UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Critérios de Teste Funcional Baseados em Máquinas de Estados Finitos Estendidas

Tese de Mestrado

Marcelo Fantinato
Orientador: Prof. Dr. Mario Jino

Banca examinadora:

Prof. Dr. Mario Jino – Presidente
DCA/FEEC/UNICAMP

Prof. Dr. Ivan Luiz Marques Ricarte
DCA/FEEC/UNICAMP

Prof. Dr. Ricardo Ribeiro Gudwin
DCA/FEEC/UNICAMP

Profa. Dra. Ana Cristina Vieira de Melo
IME/USP – São Paulo

Campinas – SP – Brasil
2002

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Fantinato, Marcelo
F218c Critérios de teste funcional baseados em máquinas de estados finitos estendidas / Marcelo Fantinato.-- Campinas, SP: [s.n.], 2002.

Orientador: Mario Jino.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Engenharia de software. 2. Software - Validação. 3. Teste de software. I. Jino, Mario. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Máquinas de Estados Finitos (MEFs) são uma das técnicas de modelagem mais utilizadas no Teste Baseado em Modelos, uma técnica de teste funcional que utiliza informações de modelos comportamentais do software para a realização do teste. Entretanto, as MEFs tradicionais não dispõem de mecanismos para modelar aspectos de fluxo de dados do comportamento do software. Conseqüentemente, os critérios de teste baseados nesta técnica de modelagem podem considerar apenas aspectos de fluxo de controle do software modelado. Este trabalho define uma extensão das MEFs tradicionais, que oferece suporte à modelagem de fluxo de dados. As MEFs Estendidas são usadas como base para definir um conjunto de critérios de teste funcional, a partir de critérios de teste estrutural conhecidos. Este trabalho discute também a implementação de alguns dos critérios definidos usando a ferramenta de teste POKE-TOOL. Além disso, os resultados de cobertura da aplicação dos critérios de teste funcional propostos sobre uma especificação funcional são comparados com os da aplicação de critérios de teste estrutural sobre a implementação correspondente.

Abstract

Finite State Machines (FSMs) are one of the most commonly used modeling techniques for Model Based Testing, a functional testing technique that makes use of information from behavioral models of the software to carry out the testing task. However, traditional FSMs do not provide mechanisms to model data flow aspects of the software behavior. Consequently, the testing criteria based on this modeling technique can use just control flow aspects of the modeled software. This work proposes an extension to the traditional FSMs, which provides data flow modeling mechanisms. The Extended FSMs are used as a basis to define a set of functional testing criteria, extending known structural testing criteria. The implementation and application of the defined criteria, using the POKE-TOOL testing tool, are also discussed. Moreover, coverage results from the application of the proposed functional testing criteria on a functional specification are compared to those from the application of structural testing criteria on the corresponding implementation.

Agradecimentos

Agradeço principalmente a Deus por ter me dado força e proteção nos momentos difíceis, sabedoria para escolher os caminhos corretos e também pelos momentos de alegria que tive durante esta caminhada.

Com atenção especial, a meu orientador Mario Jino, pela sua dedicação, profissionalismo e compreensão apresentados durante os três anos em que trabalhamos juntos. Por ter me oferecido a oportunidade de crescer pessoal e profissionalmente.

Especialmente a minha família por ter acreditado e depositado total confiança em mim. Agradeço a eles por tudo que fizeram por mim, principalmente por estarem sempre ao meu lado oferecendo amor, carinho e compreensão.

Com um carinho especial, a Sarajane Marques Peres que foi quem mais esteve ao meu lado durante a etapa final de conclusão deste trabalho. Seu apoio e compreensão foram muito importantes para mim.

Ao diretor e gerentes da Diretoria de Soluções em *Billing*, da empresa CPqD Telecom & IT Solutions, por terem me oferecido apoio e incentivo na realização e conclusão deste trabalho.

Ao órgão de apoio a pesquisa CNPq, que financiou parte desta pesquisa.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O objetivo principal da atividade de teste de software é revelar a presença de defeitos no software o mais cedo possível dentro do ciclo de desenvolvimento, buscando melhorar a qualidade do produto que está sendo desenvolvido. Como a realização de um teste exaustivo é impraticável, devem ser aplicados critérios de teste que possibilitem a seleção de um conjunto de casos de teste que possuam alta probabilidade de revelar os defeitos existentes mas cuja aplicação seja viável. Tais critérios também podem ser utilizados na avaliação de cobertura dos casos de teste utilizados durante a realização dos testes (Rapps & Weyuker, 1985; Rocha et al., 2001).

A maior parte dos testes realizados atualmente na indústria de software é feita por meio da aplicação de técnicas de teste funcional, as quais são baseadas nos requisitos funcionais do software e possuem menor custo de aplicação. Entretanto, os trabalhos de pesquisas mais formais, realizados para definir técnicas de teste sistemáticas, têm se concentrado no teste estrutural – baseado na estrutura interna do software – e, assim, os critérios de teste funcional são geralmente descritos de forma pouco rigorosa, tornando-os difíceis de serem avaliados quantitativamente. Deste modo, as empresas produtoras de software não dispõem de muitos critérios de teste formais para realizar e avaliar os testes de seus produtos e acabam utilizando tão somente heurísticas para a realização do teste de software (Offutt et al., 2000).

Essa deficiência tem incentivado as organizações a procurar por técnicas mais formais que melhorem as tradicionais abordagens do teste funcional. Uma das técnicas que tem sido bastante explorada recentemente na área de teste de software é o Teste Baseado em Modelos (TBM). Esta técnica utiliza informações sobre o comportamento funcional do software – descrito por meio de modelos – para a realização dos testes. Estes modelos são construídos a

partir dos requisitos funcionais do software e determinam basicamente quais são as ações possíveis durante sua execução e quais são as saídas esperadas (Apfelbaum & Doyle, 1997; Dalal et al., 1998).

O TBM é uma técnica que está ainda imatura e, por isso, existe uma escassez de critérios formais e de ferramentas de suporte a ela (Offutt et al., 2000). Embora já existam critérios definidos formalmente para essa técnica, estes critérios possuem uma complexidade muito grande em sua aplicação, o que desmotiva o seu uso na indústria de software, devido a seu alto custo de aplicação. Apesar de sua imaturidade, essa técnica possui grande potencial para ser utilizada no teste de grandes sistemas, principalmente por meio de sua automação. Essa automação é possível devido ao suporte oferecido pelo modelo que descreve o comportamento da aplicação, a partir do qual os casos de teste podem ser automaticamente gerados e executados, e os resultados avaliados.

Entre as técnicas de modelagem, uma das mais utilizadas no TBM são as Máquinas de Estados Finitos (MEFs), que têm se mostrado uma excelente ferramenta para modelagem, entendimento e teste de software. Embora as MEFs sejam uma técnica de aplicação simples e intuitiva, existem limitações quanto aos mecanismos de modelagem disponíveis. Uma destas limitações é que elas possibilitam apenas a modelagem do comportamento do software em relação a seu fluxo de controle, não possibilitando a modelagem de seu fluxo de dados. Infelizmente, os critérios existentes baseados apenas em análise de fluxo de controle do comportamento do software exigem apenas que elementos como estados, arcos e laços da MEF sejam exercitados (Offutt, 2000). Como a satisfação desses critérios é, na maioria dos casos, fácil de ser alcançada, a sua aplicação não leva à obtenção de boas indicações de confiabilidade e qualidade do software testado.

Por outro lado, no teste estrutural, já existem critérios de teste baseados tanto no fluxo de controle (Beizer, 1990; Pressman, 1992) quanto no fluxo de dados do software (Rapps, 1985; Maldonado, 1991), os quais são representadas por meio de grafos de programa. Portanto, os critérios de teste estrutural oferecem vários graus de cobertura da estrutura interna do programa testado, sendo amplamente utilizados na análise de cobertura de código durante a execução de um conjunto de casos de teste. Esta análise conta com o apoio de várias ferramentas de teste de software; a ferramenta POKE-TOOL, desenvolvida na UNICAMP (Chaim, 1991; Maldonado, 1991; Bueno et al., 1995), é uma delas.

1.1. Objetivos

O objetivo principal deste trabalho é propor um conjunto de critérios, para ser utilizado no TBM, que considera informações tanto de fluxo de controle como de fluxo de dados do software. Os critérios definidos são baseados em uma extensão das MEFs que oferece suporte à modelagem do fluxo de dados. As MEFs Estendidas possuem uma semelhança estrutural com grafos de programa, permitindo a definição dos critérios de teste funcional com base em critérios de teste estrutural. Entretanto, a semelhança entre as MEFEs e os grafos de programa é meramente estrutural, já que elas não compartilham do mesmo significado semântico. Com isso, por meio de uma extensão de funcionalidade, a ferramenta POKE-TOOL também pode ser utilizada para dar suporte ao uso dos critérios de teste funcional.

Enquanto os critérios baseados em análise de fluxo de controle do software modelado exigem que elementos como comandos, arcos e laços da MEF Estendida sejam exercitados, os critérios baseados em análise de fluxo de dados definidos focalizam em como são atribuídos valores às variáveis da MEF Estendida que direcionam o comportamento do software, e em como essas variáveis são usadas. Assim, ao invés de selecionar caminhos do modelo comportamental do software baseado somente em sua estrutura de controle, os critérios baseados em fluxo de dados selecionam caminhos que rastreiam as ocorrências de variáveis. Um caminho pode passar por todo o ciclo de uma variável – desde a atribuição inicial de valor, passando por possíveis alterações, até que seus valores sejam finalmente usados em um cálculo ou apresentados aos usuários.

Com a definição deste conjunto de critérios, a área de teste de software pode usufruir de um amplo conjunto de critérios de teste funcional com vários níveis de cobertura do modelo comportamental do software – assim como já era possível no teste estrutural, em relação a sua estrutura interna. Embora os critérios de teste funcional definidos sejam estruturalmente similares aos critérios de teste estrutural usados em suas definições, eles são baseados em informações de naturezas diferentes. Isso indica que podem existir características diferentes entre esses dois conjuntos de critérios, tais como: enfoque na detecção da presença de diferentes tipos de defeitos, ou seja, alguns tipos de defeitos podem ser mais sucetíveis de terem sua presença detectada por meio da aplicação de critérios de teste estrutural enquanto outros o terem por meio da aplicação de critérios de teste funcional; e diferentes graus de dificuldade para a satisfação dos critérios, ou seja, critérios de teste estrutural podem exigir a

execução de um número maior ou menor de casos de teste, em relação aos critérios de teste funcional, para sua satisfação completa.

1.2. Estrutura da Dissertação

No Capítulo 2 é apresentada uma visão geral sobre a atividade de teste de software, visando oferecer um entendimento melhor da área em que este trabalho está inserido. Para isso, são apresentados os objetivos principais do teste de software, conceitos básicos relacionados a essa atividade, as fases de execução do teste de software, e as principais técnicas e os principais critérios utilizados no teste de software.

O Capítulo 3 contém uma visão geral sobre o TBM, relacionando os principais trabalhos da área. Para isso, são apresentadas as motivações para o uso da modelagem no teste de software, as principais características do TBM, as fases em que ele é realizado, as técnicas de modelagem que podem ser utilizadas com essa técnica, e uma breve discussão sobre a automação do TBM.

No Capítulo 4 é introduzida a definição das MEFEs e dos critérios de teste funcional baseados em MEFEs. Para isso, são apresentados uma definição das MEFs tradicionais, a definição das MEFEs, um conjunto de diretrizes utilizadas para se modelar o comportamento de um software por meio dessa técnica de modelagem, um exemplo de modelagem de um software por meio dessa técnica, o conjunto de critérios definidos para essa técnica de teste, uma análise das vantagens e desvantagens da utilização dos critérios definidos, e, finalmente, uma análise da aplicabilidade dos critérios.

No Capítulo 5 são descritas a implementação dos critérios definidos na ferramenta POKE-TOOL e a aplicação desses critérios por meio de um experimento prático. Para isso, são apresentados uma descrição das características básicas da versão padrão da ferramenta POKE-TOOL, as adaptações necessárias que foram realizadas na ferramenta para a implementação dos critérios definidos, e o experimento realizado, contendo uma descrição do experimento, os resultados obtidos e uma breve análise desses resultados.

No Capítulo 6 é apresentada a conclusão deste trabalho, incluindo sugestões de trabalhos futuros que podem ser conduzidos na área de teste baseado em MEFEs.

Capítulo 2

Teste de Software

O teste de software é uma das atividades essenciais em qualquer processo de engenharia de software, independentemente do paradigma utilizado. De um modo geral, a realização dos testes inicia-se assim que o software tiver sido implementado numa forma executável por máquina. Durante essa atividade, o software é testado para que se possam descobrir defeitos de função, de lógica e de implementação (Pressman, 1992).

Existem várias estratégias que podem ser utilizadas durante a atividade de teste, incluindo a utilização de diferentes técnicas e ferramentas de teste. A escolha de quais deles utilizar depende das características do software sendo desenvolvido. Portanto, apesar da realização propriamente dita do teste ser iniciada somente depois que o código-fonte tiver sido gerado, deve existir um planejamento da atividade de teste, o qual deve ser iniciado no início do processo de engenharia de software.

2.1. Objetivos do Teste

O objetivo principal do teste é detectar a presença de defeitos no software o mais cedo possível, não somente no código-fonte, mas também nos requisitos, no projeto e na documentação do software (Software Program Managers Network, 1998). Visando atingir este objetivo, o teste deve ser planejado e projetado de modo a definir formas de execução do programa que tenha uma alta probabilidade de revelar defeitos no software (Myers, 1979). Com isso, tem-se que a atividade de teste será realizada com sucesso se ela conseguir descobrir a presença de defeitos no software (Pressman, 1992).

Segundo Pressman (1992), o objetivo secundário do teste é obter uma boa indicação de confiabilidade e alguma indicação de qualidade do software como um todo. A obtenção desta indicação é possível porque o teste, quando não detecta a presença de defeitos, demonstra que as funções do software estão aparentemente funcionando de acordo com as especificações e que os requisitos de desempenho foram aparentemente cumpridos.

Geralmente, por meio da execução do teste, não se pode garantir que um software esteja totalmente correto, pois para a maioria dos softwares o teste não pode mostrar a ausência de defeitos, mas apenas constatar a sua presença.

2.2. Conceitos Básicos

Nesta seção, são apresentadas convenções de conceitos básicos da área de teste de software que são utilizados no restante desta dissertação.

- **Domínio de entrada** – conjunto de valores que podem ser utilizados como entrada para um software.
- **Dado de teste** – valor, pertencente ao domínio de entrada, que se fornece à execução de um software durante o teste.
- **Saída esperada** – resultado esperado da execução de um software correto para um dado de teste.
- **Caso de teste** – par ordenado composto por um dado de entrada e pela respectiva saída esperada.
- **Defeito** – deficiência algorítmica que pode provocar uma saída incorreta com relação à especificação.
- **Elemento requerido** – algum componente do software, ou associado a ele, que deve ser exercitado pelo teste. Por exemplo, a leitura de um dado, a execução de um comando, a apresentação de um resultado, etc.
- **Grafo do programa** – grafo de fluxo de controle do programa, formado por nós e arcos, que define as possíveis seqüências de execução do programa.

2.3. Fases de Teste

A atividade de teste pode ser dividida em fases, de modo que, em cada fase, diferentes tipos de defeitos e aspectos do software sejam abordados, culminando no estabelecimento de estratégias adequadas de geração de dados de teste e de medidas de cobertura. De fato, esta divisão da atividade de teste em fases é uma maneira prática de minimizar a complexidade dessa atividade. Essas fases são executadas de forma incremental e complementar, em função das atividades do processo global de engenharia de software (Rocha et al., 2001).

De acordo com Pressman (1992), devem existir quatro fases de teste, as quais são: teste de unidade – que se concentra na implementação do código-fonte; teste de integração – que se concentra no projeto de software; teste de validação – que se concentra nos requisitos de software; e teste de sistema – que se concentra nos requisitos de sistema. O relacionamento entre as fases de teste e as atividades do processo de engenharia de software é apresentado na Figura 2.1. Cada um desses relacionamentos é mais bem explicado a seguir.

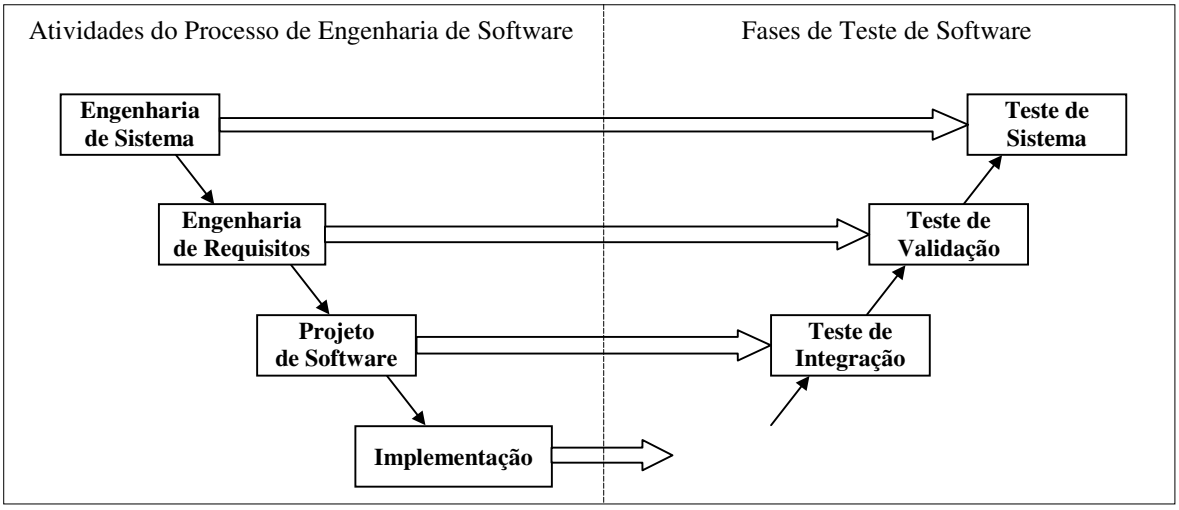

Figura 2.1 – Atividades do Processo de Engenharia de Software e Fases de Teste

A primeira fase de teste que deve ser executada é o teste de unidade, que tem por objetivo explorar a menor unidade funcional (UF) de projeto de software que foi implementada. Segundo o padrão IEEE 610.12-1990 (IEEE-AS Standards Board, 1990), uma UF é um componente de software que não pode ser subdividido. Em programas, uma UF refere-se a uma sub-rotina ou a um procedimento que é a menor parte funcional de um programa que

pode ser executada. Nessa fase de teste, o testador deve procurar identificar a presença de defeitos de lógica e de implementação de cada unidade, tentando garantir que cada uma delas funcione adequadamente.

À medida que as unidades vão sendo testadas, pode-se iniciar o teste de integração. Essa fase de teste tem por objetivo explorar as interfaces entre as unidades quando estas são integradas para construir a estrutura do software que foi estabelecida na fase de projeto. Apesar das unidades já terem sido testadas individualmente, o teste de integração é necessário visto que podem surgir problemas na interação entre essas unidades. Nessa fase de teste, o testador deve procurar identificar a presença de defeitos de comunicação entre as unidades, tentando garantir que elas funcionem em conjunto de forma adequada.

Depois que o software estiver totalmente construído, ou seja, todas as unidades tiverem sido integradas, deve-se iniciar o teste de validação. Essa fase de teste tem por objetivo explorar os requisitos estabelecidos como parte da análise de requisitos de software em relação ao software que foi construído. Nessa fase de teste, o testador deve procurar identificar defeitos de funções e características de desempenho do software que não estejam de acordo com a sua especificação, tentando garantir que o software funcione conforme os requisitos levantados.

Finalmente, depois que o software tiver sido validado, deve-se iniciar o teste de sistema, que tem por objetivo explorar o comportamento do software inserido num sistema mais amplo, contendo, por exemplo, *hardware*, pessoas, bancos de dados, etc. Nessa fase de teste, o testador deve procurar identificar a presença de defeitos de funções e de desempenho global do sistema, tentando garantir que todos os elementos funcionem em conjunto de forma adequada.

Visando a correta execução de cada uma dessas fases, a execução de cada uma delas deve envolver quatro etapas básicas – planejamento, projeto de casos de teste, execução e avaliação dos resultados, as quais devem ser conduzidas ao longo de todo o processo de engenharia de software. O planejamento do teste deve fazer parte do planejamento global do sistema, culminando em um plano de teste que constitui um dos documentos cruciais no ciclo de vida de desenvolvimento de software. Nesse documento são estimados recursos e são definidas as estratégias de teste a serem utilizados, incluindo as técnicas e ferramentas de teste (Rocha et al., 2001).

2.4. Técnicas de Teste

A única maneira de se mostrar a correção de um software seria por meio da execução de um teste exaustivo, ou seja, testar o software com todas as suas combinações de valores de entrada. Entretanto, esta prática é inviável, pois o domínio dos dados de entrada normalmente é infinito ou, pelo menos, muito grande (Myers, 1979). Assim, durante o projeto de casos de teste, torna-se necessário selecionar um subconjunto de dados de teste para ser utilizado.

Existem várias técnicas de teste que podem ser utilizadas para a seleção de subconjuntos de dados de teste. Cada uma destas técnicas possui características próprias que tornam sua aplicação mais indicada a diferentes fases de teste. Essas técnicas de teste são agrupadas em classes de técnicas similares, dependendo de suas características. Exemplos de classes de técnicas de teste são o teste funcional e o teste estrutural.

A diferença básica entre as técnicas de teste funcional e as do teste estrutural é a origem das informações utilizadas na seleção dos dados de teste a serem usados, durante a etapa de projeto de casos de teste. O teste funcional leva em consideração os requisitos funcionais do software durante a seleção; por outro lado, o teste estrutural leva em consideração a estrutura interna do software (Pressman, 1992).

De acordo com Rocha et al. (2001), um ponto que deve ser ressaltado é que a aplicação dessas técnicas de teste detectam a presença de defeitos de categorias distintas, uma vez que elas contemplam diferentes perspectivas do software. Deste modo, impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares de cada uma das classes, levando a uma atividade de teste de boa qualidade, eficaz e de baixo custo.

