Geração Automática de Cenários de Teste a partir de Modelos da Especificação de Sistemas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Ivan Rodolfo Duran Cruz Perez e aprovada pela Banca Examinadora.

Campinas, 11 de Fevereiro de 2008.

Eliane Martins

Profª. Drª. Eliane Martins
Instituto de Computação – UNICAMP
(Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues CRB8a / 2116

Perez, Ivan Rodolfo Duran Cruz
P415g Geração automática de cenários de teste a partir de modelos da especificação de sistemas /Ivan Rodolfo Duran Cruz Perez -- Campinas, [S.P. :s.n.], 2008.

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

1. Engenharia de software. 2. Software - Testes. 3. Software - Avaliação. 4. Modelos matemáticos. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Automatic generation of test scenarios from the models of systems specification.

Palavras-chave em inglês (Keywords): 1. Software engineering. 2. Software – Testing. 3. Software – Evaluation. 4. Mathematical models.

Área de concentração: Engenharia de Software

Titulação: Mestre em Ciência da Computação

Banca examinadora:

Profa. Dra. Eliane Martins (IC-UNICAMP)
Prof. Dr. Adenilson da Silva Simão (ICMC-USP-São Carlos)
Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)

Data da defesa: 11/02/2008

Programa de pós-graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 11 de fevereiro de 2008, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Adenilso da Silva Simão
ICMC - USP / São Carlos.

Profª. Drª. Cecília Mary Fischer Rubira
IC - UNICAMP.

Profª. Drª. Eliane Martins
IC - UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Geração Automática de Cenários de Teste a partir de Modelos da Especificação de Sistemas

Ivan Rodolfo Duran Cruz Perez[1]

Fevereiro de 2008

Banca Examinadora:

- Prof^a. Dr^a. Eliane Martins
  Instituto de Computação – UNICAMP (Orientadora)
- Prof. Dr. Adenilso da Silva Simão
  ICMC – USP São Carlos
- Prof^a. Dr^a. Cecília Mary Fischer Rubira
  Instituto de Computação – UNICAMP
- Prof. Dr. Ricardo de Oliveira Anido (Suplente)
  Instituto de Computação – UNICAMP

[1]Suporte financeiro de: Bolsa do Capes em 2005, Projeto CompGov - Biblioteca Compartilhada de Componentes para E-Gov, FINEP, no. 1843/04, Projeto Harpia em 2006–2008 e Universidade Estadual de Campinas 2005–2008.

Resumo

As crescentes exigências em relação à melhoria de qualidade e a redução de custos e prazos têm tornado comum à busca por soluções mais eficientes para desenvolvimento e testes de sistemas. Com relação aos testes, uma recomendação é a de começá-los mais cedo, e, de preferência, automatizar o que for possível para evitar enganos cometidos pelos testadores. Assim, uma área de pesquisa que está em evidência atualmente é a de testes baseados em modelos (MBT), onde muitos estudos têm sido realizados visando o desenvolvimento de soluções para automação de testes a partir de modelos criados durante o ciclo de desenvolvimento. Nesta dissertação é proposto um método para geração automática de cenários de teste a partir de modelos da especificação de sistemas, baseando-se em um trabalho prévio para geração de testes de componentes de software. Os modelos utilizados para geração dos testes são os Diagramas de Atividades UML, criados a partir da descrição dos casos de uso, para Testes de Sistemas. A partir deles são gerados cenários de teste que descrevem as interações do sistema, tais como, as ações dos atores e as situações esperadas, incluindo também os cenários de exceção. A aplicação deste método na prática foi feita com êxito por uma equipe de testadores em sistemas reais. De forma geral, os modelos especificados para derivação dos testes têm facilitado o entendimento do sistema pelos testadores envolvidos e as informações presentes nos cenários de teste têm apoiado a realização dos testes.

Abstract

The increasing requirements for quality improvement, reduction of costs and deadlines have promoted the search for more efficient solutions for systems development and testing. In the case of the tests, the recommendation is to start them earlier and, preferably, automatize what is possible to prevent the testers mistakes. Thus, a research area that is in evidence currently is the Model-Based Testing (MBT), in which many studies have been carried out with the aim of development solutions for test automation from the models created during the software development cycle. In this dissertation is proposed a method for automatic generation of test scenarios from the models of systems specification, it is based on a previous work for tests generation of software components. The models used for tests generation are the UML Activities Diagrams, generated from the description of the use cases for System Testing. From them, test scenarios are generated that describe the interactions of the system, such as, the actors actions and the expected situations, including also the exception scenarios. The application of this method in practice was successfully made by a team of testers in real systems. In general, the models specified for tests derivation have facilitated the agreement of the system by the involved testers and, the generated test scenarios contain information that have supported the test execution.

