1987.

[UY91] Hasan Ural and Bo Yang. A test sequence selection method for protocol testing. *IEEE Transactions on Communications*, 39(4):514–523, April 1991.

[VJL+94] S. T. Vuong, H. Janssen, Y. Lu, C. Mathieson, and B. Do. Testgen: An environment for protocol test suite generation and selection. *Computer Communications*, 17(4):257–270, april 1994.

[VMJC97] Plínio Roberto Souza Vilela, José Carlos Maldonado, Mário Jino, and Marco Lordello Chaim. Uma visão sobre o teste funcional baseado em análise de fluxo de dados. In *Workshop do Projeto Validação e Teste de Sistemas de Operação*, pages 3–14, Águas de Lindóia, SP, Janeiro 1997.