Outro exemplo de classe de técnica de teste é o Teste Baseado em Erros, que estabelece os requisitos de teste explorando os erros típicos e comuns cometidos durante o desenvolvimento de software (Rocha et al., 2001). Embora as técnicas de Teste Baseado em Erros sejam importantes e bastante pesquisadas, ele não é tratado nesta dissertação. A seguir encontra-se uma breve descrição do teste estrutural e do teste funcional.

2.4.1. Teste Estrutural

O teste estrutural – também chamado de teste de caixa branca – estabelece os elementos requeridos com base em uma certa implementação, solicitando a execução de partes ou de componentes elementares do programa (Pressman, 1992; Myers, 1979).

Usando técnicas de teste estrutural, o testador deriva casos de teste a partir de um exame de elementos da estrutura interna do programa. Embora a funcionalidade do programa seja usada para se determinar qual é a saída esperada para uma dada entrada, a escolha dos dados de teste é realizada olhando-se dentro da "caixa" e escolhendo-se dados de teste para exercitar os elementos requeridos.

Os tipos de defeitos que podem ser revelados por meio do teste estrutural são (Pressman, 1992): defeitos lógicos; pressuposições incorretas; defeitos de projeto; e defeitos tipográficos.

As técnicas estruturais são mais utilizadas nas primeiras fases do processo de teste – teste de unidade e de integração (ver Figura 2.1). O teste de unidade faz muito uso das técnicas estruturais para exercitar caminhos específicos da estrutura de controle de um módulo, a fim de garantir uma completa cobertura e máxima detecção da presença de defeitos. Já o teste de integração faz um menor uso das técnicas estruturais, usadas nessa fase para tentar garantir a cobertura de caminhos importantes de controle entre os módulos integrados (Pressman, 1992).

2.4.2. Teste Funcional

O teste funcional – também chamado de teste de caixa preta – estabelece os elementos requeridos com base na especificação, não utilizando conhecimento algum sobre a implementação. As técnicas dessa classe utilizam como base a especificação, ou seja, os requisitos funcionais, para validar o próprio software (Pressman, 1992; Myers, 1979).

Por meio da realização do teste funcional, o software ou o sistema é tratado como uma caixa preta. Ele é executado com determinadas entradas e suas saídas são verificadas em relação ao comportamento definido por meio da especificação de software. O testador deve estar preocupado somente com a funcionalidade e com características do software e seus detalhes de implementação não devem ser levados em consideração (Beizer, 1990).

O teste funcional procura descobrir a presença de defeitos nas seguintes categorias (Pressman, 1992): funções incorretas ou ausentes; defeitos de interface; defeitos nas estruturas de dados ou no acesso a bancos de dados externos; defeitos de desempenho; e defeitos de inicialização e término.

Ao contrário das técnicas estruturais, as quais são mais utilizadas nas primeiras fases do processo de teste, as técnicas funcionais tendem a serem aplicadas durante as demais fases do processo de teste – teste de integração, teste de validação e teste de sistema (ver Figura 2.1). O teste de integração, além de um pouco das técnicas estruturais, faz mais uso das técnicas funcionais; por outro lado, o teste de validação e o teste de sistema fazem uso exclusivamente das técnicas funcionais (Pressman, 1992).

2.5. Critérios de Teste

De um modo geral, um critério de teste é algum método ou diretriz que serve para direcionar a atividade de teste e/ou tomar decisões relativas ao teste. Um critério de teste define um conjunto de condições que devem ser utilizadas na atividade de teste.

Os critérios de teste podem ser utilizados de duas maneiras:

- Critério de seleção (ou critério de geração): quando o critério é utilizado para selecionar um conjunto de dados de teste;
- Critério de adequação (ou critério de cobertura): quando o critério é utilizado para avaliar a qualidade de um conjunto de dados de teste.

Assim, pode-se dizer que um critério de teste serve para selecionar e/ou avaliar casos de teste de forma a aumentar a probabilidade de se revelar a presença de defeitos ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto. Formalmente, um critério de teste define qual é o conjunto de elementos requeridos do software que deve ser exercitado (Rocha et al., 2001).

A partir de um conjunto de casos de teste, pode-se realizar uma análise de cobertura, que consiste em determinar o percentual de elementos requeridos que foram exercitados pelo conjunto de casos de teste. Com essas informações, o conjunto de casos de teste pode ser

melhorado para que os elementos ainda não abordados sejam testados com adição de novos casos de teste (Rocha et al., 2001).

Existem vários critérios de teste associados às diferentes técnicas de teste. Cada um desses critérios possui vantagens e desvantagens, as quais têm sido avaliadas por meio de estudos teóricos e empíricos (Rapps & Weyuker, 1985; Ntafos, 1988; Clarke et al., 1989; Weyuker & Jeng, 1991). A seguir, encontra-se um resumo das principais técnicas de teste – do teste estrutural e do teste funcional – juntamente com os principais critérios para cada técnica.

2.5.1. Critérios para o Teste Estrutural

As principais técnicas de teste estrutural são: Teste Baseado em Complexidade; Teste Baseado em Fluxo de Controle; e Teste Baseado em Fluxo de Dados. Uma visão geral dessas técnicas e de alguns dos critérios associados a elas é descrita a seguir.

Teste Baseado em Complexidade: essa técnica de teste estrutural utiliza informações sobre a complexidade do software para determinar os elementos requeridos. Um dos critérios dessa técnica é o Critério dos Caminhos Básicos, que utiliza uma medida da complexidade do software – chamada Complexidade Ciclomática – para derivar o conjunto de casos de teste. O valor calculado da Complexidade Ciclomática define o número de caminhos independentes existentes no grafo do programa, que são caminhos através do grafo que introduzem pelo menos um novo conjunto de instruções de processamento ou uma nova condição. Este valor oferece um limite máximo para o número de casos de teste que deve ser projetado e executado para garantir que todas as instruções sejam exercitadas pelo menos uma vez (Pressman, 1992).

Teste Baseado em Fluxo de Controle: essa técnica de teste estrutural utiliza informações sobre o controle de execução do programa, como comandos ou desvios, para determinar os elementos requeridos. Os critérios mais conhecidos dessa técnica são Todos_os_nós, Todos_os_arcos e Todos_os_caminhos, os quais exigem, respectivamente, que cada nó, cada arco e cada caminho do grafo do programa seja executado pelo menos uma vez durante o teste (Pressman, 1992; British Computer Society, 2001).

Teste Baseado em Fluxo de Dados: essa técnica de teste estrutural utiliza informações sobre as variáveis do programa, como definições e usos das variáveis, para determinar os elementos requeridos. Os critérios mais conhecidos dessa técnica são os critérios de Rapps & Weyuker

(Rapps & Weyuker, 1985), que basicamente exploram associações entre os nós do grafo do programa em que existe uma definição de variável e os nós em que existe um uso da mesma variável. Um exemplo é o critério Todos_os_usos, que exige que pelo menos um caminho entre todas as associações definição-uso de cada variável seja executado pelo menos uma vez. Um conjunto de critérios similares aos de Rapps & Weyuker são os critérios de Potenciais Usos (Maldonado, 1991). A diferença é que esses exploram associações entre os nós do grafo do programa em que existe uma definição de variável e os nós em que existe um possível uso da variável. Outros critérios dessa técnica são propostos por Laski & Korel (1983) e Ntafos (1984).

2.5.2. Critérios para o Teste Funcional

As principais técnicas de teste funcional são: Teste de Particionamento de Equivalência; Teste de Análise de Valores Limites; Teste de Grafo de Causa-Efeito; e Teste Baseado em Modelos (Pressman, 1992; Myers, 1979; British Computer Society, 2001). Uma visão geral dessas técnicas e de alguns dos critérios associados a elas é descrita a seguir.

Teste de Particionamento de Equivalência: essa técnica de teste funcional utiliza informações sobre os dados de entrada do software para determinar os elementos requeridos. Esta técnica baseia-se na hipótese de que o domínio de entrada do programa pode ser dividido em classes de equivalência, cujos elementos possuem a mesma capacidade de detecção da presença de defeitos. Embora esta hipótese não possa ser garantida, existem diretrizes que podem ser usadas visando a definição de classes de equivalência com esta característica. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico, um intervalo de valores, um conjunto de valores relacionados ou uma condição *booleana*. Um critério associado a essa técnica consiste em exigir que os casos de teste utilizem pelo menos um dado de entrada de cada classe de equivalência.

Teste de Análise de Valores Limites: essa técnica de teste funcional complementa a técnica de Teste de Particionamento de Equivalência. A diferença principal do critério dessa técnica é que em vez de exigir que os casos de teste utilizem qualquer elemento das classes de equivalência, esse critério exige que os casos de teste utilizem dados de entrada pertencentes aos limites de cada classe de equivalência. Como os dados de teste são obtidos a partir dos

limites tanto das classes com dados válidos como das classes com dados inválidos para um mesmo parâmetro de entrada de dados, então existirão casos de teste que exercitem a violação dos limites especificados para cada parâmetro. Outra diferença é que em vez de criar classes de equivalência com base somente no domínio de entrada, também são criadas classes de equivalência de saídas, ou seja, com base no domínio de saída. Esse critério é importante pois, segundo Pressman (1992), os defeitos costumam ocorrer com maior freqüência nos limites dos domínios de entrada.

Teste de Grafo de Causa-Efeito: essa técnica de teste funcional também utiliza informações sobre os dados de entrada do software, verificando o efeito combinado dos dados de entrada sobre sua execução. Primeiramente, são identificados as causas (condições de entrada) e os efeitos (ações) do software, de acordo com sua especificação. Em seguida, é construído um grafo de causa-efeito, combinando as causas e os efeitos identificados para o software, através de operadores lógicos ("e", "ou", "não") e operadores de restrição (Exclusivo, Inclusivo, Somente um, Exige e Mascara). Depois, o grafo de causa-efeito é então convertido em uma tabela de decisão. Um critério associado a essa técnica consiste em exigir que os casos de teste coloquem a prova cada uma das regras da tabela de decisão.

Teste Baseado em Modelos: essa técnica de teste funcional utiliza informações sobre o comportamento funcional do software, descrito por meio de um modelo comportamental, para determinar como será realizado o teste. O modelo comportamental, construído usando os requisitos funcionais do software, determina quais ações são possíveis em sua execução e quais saídas são esperadas (Apfelbaum & Doyle, 1997; Dalal et al., 1998). Essa técnica é bastante ampla, visto que existem várias formas de se modelar o comportamento do software, e para cada técnica de modelagem podem existir diferentes critérios associados. Exemplos de critérios dessa técnica são Todos_os_estados, Todas_as_transições e Todos_os_caminhos, os quais exigem, respectivamente, que cada estado, cada transição e cada caminho do modelo comportamental do programa seja executado pelo menos uma vez durante o teste.

2.6. Considerações Finais

O teste de software é a atividade do processo de engenharia de software responsável por revelar a presença de defeitos em um programa. Por meio da execução do teste não se pode provar que o software está correto; entretanto, se a realização do teste não detectar a presença

de defeitos, pode-se obter um alto nível de confiabilidade do software. Para que esta realização seja bem sucedida, o teste deve ser cuidadosamente planejado e projetado, levando em consideração as técnicas e as ferramentas de teste existentes.

Visando diminuir a complexidade do teste de software, ele pode ser dividido em quatro fases: teste de unidade, teste de integração, teste de validação e teste de sistema. Em cada uma dessas fases podem ser aplicadas uma ou mais técnicas de teste. As técnicas de teste estão agrupadas em classes de teste, tais como o teste estrutural, que se concentra na estrutura interna do software, e o teste funcional, que se concentra nos requisitos externos do software. É importante ressaltar que as técnicas de teste devem ser utilizadas de forma complementar, uma vez que a aplicação de cada uma detecta a presença de defeitos de categorias distintas.

Embora seja clara a necessidade de se utilizar ambas as técnicas durante a atividade de teste de software, a maioria dos testes realizados na indústria de software é conduzida apenas em nível de sistema, fazendo uso, conseqüentemente, de técnicas de teste funcional (Bourhfir et al., 1999; Offutt et al., 2000). Apesar disso, a maior parte das técnicas e critérios de teste funcional é somente descrita de forma pouco rigorosa, pois a maioria das pesquisas formais tem se concentrado no nível de unidade, ou seja, nas técnicas e critérios de teste estrutural.

Essas técnicas devem ser utilizadas de modo que as vantagens de cada uma sejam mais bem exploradas em uma estratégia de teste que leve a uma atividade de teste de boa qualidade, eficaz e de baixo custo. Dada a diversidade de critérios existentes para cada uma dessas técnicas, para o estabelecimento de uma estratégia de teste com essas características, é indispensável a condução de estudos teóricos e empíricos que permitam adquirir conhecimento sobre as vantagens, desvantagens e limitações de cada um desses critérios (Rocha et al., 2001).

Esta dissertação aborda o Teste Baseado em Modelos, uma técnica de teste funcional que utiliza informações do software descritas por meio de um modelo comportamental para determinar os elementos requeridos. O próximo capítulo descreve melhor essa técnica e o conjunto de critérios de teste associados a ela.

Capítulo 3

Teste Baseado em Modelos

O Teste Baseado em Modelos (TBM) é uma técnica de teste funcional que utiliza informações sobre o comportamento funcional do software para a realização do teste. Esse comportamento é descrito por meio de um modelo – chamado de modelo comportamental, o qual é construído a partir dos requisitos funcionais do software. Os modelos comportamentais determinam basicamente quais são as possíveis ações durante a execução de um software e quais são as saídas esperadas (Apfelbaum & Doyle, 1997; Dalal et al., 1998).

Offutt et al. (2000) afirmam que essa técnica de teste está ainda imatura, o que significa que existe uma escassez de critérios formais e de ferramentas de suporte a ela. Apesar disso, o TBM possui grande potencial para ser utilizado no teste de grandes sistemas, por meio da automação de grande parte da atividade de teste de software. Essa automação é possível devido ao suporte oferecido pelo modelo comportamental da aplicação, a partir do qual os casos de teste podem ser automaticamente gerados e executados, e os resultados avaliados.

3.1. Modelagem de Software

A modelagem é a atividade de construção de modelos que descrevem o comportamento de um sistema. A atividade de modelagem é utilizada em muitas áreas distintas para promover um melhor entendimento e para fornecer uma estrutura reutilizável – por meio de um modelo – para o desenvolvimento de produtos (Apfelbaum & Doyle, 1997). Modelos são mais simples do que o sistema que eles descrevem e, por isso, podem ajudar no entendimento de um sistema e na predição de seu comportamento (Robinson, 1999a, 1999b).

No processo de engenharia de software, modelos já vêm sendo utilizados na descrição do comportamento de sistemas. Os requisitos do sistema são primeiramente modelados durante a fase de análise de sistemas e esses modelos são usados nas fases posteriores de projeto e de implementação. Recentemente, com a adoção generalizada de metodologias orientadas a objetos, a modelagem tornou-se uma área de pesquisa na engenharia de software que tem desfrutado de grande popularidade no auxílio às atividades de análise e de projeto (Pressman, 1992; Apfelbaum & Doyle, 1997; Gronau et al., 2000).

Os modelos devem se concentrar no que o sistema deve fazer e não em como ele o faz. Segundo Pressman (1998), um modelo precisa descrever:

- a informação que o sistema modelado transforma;
- as funções (ou sub-funções) que devem ser utilizadas para que as transformações de informação ocorram; e
- o comportamento do sistema quando as transformações de informação estão se desenvolvendo.

De acordo com Apfelbaum e Doyle (1997), o desenvolvimento de especificações na forma de modelos, mesmo se feito em fase avançada no processo de software, é um meio muito efetivo de:

- descobrir defeitos no sistema, já que muitos se tornam visíveis simplesmente pela própria modelagem;
- definir rapidamente a base para os cenários de uso do sistema; e
- preservar esse investimento para versões futuras ou sistemas similares.

A modelagem é um meio muito econômico para se capturar o conhecimento sobre um sistema e depois reutilizá-lo à medida que o sistema cresce. A informação capturada e modelada também pode ser muito útil à atividade de teste de software, além de às outras fases do ciclo de vida. A aplicação de modelos na atividade de teste de software tem sido explorada no meio acadêmico, com o desenvolvimento de técnicas e, geralmente, a realização de experimentos práticos (Borrione et al., 1998; Dalal et al., 1998, 1999; El-far, 1999; Martins et al, 2000; Offutt, 2000). Entretanto, a maioria das empresas acaba não fazendo uso dessa tecnologia na realização de seus testes (Apfelbaum & Doyle, 1997).

Um benefício em especificar os requisitos do sistema na forma de um modelo comportamental e usar esse modelo no teste de software é que os casos de teste podem ser criados no início do processo de desenvolvimento do software e estar prontos para execução antes que o programa esteja completamente implementado. Adicionalmente, o testador geralmente encontrará inconsistências e ambigüidades nos modelos quando os testes forem gerados, permitindo que a especificação seja melhorada antes do programa ser escrito (Offutt et al., 2000).

Existe uma grande variedade de técnicas de modelagem distintas que podem ser usadas no desenvolvimento de modelos de software. Essas técnicas possuem diferenças entre si, tais como o grau de formalidade e os mecanismos de modelagem oferecidos por elas. Assim, cada uma dessas técnicas pode oferecer certas vantagens e desvantagens, dependendo do enfoque que se deseja e de qual fase do processo de engenharia de software o modelo será desenvolvido e em quais fases ele será utilizado (Pressman, 1992).

3.2. Características do Teste Baseado em Modelos

A principal premissa por trás do TBM é a criação de um modelo – uma representação do comportamento do software a ser testado. Esse modelo deve descrever quais ações são possíveis e quais saídas são esperadas. A partir dessa descrição, podem ser aplicados critérios de seleção de casos de teste, os quais devem ser executados e avaliados.

Embora essa abordagem de teste não seja a solução para todos os problemas da área, ela oferece uma promessa considerável na redução do custo da geração de teste, no aumento da eficiência dos testes e na redução do ciclo de teste. Uma vez que o investimento inicial tenha sido feito para se criar um modelo, pequenas alterações podem ser facilmente realizadas. Assim, esse tipo de teste pode ser especialmente eficaz para os softwares que são alterados freqüentemente, pois os testadores podem atualizar o modelo comportamental e rapidamente gerar novamente um conjunto de casos de teste, evitando a edição tediosa e sujeita a defeitos de um novo conjunto de testes derivados manualmente (Dalal et al., 1999).

O TBM permite que grandes quantidades de casos de teste sejam geradas a partir de uma descrição comportamental do software sob teste. Dado o mesmo modelo, muitos tipos de cenários podem ser exercitados e grandes áreas da aplicação sob teste podem ser cobertas, levando a um processo de teste mais eficiente e mais efetivo. A escolha de quais cenários

devem ser exercitados pode ser feita durante a execução dos testes, dependendo do objetivo de cada fase de teste. Assim, esta técnica de teste não apresenta o problema de os testes se tornarem menos eficientes com o decorrer do tempo dado que os defeitos cuja presença eles são capazes de detectar já foram consertados (Robinson & Rosaria, 2000).

Apesar de o TBM ser uma técnica de teste funcional e ter sido desenvolvido principalmente para ser aplicado aos testes de sistema, Dalal et al. (1999) mostram que essa técnica pode ser aplicada a qualquer fase de teste e não somente às últimas fases do teste de software. Assim, ela pode ser aplicada aos testes de unidade, de integração, de validação e de sistema, desde que os modelos represetem o comportamento do software no respectivo nível em que se deseja realizar o teste.

3.3. Fases do Teste Baseado em Modelos

Como ocorre na maioria das técnicas de teste, o TBM é executado em várias fases; algumas delas são similares às fases de execução de outras técnicas de teste. De um modo geral, essa técnica envolve quatro fases, as quais estão representadas na Figura 3.1.

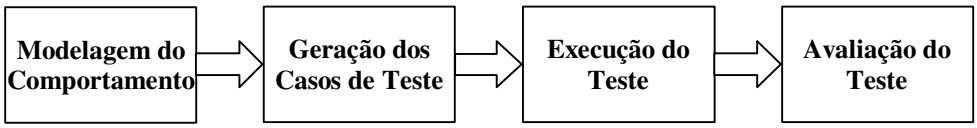

Figura 3.1 – Fases do Teste Baseado em Modelos

As quatro fases são descritas resumidamente a seguir. Na seqüência, cada uma das fases do TBM é descrita de forma mais detalhada.

- **Modelagem do comportamento do sistema:** um modelo que descreve o comportamento do sistema a ser testado é desenvolvido. Existem várias técnicas de modelagem – também chamadas de linguagens de modelagem – que podem ser utilizadas para esse propósito, as quais são descritas na próxima seção;
- **Geração dos casos de teste:** o modelo comportamental é utilizado para a geração dos casos de teste. Existem vários algoritmos que podem ser utilizados para esse propósito, os quais também são descritos posteriormente nesta seção;
- **Execução do teste:** os casos de teste são aplicados ao software sob teste;

- **Avaliação do teste:** os resultados obtidos da execução do teste são comparados com os resultados esperados.

3.3.1. Modelagem do Comportamento do Sistema

O primeiro passo no TBM é o desenvolvimento de um modelo comportamental do sistema a ser testado usando alguma técnica de modelagem. A modelagem é a fase fundamental dessa técnica de teste, já que as demais fases (geração de casos de teste, execução e avaliação dos testes) dependem da precisão do modelo criado (El-Far, 1999).

De acordo com Robinson & Rosaria (2000), o desenvolvimento de um modelo é um processo incremental que requer que os responsáveis pela modelagem levem em consideração, simultaneamente, muitos aspectos do sistema a ser testado. Como primeira tarefa na modelagem do comportamento do sistema, os responsáveis pela modelagem devem analisar a especificação do software e executar a aplicação desenvolvida para obter um entendimento geral da funcionalidade do sistema.

O modelo comportamental do sistema deve ser elaborado com base nas especificações do software. Dalal et al. (1999) mostram que o ideal seria que fosse utilizado o mesmo modelo do sistema para todas as fases do processo de engenharia de software, incluindo a fase de teste. Entretanto, conciliar os objetivos e as necessidades de cada fase do processo geralmente não é uma tarefa trivial.