Agradecimentos

Gostaria de agradecer a Deus por ter me dado a oportunidade de realizar este trabalho, que sem dúvida foi importante passo para meu amadurecimento e realização profissional. Aos meus pais que amo muito, Mauro e Ana Lucia, que apesar de estarem distantes fisicamente, sempre me apoiaram durante todo o caminho que levou a realização deste trabalho. Aos meus irmãos Cae, Júnior e Luciana, que amo muito. A minha namorada Amanda que amo muito e esteve ao meu lado, me ajudando e apoiando, sempre com muita paciência.

A Eliane Martins, uma excelente orientadora que esteve sempre com muito empenho e me oferecendo apoio durante todo o mestrado. Além dela, várias outras pessoas do grupo de pesquisa foram importantes durante a realização do mestrado entre elas, ou Bruno Tx., que também participou ativamente deste trabalho, sempre com muitas sugestões e questionamentos interessantes; A Regina, que me deu bastante apoio principalmente na fase final deste trabalho; O Júlio Viégas que ajudou muito em vários problemas e no desenvolvimento dos protótipos; A Ruth e Daniele que participaram nas definições inicias do protótipo para o projeto CompGOV; As demais pessoas do grupo de testes do projeto harpia Prof. Jino, Gabriela, Camila, Daniel, Erika e Júlio. Aos demais participantes do grupo de testes do IC Thaise, Camila Rocha e Elena.

Aos amigos em campinas, Bruno Albertini, Dadah, Frazão, Miguelito, Bolha, Sabrina, Leonel, Leo Tizzei, Patrick, Edna, Juliana, Neumar, Carlos, Moronte, Bruno Z., Otávio, Cubas, Ekler, Miani, Marlon, Norton, Cláudio, Augusto, Leo Fernandes, Cláudia, Vivi e Fernando. Enfim todos eles, incluindo os que não me lembro no momento. Aos amigos da faculdade em Campo Grande, Murphy, Kiba, Chermont, Gaúcho, Benini, Vini, Edgard, Marcio, Chambinho, Coin, Jony, X, Golão, Sorriso, Camilo, Felipe, Tiago, Bruce, Tigrão, Alexandre, Alemão, Carioca, Juliano, Marcel, Fernando, Livia, Janine, Paulo, Anderson, Rafael e outros! Devo agradecer também aos meus professores da graduação que me incentivaram a cursar o mestrado, principalmente ao Nalvo F. de Almeida e ao Henrique Mongelli.

Por último gostaria de agradecer a Unicamp, especialmente ao Instituto de Computação, por ter fornecido uma ótima infra-estrutura no decorrer deste mestrado. Ao

apoio financeiro recebido do CAPES, através de bolsa no inicio do mestrado, ao projeto FINEP CompGOV, e ao projeto Harpia da Receita Federal pelo apoio financeiro e por ter possibilitado o desenvolvimento da pesquisa e realização do estudo de caso no projeto.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Acrônimos