Existem várias técnicas de modelagem que podem ser usadas para se construir os modelos comportamentais do software. Para cada uma dessas técnicas pode haver uma técnica específica de teste dentro do TBM. Dentre as técnicas de modelagem, uma das mais utilizadas são as Máquinas de Estados Finitos, cuja aplicação no TBM originou o Teste Baseado em Máquinas de Estados Finitos (Chow, 1978; Fujiwara et al., 1991; Bourhfir et al., 1996; Offutt et al., 2000).

3.3.2. Geração dos Casos de Teste

Subseqüentemente à construção de um modelo comportamental do sistema, a questão da geração dos casos de teste deve ser levada em consideração. A partir do modelo elaborado,

devem ser identificados os cenários de uso que devem ser exercitados e os respectivos dados de entrada e resultados esperados para cada cenário, formando assim os casos de teste. Devido à grande complexidade dos sistemas atuais, é impossível executar todos os casos de teste que podem ser gerados a partir de um modelo do sistema. Portanto, devem ser aplicados critérios de teste que possibilitem a seleção de um conjunto de casos de teste que possuam alta probabilidade de revelar os defeitos existentes no produto mas cuja aplicação seja viável.

Da mesma forma como existem várias técnicas de modelagem que podem ser usadas para se criar um modelo, também existem vários critérios que podem ser usados para selecionar os casos de teste a partir de um modelo. A escolha de quais critérios utilizar também é uma decisão muito importante, pois um bom resultado na revelação de defeitos existentes no produto depende dos critérios utilizados (El-Far, 1999).

Os critérios escolhidos para a seleção de casos de teste podem ser usados também como critérios de adequação de casos de teste. Ou seja, eles podem ser utilizados para verificar qual parte do modelo já foi coberta por meio da execução de um determinado conjunto de casos de teste.

Em se tratando de TBM, os casos de teste não representam a abordagem simples em que apenas um valor de entrada (ou um conjunto de valores) deve ser aplicado ao programa sob teste e apenas um valor (ou um conjunto de valores) é esperado como saída. Ao invés disso, os casos de teste devem ser descritos em função dos elementos existentes no próprio modelo. Por exemplo, para modelos descritos por meio de MEFs, os casos de teste podem ser descritos como função dos estados e das transições de estado.

A geração de um conjunto de casos de teste não é uma tarefa trivial e, por isso, é bastante recomendável que essa fase do TBM seja automatizada. Uma ferramenta de teste pode ser utilizada para, dado um modelo comportamental do software, selecionar automaticamente os casos de teste de acordo com algum critério de teste escolhido. A automação da geração dos casos de teste pode ser realizada de forma parcial – com a interação do projetista de teste – ou completa, o que depende de fatores como: a técnica e a ferramenta utilizadas, a complexidade do software a ser testado e do modelo que descreve seu comportamento e o nível de abrangência desejada para o teste.

3.3.3. Execução do Teste

Nesta fase, os casos de teste selecionados na fase anterior devem ser aplicados ao software sob teste. A execução dos casos de teste envolve determinar como simular as ações do usuário para que o software comporte-se como se estivesse em seu ambiente normal de execução. Essa tarefa também pode ser feita manualmente ou automatizada. A execução automática dos casos de teste tem se tornado uma tarefa simples, visto que existem numerosas ferramentas comerciais dedicadas a simular as entradas para o software.

A automação dessa fase pode ser realizada por um executor de teste (chamado em inglês de *test driver* ou *test harness*), que é um programa que aplica os casos de teste (uma seqüência de entradas) ao sistema sob teste. Para casos de teste gerados a partir de modelos descritos por MEFs, o *test driver* deve obter uma seqüência de transições a partir do critério escolhido, ler a seqüência de teste, entrada por entrada, determinar qual transição deve ser aplicada para cada entrada e fazer com que a aplicação execute aquela transição, levando a um novo estado.

3.3.4. Avaliação do Teste

A avaliação do teste envolve a comparação dos resultados obtidos durante a execução do teste contra os resultados esperados, obtidos a partir de algum tipo de especificação. Geralmente, essa fase de avaliação do teste é realizada pelo mesmo módulo responsável pela execução dos casos de teste, isso é, o *test driver*.

Fundamental à realização dessa fase é a existência de algum tipo de oráculo. Um oráculo é um mecanismo para determinar se a aplicação comportou-se corretamente ou não. O oráculo é uma entidade independente que determina se o resultado observado da execução do teste satisfaz as expectativas, isso é, se as saídas corretas foram produzidas, ou se as seqüências de controle corretas foram seguidas (El-Far, 1999).

Geralmente, o desenvolvimento de um oráculo não é uma tarefa trivial e pode ser tão complexo quanto o desenvolvimento da própria aplicação sob teste, representando um dos maiores custos da atividade de teste. Um dos grandes benefícios do TBM é a habilidade de se utilizar o modelo comportamental do sistema para a verificação das saídas obtidas (Offutt et al., 2000). Em se tratando de modelos especificados por MEFs, por exemplo, é possível saber,

a partir do modelo, quais transições deveriam estar disponíveis a partir de cada estado e a saída esperada de cada transição podendo-se saber qual o estado final esperado (Robinson, 2000).

3.4. Técnicas de Modelagem para o Teste Baseado em Modelos

Nesta seção, são apresentadas as principais técnicas de modelagem que podem ser utilizadas com o TBM, enfocando a técnica de MEFs. Para essa técnica, são apresentados também alguns dos principais critérios de teste associados a ela. Outras técnicas apresentadas são Statecharts e Redes de Petri.

3.4.1. Máquinas de Estados Finitos

As Máquinas de Estados Finitos são uma técnica formal que tem se mostrado bastante útil para tratar do comportamento de sistemas e para ser utilizada no teste de software (Apfelbaum & Doyle, 1997; Robinson & Rosaria, 2000). Esta técnica é especialmente útil para modelar o comportamento de sistemas reativos, os quais são essencialmente dirigidos a eventos e dominados por controle. Além disso, as MEFs possuem uma gama de aplicação bastante grande e genérica, podendo ser utilizadas na modelagem de vários tipos de sistema.

Um modelo representado por uma MEF consiste basicamente de um conjunto de estados e um conjunto de transições de estados. Dado um estado atual e uma entrada, podem-se determinar o próximo estado no modelo e uma saída. Os elementos das MEFs – estados e transições – podem ser representados de três formas:

- Representação textual, em que os elementos pertencentes à MEF são descritos textualmente, seguindo algumas regras sintáticas (vide exemplo na Figura 3.2);
- Representação gráfica – chamada de grafo de estados, útil para a visualização dos elementos pertencentes à MEF (vide exemplo na Figura 3.3); e
- Representação tabular – chamada de tabela de transições de estados, úteis para o armazenamento e manipulação computacional das informações contidas na MEF (vide exemplo na Figura 3.4).

As MEFs podem ser incrementadas com o uso de recursos adicionais de modelagem como o uso de condicionais ou de predicados, utilizados para tornar as transições dependentes de variáveis ou do contexto atual do sistema.

```
Estados = {A, B, C};
Entradas = {Entrada 1, Entrada 2}
Saídas = {Saída 1, Saída 2}
Transição1(A, Entrada 1) = (B, Saída 1)
Transição2(B, Entrada 2) = (C, Saída 2)
Transição3(C, Entrada 2) = (A, Saída 1)
```

Figura 3.2 – Exemplo de Representação Textual de uma MEF

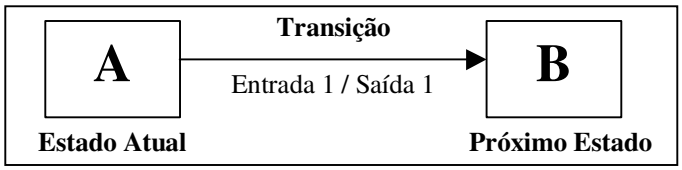

Figura 3.3 – Exemplo de Representação Gráfica de uma MEF

	Entrada 1	**Entrada 2**
A	B / Saída 1	-
B	-	C / Saída 2
C	-	A / Saída 1

Figura 3.4 – Exemplo de Representação Tabular de uma MEF

O modelo de MEFs apresentado nas figuras anteriores é conhecido como modelo de Mealy. Neste modelo, as saídas estão associadas a transições e os estados são passivos. Existe também o modelo de Moore, no qual as saídas estão associadas a estados, os quais são, portanto, ativos. Neste capítulo é tratado apenas o modelo de Mealy, mais amplamente utilizado no TBM.

Devido à grande gama de aplicação das MEFs como técnica de modelagem, existe para o TBM uma técnica de teste específica chamada normalmente de Teste Baseado em Máquinas de Estados Finitos. Outros nomes que essa técnica recebe são Teste Baseado em Especificações Baseadas em Estados ou simplesmente Teste Baseado em Estados (Offutt et al., 2000), Teste de Máquinas de Estados Finitos (Lee & Yannakakis, 1995), entre outros nomes similares.

Para modelos descritos por meio de MEFs, no caso geral, os dados de entrada de um caso de teste podem ser descritos como uma seqüência de ações que levam a uma seqüência de estados e a saída esperada do caso de teste pode ser descrita como as saídas esperadas para cada transição e um estado final.

Os critérios de teste baseados em MEFs devem definir conjuntos de elementos requeridos do modelo, tais como estados e transições, que devem ser exercitados durante a execução do teste. Como as MEFs podem ser consideradas essencialmente grafos, muitos algoritmos da teoria de grafos podem ser utilizados como formas de geração ou avaliação de elementos requeridos pelos critérios de teste. A maioria desses algoritmos está relacionada ao conceito de caminho. Um caminho é uma seqüência de transições através dos elementos do modelo (estados).

Como acontece com a maioria dos critérios de técnicas funcionais, uma grande parte dos critérios de teste Baseado em MEFs está descrita de forma pouco rigorosa na literatura (Offutt et al., 2000). A seguir encontra-se uma breve descrição de alguns desses critérios, os quais são bastante utilizados por testadores de empresas produtoras de software.

Seleção_aleatória: essa é uma das escolhas mais populares para esse propósito, que permite que os caminhos da MEF sejam selecionados aleatoriamente, selecionando qualquer transição disponível a partir de um estado. Esse critério de teste, por gerar infinitos casos de teste, precisa de um critério de parada, o qual geralmente é dependente do tempo. Dado tempo suficiente, esses percursos aleatórios podem cobrir uma boa parte da aplicação. A natureza aleatória dessa escolha significa que elas tendem a produzir combinações não usuais que testadores humanos acabariam não considerando para o teste (Robinson, 1999a, 1999b, 2000).

Todos_os_estados: esse é um dos critérios mais simples existentes, que define que todos os estados dentro da MEF devem ser exercitados. Similarmente ao critério Todos_os_nós do teste estrutural, a satisfação desse critério oferece uma cobertura muito pequena da MEF (Robinson, 1999b).

Todas_as_transições: esse também é um critério bastante simples, que define que todas as transições dentro da MEF devem ser exercitadas. Sua satisfação também oferece uma cobertura pequena da MEF (Robinson, 1999b; Offutt et al., 2000). Um algoritmo que pode ser

utilizado como forma de geração de elementos requeridos por esse critério é o Carteiro Chinês, que gera um caminho que exercita todas as transições na MEF com o menor número de transições possível (Robinson, 1999a, 2000; Robinson & Rosaria, 2000). Variações do Carteiro Chinês – como o Carteiro Chinês Escolhendo Estados e o Carteiro Chinês Capacitado – também podem ser utilizadas com este critério (Robinson & Rosaria, 2000).

Todas_as_combinações: esse é o critério cuja satisfação oferece a maior cobertura de todos os critérios, já que todas as combinações de transições dentro da MEF devem ser executadas. Entretanto, similarmente ao critério Todos_os_caminhos do teste estrutural, dependendo do tamanho da MEF, esse critério pode ser considerado inviável. Em se tratando de teste de software baseado em estados, essa abordagem é também chamada de *switch cover* (Bourhfir et al., 1996; Robinson, 1999b; Offutt et al., 2000).

Além dos critérios descritos acima, existem critérios que foram definidos formalmente com base em trabalhos de pesquisa realizados especificamente para o Teste Baseado em MEFs. De acordo com Fujiwara et al. (1991), os mais conhecidos desses critérios formais são: método *Distinguishing Sequence* – DS (Gill, 1962; Gonenc, 1970; Kohavi, 1978); método W (Chow, 1978); *Transition Tour* (Naito & Tsunoyama, 1981); método *Unique-Input-Output* – UIO (Sabnani & Dahbura, 1988); e método Wp (Fujiwara et al., 1991). Embora a utilização desses critérios ofereça uma maior garantia de que o software possui alta qualidade e alta confiabilidade por meio da realização dos testes, eles são critérios mais complexos, apresentando conseqüentemente um maior custo de aplicação.

Embora as MEFs sejam uma ferramenta excelente para o entendimento e o teste de software, o modelo pode se tornar muito complexo e de difícil manutenção para sistemas grandes. Para solucionar esse problema, existem extensões dessa técnica que suportam modelos hierárquicos, nos quais um estado pode ser substituído por uma chamada a um outro modelo que define o comportamento dentro do estado. Modelos hierárquicos permitem que comportamentos complexos sejam decompostos em modelos de vários níveis.

Além da ausência de mecanismos para a modelagem de hierarquia, existem outras limitações das MEFs tradicionais que prejudicam o seu uso como técnica de modelagem no TBM. Exemplos de outras características que não podem ser modeladas pelas MEFs tradicionais são: concorrência, comunicação, processos distribuídos, paralelismo e fluxo de dados.

3.4.2. Statecharts

A técnica de Statecharts é uma extensão das MEFs tradicionais que inclui, além de estados e transições, três elementos adicionais – hierarquia de estados, concorrência entre estados e comunicação entre estados. Essa técnica foi projetada primariamente para modelar o comportamento de sistemas reativos, tais como os usados em aviação e redes de comunicação (Harel, 1987a, 1987b; Bourhfir et al., 1996).

Os elementos básicos modelados por Statecharts são:

- **Estado:** situação em que um sistema se encontra em um determinado instante do tempo;
- **Transição:** relacionamento entre dois estados que indica uma seqüência entre eles;
- **Evento:** acontecimento que ao ser percebido pelo sistema provoca uma transição de estados;
- **Condição:** predicado opcional associado a um evento que habilita o sistema a efetuar uma transição de estados;
- **Ação:** acontecimento gerado para o ambiente externo como resultado de uma transição de estados.

O sistema só passa de um estado para outro caso ocorra um evento e se, e somente se, a condição associada for verdadeira no momento do evento. Caso a transição de estados ocorra, uma ação pode ser executada como resposta a transição de estados efetuada.

Existem três tipos de estados: estados atômicos, estados AND e estados OR. Os estados atômicos não se decompõem em sub-estados, enquanto os estados AND e OR se decompõem. É com o uso dos estados AND e OR que pode ser realizada uma decomposição hierárquica de estados do modelo, isto é, os estados podem ser agrupados em outros estados formando níveis. Portanto, um estado s_i é um sub-estado do estado S quando este for ancestral direto ou indireto de s_i ao longo da hierarquia. Todo Statechart possui um estado que não tem qualquer estado-pai, denominado raiz do Statechart.

O estado que se encontra o sistema modelado pelo Statechart representa uma situação do sistema em um determinado instante considerado. Se o sistema estiver em um estado OR,

então ele está exclusivamente em apenas um de seus sub-estados; se o sistema estiver em um estado AND, então ele está simultaneamente em todos seus sub-estados.

Por meio do uso da decomposição de estados em sub-estados, os Statecharts permitem que seja evitada a explosão exponencial de estados que pode ocorrer nas MEFs tradicionais, as quais são essencialmente de apenas um nível e seqüenciais. Quando a especificação em Statechart torna-se muito complexa, o que acontece com modelagens de problemas reais, pode-se decompor o Statechart em diagramas separados. Isto permite a visualização em níveis de abstração diferentes, conservando a mesma hierarquia.

Os eventos, que podem ser externos ou internos, permitem que as especificações em Statecharts descrevam interações com o ambiente externo. Fisicamente, os eventos correspondem a ações do usuário sobre dispositivos de entrada ou são gerados pela aplicação como conseqüência de uma transição.

Na Figura 3.5 é ilustrado um Statechart que possui cinco estados (*S*, *A*, *B*, *C* e *D*) e três transições (t_1, t_2 e t_3). Os estados são representados por quadrados com os cantos arredondados e as transições por setas unidirecionais que ligam os estados de origem aos de destino. Graficamente, o estado raiz (estado *S*, no exemplo), os estados hachurados (estado *C*, no exemplo) e os que possuem borda pontilhada (estado *A*, no exemplo) representam os estados ativos, isto é, que fazem parte da configuração de estados na qual o sistema se encontra em um determinado momento. Neste exemplo, quando a transição *t3: f[c]/a* for disparada (ou seja, quando a expressão condicional *c* for verdadeira e o evento *f* ocorrer), a ação *a* será executada e o estado atômico *C* deixará de estar ativo e o estado atômico *D* será ativado. Deste modo, o estado *D* estará hachurado, e não mais o estado *C*.

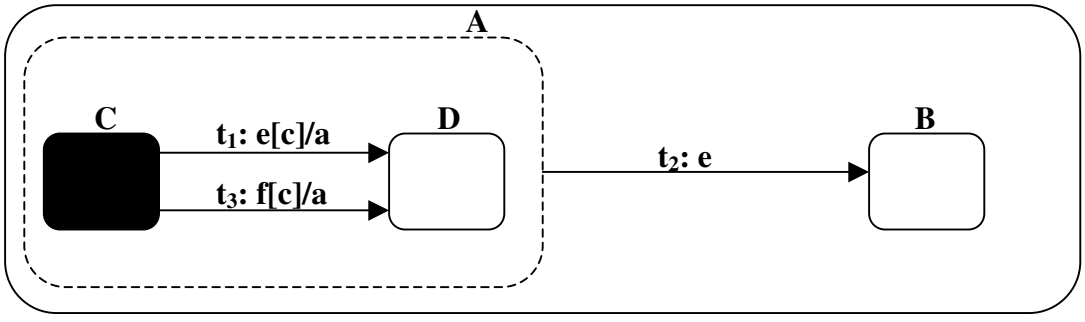

Figura 3.5 – Exemplo de um Statechart

3.4.3. Redes de Petri

Redes de Petri são uma técnica de modelagem gráfica e algébrica que apresenta um bom nível de abstração em comparação com outras técnicas gráficas. Elas são uma técnica promissora para a descrição e o estudo de vários tipos de sistemas de processamento de informação que não podem ser bem descritos usando outras técnicas de modelagem. Os tipos de processos que podem ser modelados pelas Redes de Petri são: processos concorrentes, processos assíncronos, processos distribuídos, processos paralelos, processos não-determinísticos e processos estocásticos (Bourhfir et al., 1996).

Existem vários tipos de Redes de Petri, desde redes mais elementares – como as Redes de Petri Condição-Evento ou Redes de Petri Lugar-Transição até redes mais sofisticadas – como as Redes de Petri Coloridas. Nesta dissertação são apresentadas apenas o primeiro tipo.

A estrutura básica das Redes de Petri Condição-Evento se concentra em dois conceitos primitivos: eventos e condições. Eventos são ações que ocorrem no sistema e que são controladas pelo estado do sistema. Cada evento possui pré-condições que vão permitir sua ocorrência e pós-condições decorrentes desta, as quais são, por sua vez, pré-condições de outros eventos posteriores.

Uma Rede de Petri é vista como um tipo particular de grafo orientado que permite modelar as propriedades estáticas de um sistema a eventos discretos, constituído de dois tipos de nós: as transições (que correspondem aos eventos que caracterizam as mudanças de estado do sistema) e os lugares (que correspondem às condições que devem ser certificadas para os eventos acontecerem) interligados por arcos direcionados ponderados (Peterson, 1981).

Uma Rede de Petri é composta de quatro conjuntos de elementos: um conjunto de lugares P, um conjunto de transições T, uma aplicação de entrada I ou Pré, e uma aplicação de saída O ou Pós. As funções de entrada e saída relacionam transições e lugares. Sendo assim a estrutura das Redes de Petri é definida por seus lugares, suas transições, a função de entrada I (ou Pré), e a função de saída O (ou Pós).

A Figura 3.6 apresenta um exemplo gráfico de uma Rede de Petri que modela duas atividades paralelas. No modelo, as transições (ou eventos) são representadas por barras (T_1, T_2, T_3, T_4) e

os lugares (ou condições) são representados por círculos (P_1, P_2, P_3, P_4, P_5). Os eventos são relacionados às condições por meio dos arcos direcionados que interligam as transições a lugares e vice-versa.

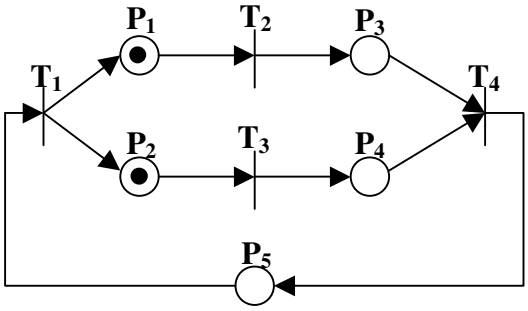

Figura 3.6 – Exemplo de uma Rede de Petri

As Redes de Petri estão entre as melhores técnicas para a descrição do comportamento de sistemas reativos. Elas são uma técnica que permite a descrição precisa de sistemas e que possui um enorme campo de pesquisa, acumulado por aproximadamente trinta anos. Elas são fortemente dirigidas a eventos e possuem grande facilidade de modelagem de concorrência.