AC Aresta de Chamada

AR Aresta de Retorno

BFS *Breadth First Search*

CDU Caso de Uso

CFG *Control Flow Graph*

CT Caso de Teste

DA Diagrama de Atividades

DACI Diagrama de Atividades com Cenários Internos do Caso de Uso

DAFC Diagrama de Fluxo de Controle entre Casos de Uso

DBC Desenvolvimento Baseado em Componentes

DFS *Depth First Search*

IACFG *Inter-activity Control Flow Graph*

ICFG *Interprocedural Control Flow Graph*

IIFG *Interprocedural Inlined Flow Graph*

FSM *Finite-State Machine*

MBT *Model-Based Testing*

MDA *Model-Driven Architecture*

MDD *Model-Driven Development*

ODC *Orthogonal Defect Classification*

OMG *Object Management Group*

PIM *Platform Independent Model*

PSM *Platform Specific Model*

SUT *Software Under Test*

UML *Unified Modeling Language*

U2TP *UML 2.0 Testing Profile*

VC Vértice de Chamada

VR Vértice de Retorno

XMI *XML Metadata Interchange*

Capítulo 1

Introdução

As crescentes exigências em relação à melhoria de qualidade e a redução de custos e prazos têm tornado comum a busca por soluções que propiciem a reutilização e a automação durante o ciclo de desenvolvimento de *software*. Neste sentido, tanto as universidades quando a indústria de desenvolvimento de *software* têm apostado cada vez mais em abordagens de desenvolvimento dirigido a modelos (em inglês, *Model-Driven Development* – MDD) [MCF03, Sel03, KR03]. Um fator importante que motivou a aceitação de MDD foi o surgimento de padronizações para criação de modelos providas pela UML (em inglês, *Unified Modeling Language*) [Gro03b, Gro04].

Em MDD os modelos são a principal fonte de informação sobre o sistema e, durante as fases de desenvolvimento, eles podem ser refinados ou sofrer transformações dando origem a novos modelos [MCF03]. Além disso, os modelos podem ser utilizados para geração automática de código ou outros artefatos nas diversas fases do ciclo de desenvolvimento [AD97, MCF03]. Uma das fases na qual os modelos podem ser utilizados é a fase de testes de software, que consiste no processo de executar um sistema com o objetivo de encontrar defeitos (em inglês, *failures*) [Mye79, Pre97]. Durante os testes, os modelos são a principal forma de se obter informações sobre a especificação do sistema que está sendo desenvolvido, pois eles mantém somente os aspectos necessários em cada fase de desenvolvimento [AD97].

Neste cenário, uma área de concentração que está em evidência atualmente é a de testes baseados em modelos (em inglês, *Model-Based Testing* – MBT), onde muitos estudos têm sido realizados visando o desenvolvimento de soluções de automação de testes. A automação proporciona a redução do esforço empregado para realização dos testes e, além disso, proporciona a redução do número de enganos (erros) que podem ser cometidos pelos testadores durante a realização dessas atividades.

Em MBT, várias abordagens têm sido propostas para automação de testes [PAK+07, VLH+06, CCD03, BBM02, BL02, OA99] e, geralmente, o principal objetivo das aborda-

gens MBT é automatizar atividades de geração e execução de testes. Neste aspecto, os modelos podem ser utilizados em diversas fases de teste, durante os testes de unidade, componente, integração e sistemas. O foco desta dissertação é na fase de testes de sistema, onde o sistema deve ser testado para validar se sua implementação está de acordo com o que foi especificado ou requerido pelo cliente. Desta forma, devem ser avaliadas as funcionalidades do Sistema em Testes (em inglês, *Software Under Test* – SUT), o desempenho dele utilizando uma massa de dados e os demais requisitos de qualidade especificados [Pre01].

1.1 Contexto e Motivação

Um dos desafios em testes de software é como automatizar a geração e execução de testes. A geração de dados de teste em grande escala é uma atividade muito custosa e a especificação e execução de testes manualmente é propensa a erros ou enganos que podem ser cometidos pelos testadores. Além disso, tornar os testes independentes da implementação do sistema e antecipar as atividades de teste, principalmente os testes de sistema, que são baseados na especificação de requisitos do SUT também são desafios. Os requisitos do SUT geralmente são especificados (na fase de Requisitos) através de diagramas de caso de uso [Rum94, Gro03b, CL01, Coc01b]. Entretanto, os casos de uso geralmente são descritos de forma textual e podem conter ambigüidades [Gel04], o que pode dificultar a geração de testes. Além disso, os testes a partir de requisitos de sistema devem representar cenários de uso do sistema e não somente as interações de cada caso de uso isoladamente.

Uma alternativa à descrição textual dos casos de uso é a utilização de outros modelos para especificação e geração de testes. Entretanto, como criar modelos que possibilitem a realização de testes de forma eficiente no SUT? É importante ressaltar que existem vários aspectos que podem ser tratados e observados na criação dos modelos para o projeto e seleção de testes, tais como a representação do fluxo de controle do sistema, fluxo de dados, os cenários de exceção e cenários concorrentes. Além disso, quais destes aspectos e como eles devem ser apresentados nos modelos para possibilitar a seleção (ou geração) automática de testes a partir de modelos? A resposta para estas questões são tópicos de pesquisa e desafios em testes de software [Ber07].

Para geração de testes de sistemas, algumas abordagens utilizam modelos que mantêm somente informações da especificação, ou seja, independentes da plataforma de desenvolvimento (conhecidos em MDD como, *Platform Independent Model* – PIM). Assim, o conjunto de testes gerado a partir dos modelos também é independente da implementação do SUT. Uma vantagem nessas abordagens é que os testes gerados a partir dos modelos PIM podem ser utilizados para qualquer implementação do SUT e só é necessária a ma-

nutenção do conjunto de testes quando houver alterações na especificação do SUT. Por outro lado, devido a falta de informações sobre a plataforma de desenvolvimento nessas abordagens, nem sempre é possível automatizar a geração de dados de testes e a execução dos testes a partir destes modelos.

Em abordagens que utilizam modelos com informações da implementação do SUT para geração de testes, ou seja, modelos específicos da plataforma (em inglês, *Platform Specific Model* – PSM), a automação da execução dos teste é mais comumente empregada. Entretanto, não só alterações na especificação, mas também alterações na implementação do SUT, podem implicar na necessidade de manutenção do conjunto de testes gerado. Isso pode tornar a manutenção dos modelos e do conjunto de testes gerado muito custosa. Uma vantagem dos PIM com relação aos PSM é que as atividades de planejamento e projeto de testes podem ser realizadas de forma independente da implementação do SUT. Desta forma, uma questão importante é como gerar testes a partir de PIMs ou de PSM para realização de testes de sistema.

1.2 Objetivo

Na busca por respostas para as questões de pesquisa levantadas na Seção 1.1, esta dissertação apresenta um método para geração automática de testes de sistemas a partir da especificação do sistema. No método que será proposto aqui os testes são criados a partir de modelos que representam os casos de uso do SUT.

O método gera os casos de teste baseando-se nas informações obtidas nos cenários internos dos casos de uso e no fluxo de execução entre os casos de uso. O fluxo de controle entre os casos de uso é obtido a partir do processo de negócio do sistema. Os cenários internos são utilizando um padrão para descrição dos casos de uso, que visa reduzir as ambigüidades da especificação do sistema e dos testes gerados.

1.3 Solução Proposta

Os diagramas de casos de uso são utilizados para o detalhamento de requisitos do sistema, durante fase de Análise de Requisitos. O padrão para descrição dos casos de uso baseia-se no padrão proposto por Gelperin [Gel04] e os cenários de exceção dos casos de uso são classificados de acordo com a representação de cenários de exceção proposta por Ferreira [dMF01]. Assim, o método de teste pode ser iniciado após a fase de Análise de Requisitos do sistema.

Na primeira etapa do método de teste devem ser criados modelos para representação dos cenários internos dos CDUs e das dependências ou ordem de execução entre os CDUs.

Nos cenários internos são representados o fluxo de controle do CDU e os cenários de exceção especificados. A representação da ordem de execução entre os CDUs possibilita a obtenção dos cenários de uso do SUT. Os modelos adotados para representar os cenários internos e a ordem de execução dos CDUs foram os Diagramas de Atividades (DAs).

Os Diagrama de Atividadess são usados para representar seqüências possíveis de execução entre atividades (ações) e permitem até mesmo a representação de ações concorrentes. Eles são úteis, por exemplo, para modelar fluxo de processo de negócios e *workflows* [Bin99, Capítulo 8]. Os DAs foram utilizados aqui pois, é possível relacioná-los de forma hierárquica, permitindo assim a representação direta de inclusões e extensões de CDUs. Além disso, é possível representar a especificação excepcional dos casos de uso de forma diferenciada, que é um fator importante devido aos cenários de exceção representarem situações críticas a serem testadas no SUT. A utilização dos DAs nesta dissertação baseia-se em um trabalho anterior em que estes modelos foram utilizados para representação de cenários de teste para componentes de software [Roc05].

Na segunda etapa do método de teste proposto, é criado um Grafo de Fluxo de Controle (em inglês, *Control Flow Graph* – CFG) que interliga todos os DAs. Essa abordagem possibilita a geração de testes que representam os cenários de uso do SUT, que sejam completos e não somente os cenários internos dos casos de uso isoladamente. Isto é possível pois o CFG interligado mantém os relacionamentos e a precedência de execução entre os DAs, que representam a descrição dos CDUs. Este CFG é criado de forma automática pelo método de teste proposto e foi baseado na proposta de Harrold *et al* [HRS98]. Harrold utiliza o CFG para análise de dependência entre instruções de programas, com o CFG interligando todos os CFGs que representam os procedimentos do programa.