3.4.4. Outras Técnicas

Além das técnicas descritas nos itens anteriores, existem outras técnicas de modelagem que têm sido empregadas no TBM, tais como:

- **SDL –** uma linguagem de descrição e especificação formal, recomendada pela ITU-T (1998), para a descrição e especificação do comportamento de sistemas de telecomunicações. Possui reconhecimento como padrão internacional. Permite modelar concorrência e não-determinismo e a linguagem foi estendida para incluir também certas características de modelos orientados a objeto;
- **Linguagem Z** – uma técnica formal, baseada em lógica de primeira ordem e teoria dos conjuntos, que tem sido usada no setor de telecomunicações (Spiveu, 1992);
- **Estelle** – uma linguagem de modelagem desenvolvida pela ISO para a especificação de serviços e protocolos de comunicação (Budkowski & Dembinski, 1987). Essa técnica de modelagem formal é baseada em MEFs e na linguagem de programação PASCAL. Estelle tem sido usada em técnicas de teste envolvendo não

somente informações sobre o fluxo de controle, mas também sobre fluxo de dados (Ural & Yang, 1991; Miller & Paul, 1992);

- **Diagrama de Estados e Diagrama de Atividades da UML –** um diagrama de estados da UML é uma máquina de estados usada para mostrar todos os possíveis estados de uma classe e quais eventos causam a mudança de estados; enquanto que um diagrama de atividades da UML é um caso especial de um diagrama de estados no qual todos, ou a maioria dos estados, são estados de ação e no qual todas, ou a maioria das transições são ativados pelo processamento de ações nos estados de origem (UML, 2003).

3.5. Automação do Teste Baseado em Modelos

Uma importante característica do TBM é que ele possibilita a automação de grande parte da atividade de teste de software. Essa automação é possível devido ao suporte oferecido pelo modelo comportamental da aplicação, a partir do qual os casos de teste podem ser automaticamente gerados e executados, e os resultados avaliados para se obter os resultados do teste (Dalal et al., 1998; Robinson & Rosaria, 2000).

A automação das fases do TBM oferece a essa técnica uma grande agilidade à execução de testes. Dado um mesmo modelo comportamental do software sob teste e um *test driver*, grandes quantidades de casos de teste podem ser gerados e executados. Além disso, se o comportamento do software for modificado, basta que o modelo seja atualizado, para que novos conjuntos de casos de teste sejam gerados e executados (Dalal et al., 1998, 1999).

Além de agilidade, a automação oferece também grande flexibilidade à execução de testes. Por exemplo, a geração dos casos de teste pode ser direcionada para diferentes áreas de foco de teste, tais como teste de *stress*, teste de regressão e a verificação de que uma versão do software possui funcionalidades básicas. Além disso, entradas que levam a defeitos conhecidos podem ser temporariamente desabilitadas do modelo. Assim, quaisquer casos de teste baseados nesse modelo alterado evitarão os defeitos conhecidos e o código implementado para o teste (*scripts* de teste) não precisa ser alterado para isso (Robinson, 2000; Robinson & Rosaria, 2000).

As ferramentas utilizadas para a automação do TBM são, em geral, de dois tipos diferentes. O primeiro tipo são as ferramentas de teste genéricas, que podem ser utilizadas para o teste de qualquer sistema. O outro tipo são as ferramentas de teste específicas, que precisam ser desenvolvidas para o teste de cada sistema diferente. Dado esses dois tipos de ferramentas, pode-se utilizar uma estratégia de automação de teste que utilize os dois tipos diferentes de ferramentas para diferentes fases do TBM.

Algumas ferramentas têm sido desenvolvidas para a automação do TBM. Uma das ferramentas mais citadas na literatura é a TestMaster (produzida pela Teradyne, Inc.) (Berger et al., 1997; Clarke, 1998). Essa ferramenta oferece um sistema de geração de casos de teste, incluindo um conjunto de ferramentas visuais com suporte à entrada, análise, visualização, especificação, depuração, checagem de consistência e manutenção de modelos.

Exemplos de outras ferramentas são: Visual Test (produzida pela *Rational* Software) (Robinson, 1999a, 2000; Robinson & Rosaria, 2000), MGASet-Java (Candolo et al., 2001), ConData (Martins et al., 2000) e TAG (Tan et al., 1995).

3.6. Considerações Finais

O TBM é uma técnica de teste funcional que tem se mostrado como uma forma eficiente e adaptável de teste de software. Essa técnica de teste é suportada pela criação e uso de um modelo que descreve o comportamento do sistema sob teste, a partir do qual os casos de teste podem ser gerados e executados e os resultados dessa execução avaliados.

O TBM é realizado geralmente em quatro fases: 1) modelagem do comportamento do sistema; 2) geração dos casos de teste; 3) execução do teste; e 4) avaliação do teste. A fase fundamental do TBM é a modelagem do comportamento do sistema, pois é da precisão do modelo criado que dependem as demais fases. A técnica de modelagem mais explorada pelas pesquisas existentes na área são as MEFs, para as quais já existem critérios de teste definidos. Entretanto, a maior parte dos critérios que são utilizados na indústria de software são descritos de forma pouco rigorosa.

Embora esta técnica de teste ainda não tenha sido amplamente explorada, ela possui um grande potencial para ser utilizada no teste de grandes sistemas, por meio da automação de

atividades de teste de software. Por meio dessa automação, grandes quantidades de casos de teste podem ser gerados a partir de um modelo comportamental do software, usando diferentes critérios. Depois de gerados, um *test driver* pode então executar os casos de teste no sistema sob teste. Assim, muitas áreas de foco de teste podem ser implementadas e diferentes níveis de cobertura do modelo podem ser atingidos por meio do uso do mesmo modelo e do mesmo *test driver*.

Vários estudos de casos relacionados à aplicação da abordagem de TBM em vários tipos de softwares são apresentados na literatura (Apfelbaum & Doyle, 1997; Clarke, 1998; Dalal et al., 1998, 1999; Gronau et al., 2000). Em todos eles, as execuções dos testes apresentaram ótimos resultados na detecção da presença de defeitos, os quais não foram ou dificilmente seriam detectados por abordagens de teste tradicionais.

Entretanto, para tornar essa abordagem uma solução prática, fatores econômicos devem ser levados em consideração. De acordo com Apfelbaum e Doyle (1997), uma nova técnica, para ser adotada e utilizada, deve permitir a redução de custo. As métricas tradicionais usadas para justificar a compra de softwares ou para garantir uma mudança em um processo já existente incluem: baixo custo, aumento da qualidade e redução do tempo de entrega do produto para o mercado. Estudos mostram que o TBM é uma técnica de teste que tem habilidade em fornecer melhorias dramáticas nos três campos dessas métricas. Assim, o maior desafio para uma aceitação mais ampla dessa técnica é a educação (Apfelbaum & Doyle, 1997).

Quanto às desvantagens do TBM, poucas são apresentadas na literatura. Uma desvantagem mencionada é que uma certa quantidade de esforço é necessária para desenvolver o modelo inicial para sistemas relativamente complexos. Outra desvantagem é relacionada às técnicas de testes em que um *test driver* deve ser desenvolvido para cada aplicação sob teste, já que isso requer que os testadores sejam capazes de programar, como no caso do uso da ferramenta Visual Test (Robinson & Rosaria, 2000). Além disso, ainda existe o problema de que muitas vezes os requisitos do software não estão completos, tornando difícil a geração dos modelos que traduzem a realidade do software implementado e que deve ser testado.

O próximo capítulo apresenta a definição de nova técnica de modelagem – que é uma extensão das MEFs tradicionais – e a definição de um conjunto de critérios de teste funcional baseados nesta técnica de modelagem.

Capítulo 4

Critérios de Teste Funcional Baseados em MEFs Estendidas

Embora Máquinas de Estados Finitos (MEFs) sejam uma das técnicas mais utilizadas para a modelagem comportamental de software no Teste Baseado em Modelos (TBM), existem limitações quanto aos mecanismos de modelagem disponíveis. Uma das principais limitações é que elas não possuem recursos para a modelagem de fluxo de dados. Deste modo, as informações sobre o fluxo de dados do sistema alvo não podem ser levadas em consideração para o estabelecimento de requisitos de teste.

Visando eliminar essa limitação do uso das MEFs, é definida uma extensão desta técnica de modelagem que considera, além do fluxo de controle, o fluxo de dados. Com base nas MEFs Estendidas (MEFEs), é definido um conjunto de critérios de teste funcional, com o objetivo de fornecer ao TBM um amplo conjunto de critérios de teste definidos de forma rigorosa, que oferecem vários níveis de cobertura do modelo comportamental. Os critérios de teste definidos neste trabalho são critérios de teste funcional, já que eles utilizam informações sobre o comportamento do software modelado por meio de MEFEs e não informações sobre a estrutura interna do software implementado.

4.1. Máquinas de Estados Finitos

Basicamente, uma MEF é formada por um conjunto de estados e um conjunto de transições de estados. Além disso, uma MEF apresenta outros elementos, tais como: um estado inicial, um conjunto de entradas e um conjunto de saídas. Com o objetivo de facilitar o entendimento da

extensão das MEFs apresentada posteriormente, nesta seção é apresentada a definição de uma MEF tradicional baseada no Modelo de Mealy.

Uma Máquina de Estados Finitos M é definida como uma 7-tupla $M = < S, s_0, I, O, D, \delta, \lambda >$, onde:

- S é um conjunto finito e não vazio de estados;
- $s_0 \in S$ é o estado inicial;
- I é um conjunto finito de entradas, que pode incluir a entrada nula;
- O é um conjunto finito de saídas, que pode incluir a saída nula;
- $D \subseteq S \times I$ é o domínio da especificação;
- $\delta: D \rightarrow S$ é a função de transições de estados;
- $\lambda: D \rightarrow O$ é a função de saída.

Uma transição de estados da MEF M é representada por $\delta(s_i, i) = s_j$ e a saída obtida pela realização dessa transição de estados é representada por $\lambda(s_i, i) = o$. Isso significa que, quando a MEF M está no estado $s_i \in S$, ao receber uma entrada $i \in I$, ela se move para o estado $s_j \in S$, produzindo uma saída $o \in O$; s_i é o estado atual ou estado corrente e s_j é o próximo estado.

Uma MEF M pode ser determinística ou não determinística. Se a quantidade de transições de um estado $s \in S$ sobre um entrada $i \in I$ é igual a um, então a máquina é determinística. Ou seja, uma MEF é determinística se, a partir de um estado s e sobre uma entrada i, a máquina segue apenas uma única transição até um próximo estado. Caso contrário, a máquina é não determinística, ou seja, não é possível definir previamente qual transição deve ser seguida e, conseqüentemente, podem ser produzidas saídas diferentes.

Além da representação textual de uma MEF, a qual pode ser feita simplesmente por meio da descrição textual de cada um de seus elementos, conforme apresentados em sua definição, existem dois outros tipos de representação que podem ser usados para descrever uma MEF: uma forma gráfica – chamada de diagrama de transições de estados, e uma forma tabular – chamada de tabela de transições de estados.

4.1.1. Diagrama de Transições de Estados

Os diagramas de transições de estados, também chamados simplesmente de grafos de estados, são muito úteis para a visualização das informações contidas nas MEFs. Uma representação desse tipo de diagrama é apresentada na Figura 4.1. Uma MEF M pode ser representada por um grafo dirigido G = (V, E), onde:

- V é um conjunto de vértices, que representa o conjunto de estados S da MEF M. Cada $v \in$ V representa um estado $s \in$ S;
- E = {(v_i, v_j; *i*/o), v_i, $v_j \in$ V} é um conjunto de arcos dirigidos, que representa as transições da MEF M. Cada e = (v_i, v_j; *i*/*o*), representa uma transição de estados de v_i para v_j, com entrada *i* e saída *o*.

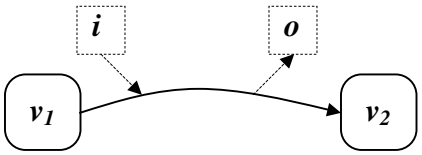

Figura 4.1 – Representação do Diagrama de Transições de Estados de uma MEF

4.1.2. Tabela de Transições de Estados

As tabelas de transições de estados são muito úteis para o armazenamento e manipulação computacional das informações das MEFs. Uma MEF M pode ser representada por uma tabela de transições de estados, conforme apresentado na Figura 4.2, onde cada linha representa um estado e cada coluna representa uma entrada. Para uma combinação de um estado atual $s_i \in$ S e uma entrada $i_j \in$ I, a posição correspondente da tabela especifica o próximo estado $s \in$ S e a saída $o \in$ O.

	i_1	i_2	...	i_m
s_1	s / o	s / o	...	s / o
s_2	s / o	s / o	...	s / o
...	...	...	...	...
s_n	s / o	s / o	...	s / o

Figura 4.2 – Tabela de Transições de Estados de uma MEF

4.2. Máquinas de Estados Finitos Estendidas

Embora as MEFs sejam uma técnica de aplicação simples e intuitiva, existem limitações quanto aos mecanismos de modelagem disponíveis. Uma destas limitações é que elas possibilitam apenas a modelagem do comportamento do software em relação a seu fluxo de controle, não possibilitando a modelagem de seu fluxo de dados.

Infelizmente, os critérios de teste existentes baseados apenas em análise de fluxo de controle do comportamento do software exigem apenas que elementos como estados, arcos e laços da MEF sejam exercitados (Offutt, 2000). Como a satisfação desses critérios é, na maioria dos casos, fácil de ser alcançada, a sua aplicação não leva à obtenção de boas indicações de confiabilidade e qualidade do software testado.

Por outro lado, no teste estrutural, já existem critérios de teste baseados tanto no fluxo de controle (Beizer, 1990; Pressman, 1992) quanto no fluxo de dados das estruturas internas do software (Rapps, 1985; Maldonado, 1991), as quais são representadas por meio de grafos de programa. Portanto, os critérios de teste estrutural oferecem vários graus de cobertura da estrutura interna do programa testado.

Esses critérios de teste estrutural podem ser utilizados como base para a definição de um conjunto de critérios de teste funcional que leve em consideração informações tanto de fluxo de controle como de fluxo de dados do comportamento do software. Para possibilitar a definição desses novos critérios, as MEFs tradicionais devem ser estendidas para oferecer suporte à modelagem do fluxo de dados. Além disso, as MEFs Estendidas devem possuir uma semelhança estrutural com grafos de programa para que seja possível a definição dos critérios de teste funcional com base nos critérios de teste estrutural.

Algumas extensões de MEFs têm sido propostas para oferecer suporte à modelagem de fluxo de dados (Lee & Yannakakis, 1995; Shehady & Siewiorek, 1997; Borrione et al., 1998; Bourhfir et al., 1996; Martins et al., 2000). Os modelos criados a partir dessas extensões possuem todas as informações relativas ao fluxo de dados nas transições de estado.

Por exemplo, as MEFs Estendidas apresentadas por Borrione et al. (1998) é definida como uma 6-tupla $M = <S, s_0, I, O, T, V >$, onde:

- S é um conjunto finito e não vazio de estados;
- $s_0 \in$ S é o estado inicial;
- I é um conjunto finito e não vazio de entradas, incluindo a entrada vazia;
- O é um conjunto finito e não vazio de saídas, incluindo a saída vazia;
- T é um conjunto finito e não vazio de transições;
- V é um conjunto finito de variáveis.

Cada elemento $t \in$ T é uma 5-tupla t = (estado_fonte, estado_destino, entrada, predicado, bloco_de_ações). Dos elementos de t, *estado_fonte* $\in$ S e *estado_destino* $\in$ S repreentam, respectivamente, o estado inicial e o estado finalda transição *t*; *entrada* $\in$ I; *predicado* é um predicado *Pascal-like* expresso em termos das variáveis de V, os parametros das entradas e algumas constantes; o *bloco_de_ações* é um bloco de computações que consiste de instruções de atribuição *Pascal-like* e instruções de saída.

Essas extensões já propostas – como a exemplificada anteriormente – são baseadas puramente no modelo de Mealy (descrita na Seção 3.4), em que os estados são elementos passivos dentro do modelo. Assim, a representação de todas as informações relativas a ocorrências de variáveis – entrada de dados, atribuição, saída de dados e predicado – são agrupadas dentro das transições. Desta forma, esses modelos não possuem a similaridade estrutural com grafos de programa – em que as informações de fluxo de dados são distribuídas entre os nós e os arcos do grafo – desejada para a definição dos critérios de teste funcional com base nos critérios de teste estrutural.

A extensão de MEFs proposta neste trabalho representa informações sobre fluxo de dados sem estarem associadas necessariamente às transições. Algumas ocorrências de variáveis, como atribuição, entrada de dados e saída de dados, são representadas dentro dos estados da MEFs, enquanto apenas as ocorrências de variáveis em predicados são representadas nas transições de estados. Essa extensão das MEFs é baseada em ambos os modelos de Mealy e de Moore, visto que os estados são dinâmicos – como nos modelos de Moore, mas as saídas associadas a transições são mantidas – como nos modelos de Mealy.

As MEFEs porpostas neste trabalho foram definidas buscando uma semelhança estrutural com grafos de programa, para que os critérios de teste funcional pudessem ser definidos diretamente a partir de critérios de teste estrutural já existentes. Apesar da semelhança

estrutural, os grafos de programa e as MEFEs não compartilham do mesmo significado semântico. A seguir apresenta-se a definição de uma MEFE para a qual foi definido o conjunto de critérios de teste funcional objeto deste trabalho.

Uma MEFE M é definida como uma 11-tupla $M = < S, s_0, I, O, V, P, N, \rho, D, \delta, \lambda, >$, onde:

- S é um conjunto finito e não vazio de estados;
- $s_0 \in S$ é o estado inicial;
- I é um conjunto finito de entradas, que pode incluir a entrada nula;
- O é um conjunto finito de saídas, que pode incluir a saída nula;
- V é um conjunto finito de variáveis;
- P é um conjunto finito de predicados envolvendo variáves de V;
- N é um conjunto finito de instruções envolvendo variáveis de V;
- ρ: SxVxN é a relação de ocorrênias de variáveis;
- $D \subseteq S \times I \times P$ é o domínio da especificação;
- $\delta: D \rightarrow S$ é a função de transições de estados;
- $\lambda: D \rightarrow O$ é a função de saída.

Cada predicado $p \in P$ pode ser formado por variáveis de V, constantes e operadores. Os operadores possíveis são os operadores lógicos E, OU e NÃO (representado por ~), e os operadores aritiméticos +, -, *, / e =.

Cada instrução $n \in N$ pode ser formado por variáveis de V, constantes, operadores, comandos e chamadas de função. Os operadores possíveis são os operadores aritiméticos +, -, * e /, e o operador de atribuição $\leftarrow$. Os comandos possíveis são o comando de entrada de dados READ e o comando de saída de dados WRITE. As chamadas de função são representadas por $f_i(v_1, v_2, ..., v_n)$, onde $i, n \in (1, 2, 3, ...)$ e $v \in V$.

Uma chamada de função representa uma função qualquer que usa os valores das variáveis que lhe são passadas como parâmetro para fazer algum tipo de cálculo e retornar um valor. Embora, para a utilização da MEFE nos critérios de teste funcional definidos, não seja necessário descrever as funções utilizadas no modelo, recomenda-se descrever cada uma das funções utilizadas, o que pode ser feito por meio de uma legenda ou comentário no modelo.

As instruções *n*, associadas às ocorrências de variáveis *ov*, podem ser de três tipos:

- **instrução de entrada de dados** – representada por READ(v), onde $v \in$ V;
- **instrução de saída de dados** – representada por WRITE(v) , onde $v \in$ V;
- **instrução de atribuição de dados** – representada por $v \leftarrow exp$, onde *exp* é uma expressão formada por variáveis de V, constantes, operadores e chamadas de funções.

Uma ocorrência de variável é representa por $\rho(s, v, n)$. Isso significa que existe uma ocorrência da variável $v \in$ V, no estado $s \in$ S, na instrução $n \in$ N. Em um mesmo estado *s* pode existir mais que uma ocorrência da variável *v*, em diferentes instruções *n*.

Uma transição de estados da MEFE M é representada por $\delta(s_i, i, p) = s_j$ e a saída obtida pela realização dessa transição de estados é representada por $\lambda(s_i, i, p) = o$. Isso significa que, quando a MEFE M está no estado $s_i \in$ S, ao receber uma entrada $i \in$ I, se o predicado $p \in$ P for satisfeito, ela se move para o estado $s_j \in$ S, produzindo uma saída $o \in$ O; s_i é o estado atual ou estado corrente e s_j é o próximo estado.

As variáveis das MEFEs são necessárias para representar o fluxo de dados associado ao comportamento do software modelado. Essas variáveis não são variáveis internas de um programa que implemente a MEFE. Na realidade, as variáveis de MEFEs estão em um nível de abstração mais alto do que o das utilizadas na implementação do software. Assim, apesar de possível, não é necessário nem desejável que exista um mapeamento de um para um entre as variáveis de uma MEFE com as variáveis utilizadas em um programa que implemente a MEFE. O mesmo vale para os predicados de MEFEs, os quais também estão em um nível de abstração mais alto que o dos utilizados na implementação do software. Ambos são utilizados apenas para representar o comportamento que o software deve possuir quando implementado.

Os elementos *entrada* e *saída* pertencentes, respectivamente, aos conjuntos I e O da MEFE, não são utilizados diretamente na definição dos critérios de teste funcional baseados em MEFEs apresentados neste trabalho. Entretanto, eles foram mantidos na definição das MEFEs para manter a estrutura existente nas MEFs tradicionais. As *entradas* e *saídas* podem ser necessárias à definição completa do comportamento de software através de MEFs – tradicionais ou estendidas, como é apresentado na próxima seção.

Como ocorre com as MEFs tradicionais, uma MEFE M também pode ser determinística ou não determinística. Se a quantidade de transições de um estado $s \in S$, sobre um entrada $i \in I$ e satisfazendo o predicado $p \in S$ é igual a um, então a máquina é determinística. Ou seja, uma MEFE é determinística se, sobre uma entrada i e com a satisfação de um predicado p, a máquina segue apenas uma única transição até um próximo estado. Caso contrário, a máquina é não determinística, ou seja, não é possível definir previamente qual transição deve ser seguida e, conseqüentemente, podem ser produzidas saídas diferentes.

Novamente, como também ocorre nas MEFs tradicionais, além da representação textual de uma MEFE, existem dois tipos de representação que podem ser usadas para descrever uma MEFE. Esses tipos de representação são o diagrama de transições de estados e a tabela de transições de estados, os quais são apresentados a seguir.