Para geração automática de cenários de teste foram empregadas as técnicas de teste estrutural, que são utilizadas para geração de testes em CFGs [Bei90, Pre97]. Foram selecionados caminhos de teste no CFG para representar os cenários de uso completo do sistema. A partir dos cenários de uso foram obtidos os dados de teste para criação de um ou mais casos de teste para executar os cenários. A criação dos casos de teste e dos dados de teste foi feita manualmente durante a avaliação do método para geração dos cenários de teste, pois essas etapas fazem parte do escopo deste mestrado. Para especificar os modelos, preparar e realizar os testes foi necessário adotar um processo de teste, que se baseia no processo de testes especificado por Filho [dPPF01], mas as atividades do processo de teste são detalhadas para apoiar o método de geração automática de testes proposto nesta dissertação.

Várias abordagens de teste têm sido propostas com intuito de gerar cenários de teste para sistemas utilizando os DAs, entretanto algumas consideram somente a geração de cenários de cada caso de uso isoladamente [LJX+04, HVFR04, CLL05]. Outras abordagens propõem a geração de cenários completos, mas não automatizam todas as etapas

de geração, sendo necessário o pré-processamento manual dos DAs para obter modelos intermediários ou a descrição mais detalhada dos modelos para que seja possível gerar os cenários de teste [BL02, BLL04a, VLH+06].

É importante ressaltar que a geração de testes é realizada utilizando modelos da especificação do SUT e os testes gerados representam cenários de uso do SUT que possuem o mesmo nível de abstração dos modelos especificados. Cada cenário de uso criado detalha as ações que devem ser realizadas e observadas por um ator como, por exemplo, um usuário ou sistema externo que interage com SUT através de suas funcionalidades. Para verificar a viabilidade da aplicação do método de teste proposto em sistemas reais foi necessário o desenvolvimento de um protótipo para geração de cenários de teste como prova de conceito da viabilidade do método proposto nesta dissertação. Esse protótipo foi utilizado para geração de testes no âmbito do projeto Harpia, que é uma parceria realizada entre a Receita Federal do Brasil, Universidade Estadual de Campinas – Unicamp e Instituto Tecnológico da Aeronáutica – ITA. A realização dos testes utilizando os cenários gerados foi feita pela equipe de testes do projeto Harpia. Além disso, os testadores envolvidos foram submetidos a um questionário para avaliação do método proposto.

Como resultado da aplicação deste método de teste, foi possível observar melhorias nas atividades de teste de sistemas. Entre as melhorias proporcionadas, a manutenção dos casos de teste foi facilitada, já que alterações na especificação não implicaram alterações em cada cenário de teste. Outra melhoria foi a criação de testes para cobrir toda a especificação do SUT, pois quando os testes são criados de forma manual, é difícil garantir a cobertura de toda a especificação, já que o trabalho manual é propenso a erros. O método proposto também foi utilizado para realização de teste em um projeto de conclusão de curso de G. O. Patuci [Pat07]. Além disso, segundo os testadores que participaram da criação dos modelos e da execução dos testes, foi fácil utilizar os cenários de teste gerados pelo método proposto.

1.4 Organização do Texto

Esta dissertação foi dividida em oito capítulos, que são organizados da seguinte forma: o Capítulo 2 apresenta os fundamentos teóricos de testes de *software*, Grafos de Fluxo de Controle, que são necessários para o entendimento do texto. Além disso, o Capítulo 2 também apresenta o processo de teste adotado para apoiar o método de geração de testes proposto. O Capítulo 3 apresenta os modelos utilizados para geração dos testes e como esses modelos devem ser especificados. O método para geração de testes é apresentado no Capítulo 4. O Capítulo 5 apresenta os resultados da aplicação do método de teste em um estudo de caso, e a avaliação do método durante o estudo de caso que foi realizado pelos testadores envolvidos. Capítulo 6 apresenta os trabalhos relacionados, com maior atenção

aos trabalhos que propõem a geração automática de testes utilizando DAs. Finalmente, o Capítulo 7 apresenta as conclusões, contribuições e trabalhos futuros.

Além dos capítulos mencionados acima, esta dissertação possui dois apêndices que são organizados da seguinte maneira: o Apêndice A apresenta o questionário utilizado para avaliação do método de teste pelos testadores envolvidos no estudo de caso; e o Apêndice B apresenta as respostas obtidas na aplicação do questionário.

Capítulo 2

Conceitos de Testes de Software e Grafos de Fluxo de Controle

Este capítulo apresenta as terminologias e conceitos usados no decorrer deste trabalho. Primeiramente, na Seção 2.1 são apresentados os conceitos de Validação e Verificação. Em seguida, são apresentados os conceitos de *Testes de Software*, na Seção 2.2, onde são mostrados os principais tipos, técnicas e critérios para realização de testes; Depois são introduzidos os conceitos de teste funcional, que se encontram na Seção 2.3; na Seção 2.4 são apresentados os conceitos de teste estrutural, bem como os conceitos de Grafos de Fluxo de Controle (CFG). Na Seção 2.5 são apresentados alguns critérios e técnicas de teste orientado a caminhos, que são utilizadas geralmente na realização de testes estruturais. Por último, a Seção 2.6.1 detalha o processo de teste adotado para apoiar o método de geração de testes.

2.1 Verificação e Validação

Quando se busca a melhoria da qualidade do software, as atividades de Verificação e Validação (V&V) [Mye79] devem ser realizadas durante todo o ciclo de desenvolvimento. Essa prática é ainda mais essencial no desenvolvimento de sistemas complexos e críticos. As atividades de V&V têm como principal objetivo verificar se o sistema desenvolvido está em conformidade com sua especificação e validar se o sistema atende às necessidades de seus usuários.

As técnicas para realização de V&V podem ser divididas em dois principais grupos: estáticas e dinâmicas. As técnicas estáticas de V&V visam identificar defeitos sem propriamente executar o sistema que está sendo proposto (ou desenvolvido). Algumas técnicas estáticas mais usadas em V&V são: técnicas de inspeção [Lai07] tais como, revisão e inspeção de código; verificação de modelos [VHB+03] e execução simbólica [Kin76]. As

técnicas dinâmicas de V&V propostas também têm por objetivo revelar defeitos no sistema em desenvolvimento. Entretanto, sua realização é através da execução do sistema, controlando quais são os dados de entrada e observando o comportamento do sistema para verificar se foram ou não observados defeitos. Neste âmbito, as principais técnicas de V&V dinâmicas são as de *Testes de Software*, que têm como objetivo executar o sistema para revelar defeitos.

Os termos *fault*, *error* e *failure*, foram adotados de acordo com a terminologia em português definida por Leite e Loques [LL87], sendo, respectivamente, falha, erro e defeito.

O padrão IEEE (*IEEE STD. Glossary of Software Engineering Terminology*, padrão 610.12/1990) [IEE90] define que os problemas introduzidos no software pelo desenvolvedor são chamados de falhas (*fault*). Enganos podem ser cometidos tanto na especificação quanto no código do sistema. Quando uma falha é ativada durante a execução do software, um erro é gerado (*error*). Caso o problema se manifeste nas fronteiras do sistema, ocorre um defeito (*failure*), que pode ser percebido pelo usuário. Pode ser considerado como defeito qualquer resultado obtido, durante a execução do software, que seja diferente do especificado (ou esperado).

2.2 Conceitos Básicos de Testes de Software

Esta seção apresenta as definições e conceitos de *Testes de Software* utilizadas nesta dissertação. *Teste de Software* é o processo de executar um sistema com o objetivo de encontrar defeitos [Mye79]. A ocorrência de defeito indica a presença de falhas no software. Um caso de teste bem sucedido é aquele que revela uma falha no sistema, que ainda não tinha sido descoberta. Os testes não são capazes de provar a ausência de falhas em um sistema. Caso nenhum defeito ocorra durante os testes, não se pode concluir que o sistema não tenha falhas.

Uma limitação dos testes é o fato de não ser possível identificar, a priori, onde estão as falhas. Também não é possível realizar testes exaustivos, isto é, aplicar ao sistema todas as combinações de entradas possíveis ou exercitar todos os caminhos de execução possíveis. Isso é impraticável, pois mesmo em sistemas triviais, o número de casos de teste seria grande e, em alguns casos, até infinito, o que inviabilizaria sua execução. Assim, é importante adotar critérios para selecionar os testes com alta probabilidade de ativar defeitos, revelando assim a presença de falhas. Para a criação dos casos de teste, é recomendada a adoção de uma estratégia que possibilite sua criação em paralelo com o desenvolvimento, tornando assim possível, o agrupamento das atividades para realização dos testes em fases bem definidas.