4.2.1. Diagrama de Transições de Estados

Para as MEFEs, os diagramas de transições de estados também são muito úteis para a visualização das informações. Uma representação desse tipo de diagrama é apresentada na Figura 4.3. Uma MEFE M pode ser representada por um grafo dirigido G = (V, E), onde:

- V é um conjunto de vértices, que representa o conjunto de estados S da MEFE M. Cada $v \in V$ representa um estado $s \in S$ e contém as ocorrências de variáveis $ov \in OV$ associadas ao estado s;
- $E = \{(v_i, v_j; i/o; p), v_i, v_j \in V\}$ é um conjunto de arcos dirigidos, que representa as transições da MEFE M. Cada $e = (v_i, v_j; i/o; p)$ representa uma transição de estado de v_i para v_j, com entrada i e saída o, que só é realizada se o predicado p for satisfeito.

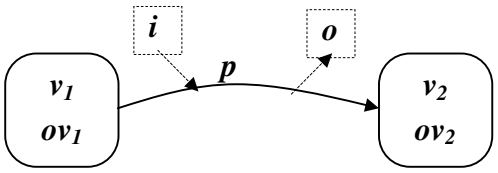

Figura 4.3 – Representação do Diagrama de Transições de Estados de uma MEFE

4.2.2. Tabela de Transições de Estados

As tabelas de transições de estados são muito úteis para o armazenamento e manipulação computacional das informações das MEFEs, assim como para as MEFs tradicionais. Uma MEFE M pode ser representada por uma tabela de transições de estados, conforme apresentado na Figura 4.4, onde cada linha representa um estado e cada coluna representa uma entrada e um predicado a ser satisfeito. Para uma combinação de um estado atual $s_i \in$ S, uma entrada $i_j \in$ I e um predicado $p_k \in$ P a ser satisfeito, a posição correspondente da tabela especifica o próximo estado $s \in$ S e a saída $o \in$ O.

	i_1 / p_1	i_2 / p_2	...	i_m / p_k
s_1	s / o	s / o	...	s / o
s_2	s / o	s / o	...	s / o
...	...	...	...	...
s_n	s / o	s / o	...	s / o

Figura 4.4 – Tabela de Transições de Estados de uma MEFE

Com o propósito de representar e/ou armazenar as informações de fluxo de dados, deve existir uma tabela adicional para essas informações. Nessa tabela, conforme apresentado na Figura 4.5, cada linha também representa um estado e cada coluna representa uma variável. Para uma combinação de um estado atual $s_i \in$ S e uma variável $v_j \in$ V, a posição correspondente da tabela especifica o conjunto de instruções (n_1, n_2, ..., n_k), com $k \geq 0$, em que a variável v é referenciada no estado i.

	v_1	V_2	...	v_m
s_1	(n_1, n_2, ..., n_k)	(n_1, n_2, ..., n_k)	...	(n_1, n_2, ..., n_k)
s_2	(n_1, n_2, ..., n_k)	(n_1, n_2, ..., n_k)	...	(n_1, n_2, ..., n_k)
...	...	...	...	...
s_n	(n_1, n_2, ..., n_k)	(n_1, n_2, ..., n_k)	...	(n_1, n_2, ..., n_k)

Figura 4.5 – Tabela de Ocorrências de Variáveis de uma MEFE

4.3. Modelagem Comportamental de Software por meio de MEFEs

A MEFE apresentada na seção anterior é a definição de uma técnica que pode ser utilizada para modelar o comportamento de um software. Diretrizes devem ser seguidas para a obtenção de um modelo que represente corretamente o comportamento desejado do software e que possa ser utilizado como base para a execução da atividade de teste. Essas diretrizes dizem respeito ao mapeamento que deve existir entre os elementos do software e os elementos da MEFE.

- **Estado:** um estado da MEFE deve representar a execução de uma funcionalidade do software modelado. A granulosidade da funcionalidade representada por um estado depende do rigor desejado para o modelo e para o teste de software baseado nesse modelo. Recomenda-se criar um estado para cada função propriamente dita que será ou já foi implementada no software, dependendo da fase do processo de engenharia de software em que o modelo está sendo criado.

- **Transição:** uma transição de estados da MEFE deve representar o término da execução de uma funcionalidade e o início da execução de uma outra funcionalidade do software modelado. Uma transição deve representar somente uma mudança de estados propriamente dita, na forma de um evento praticamente instantâneo, sem incluir qualquer operação ou ação dentro do software.

- **Entrada:** uma entrada da MEFE deve representar um evento externo fixo e determinado em tempo de modelagem que deve ocorrer para que uma determinada transição de estados do software modelado seja realizada. Por exemplo, o clique com o *mouse* em um botão OK ou o acionamento de algum botão por meio de um dispositivo de entrada. Nem todas as transições precisam ter uma entrada associada, pois o conjunto de entradas pode conter a entrada nula. Além disso, entradas relativamente óbvias, como a digitação da tecla <Enter>, por exemplo, não precisam ser modeladas, podendo ser consideradas como entradas nulas. As entradas não podem representar instruções de entrada de dados para ser armazenados em variáveis, já que os eventos que elas representam são eventos padrão e pré-determinados, os quais não podem ser alterados. As instruções de entrada de dados variáveis devem ser representadas por ocorrências de variável.

- **Saída:** uma saída da MEFE deve representar um evento externo fixo e determinado em tempo de modelagem a ser gerado sempre que ocorrer a realização de uma transição de estados do software modelado. Por exemplo, a apresentação de uma mensagem de erro padrão, a apresentação de um sinal sonoro padrão ou o acionamento de um dispositivo – como uma lâmpada. Similarmente para as entradas da MEFE, nem todas as transições precisam ter uma saída associada, pois o conjunto de saídas pode conter a saída nula. As saídas não podem representar instruções de saída de dados armazenados em variáveis, já que os eventos que elas representam são eventos padrão e pré-determinados, os quais não podem ser alterados. As instruções de saída de dados variáveis devem ser representadas por ocorrências de variável.

- **Variável:** uma variável global da MEFE deve representar uma variável qualquer necessária para armazenar informações durante a execução do software modelado. Uma ocorrência de variável da MEFE existe quando uma variável é referenciada em uma instrução. Esta instrução deve ser executada dentro de uma funcionalidade do software modelado, ou seja, uma ocorrência de variável deve existir dentro de um estado da MEFE. Três tipos de instrução podem ser modelados por MEFEs: instrução de entrada de dados de valor para variável; instrução de saída de dados de valor de variável; e instrução de atribuição de dados de valor para variável.

- **Funções:** as funções utilizadas nas expressões que, por sua vez, podem ser utilizadas dentro de instruções de atribuição de dados, não precisam ser descritas na MEFE; deste modo, evita-se o detalhamento de informações referentes ao projeto de software. Para manter o entendimento claro do modelo, recomenda-se utilizar nomes sugestivos que indiquem a operação da função.

- **Predicado:** um predicado da MEFE deve representar uma condição que deve ser satisfeita para que uma transição de estados do software modelado ocorra. Ou seja, ele deve indicar a condição lógica que deve ser satisfeita para que o software termine a execução de uma funcionalidade e inicie a execução de uma outra funcionalidade. Um predicado deve ser uma expressão formada por variáveis globais definidas para a MEFE, constantes e/ou operadores.

4.4. Exemplo de Modelagem Comportamental de Software

Para um melhor entendimento das MEFEs – descritas na Seção 4.2 – e das diretrizes para se modelar o comportamento de software por meio de seu uso – descritas na Seção 4.3, esta seção apresenta o exemplo da modelagem comportamental de um sistema chamado Tele. São apresentados: os requisitos do sistema, especificados de forma textual; a MEFE que especifica o comportamento do sistema descrito; e o diagrama de transições de estados.

4.4.1. Requisitos do Sistema Tele

O sistema Tele é utilizado para a realização de ligações telefônicas a partir de uma cabine telefônica informatizada. Esta cabine é dotada de um aparelho telefônico acoplado em um sistema computacional, em que é executado o sistema Tele, por meio do qual o usuário realiza suas ligações telefônicas. O usuário utiliza o sistema Tele para escolher a forma de pagamento a ser utilizada, informar o número de telefone a ser chamado, realizar a ligação e, depois do término da ligação, obter dados a respeito da ligação realizada.

O sistema Tele oferece três possibilidades de pagamento:

- O valor referente à ligação pode ser incluído em uma fatura mensal, sendo necessário para isso que o usuário possua um cartão telefônico da operadora;
- A ligação pode ser paga por meio de cartão de crédito; ou
- A ligação pode ser feita a cobrar.

Caso o usuário opte por pagar a ligação com cartão telefônico, ele deve fornecer o número de seu cartão e sua senha, os quais devem ser verificados. Caso o usuário opte por pagar a ligação com cartão de crédito, ele deve fornecer o número de seu cartão, o qual também deve ser verificado. Caso o usuário opte por fazer a ligação a cobrar, para que ela se complete, o receptor deve aceitá-la no momento de seu atendimento.

Depois de escolhida a forma de pagamento, o usuário deve fornecer o número de telefone a ser chamado. O sistema deve identificar o valor da tarifa (por minuto) a ser cobrada para a ligação, a qual depende do número fornecido, e depois discar o número desejado. Assim que o receptor atender a ligação (ou aceitá-la, caso ela seja a cobrar), o sistema deve identificar o

momento inicial da ligação. Depois que ela é encerrada, o sistema deve identificar o momento final e, assim, calcular o tempo total e o valor total da ligação. Depois desses valores identificados ou calculados, o sistema deve armazená-los em um banco de dados para uso posterior.

Ao terminar uma ligação, caso ela não seja a cobrar, o usuário tem a possibilidade de receber dados sobre a ligação realizada, tais como: número discado, tempo total e valor total da ligação. Depois que uma ligação for encerrada, o usuário tem a possibilidade de realizar outras ligações e, nesse caso, a possibilidade de mudar a forma de pagamento para uma nova ligação.

4.4.2. Máquina de Estados Finitos Estendida do Sistema Tele

Tele = < S; s_0; I; O; V; OV; P; D; δ; λ >, onde:

S = {Início da Sessão;
Determinar Forma de Pagamento;
Obter Dados do Cartão Telefônico;
Obter Dados do Cartão de Crédito;
Realizar a Ligação;
Receber Confirmação do Receptor;
Conversação;
Terminar a Ligação;
Armazenar Dados em BD;
Informar Dados sobre a Ligação?;
Apresentar Dados;
Realizar Outra Ligação?;
Mudar Forma de Pagamento?;
Fim da Sessão}

s_0 = {Início da Sessão}

I = {Receber Chamada;
Cancelar;
Atender;

Desligar}

O = {Apresentar Mensagem de Bem-Vindo;
Apresentar Mensagem de Ligação a Cobrar;
Apresentar Mensagem de Agradecimento}

V = {f_p; // forma de pagamento
n_c; // número do cartão
senha; // senha do cartão
res; // resposta
n_t; // número de telefone
tarifa; // valor da tarifa
o_r; // opção do receptor
t_i; // tempo inicial
t_f; // tempo final
t; // tempo total
v; // valor total da ligação
op1; // opção de escolha – se deseja inofrmar dados sobre a ligação
op2; // opção de escolha – se deseja realizar outra ligação
op3} // opção de escolha – se deseja mudar a forma de pagamento

P = {f_p = c_telefônico;
f_p = c_crédito;
f_p = a_cobrar;
!(f_p = a_cobrar);
f_p = cancelar;
res = V;
res = F;
o_r = V;
o_r = F;
op1 = V;
op1 = F;
op2 = V;
op2 = F;

op3 = V;
op3 = F}

N = {READ(f_p);
READ(n_c);
READ(senha);
READ(n_t);
READ(o_r);
READ(op1);
READ(op2);
READ(op3);
WRITE(n_t);
WRITE(t);
WRITE(v);
res ← f_1(n_c, senha); //f_1 – verifica validade de n_c e senha
res ← f_2(n_c); //f_2 – verifica validade de n_c
tarifa ← f_3(n_t); //f_3 – obtém o valor da tarifa para o n_t
t_i ← f_4(); //f_4 – obtém a hora extana atual
t_f ← f_4(); //f_4 – obtém a hora extana atual
t ← t_f - t_i;
v ← t * tarifa;
BD ← f_p;
BD ← n_c;
BD ← n_t;
BD ← t_i;
BD ← t;
BD ← v}

ρ(Determinar Forma de Pagamento, f_p, READ(f_p));
ρ(Obter Dados do Cartão Telefônico, n_c, READ(n_c));
ρ(Obter Dados do Cartão Telefônico, senha, READ(senha));
ρ(Obter Dados do Cartão Telefônico, res, res ← f(n_c, senha));
ρ(Obter Dados do Cartão Telefônico, n_c, res ← f(n_c, senha));

ρ(Obter Dados do Cartão Telefônico, senha, res ← f(n_c, senha));
ρ(Obter Dados do Cartão de Crédito, n_c, READ(n_c));
ρ(Obter Dados do Cartão de crédito, res, res ← f(n_c));
ρ(Obter Dados do Cartão Telefônico, n_c, res ← f(n_c));
ρ(Realizar a Ligação, n_t, READ(n_t));
ρ(Realizar a Ligação, tarifa, tarifa ← f(n_t));
ρ(Realizar a Ligação, n_t, tarifa ← f(n_t));
ρ(Receber Confirmação do Receptor, o_r, READ(o_r));
ρ(Conversação, t_i, t_i ← f());
ρ(Terminar a Ligação, t_f, t_f ← f());
ρ(Terminar a Ligação, t, t ← t_f - t_i);
ρ(Terminar a Ligação, t_f, t ← t_f - t_i);
ρ(Terminar a Ligação, t_i, t ← t_f - t_i);
ρ(Terminar a Ligação, v, v ← t * tarifa);
ρ(Terminar a Ligação, t, v ← t * tarifa);
ρ(Terminar a Ligação, tarifa, v ← t * tarifa);
ρ(Armazenar Dados em BD, f_p, BD ← f_p);
ρ(Armazenar Dados em BD, n_c, BD ← n_c);
ρ(Armazenar Dados em BD, n_t, BD ← n_t);
ρ(Armazenar Dados em BD, t_i, BD ← t_i);
ρ(Armazenar Dados em BD, t, BD ← t);
ρ(Armazenar Dados em BD, v, BD ← v);
ρ(Informar Dados sobre a Ligação?, op1, READ(op1));
ρ(Apresentar Dados?, n_t, WRITE(n_t));
ρ(Apresentar Dados?, t, WRITE(t));
ρ(Apresentar Dados?, v, WRITE(v));
ρ(Realizar Outra Ligação?, op2, READ(op2));
ρ(Mudar Forma de Pagamento?, op3, READ(op3));

δ(Início da Sessão, Receber Chamada,) = Determinar Forma de Pagamento;
δ(Determinar Forma de Pagamento, , f_p = c_telef) = Obter Dados do Cartão Telefônico;
δ(Determinar Forma de Pagamento, , f_p = c_crédito) = Obter Dados do Cartão de Crédito;

δ(Determinar Forma de Pagamento, , f_p = a_cobrar) = Realizar a Ligação;
δ(Determinar Forma de Pagamento, , f_p = cancelar) = Fim da Sessão;
δ(Obter Dados do Cartão Telefônico, Cancelar,) = Determinar Forma de Pagamento;
δ(Obter Dados do Cartão Telefônico, , res = V) = Realizar a Ligação;
δ(Obter Dados do Cartão Telefônico, , res = F) = Obter Dados do Cartão Telefônico;
δ(Obter Dados do Cartão de Crédito, Cancelar,) = Determinar Forma de Pagamento;
δ(Obter Dados do Cartão de Crédito, , res = V) = Realizar a Ligação;
δ(Obter Dados do Cartão de Crédito, , res = F) = Obter Dados do Cartão de Crédito;
δ(Realizar a Ligação, Atender, f_p = a_cobrar) = Receber Confirmação do Receptor;
δ(Realizar a Ligação, Atender, !(f_p = a_cobrar)) = Conversação;
δ(Receber Confirmação do Receptor, , o_r = V) = Conversação;
δ(Receber Confirmação do Receptor, , o_r = F) = Realizar Outra Ligação?;
δ(Conversação, Desligar,) = Terminar a Ligação;
δ(Terminar a Ligação, ,) = Armazenar Dados em BD;
δ(Armazenar Dados em BD, , f_p = a_cobrar) = Realizar Outra Ligação?;
δ(Armazenar Dados em BD, , !(f_p = a_cobrar)) = Informar Dados sobre a Ligação?;
δ(Informar Dados sobre a Ligação?, , op1 = V) = Apresentar Dados;
δ(Informar Dados sobre a Ligação?, , op1 = F) = Realizar Outra Ligação?;
δ(Apresentar Dados, ,) = Realizar Outra Ligação?;
δ(Realizar Outra Ligação?, , op2 = V) = Mudar Forma de Pagamento?;
δ(Realizar Outra Ligação?, , op2 = F) = Fim da Sessão;
δ(Mudar Forma de Pagamento?, , op3 = V) = Determinar Forma de Pagamento;
δ(Mudar Forma de Pagamento?, , op3 = F) = Realizar a Ligação.

λ(Início da Sessão, Receber Chamada,) = Apresentar Mensagem de Bem-Vindo;
λ(Determinar Forma de Pagamento, , f_p = cancelar) = Apresentar Mens. de Agradecimento;
λ(Realizar a Ligação, Atender, f_p = a_cobrar) = Apresentar Mensagem de Ligação a Cobrar;
λ(Realizar Outra Ligação, , op2 = F) = Apresentar Mensagem de Agradecimento.

4.4.3. Diagrama de Transições de Estados do Sistema Tele

O diagrama de transições de estados para a MEFE Tele é apresentado na Figura 4.6.

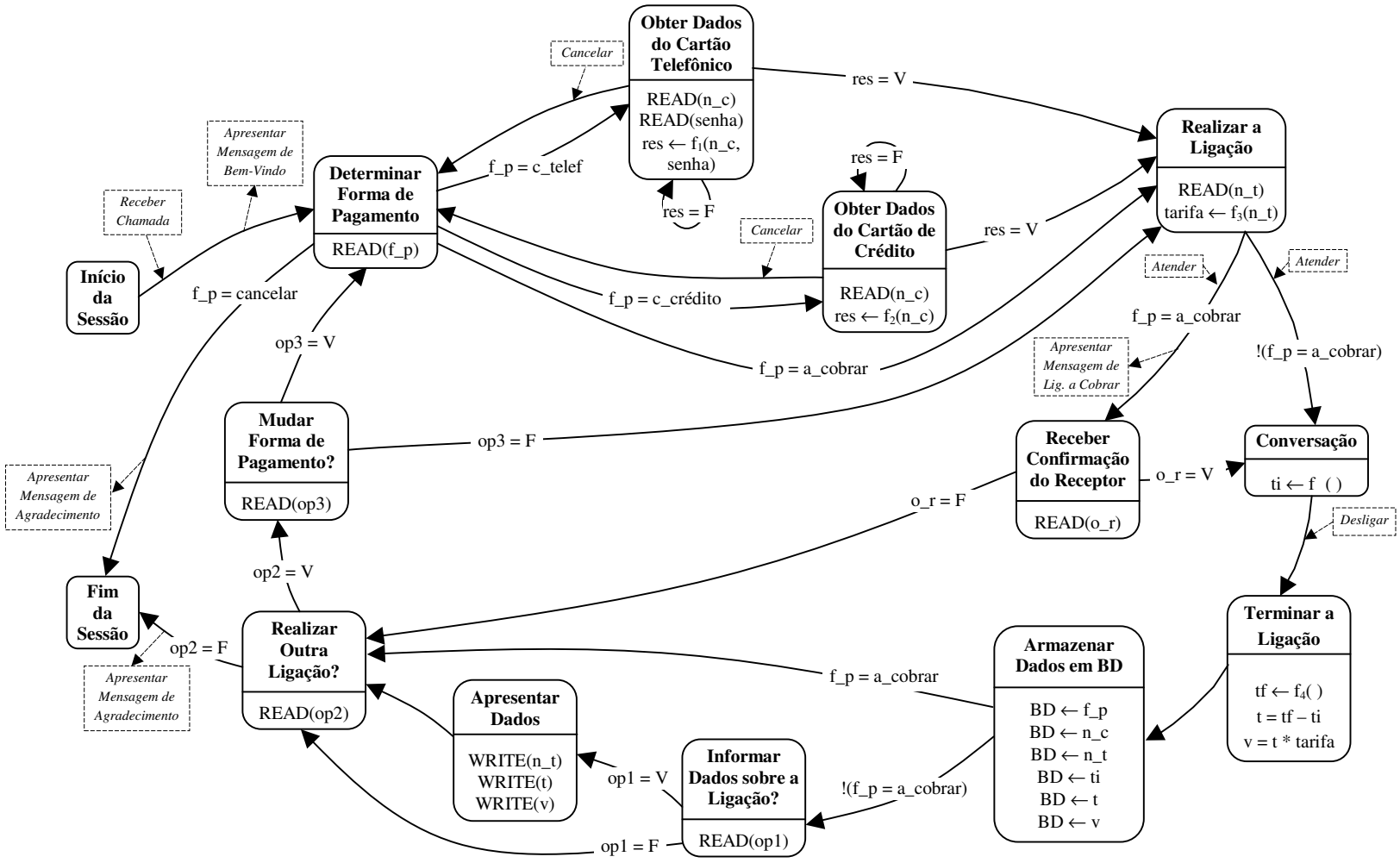

Figura 4.6 – Diagrama de Transições de Estados da MEFE Tele

4.5. Critérios de Teste Funcional Baseados em MEFEs

Nesta seção são apresentadas as definições de um conjunto de critérios de teste funcional baseados em MEFEs que consideram o fluxo de controle e o fluxo de dados do sistema modelado. A definição desses critérios foi realizada a partir da adaptação de critérios de teste estrutural. Essa adaptação foi possível devido ao mapeamento existente entre os elementos das MEFEs e os elementos de grafos de fluxo de programas em que se baseiam as definições dos critérios de teste estrutural.

Foram considerados os seguintes critérios de teste estrutural:

1) Critérios de teste baseado em fluxo de controle, incluindo os critérios Todos_os_nós, Todos_os_arcos e Todos_os_caminhos (Beizer, 1990; Pressman, 1992); e
2) Critérios de teste baseados em fluxo de dados, incluindo os Critérios de Rapps & Weyuker (Rapps & Weyuker, 1985) e os critérios Potenciais Usos (Maldonado, 1991).

4.5.1. Definições

Algumas definições são necessárias para a apresentação dos critérios baseados em MEFEs. Essas definições são similares às apresentadas em (Rapps & Weyuker, 1985), empregadas na apresentação dos critérios de teste estrutural baseados em fluxo de controle e fluxo de dados.

As ocorrências de variáveis na MEFE podem ser de três tipos:

- **definição de variável** (indicada por ***def***): uma ocorrência de uma variável *v* é do tipo *def* quando ela faz parte de uma instrução de entrada de dados, representada por READ(*v*), ou faz parte do lado esquerdo de uma instrução de atribuição de dados;
- **uso computacional de variável** (indicada por ***c-uso***): uma ocorrência de uma variável *v* é do tipo *c-uso* quando ela faz parte de uma instrução de saída de dados,

representada por WRITE(v), ou faz parte da expressão localizada no lado direito de uma instrução de atribuição de dados; e

- **uso predicativo de variável** (indicada por ***p-uso***): uma ocorrência é do tipo *p-uso* quando ela faz parte da expressão de um predicado.

De modo geral, uma ocorrência de variável é do tipo *def* quando a variável recebe um valor. Uma ocorrência de variável é do tipo *c-uso* ou *p-uso* quando o valor da variável está sendo usado de alguma forma. A diferença entre um *c-uso* e um *p-uso* é que, enquanto o primeiro tipo afeta diretamente a computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado, o segundo tipo afeta diretamente o fluxo de controle do programa.

Em função dos tipos de ocorrência de variável em MEFEs, os seguintes conjuntos de variáveis são definidos:

- ***def*(*s*):** é o conjunto de variáveis para as quais o estado s contém uma definição;
- ***c-uso*(*s*):** é o conjunto de variáveis para as quais o estado s contém um *c-uso*;
- ***p-uso*(s_i, s_j):** é o conjunto de variáveis para as quais a transição (s_i, s_j) contém um *p-uso*;

Existem vários tipos de caminhos em uma MEFE, definidos a seguir:

- **caminho:** seqüência finita de estados (s_1, ..., s_k), $k \geq 2$, tal que existe uma transição de s_i até s_{i+1} para $i = 1, 2, ..., k - 1$;
- **caminho simples:** caminho em que todos os estados, exceto possivelmente o primeiro e o último, são distintos;
- **caminho livre de laço:** caminho em que todos os estados, sem exceção, são distintos;
- **caminho livre de definição:** um *caminho livre de definição c.r.a.* (com relação a) x do estado i até o estado j ou do estado i até a transição (s_m, j) é um *caminho* (i, s_1, ..., s_m, j), sem nenhuma *def* de x nos estados s_1, ..., s_m;
- **du-caminho:** um *du-caminho c.r.a. x* é um *caminho* (s_1, ..., s_j, s_k) tal que o estado s_1 tem uma *def* de x e: 1) s_k tem um *c-uso* de x e (s_1, ..., s_j, s_k) é um *caminho simples*

livre de definição c.r.a. x; ou 2) (s_j, s_k) tem um *p-uso* de *x* e (s_1, ..., s_j) é um *caminho livre de laço e livre de definição c.r.a. x*;

- **potencial du-caminho:** um *potencial du-caminho c.r.a. x* é um *caminho livre de laço e livre de definição c.r.a. x* (s_1, ..., s_k) tal que o estado s_1 tem uma *def* de *x*.

Os seguintes conjuntos são necessários para a definição dos critérios baseados em MEFE. Sendo *s* um estado qualquer e *x* uma variável tal que $x \in def(s)$:

- **dcu(*x*, *s*):** é o conjunto de todos os estados s_i tais que $x \in c\text{-}uso(s_i)$ e para os quais existe um *caminho livre de definição c.r.a. x* de *s* até s_i;
- **dpu(*x*, *s*):** é o *conjunto* de todas as transições (s_i, s_j) tais que $x \in p\text{-}uso(s_i, s_j)$ e para os quais existe um *caminho livre de definição c.r.a. x* de *s* até s_i.

4.5.2. Definição dos Critérios

Dada uma MEFE, um critério de teste define um conjunto de elementos requeridos da MEFE. A seguir, são apresentadas as definições formais dos critérios propostos de teste funcional baseados em MEFEs. Para cada critério, é apresentado um exemplo de um elemento requerido relativo a MEFE Tele da Figura 4.6.

Dada uma MEFE determinística e um conjunto de casos de teste, são definidos os seguintes critérios baseados em MEFEs:

4.5.2.1. Critérios Baseados em Fluxo de Controle

- **Critério Todos_os_estados (*all-states*):** requer que cada estado *s* da MEFE seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: o estado *Determinar Forma de Pagamento*.

- **Critério Todas_as_transições (*all-trans*):** requer que cada transição (s_i, s_j) da MEFE seja exercitada pelo menos uma vez pelo conjunto de casos de teste.

Exemplo de elemento requerido: a transição (*Obter Dados do Cartão Telefônico, Realizar a Ligação*).

- **Critério Todos_os_pares_de_transição (*all-trans-pair*):** requer que cada combinação de transição de entrada (s_i, s) e transição de saída (s, s_j) de todos os estados s da MEFE seja exercitada pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: para o estado *Conversação,* as transições (*Realizar a Ligação*, *Conversação*) e (*Conversação*, *Terminar a Ligação*); e (*Obter Dados do Cartão de Crédito*, *Conversação*) e (*Conversação*, *Terminar a Ligação*).

- **Critério Todos_os_caminhos (*all-paths*):** requer que cada caminho (s_1, ..., s_k), com $k \geq 2$, da MEFE seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: o caminho (*Início da Sessão*, *Determinar Forma de Pagamento*, *Fim da Sessão*).

Estes critérios poderiam ser aplicados diretamente a uma MEF tradicional, visto que eles consideram apenas informações de fluxo de controle da modelagem comportamental do software. Na realidade, alguns desses critérios já foram definidos para MEFs tradicionais por Offutt (2000). Eles são apresentados aqui para efeito de completude de critérios de teste funcional baseados em MEFEs.

4.5.2.2. Critérios Baseados em Fluxo de Dados

- **Critério Todas_as_definições (*all-defs*):** requer que, para cada estado s da MEFE e cada variável $x \in$ def(s), pelo menos um caminho livre de definição c.r.a. x de cada estado s até algum estado de dcu(x, s) ou alguma transição de dpu(x, s) seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Realizar a Ligação* e o estado *Apresentar Dados*, relativa à variável *n_t*.

- **Critério Todos_os_c-usos (*all-c-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada estado de dcu(*x*, *s*) seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Determinar Forma de Pagamento* e o estado *Armazenar Dados em BD*, relativa à variável *f_p*.

- **Critério Todos_os_p-usos (*all-p-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada transição de dpu(*x*, *s*) seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Determinar Forma de Pagamento* e a transição (*Realizar a Ligação*, *Receber Confirmação do Receptor*), relativa à variável *f_p*.

- **Critério Todos_os_c-usos/alguns_p-usos (*all-c-uses/some-p-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada estado de dcu(*x*, *s*), ou – se dcu(*x*, *s*) é vazio – até alguma transição de dpu(*x*, *s*), seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplos de elementos requeridos: a associação entre o estado *Determinar Forma de Pagamento* e o estado *Armazenar Dados em BD*, relativa à variável *f_p* (elemento requerido do tipo c-uso); ou a associação entre o estado *Receber Confirmação do Receptor* e a transição (*Receber Confirmação do Receptor*, *Conversação*), relativa à variável *o_r* (elemento requerido do tipo p-uso).

- **Critério Todos_os_p-usos/alguns_c-usos (*all-p-uses/some-c-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada transição de dpu(*x*, *s*), ou – se dpu(*x*, *s*) é vazio – até algum estado de dcu(*x*, *s*), seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

Exemplos de elementos requeridos: a associação entre o estado *Determinar Forma de Pagamento* e a transição (*Realizar a Ligação*, *Receber Confirmação do Receptor*), relativa à variável *f_p* (elemento requerido do tipo p-uso); ou a associação entre o estado *Obter Dados do Cartão Telefônico* e o estado *Armazenar Dados em BD*, relativa à variável *n_c* (elemento requerido do tipo c-uso).

- **Critério Todos_os_usos (*all-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada estado de dcu(*x*, *s*) e até cada transição de dpu(*x*, *s*) seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Informar Dados sobre a Ligação?* e a transição (*Informar Dados sobre a Ligação?*, *Apresentar Dados*), relativa à variável *op1*.

- **Critério Todos_os_du-caminhos (*all-du-paths*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), cada du-caminho c.r.a. *x* seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Realizar a Ligação* e o estado *Terminar a Ligação*, relativa à variável *tarifa*.

- **Critério Todos_potenciais_usos (*all-pot-uses*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um caminho livre de definição c.r.a. *x* de cada estado *s* até cada estado e transição alcançável a partir de *s* seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Conversação* e a transição (*Informar Dados sobre a Ligação?*, *Apresentar Dados*), relativa à variável *ti*.

- **Critério Todos_potenciais_usos/du (*all-pot-uses/du*):** requer que, para cada estado *s* da MEFE e cada variável $x \in$ def(*s*), pelo menos um du-caminho c.r.a. *x* de cada estado *s* até cada estado e transição alcançável a partir de *s* seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

Exemplo de elemento requerido: a associação entre o estado *Informar Dados sobre a Ligação?* e a transição (*Mudar Forma de Pagamento*, *Determinar Forma de Pagamento*), relativa à variável *op1*.

- **Critério Todos_potenciais_du-caminhos (*all-pot-du-paths*):** requer que, para cada estado *s* da MEFE e cada variável $x \in \text{def}(s)$, cada potencial du-caminho c.r.a. *x* seja exercitado pelo menos uma vez pelo conjunto de casos de teste.

  Exemplo de elemento requerido: a associação entre o estado *Terminar a Ligação* e o estado Fim da Sessão, relativa à variável *v*.

4.6. Vantagens e Desvantagens da Aplicação dos Critérios Definidos

Os critérios de teste funcional baseados em MEFEs definidos oferecem vantagens tanto em relação aos critérios de teste funcional baseados apenas em fluxo de controle de MEFs tradicionais quanto em relação aos critérios de teste estrutural baseados na estrutura interna do software.

No primeiro caso, como ambos os conjuntos de critérios são de teste funcional, a vantagem é em relação ao nível de cobertura do teste a ser realizado. Enquanto os critérios baseados em análise de fluxo de controle do software modelado exigem que elementos como comandos, arcos e laços da MEF Estendida sejam exercitados, os critérios baseados em análise de fluxo de dados definidos se focalizam em como são atribuídos valores às variáveis da MEF Estendida que direcionam o comportamento do software, e em como essas variáveis são usadas.

Ao invés de selecionar caminhos do modelo comportamental do software baseado somente em sua estrutura de controle, os critérios baseados em fluxo de dados selecionam caminhos que rastreiam as ocorrências de variáveis. Um caminho pode passar por todo o ciclo de uma variável – desde a atribuição inicial de valor, passando por possíveis alterações, até que seus valores sejam finalmente usados em um cálculo ou apresentados aos usuários. Portanto, a satisfação dos critérios baseados em fluxo de dados, quando não detectam a presença de defeitos, confere ao software testado uma maior indicação de confiabilidade e de qualidade, em relação a satisfação de critérios baseados em MEFs tradicionais.

No segundo caso, como os dois conjuntos de critérios comparados são, respectivamente, de teste funcional e estrutural, a vantagem principal está relacionada com os tipos de erros cuja presença pode ser detectada pela sua utilização. Por exemplo, a execução de casos de teste, por meio do uso de critérios de teste funcional baseados em MEFEs, pode detectar a ausência de uma funcionalidade especificada no modelo e que não foi implementada no software.

Normalmente, a estrutura do software é mais complexa do que sua própria especificação, pois a implementação de um sistema precisa levar em consideração mais detalhes do que os definidos na especificação. Essa diferença de complexidade pode fazer com que a quantidade necessária de casos de teste para satisfazer os critérios de teste funcional baseados em MEFEs seja menor do que a quantidade necessária para satisfazer os critérios de teste estrutural baseados na estrutura interna do software.

Para o caso geral, os critérios de teste funcional baseados em MEFEs apresentaram-se como uma solução intermediária, em relação ao nível de cobertura do teste a ser realizado, entre os critérios de teste funcional baseados em MEFs tradicionais, que modelam o comportamento do software, e os critérios de teste estrutural baseados na estrutura interna do software. Se por um lado a aplicação de critérios de teste funcional baseados em MEFs tradicionais pode levar a descoberta da presença de poucos defeitos no software, devido a seu baixo nível de exigência; por outro lado, a aplicação de critérios de teste estrutural podem demandar um alto custo de aplicação, devido a seu alto nível de exigência.

Entretanto, a necessidade de elaboração de uma MEFE que modele o comportamento do software implica o ônus da especificação das informações referentes ao fluxo de dados. Portanto, antes do uso da abordagem proposta neste trabalho, é necessário analisar a relação custo/benefício em função do tipo de software a que os critérios podem ser aplicados. Considerando apenas a construção da MEFE em si, o esforço pode ser muito grande; mas, considerando o esforço já incorrido na construção de uma MEF tradicional, o esforço adicional é pouco significativo.

4.7. Aplicabilidade dos Critérios Definidos

Os critérios de teste funcional baseados em MEFEs apresentados neste trabalho são definidos para aplicação principalmente na fase de teste de validação e de sistema, embora eles também

possam ser usados, talvez com menor benefício, nas fases de teste de integração e teste de unidades. Neste caso, deve ser realizado um estudo para definir como as MEFEs podem ser utilizadas para modelar estruturas de projeto do sistema a ser desenvolvido.

Uma estratégia comumente utilizada e que pode ser aplicada com estes conjuntos de critérios, é a utilização de teste funcional e teste estrutural de modo complementar. Deste modo, os critérios de teste estrutural podem ser aplicados para os testes de unidade e de integração; e os critérios de teste funcional definidos podem ser utilizados para os testes de integração, de validação e de sistema.

A utilização desses critérios visa conferir maior qualidade ao teste de sistemas reativos, que são sistemas que respondem à ocorrência de eventos, com entrada de dados, e são dominados por controle. Exemplos típicos são os sistemas que possuem interação com o usuário e cujo fluxo de execução depende de dados fornecidos por esta interação. Apesar disso, as MEFEs também podem ser utilizadas para modelar o comportamento de sistemas não reativos e, assim, o uso dos critérios de teste funcional baseados em MEFEs pode ser estendido para o teste de outros tipos de sistema.

Durante a realização deste trabalho não foi considerada a aplicação dos critérios definidos para outros sistemas que não software. Como as MEFs tradicionais podem ser utilizadas para a modelagem destes outros tipos de sistema, existe uma grande possibilidade de que estes sistemas também possam ser modelados por meio de MEFEs. Assim, o teste destes sistemas poderia ser realizado com o apoio dos critérios definidos neste trabalho e desfrutar das mesmas vantagens oferecidas ao teste de sistemas de software. Entretanto, pesquisas adicionais devem ser realizadas para verificar a real possibilidade dessa aplicação.

4.8. Considerações Finais

Embora as MEFs sejam uma das técnicas mais utilizadas para a modelagem comportamental de software no TBM, elas não consideram aspectos importantes para a atividade de teste de software, como o fluxo de dados. As MEFEs apresentadas neste capítulo têm o objetivo de oferecer mecanismos para a modelagem de informações relativas ao fluxo de dados do software para que essas sejam consideradas durante o teste de software.

Uma das vantagens oferecidas pelos critérios de teste funcional baseados em MEFEs é que eles oferecem um nível intermediário de cobertura do teste em relação aos critérios de teste funcional baseados em MEFs tradicionais, que modelam o comportamento do software, e os critérios de teste estrutural baseados na estrutura interna do software.

As MEFEs são apenas uma linguagem de modelagem que podem ser utilizadas para especificar vários tipos de sistema. Portanto, para que essa técnica seja utilizada na modelagem do comportamento de um sistema computacional, diretrizes devem ser usadas para que o analista mapeie corretamente os elementos do sistema e os elementos da MEFE. Essas diretrizes – apresentadas na Seção 4.3 – mostram, por exemplo, que tipo de informação no sistema deve ser modelado como um estado na MEFE.

A definição das MEFEs teve como ponto central a obtenção de uma correspondência entre seus elementos e os elementos de grafos de fluxo de programa, os quais são utilizados nas definições de critérios de teste estrutural. Por meio desse mapeamento foi possível definir um conjunto de critérios de teste funcional baseados em MEFEs adaptando os critérios de fluxo de controle e de fluxo de dados utilizados no teste estrutural, para os quais já existem vários estudos teóricos e empíricos realizados.

Os critérios definidos e apresentados neste capítulo podem ser usados tanto como critérios de seleção de casos de teste quanto como critérios de adequação de casos de teste. Alguns desses critérios foram implementados em uma ferramenta de teste que foi projetada e implementada inicialmente para ser utilizada na análise de adequação de casos de teste gerados por meio de critérios de teste estrutural. A implementação e os resultados da aplicação desses critérios são apresentados no próximo capítulo.

Capítulo 5

Implementação e Aplicação dos Critérios

Alguns dos critérios de teste funcional baseados em Máquinas de Estados Finitos Estendidas definidos no Capítulo 4 foram implementados na POKE-TOOL, uma ferramenta de apoio ao teste de software desenvolvida para critérios de teste estrutural. A implementação dos critérios de teste funcional consistiu de adaptações na ferramenta para que ela suportasse também os critérios de teste funcional.

O uso da POKE-TOOL para a implementação e aplicação dos critérios de teste funcional não implica que estes utilizam conhecimento interno da estrutura dos programas. Na realidade, a ferramenta é utilizada apenas para realizar a análise de cobertura das MEFEs que especificam o comportamento de um software.

Foi realizado um experimento para comparar empiricamente a relação entre a aplicação, sobre um mesmo programa, dos critérios de teste funcional e dos critérios de teste estrutural correspondentes. A comparação levou em consideração a quantidade de casos de teste necessária para a cobertura completa dos elementos requeridos pelos critérios de teste funcional e a cobertura atingida, com os mesmos casos de teste, dos elementos requeridos pelos critérios de teste estrutural correspondentes.

5.1. A Ferramenta POKE-TOOL

A ferramenta POKE-TOOL (*Potential Uses Criteria Tool for Program Testing*) é uma ferramenta interativa que visa auxiliar o usuário na atividade de teste de software. Ela foi desenvolvida na Faculdade de Engenharia Elétrica e de Computação – FEEC – da

Universidade Estadual de Campinas – UNICAMP (Maldonado, 1989; Chaim, 1991; Bueno et al., 1995).

Essa ferramenta apóia a aplicação de critérios de teste estrutural principalmente como critérios de adequação, ou seja, a ferramenta é utilizada para verificar o quanto um conjunto de casos de teste satisfaz um certo critério de teste. Ou seja, dado um conjunto de casos de teste, a ferramenta pode ser utilizada para realizar uma análise de cobertura, que consiste em determinar o percentual de elementos requeridos que foram exercitados pelo conjunto de casos de teste.

A POKE-TOOL também pode ser utilizada como auxílio na seleção de casos de teste. Dado um conjunto de casos de teste e um critério de teste estrutural, a análise de cobertura permite determinar quais os elementos requeridos que ainda não foram exercitados, fornecendo subsídios para a seleção de novos casos de teste que aumentem o grau de cobertura.

Existem várias versões dessa ferramenta. A versão padrão é utilizada em estações UNIX/SUN, é executada por meio de linha de comando e oferece suporte ao teste de programas escritos na linguagem C. Além dessa versão já existem configurações para o teste de programas escritos nas linguagens COBOL e Fortran.

A POKE-TOOL foi projetada e desenvolvida com o objetivo principal de apoiar a aplicação dos critérios Potenciais Usos – Todos_potenciais_usos, Todos_potenciais_usos/du e Todos_ _potenciais_du-caminhos (Maldonado, 1991). A versão atual da ferramenta oferece suporte também aos critérios Todos_os_nós e Todos_os_arcos e a alguns dos critérios de Rapps & Weyuker – Todos_os_p-usos, Todos_os_usos e Todos_os_du-caminhos (Rapps & Weyuker, 1985).

Três entradas devem ser fornecidas à POKE-TOOL: o código-fonte de um programa; um conjunto de casos de teste; e o conjunto de critérios que serão utilizados. O seu uso é realizado de forma orientada à sessão de trabalho, de modo que, em cada sessão, o usuário pode realizar várias tarefas relacionadas ao teste de software. Cada sessão de trabalho é dividida em duas fases: a fase estática e a fase dinâmica, as quais são descritas a seguir.

A fase estática consiste na análise do código-fonte para preparar a execução da fase dinâmica. São obtidas informações necessárias para a aplicação dos critérios selecionados pelo usuário e

é realizada a instrumentação do código-fonte. O conjunto de elementos requeridos por cada um dos critérios selecionados pelo usuário é determinado. As informações obtidas da análise estática podem também ser utilizadas pelo usuário para a seleção de casos de teste que visem executar esses elementos requeridos. A instrumentação do código-fonte é feita por meio da inserção de pontas de prova (instruções de escrita que produzem um "rastro" do caminho executado) no programa. Essa instrumentação gera uma nova versão do programa em teste para gerar informação para avaliar a adequação de um conjunto de casos de teste executados.

A fase dinâmica consiste na tarefa de execução e avaliação de adequação de casos de teste. Primeiramente é necessário que o programa executável seja gerado a partir do código-fonte instrumentado. O programa gerado é executado e os dados dos casos de teste são fornecidos como entrada ao programa. Realiza-se uma avaliação da cobertura atingida pelos casos de teste utilizados para cada um dos critérios selecionados pelo usuário. O resultado dessa avaliação é, para cada critério escolhido: a cobertura atingida dos elementos requeridos, em porcentagem; o conjunto de elementos requeridos que foram exercitados; e o conjunto de elementos requeridos que não foram exercitados.