2.2.1 Fases de Teste

O desenvolvimento de sistemas pode ser estruturado em fases [Pre01, Som95]. Assim, acompanhando as fases de desenvolvimento pode ser definido um conjunto de fases para preparação dos testes e, após a implementação do sistema (ou parte dele), executar os testes. Os testes devem ser previamente planejados e projetados (durante fases de preparação), para serem executados de forma controlada. Na Figura 2.1 é apresentado como as fases para preparação dos teses podem ser estruturadas e executadas paralelamente às fases de desenvolvimento. Este modelo para estruturar as fases de teste é conhecido como modelo V [dPPF01]. Durante a realização de cada fase de desenvolvimento são fornecidas informações para preparação dos diversos tipos de testes.

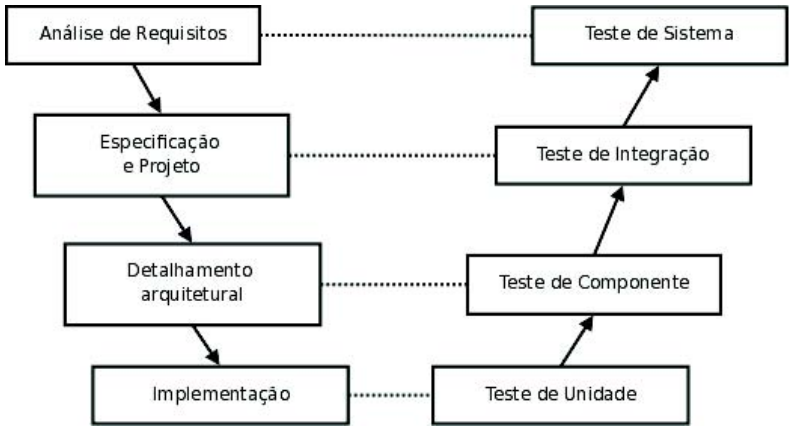

Figura 2.1: Modelo de Processo V, as atividades de preparação de testes são realizadas paralelamente às atividades de desenvolvimento [dPPF01].

As fases de testes envolvem tanto testes de baixo nível, que verificam se uma pequena parte do código fonte foi corretamente implementada (*Testes de Unidade*), quanto testes de alto nível, que validam funções do sistema relativas aos requisitos do cliente (*Testes de Sistemas*), conforme mostra a Figura 2.1. Esta divisão, em quatro fases principais, foi proposta por Beizer [Bei90] e cada uma das fases é melhor detalhada a seguir:

1. *Teste de Unidade*: Uma unidade é a menor porção de um software que pode ser executada. O teste de unidade verifica se uma porção do código executa adequadamente a sua funcionalidade, isoladamente do resto do sistema. Por isso, geralmente o caso de teste faz uso de *drivers* e *stubs*. Um *driver* é um elemento (classe, programa principal ou software externo) que aplica os casos de teste à unidade sob teste [Bin99]. O *driver* é responsável por fornecer os dados de entrada, coletar os dados de saída e apresentá-los ao usuário. Um *stub* substitui módulos necessários para a

execução da unidade, simulando seu comportamento [Mes04, Bin99]. A utilização de *drivers* e *stubs* possibilita que a unidade seja testada isoladamente.

2. *Teste de Componente*: Um componente é a integração de diversas unidades, com interfaces bem definidas [Szy98]. Nesta fase, o componente é testado de acordo com a especificação das funcionalidades e de sua estrutura. Também podem ser necessários *drivers* e *stubs*. O desenvolvimento de software baseado em componentes é cada vez mais utilizado atualmente. Um de seus principais atrativos é a possibilidade de redução do tempo e custo de desenvolvimento, através da reutilização de código. Apesar disso, o reuso de código não garante a qualidade do sistema, sendo necessária a realização de testes, a cada reutilização, quando estes componentes são integrados ou utilizados em um novo contexto [Wey98]. Vários trabalhos propondo métodos e processos de teste para desenvolvimento baseado em componentes têm sido realizados [Roc05, GGGS02, Fre91].