A ferramenta POKE-TOOL é constituída basicamente por três módulos integrados. Esses módulos implementam as funcionalidades ou parte das funcionalidades descritas nos parágrafos anteriores e se comunicam entre si por meio de arquivos. Esses três módulos, bem como o relacionamento entre eles, e as entradas e saídas de cada um são apresentados na Figura 5.1. A descrição de cada um dos módulos é apresentada em seguida.

***newpoketool*:** esse módulo inicia a fase estática do uso da ferramenta, realizando uma análise sintática do código-fonte e gerando informações necessárias para a fase dinâmica. Suas entradas principais são o código-fonte e a lista de critérios que serão utilizados. Suas saídas principais são o programa-fonte instrumentado para o teste e, para cada critério, o conjunto de elementos requeridos.

***pokeexec*:** esse módulo monitora e captura informação da execução dos casos de teste. Suas entradas principais são o programa instrumentado executável (gerado pelo módulo *newpoketool*) e os casos de teste a serem executados. Suas saídas principais são, para cada caso de teste executado, um arquivo com os dados de entrada, um arquivo com as saídas obtidas e um arquivo com o caminho executado.

***newpokeaval*:** esse módulo realiza a avaliação da cobertura atingida pelo conjunto de casos de teste executados para os critérios escolhidos. Suas entradas principais são, para cada critério, o conjunto de elementos requeridos (gerado pelo módulo *newpoketool*) e os caminhos executados (gerados pelo módulo *pokeexec*). Suas saídas principais são, para cada critério, a cobertura atingida com a execução do conjunto de casos de teste, o conjunto de elementos requeridos executados e o conjunto de elementos requeridos não executados.

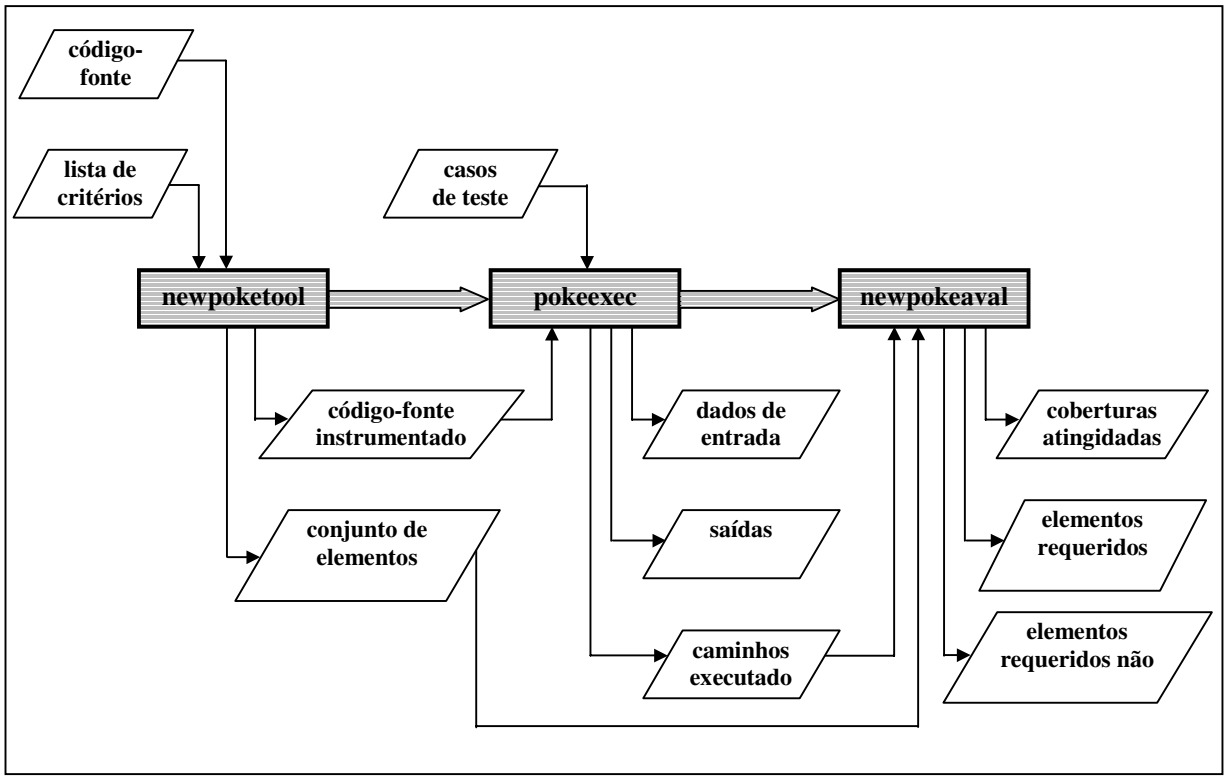

Figura 5.1 – Entradas e Saídas dos Módulos da Ferramenta POKE-TOOL

5.2. Adaptações da Ferramenta POKE-TOOL

A implementação dos critérios de teste funcional na ferramenta POKE-TOOL foi possível devido ao mapeamento conseguido entre os critérios de teste estrutural suportados pela ferramenta e os critérios de teste funcional definidos nesta dissertação. Os critérios implementados são os que possuem os critérios de teste estrutural correspondentes já implementados na ferramenta. Essa implementação consistiu de adaptações na ferramenta POKE-TOOL que se limitaram ao módulo *newpoketool*, responsável pela realização da análise estática do código-fonte a ser testado.

Uma das tarefas que teve que ser adaptada é a de geração de dois arquivos que contêm informações sobre o fluxo de controle e o fluxo de dados do programa. Esses arquivos são chamados, respectivamente, de *<nome_função>.gfc* e *tabvardef.tes*. Na versão padrão da ferramenta as informações desses arquivos são geradas a partir de informações do código-fonte e são utilizadas pelo próprio módulo *newpoketool* durante a geração do conjunto de elementos requeridos para cada critério.

Assim, a primeira adaptação da ferramenta POKE-TOOL é que os arquivos *<nome_função>.gfc* e *tabvardef.tes* fossem gerados por meio de informações retiradas da MEFE, necessária para que os conjuntos de elementos requeridos fossem gerados de acordo com os critérios de teste funcional, ou sejam, sobre informações relacionadas à MEFE, e não de acordo com os critérios de teste estrutural.

Outra tarefa que teve que ser adaptada é a de geração do código-fonte instrumentado. Na versão padrão da ferramenta, essa instrumentação é realizada com base em informações de fluxo de controle contidas no próprio código-fonte com o objetivo de que, durante a execução do programa, sejam registrados os caminhos do programa que são executados durante a execução dos casos de teste.

Portanto, a segunda adaptação da ferramenta POKE-TOOL foi a instrumentação do código-fonte realizada com base em informações retiradas da MEFE, necessária para que fossem registrados os caminhos executados da MEFE para uma análise posterior de cobertura dos elementos requeridos da MEFE.

Dados uma MEFE e um programa que implementa essa máquina, o código-fonte do programa deve ser instrumentado para que, durante a execução do programa, registre-se a seqüência de estados da MEFE que foram exercitados, tornando necessário mapear tais estados para pontos do código-fonte. O mapeamento desses elementos consiste em identificar em que parte do código-fonte é que se inicia cada um dos estados modelados pela MEFE.

O ideal seria que o programador implementasse uma função para cada estado da MEFE e que a função possuísse o mesmo nome do estado correspondente. Entretanto, o programador pode não seguir necessariamente toda a estrutura da MEFE, utilizando, por exemplo, outros nomes para as funções, que não os nomes dos estados da MEFE. Além disso, o programador pode

também implementar duas funções para um mesmo estado ou ainda apenas uma função para dois estados da MEFE. Como esse mapeamento é uma tarefa complexa, sua automação está fora do escopo deste trabalho.

O mapeamento manual consiste em identificar em que parte do código-fonte é que se inicia cada um dos estados modelados pela MEFE. Para isso, basta inserir em cada uma das partes identificadas no código-fonte o seguinte comentário "// ESTADO: <$estado_x$>", onde $estado_x$ deve ser substituído pelo nome completo do estado, conforme encontrado na MEFE. Para cada um dos estados da MEFE, deve existir um comentário desse tipo no código-fonte. Feito isso, o restante da instrumentação necessária do código-fonte é realizada automaticamente pelo módulo responsável.

Com base nessas duas adaptações, desenvolveu-se um novo módulo similar ao *newpoketool*, chamado de *funcpoketool*. Esse módulo executa basicamente as mesmas tarefas do módulo *newpoketool*; entretanto, para a geração dos conjuntos de elementos requeridos para cada critério selecionado e para a instrumentação do código-fonte, ele busca informações da MEFE que modela o comportamento do programa, ao invés do código-fonte do programa. Os outros dois módulos da ferramenta POKE-TOOL – *pokeexec* e *newpokeaval* não precisaram ser alterados.

Exemplos dos arquivos <*nome_função*>*.gfc* e *tabvardef.tes* e de um arquivo instrumentado são apresentados no Apêndice A.

5.3. Experimento Realizado

O experimento teve o objetivo de demonstrar a aplicação de alguns dos critérios de teste funcional definidos e comparar sua capacidade de cobertura do software em relação à aplicação dos critérios de teste estrutural correspondentes. Portanto, não se buscou por meio desse experimento detectar a presença de possíveis defeitos no programa utilizado, não sendo realizada, portanto, a injeção de defeitos no programa.

5.3.1. Descrição do Experimento

O experimento consistiu nos seguintes passos:

1) Implementar um sistema que simulasse a execução do sistema especificado pela MEFE Tele da Figura 4.6;
2) Projetar e executar casos de teste que atingissem 100% de cobertura para os critérios de teste funcional utilizados;
3) Comparar a quantidade de casos de teste necessária para atingir 100% de cobertura de cada um dos critérios de teste funcional utilizados;
4) Executar os casos de teste do passo 2 (100% de cobertura para os critérios de teste funcional) para medir a cobertura dos critérios de teste estrutural;
5) Comparar a cobertura atingida para os critérios de teste funcional e para os critérios de teste estrutural com o mesmo conjunto de casos de teste.

Foram utilizados seis dos oito critérios suportados pela ferramenta POKE-TOOL: Todos_ _os_estados e Todas_as_transições (chamados, respectivamente, de Todos_os_nós e Todos_ _os_arcos no teste estrutural); os critérios de Rapps & Weyuker – Todos_os_p-usos e Todos_ _os_usos; e os critérios Potenciais Usos – Todos_potenciais_usos e Todos_potenciais_usos/ /du. Os outros dois critérios não foram utilizados devido à grande dificuldade em satisfazê-los.

Um caso de teste derivado a partir de uma MEFE não pode ser descrito apenas por um valor de entrada (ou um conjunto de valores) que deve ser aplicado ao programa sob teste e um valor de saída (ou um conjunto de valores) esperado como resultado do processamento. Ao invés disso, o caso de teste é descrito por um caminho a ser percorrido na MEFE (representado por uma seqüência de transições através dos estados), mais os valores de entrada necessários para que o caminho seja percorrido e os valores de saída que devem ser produzidos pela execução do caminho.

A transição (*Realizar Outra Ligação?*, *Mudar Forma de Pagamento?*) da MEFE Tele permite que seja definido um caminho no modelo que possua um laço infinito. Assim, apenas um caso de teste poderia cobrir todos os elementos requeridos de alguns critérios, não servindo como métrica de comparação entre eles. Para que a comparação pudesse ser realizada, foi imposta uma restrição para a definição dos casos de teste a serem utilizados no experimento. Eles devem ser definidos de modo a conter o menor número possível de iterações do laço. As iterações do laço apenas devem estar contidas em casos de teste quando elas forem estritamente necessárias para exercitar algum elemento requerido por um critério de teste.

5.3.2. Resultados do Experimento

Durante a primeira parte do experimento, considerando apenas os critérios de teste funcional, foram executados 25 casos de teste. Essa quantidade de casos de teste foi necessária para que todos os elementos requeridos pelos seis critérios de teste funcional fossem exercitados (cobertura de 100%). A Tabela 1 apresenta a quantidade de elementos requeridos e a quantidade de casos de teste necessários para a satisfação de cada critério de teste funcional.

Tabela 5.1 – Quantidade de Casos de Teste Necessários por Critério

Critério	Nº de Elementos Requeridos	Nº de Casos de Teste
Todos_os_estados	14	3
Todas_as_transições	17	6
Todos_os_p-usos	28	8
Todos_os_usos	48	8
Todos_potenciais_usos	185	23
Todos_potenciais_usos/du	185	25

Esses resultados indicam que, para o programa testado, os critérios de teste funcional possuem diferentes graus de facilidade de cobertura dos elementos requeridos, assim como acontece com os critérios de teste estrutural. Por exemplo, o critério Todos_os_estados requer 14 elementos, precisando de apenas três casos de teste para ser satisfeito; o critério Todos_potenciais_usos/du requer 185 elementos, precisando de 25 casos de teste para ser satisfeito. Isto indica que o critério Todos_potenciais_usos/du é muito mais exigente do que o critério Todos_os_estados.

Rapps & Weyuker (1985) e Maldonado (1991) utilizam a relação de "inclusão estrita" entre os critérios de teste estrutural que demonstra que alguns critérios cobrem todos os elementos requeridos por outros critérios. Dado o mapeamento direto entre esses critérios de teste estrutural e os critérios de teste funcional definidos aqui, as mesmas relações demonstradas para os critérios de teste estrutural devem valer para os critérios de teste funcional.

Dos resultados do experimento não se pode concluir que os critérios que precisaram de um maior número de casos de teste para serem satisfeitos possuem maior chance de revelar a presença de defeitos. A única conclusão que se pode tirar dos resultados é que os critérios

mais difíceis de serem satisfeitos precisam de um maior número de casos de teste, ou seja, exercitam mais o programa em teste e, portanto, servem como parâmetro para indicar o quão exaustivamente o programa foi testado. Entretanto, por serem mais exigentes e exercitarem mais o programa, suas satisfações, quando não detectam a presença de defeitos, confere ao software testado uma maior indicação de confiabilidade e de qualidade, em relação a satisfação dos critérios menos exigentes.

Na segunda parte do experimento foram considerados os critérios de teste estrutural correspondentes aos critérios de teste funcional. Os mesmos 25 casos de teste projetados e executados na primeira parte do experimento foram executados sobre o programa que implementa a especificação, realizando-se a análise de cobertura em relação aos critérios de teste estrutural. A Tabela 2 apresenta a quantidade de elementos requeridos, a quantidade de casos de teste executados (os mesmos casos de teste usados para atingir 100% de cobertura dos critérios de teste funcional) e a cobertura atingida para cada critério de teste estrutural.

Tabela 5.2 – Cobertura Atingida por Critério

Critério	**Nº de Elementos Requeridos**	**Nº de Casos de Teste**	**Cobertura Atingida**
Todos_os_nós	78	3	78%
Todos_os_arcos	93	6	74%
Todos_os_p-usos	90	8	80%
Todos_os_usos	107	8	69%
Todos_potenciais_usos	384	23	81%
Todos_potenciais_usos/du	384	25	80%

Esses resultados indicam que, para o programa testado, os critérios de teste funcional são menos exigentes do que os critérios de teste estrutural correspondentes sendo, portanto, mais fáceis de serem satisfeitos. Em média, os casos de teste cuja execução atinge 100% de cobertura dos critérios de teste funcional, atinge apenas 77% de cobertura dos critérios de teste estrutural correspondentes. Este resultado é explicado pelo fato de que normalmente a implementação de um sistema precisa levar em consideração mais detalhes do que a própria especificação.

5.4. Considerações Finais

O experimento realizado teve apenas o objetivo de demonstrar a aplicação dos critérios de teste funcional definidos e de realizar uma primeira comparação experimental entre eles e os critérios de teste estrutural utilizados em suas definições. Não era, portanto, objetivo desse experimento a comprovação empírica da eficácia dos critérios de teste funcional ou de uma suposta relação de inclusão entre os critérios.

Para a realização desse experimento, a ferramenta POKE-TOOL precisou ser adaptada para suportar também os critérios de teste funcional definidos. Os resultados indicam que, para o programa testado, existem diferentes graus de facilidade de cobertura dos elementos requeridos pelos critérios de teste funcional. Além disso, que os critérios de teste funcional são menos exigentes do que os critérios de teste estrutural correspondentes sendo, portanto, mais fáceis de serem satisfeitos.

O próximo capítulo apresenta a conclusão deste trabalho, incluindo sugestões de trabalhos futuros que devem ser conduzidos na área de TBM, usando as MEFEs como técnica de modelagem.

Capítulo 6

Conclusão

As empresas produtoras de software têm enfrentado um grande desafio – a realização de testes bem sucedidos que ofereçam boas indicações de confiabilidade e de qualidade do software que está sendo produzido. A maior parte dessas empresas busca alcançar esses resultados por meio da aplicação de técnicas de teste funcional, as quais são baseadas nos requisitos funcionais do software e possuem menor custo de aplicação. Entretanto, a obtenção dos resultados desejados está ligada a uma execução sistemática dos testes, por meio do uso de técnicas definidas de forma mais rigorosa, o que geralmente não é oferecido pelas técnicas de teste funcional.

Desse modo, a indústria de software tem necessidade de investir em técnicas de teste funcional que possam ser definidas de forma mais rigorosa e usadas de forma sistemática. Nesse contexto, uma técnica que apresenta essas características é o Teste Baseado em Modelos (TBM). Essa é uma técnica funcional que já vem sendo bastante explorada na área de teste de software e apresenta uma série de vantagens em relação a outras técnicas de teste. Entretanto, como qualquer outra técnica de teste, essa abordagem não deve ser encarada com uma solução para todos os problemas da área, embora ela ofereça uma promessa considerável na redução do custo da geração de teste, no aumento da eficiência dos testes, e na redução do ciclo de teste.

O TBM direciona a realização dos testes com base em informações descritas em modelos comportamentais do software, os quais são construídos a partir de seus requisitos funcionais. As informações contidas nesses modelos determinam basicamente quais são as ações possíveis durante a execução de um software e quais são as saídas esperadas. É a partir dessas informações que os casos de teste são gerados e executados e os resultados dessa execução são avaliados.

A criação de modelos comportamentais do software pode ser realizada por meio da aplicação de várias técnicas de modelagem, como, por exemplo, as MEFs. Associadas a cada uma dessas técnicas de modelagem, existe um conjunto de critérios de teste definidos. Esses critérios são baseados nos tipos de estruturas que são usados na construção de cada tipo de modelo. Basicamente, esses critérios referem-se aos caminhos que devem ser percorridos desde um ponto inicial do modelo, até um ponto final, visando cobrir um determinado conjunto de elementos requeridos.

As MEFs são a técnica de modelagem mais utilizada atualmente no TBM. Apesar das vantagens que essa técnica oferece, como simplicidade e baixo custo de aplicação, ela possui limitações em relação aos seus mecanismos de modelagem, como a impossibilidade de se modelar, por exemplo, concorrência, hierarquia, comunicação, paralelismo, etc. Embora outras técnicas de modelagem, como *Statecharts*, Redes de Petri, SDL e Estelle, possuam menos limitações que as MEFs, elas ainda não são muito utilizadas no TBM.

Uma das principais limitações das MEFs é que essa técnica não possibilita a modelagem de fluxo de dados do software. Assim, os critérios de teste associados aos modelos criados por meio das MEFs não podem utilizar informações associadas ao fluxo de dados.

Levando-se em consideração a falta de rigor na definição das técnicas e critérios de teste funcional, a contribuição deste trabalho para a área de teste de software foi a definição de um novo conjunto de critérios formais para ser aplicado no TBM. Esses critérios foram definidos com base em uma extensão das MEFs tradicionais que oferece suporte à modelagem de fluxo de dados. As outras limitações das MEFs tradicionais não foram tratadas neste trabalho.

A estratégia para a definição desses critérios teve como ponto de partida a definição da extensão das MEFs tradicionais, de modo que fosse possível modelar também o fluxo de dados por meio delas. A definição das MEFs Estendidas (MEFEs) foi feita de modo a se obter uma equivalência estrutural entre seus elementos e os elementos de grafos de programas. Assim, foi possível a definição dos critérios baseados em MEFEs com base em critérios das técnicas baseadas em fluxo de controle e em fluxo de dados do teste estrutural, os quais já estavam definidos.

Enquanto os critérios baseados em análise de fluxo de controle de MEFs exigem que elementos como comandos, arcos e laços do modelo sejam exercitados, os critérios baseados em análise de fluxo de dados se focam em como são atribuídos valores às variáveis da MEF Estendida que regulam o comportamento do software e em como essas variáveis são usadas. A utilização deste amplo conjunto de critérios oferece vários graus de cobertura do modelo comportamental do software. Assim, está preenchida a lacuna entre os critérios baseados em MEFs tradicionais que são ou muito exigentes, como Todos_os_caminhos, ou pouco exigentes, como Todos_os_estados e Todas_as_transições.

A natureza estritamente funcional dos critérios de teste funcional baseados em MEFEs permite que os casos de teste sejam projetados antes mesmo que o software seja implementado, com base em seu modelo comportamental; enquanto os critérios de teste estrutural só podem ser aplicados para a seleção de casos de teste depois que o software já foi implementado e, conseqüentemente, já possui sua estrutura interna definida.

Assim, apesar de os critérios de teste funcional definidos serem estruturalmente similares aos critérios de teste estrutural nos quais eles são baseados, existe uma diferença essencial no resultado de suas aplicações. Enquanto os critérios de teste estrutural exigem que elementos internos da estrutura do software sejam exercitados para sua satisfação, os critérios de teste funcional exigem que elementos comportamentais do software sejam exercitados. Portanto, a aplicação de critérios de teste estrutural ou de critérios de teste funcional pode revelar a presença de diferentes tipos de defeitos no software, ou seja, são critérios de teste complementares.

Normalmente, a estrutura do software é mais complexa do que sua própria especificação, pois a implementação de um sistema precisa levar em consideração mais detalhes do que os definidos na especificação. Essa diferença de complexidade pode fazer com que a quantidade necessária de casos de teste para satisfazer os critérios de teste funcional baseados em MEFEs seja menor do que a quantidade necessária para satisfazer os critérios de teste estrutural baseados na estrutura interna do software. Essa relação foi mostrada através dos resultados do experimento descrito neste trabalho, que comparou empiricamente a relação entre a aplicação, em um mesmo programa, dos critérios de teste funcional e dos respectivos critérios de teste estrutural.

6.1. Trabalhos Futuros

Como trabalhos futuros, dois outros aspectos devem ser investigados:

- Análise mais detalhada dos tipos de defeitos mais suscetíveis de terem a presença detectada por meio da aplicação dos critérios de teste funcional baseados em MEFEs e o custo da aplicação de tais critérios;
- Desenvolvimento de uma ferramenta para dar suporte completo ao teste baseado em MEFEs. Assim, além do suporte oferecido pela extensão da POKE-TOOL para a análise de adequação de casos de teste em relação aos critérios de teste funcional, esta ferramenta deveria suportar: a modelagem do comportamento do software por meio de MEFEs; a geração de casos de teste com base no modelo criado e nos critérios existentes; e a execução e a avaliação de resultados de forma automática;

Referências Bibliográficas

Apfelbaum, L. & Doyle, J. (Maio de 1997). Model-based testing. Proceedings of the *Software Quality Week Conference*.

British Computer Society Specialist Interest Group in Software Testing. (2001). Standard for software componet testing. Disponível em: http://www.testingstdards.co.uk.

Beizer, B. (1990). *Software testing techniques*. Segunda Edição. Nova York.

Beizer, B. (1995). *Black-box testing – Techniques for functional testing of software and systems.* John Wiley & Sons, International Thomson Computer Press.

Berger, B.; Abuelbassal, M. & Hossain, M. (Março de 1997). Model driven testing. DNA Enterprises, Inc., Disponível em: www.model-based-testing.com.

Borrione, D.; Dushina, J. & Pierre, L. (1998). Formalization of finite state machines with data path for the verification of high-level synthesis. Université de Provence, França.

Bourhfir, C.; Dssouli, R.; Aboulhamid, E. M. (1996). Automatic test generation for EFSM-based systems. Publication Departamentale #1043, Disponível em: www.umontreal.ca/labs/teleinfo/publistindex.html.

Bueno, P. M. S., Chaim, M. L., Maldonado, J. C., Jino, M. & Vilela, P. R. S. (1995). Manual do usuário da POKE-TOOL. Relatório Técnico, DCA/FEEC, UNICAMP, Campinas, SP.

Budkowski, T. & Dembinski, P. (1987). An introduction to ESTELLE: A specification language for distributed systems. *Computer Networks and ISDN*, Vol. 14, No. 1.

Candolo, M. A. P., Simão, A. S. & Maldonado, J. C. (2001). MGASet java – Uma ferramenta para apoiar o teste e validação de especificações baseadas em máquinas de estado finitos. DC, ICMC, USP, São Carlos, SP.

Chaim, M. L. (Abril de 1991). POKE-TOOL – Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Dissertação de Mestrado, DCA/FEEC, UNICAMP, Campinas, SP.

Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE(4(3)).

Clarke, L. A.; Podgurski, A; Richardson, D. J & Zeil, S. J. (Novembro de 1989). A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, SE(15(11)).

Clarke, J. M. (Maio de 1998). Automated test generation from a behavioral model. Proceedings of the *Software Quality Week Conference*.

Dalal, S. R.; Jain, A.; Karunanithi, N.; Leaton, J. M. & Lott, C. M. (Novembro de 1998). Model-based testing of a highly programmable system. Proceedings of the *International Simposium on Software Reability Enginerring 1998*, pp.174-178, IEEE Computer Society Press.

Dalal, S. R.; Jain, A.; Karunanithi, N.; Leaton, J. M.; Lott, C. M.; Patton, G. C. & Horowitx, B. M. (Maio de 1999). Model-based testing in practice. Proceedings of the *International Conference on Software Engineering 1999*, ACM Press.

El-Far, I. K. I. (Maio de 1999). Automated construction of software behavior models. Tese de Mestrado, Florida Institure of Technology, Melbourne, Florida. Disponível em: www.model-based-testing.com.

Fujiwara, S.; Bochmann, G. V.; Khendek, F.; Amalou, M. & Ghedamsi, A. (Junho de 1991). Test selection based on finite state models, *IEEE Transactions on Software Engineering*, SE(17(6)).

Gill, A. (1962). *Introduction to the theory of finite state machines*. McGraw-Hill Electronic Science Series.

Gonenc, G. (Junho de 1970). A method for the design of fault-detection experiments. *IEEE Transactions on Computers*, Vol C-19, pp 551-558.

Gronau, I.; Hartman, A.; Kirshin, A.; Nagin, K. & Olvovsky, S. (2000). A methodology and architecture for automated software testing. IBM Research Laboratory in Haifa. Technical Report. Disponível em: www.model-based-testing.com.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, Vol 8, pp 231-274.

Harel, D (Junho de 1987). On visual formalisms. *Rehovot: The Weizmann Inst. Of Science*.

IEEE-AS Standards Board. (1990). IEEE Std 610.12-1990 – IEEE standard glossary of software engineering terminology. Software Engineering Technical Committee of the IEEE Computer Society.

ITU-T. (1998). ITU-T recommendation z.100: Specification and de*script*ion language (SDL). ITU-T, Genebra. Mais informações podem ser encontradas Em http://www.sdl-forum.org.

Kohavi, Z. (1978). *Switching and finite automata theory.* McGraw-Hill, New York.

Laski, J. W. & Korel, B. (Maio de 1983). A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE(9(3)).

Lee, D. & Yannakakis, M. (1995). Principles and methods of testing finite state machines. AT&T Bell Laboratories, Murray Hill, New Jersey.

Maldonado, J. C. (Julho de 1991). Critérios potenciais–usos: Uma contribuição ao teste estrutural de software. Tese de doutorado, DCA/FEEC, UNICAMP, Campinas, SP.

Martins, E., Sabião, S. B. & Ambrosio, A. M. (2000). Condata: A tool for automating specification-based test case generation for communication systems. Proceedings of *The 33rd Hawaii International Conference on System Sciences*.

Miller, R. E. & Paul, S. (1992). Generating conformance test sequences for combined control and data flow of communication protocols. Proceedings of the *Protocol Specification, Testing And Verification* (PSTV'92), Florida, EUA.

Myers, G. (1979). *The art of software testing*. John Wiley & Sons.

Naito, S. & Tsunoyama, M. (1981). Fault detection for sequenctial machines by transition-tours, Proceedings of the *FTCS – Fault Tolerant Computer Systems*, pp. 238-243.

Nakazato, K. K., Alexandrino, M., Maldonado, J. C., Fabbri, S. C. P. F. & Masiero, P. C. (1995). MGASet – Módulo de geração de seqüências de teste. Anais da Sessão de Ferramentas do *IX Simpósio Brasileiro de Engenharia de Software*, pp. 479-482, Recife, PE.

Ntafos, S. C. (Novembro de 1984). On required element testing. *IEEE Transactions on Software Engineering*, SE(10(6)).

Ntafos, S. C. (Julho de 1988). A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering,* SE(14(6)).

Offutt, A. J.; Liu, S. & Abdurazik, A. (2000). Generating test data from state-based specifications. Proceedings of *The Jornal Of Software Testing, Verification And Reliability - JSTVR.*

Peterson, J. L. (1981). *Petri net theory and the modelling of system*. Prentice-Hal.

Pressman, R. S. (1992). *Engenharia de software*. Makron Books do Brasil Editora Ltda.

Rapps, S. & Weyuker, E. J. (Abril de 1985). Selecting software testing data using data flow information. *IEEE Transactions on Software Engineering*, SE(11(4)).

Robinson, H. (Novembro de 1999). Finite state model-based testing on a shoestring. Proceedings of the *Software Testing Analysis And Review Conference*, San Jose, CA, EUA.

Robinson, H. (1999). Graph theory techniques in model-based testing. Proceedings of the *International Conference on Testing Computer Software*.

Robinson, H. & Rosaria, S. (2000). Applying models in your testing process. *Information and Software Technology,* 42, pp.815-824.

Robinson, H. (Setembro/Outubro de 2000). Intelligent test automation. *Software Testing & Quality Engineering Magazine*, pp. 24-32.

Rocha, A. R. C.; Maldonado, J. C. & Weber, K. C. (2001*). Qualidade de software – Teoria e prática*. Prentice Hall.

Sabnani, K. K. & Dahbura, A. T. (1988). A protocol testing procedure, *Computer Networks and ISDN System*, Vol. 15, N. 4, pp. 285-297.

Sato, F., Munemori, J., & Mizuno, T. (1989). Test sequence generatin method based on finite automata – Single transition checking method using W set, *Transactions on EIC*, Vol. J72-B-I, No. 3, pp. 183-192.

Shehady, R. K. & Siewiorek, D. P. (1997). A method to automate user interface testing using variable finite sate machines. Proceedings of *The 27th International Symposium on Fault-Torerant Computing – FTCS'97.*

Spiveu, M. (1992). *The Z notation: A reference manual.* Segunda Edição, Prentice-Hall International.

Software Program Managers Network. (Junho de 1998). *Little book of testing.* Volume I, Overview and Best Practices, Computers & Concepts Associates.

Tan, Q. M., Petrenko, A. & Bochmann, G. V. (1995). A test generation tool for specifications in the form of state machines. Department Díro, Université de Montreal, Canadá.

UML. (2003). Glossário da UML. Disponível em: http://usuarios.dialdata.com.br /deboni/glossar.htm.

Ural, H. & Yang, B. (Abril de 1991). A test sequence selection method for protocol testing. *IEEE Transactions on Communications*, Vol 39, Nº 4.

Vuong, S. T., Chan, W. W. K, & Ito, M. R. (Outubro de 1989). The UIOv-method for protocol test sequence generation, Proceedings of the *2nd Int. Workshop on Protocol Test Systems*, Berlim.

Weyuker, E. J. & Jeng, B. (Julho de 1991). Analyzing partition testing strategies. *IEEE Transactions on Software Engineeering*, SE(17(7)).

Apêndice A

Arquivos da POKE-TOOL

Este apêndice apresenta exemplos de arquivos manipulados pela ferramenta de teste POKE-TOOL. São apresentados três arquivos: o arquivo tele.gfc, que contém informações sobre o fluxo de controle da MEFE Tele; o arquivo tabvardef.tes, que contém informações sobre o fluxo de dados da MEFE Tele; e o arquivo testprog.c., que contém o código-fonte que implementa a MEFE Tele instrumentado. Os dois primeiros são gerados automaticamente por um módulo da ferramenta POKE-TOOL a partir da MEFE Tele, para depois serem usados por um outro módulo. E o terceiro é gerado manualmente para ser usado pela ferramenta POKE-TOOL.

A.1. tele.gfc

```
14

1
  2 0
2
  3 4 5 14 0
3
  2 3 5 0
4
  2 4 5 0
5
  6 7 0
6
  7 12 0
7
  8 0
8
  9 0
9
  10 12 0
```

```
10
  11 12 0
11
  12 0
12
  13 14 0
13
  2 5 0
14
  0
```

A.2. tabvardef.tes

```
# Tabela de Variaveis Definidas do Modulo main

$ 14
@ 0 f_p
@ 1 n_c
@ 2 senha
@ 3 res
@ 4 n_t
@ 5 tarifa
@ 6 o_r
@ 7 ti
@ 8 tf
@ 9 t
@ 10 v
@ 11 op1
@ 12 op2
@ 13 op3
@@

# Grafo Def Sintetico do Modulo main

@@ 1
Defs: @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 2
Defs: 0 @
C-Uses: @
P-Uses: 0 @
Refs: @
Undefs: @
@@ 3
Defs: 1 2 3 @
C-Uses: 1 2 @
P-Uses: 3 @
Refs: @
Undefs: @
@@ 4
```

```
Defs: 1 3 @
C-Uses: 1 @
P-Uses: 3 @
Refs: @
Undefs: @
@@ 5
Defs: 4 5 @
C-Uses: 4 @
P-Uses: 0 @
Refs: @
Undefs: @
@@ 6
Defs: 6 @
C-Uses: @
P-Uses: 6 @
Refs: @
Undefs: @
@@ 7
Defs: 7 @
C-Uses: @
P-Uses: @
Refs: @
Undefs: @
@@ 8
Defs: 8 9 10 @
C-Uses: 5 7 8 9 @
P-Uses: @
Refs: @
Undefs: @
@@ 9
Defs: @
C-Uses: 0 1 4 7 9 10 @
P-Uses: 0 @
Refs: @
Undefs: @
@@ 10
Defs: 11 @
C-Uses: @
P-Uses: 11 @
Refs: @
Undefs: @
@@ 11
Defs: @
C-Uses: 4 9 10 @
P-Uses: @
Refs: @
Undefs: @
@@ 12
Defs: 12 @
C-Uses: @
P-Uses: 12 @
Refs: @
Undefs: @
@@ 13
```

```
Defs: 13 @
C-Uses: @
P-Uses: 13 @
Refs: @
Undefs: @
```

A.3. testprog.c

```
/* Instrumentação */
#include "poketool.h"
/* Instrumentação */

#include <stdio.h>
#include <time.h>

void inicio_da_sessao();
char determinar_forma_de_pagamento();
void obter_numero_do_cartao_telefonico();
void obter_numero_do_cartao_de_credito();
void realizar_ligacao();
float definir_tarifa();
int ligacao_a_cobrar();
void conversacao();
void terminar_ligacao();
void enviar_dados_para_bd();
int obter_opcao_do_usuario(char *);
void apresentar_dados();
void fim_da_sessao();

char n_cartao[16];
char n_telefone[16];
float tarifa;
time_t t_inicial, t_final;
float t_total;
int h, m, s;
float valor_ligacao;
char senha[4];
int res_cartao;
int res_senha;
int i;
int res = 0;

const tam_cartao_telefonico = 6;
const tam_cartao_credito = 16;
const char * senhas[10][2] = {{"111111", "aaaa"},
                              {"222222", "bbbb"},
                              {"333333", "cccc"},
                              {"444444", "dddd"},
                              {"555555", "eeee"},
                              {"666666", "ffff"},
                              {"777777", "gggg"},
                              {"888888", "hhhh"},
                              {"999999", "iiii"},
```

```
                              {"000000", "jjjj"}};

main()
{

/* Instrumentação */
FILE * path = fopen("tele/path.tes","a");
static int printed_nodes = 0;
/* Instrumentação */

      char pag;
      int acao_receptor;
      int ligacao_completada;
      int op1, op2, op3;

/* Instrumentação */
ponta_de_prova(1);
/* Instrumentação */

      inicio_da_sessao();
      do
      {
            do
            {

/* Instrumentação */
ponta_de_prova(2);
/* Instrumentação */

                  pag = determinar_forma_de_pagamento();
                  switch (pag)
                  {
                        case '1' :
                                    res_cartao = 0;
                                    res_senha = 0;
                                    i = 0;
                                    printf("\nDigite o número de seu cartão
telefônico [Digite '0' para Cancelar]: ");
                                    do
                                    {

/* Instrumentação */
ponta_de_prova(3);
/* Instrumentação */

                                          obter_numero_do_cartao_telefonico();
                                    } while (res_senha);
                                    break;
                        case '2' :
                                    res = 1;
                                    printf("\nDigite o número de seu cartão de
crédito [Digite '0' para Cancelar]: ");
                                    do
                                    {
```

```
/* Instrumentação */
ponta_de_prova(4);
/* Instrumentação */

                                        obter_numero_do_cartao_de_credito();
                                    } while ((strlen(n_cartao) !=
tam_cartao_credito) && (res));
                                    break;
                            case '3' : break;
                            case 'c' :
                            case 'C' :

/* Instrumentação */
ponta_de_prova(14);
/* Instrumentação */

                                    fim_da_sessao();
                        }
                    } while (!strcmp(n_cartao, "0"));
                    do
                    {

/* Instrumentação */
ponta_de_prova(5);
/* Instrumentação */

                        realizar_ligacao();
                        if (pag == '1' || pag == '2')
                        {

/* Instrumentação */
ponta_de_prova(7);
/* Instrumentação */

                            conversacao();
                            ligacao_completada = 1;
                        }
                        else
                        {

/* Instrumentação */
ponta_de_prova(6);
/* Instrumentação */

                            acao_receptor = ligacao_a_cobrar();
                            if (acao_receptor == 1)
                            {

/* Instrumentação */
ponta_de_prova(7);
/* Instrumentação */

                                conversacao();
                                ligacao_completada = 1;
```

```
                        }
                        else
                        {
                              ligacao_completada = 0;
                        }
                  }
                  if (ligacao_completada)
                  {

/* Instrumentação */
ponta_de_prova(8);
/* Instrumentação */

                        terminar_ligacao();

/* Instrumentação */
ponta_de_prova(9);
/* Instrumentação */

                        enviar_dados_para_bd();
                        if (pag == '1' || pag == '2')
                        {

/* Instrumentação */
ponta_de_prova(10);
/* Instrumentação */

                              op1 = obter_opcao_do_usuario("\nDeseja saber
os dados sobre esta ligação");
                              if (op1)
                              {

/* Instrumentação */
ponta_de_prova(11);
/* Instrumentação */

                                    apresentar_dados();
                              }
                        }
                  }

/* Instrumentação */
ponta_de_prova(12);
/* Instrumentação */

                  op2 = obter_opcao_do_usuario("\nDeseja realizar outra
ligação");
                  if (op2)
                  {

/* Instrumentação */
ponta_de_prova(13);
/* Instrumentação */
```

```
                        op3 = obter_opcao_do_usuario("\nDeseja mudar a forma
de pagamento");
                  }
            } while (op2 && (!op3));
      } while (op2 && op3);

/* Instrumentação */
ponta_de_prova(14);
/* Instrumentação */

      fim_da_sessao();
}

void inicio_da_sessao()
{
      printf("\nBem Vindo ao Sistema Automático de Ligações
Telefônicas!\n");
}

char determinar_forma_de_pagamento()
{
      char opcao;
      printf("\nEscolha a forma de pagamento desta ligação.");
      printf("\n [1] - Cartão Telefônico");
      printf("\n [2] - Cartão de Crédito");
      printf("\n [3] - A Cobrar");
      printf("\n [c] - Cancelar");
      do
      {
            printf("\nOpção: ");
            scanf("%s", &opcao);
      } while (!strchr("123cC", opcao));
      return opcao;
}

void obter_numero_do_cartao_telefonico()
{
            i = 0;
            res_cartao = 1;
            res_senha = 1;
            scanf("%s", n_cartao);
            if (!strcmp(n_cartao, "0"))
            {
                  res_senha = 0;
                  return;
            }
            if (strlen(n_cartao) != tam_cartao_telefonico)
            {
                  printf("\nNúmero do cartão inválido.");
                  printf("\nDigite novamente o número de seu cartão
telefônico [Digite '0' para Cancelar]: ");
            }
            else
            {
```

```
                  printf("Digite sua senha: ");
                  scanf("%s", senha);
                  do
                  {
                        res_cartao = strcmp(senhas[i][0], n_cartao);
                        i++;
                  } while (res_cartao && i < 10);
                  if (!res_cartao)
                  {
                        res_senha = strcmp(senhas[i-1][1], senha);
                  }
                  if (res_senha)
                  {
                        printf("\nSenha inválida.");
                        printf("\nDigite novamente o número de seu cartão
telefônico [Digite '0' para Cancelar]: ");
                  }
            }
}

void obter_numero_do_cartao_de_credito()
{
            scanf("%s", n_cartao);
            if (!strcmp(n_cartao, "0"))
            {
                  res = 0;
                  return;
            }
            if (strlen(n_cartao) != tam_cartao_credito)
            {
                  printf("\nNúmero do cartão inválido.");
                  printf("\nDigite novamente o número de seu cartão de
crédito [Digite '0' para Cancelar]: ");
            }
}

void realizar_ligacao()
{
      int res = 0;
      printf("\nDigite o numero de telefone desejado: ");
      scanf("%s", n_telefone);
      tarifa = definir_tarifa();
      printf("\n(SIMULAÇÃO) - Discando...");
      sleep(2);
      printf("\n(SIMULAÇÃO) - Chamando...");
      sleep(3);
      printf("\n(SIMULAÇÃO) - Atendeu");
}

float definir_tarifa()
{
      if (strlen(n_telefone) == 3)
      {
            return 0;
```

```
	}
	else if (strlen(n_telefone) == 7 || strlen(n_telefone) == 8)
	{
		return 0.05;
	}
	else
	{
		return 0.15;
	}
}

int ligacao_a_cobrar()
{
	char res;
	printf("\n(SIMULAÇÃO) - Receptor aceita ligação a cobrar [S/N]: ");
	scanf("%s", &res);
	if (strchr("sS", res))
	{
		return 1;
	}
	else
	{
		return 0;
	}
}

void conversacao()
{
	t_inicial = time(NULL);
	printf("\n\nTecle <Enter> para terminar a ligação");
	getchar();
	getchar();
}

void terminar_ligacao()
{
	int resto;
	int tempo;
	t_final = time(NULL);
	t_total = difftime(t_final, t_inicial);
	valor_ligacao = (t_total/60) * tarifa;
	if (valor_ligacao > 0 && valor_ligacao < 0.01)
	{
		valor_ligacao = 0.01;
	}
	tempo = t_total;
	h = tempo / 3600;
	resto = tempo % 3600;
	m = resto / 60;
	s = resto % 60;
}

void enviar_dados_para_bd()
{
```

```
      printf("\n(SIMULAÇÃO) - Dados foram enviados para o banco de dados.");
}

int obter_opcao_do_usuario(char * msg)
{
      char res;
      printf("\n%s [s/n]? ", msg);
      scanf("%s", &res);
      if (strchr("sS", res))
      {
            return 1;
      }
      else
      {
            return 0;
      }
}

void apresentar_dados()
{
      printf("\nNúmero de telefone ligado: %s", n_telefone);
      printf("\nTempo da ligação: %dh %dmin %ds", h, m, s);
      printf("\nValor cobrado da ligação: R$ %1.2f", valor_ligacao);
}

void fim_da_sessao()
{
      printf("\nObrigado por ter usado o Sistema Automático de Ligações
Telefônicas!\n\n");
      exit(0);
}
```