3. *Teste de Integração*: Nesta fase, os componentes são integrados, e os casos de teste são direcionados à descoberta de erros arquiteturais, relacionados às interfaces dos componentes. A integração pode se dar com o uso de uma abordagem *big-bang* [Pre97], na qual todos os componentes são integrados de uma só vez, dispensando o uso de *drivers* e *stubs* mas, nesse caso, a localização de falhas é dificultada. O mais recomendado é, portanto, o uso de uma abordagem incremental, na qual os componentes são integrados pouco a pouco. O uso de *drivers* e *stubs* pode ser necessário neste caso. Testes de integração reutilizam casos de testes gerados durante as fases de teste anteriores para verificar se o novo componente integrado não afeta as demais partes do sistema.

4. *Teste de Sistema*: Nesta fase, todo o sistema é integrado, incluindo outros sistemas, hardware, etc. São testadas todas as funcionalidades propostas na especificação do sistema, ou requisitos funcionais do sistema. Para isso o sistema deve ser executado em condições similares ao seu ambiente real de uso. Além disso, a partir dos requisitos não funcionais do sistema podem ser avaliados desempenho, segurança e confiabilidade [Pre97, Som95] e, também, os demais atributos de qualidade especificados (requeridos).

Além das quatro fases acima, existem ainda os *Testes de Regressão*, que devem ser aplicados quando houver modificações no SUT, que podem ocorrer durante o desenvolvimento incremental, testes de integração e na realização de manutenção. Toda modificação durante atividades de manutenção podem causar efeitos colaterais em partes do software que funcionavam corretamente antes daquela modificação e que não deveriam ter sido

afetadas pela mesma, nessa situação diz-se que houve uma regressão [Bin94], ou seja, surgiram novos defeitos em partes do sistema já testadas em etapas anteriores.

Os Testes de regressão são obtidos a partir do conjunto de testes do sistema e, nesta fase, um conjunto de casos de teste é novamente executado com o objetivo de atestar que partes do SUT não foram afetadas pelas mudanças realizadas. Existem basicamente duas estratégias de execução de *Testes de Regressão*: retestar-tudo e seletiva [RH94]. A estratégia retestar-tudo consiste simplesmente em reaplicar todo o conjunto de testes original à nova versão do SUT. A estratégia seletiva consiste em reaplicar apenas um subconjunto dos casos de testes originais. A estratégia retestar-tudo é desaconselhável pois, apesar de segura, requer um grande esforço e custo na existência de um grande volume de casos de testes.

A divisão dos testes em fases estrutura as atividades de teste e facilita a ordenação das atividades a serem realizadas. Apesar disso, a estruturação em fases ainda não garante que os testes a serem realizados têm potencial para revelar novas falhas de forma controlada e em um tempo viável. Para garantir a realização das atividades de teste em um tempo viável, pode-se utilizar ou definir alguns critérios para seleção, criação e (ou) realização de testes.

2.2.2 Critérios de Teste

A divisão dos testes em fases organiza as atividades necessárias a cada etapa de desenvolvimento do sistema. Por outro lado, a definição e uso de critérios de teste (independentemente da fase de teste) é importante devido à inviabilidade de realização de todos os testes possíveis em um SUT. Este fato ocorre pois grande parte das vezes o conjunto de possíveis entradas ou caminhos de execução em um sistema é muito grande, tendendo ao infinito. Para resolver este problema, a cada fase de testes, é necessária a seleção de testes usando critérios bem definidos.

Um critério de testes é uma condição (ou um conjunto de condições) que define quão bem um sistema foi testado baseando-se em sua especificação [Zhu95]. Os critérios devem ser adotados de forma a estabelecer condições a serem satisfeitas durante os testes. Dentre vários critérios de testes, os três principais critérios são: o critério funcional, o estrutural e o baseado em falhas. Além desses critérios, será detalhado também os critérios de Teste Orientado a Caminhos, que podem ser adotados tanto com critérios de teste estrutural [Zhu95] quanto funcional [Edw00, PBC93].

O critério baseado em falhas busca testar o comportamento do sistema na presença de falha. Falhas podem acontecer tanto no SUT, quanto nos sistemas com os quais ele interage. As principais técnicas são mutação e injeção de falhas. Em uma abordagem de testes que utiliza mutação, as falhas devem ser introduzidas diretamente no código

do SUT para avaliar se os casos de teste executados identificam a presença dessas falhas introduzidas [DLS78]. Desta forma, se este tipo de falha existir em outros locais do código fonte, este caso de teste estará apto a revelá-la. A técnica de injeção de falhas consiste em modificar o código objeto ou o estado do SUT durante sua execução. Seu objetivo é verificar o comportamento do sistema durante a execução em presença de falhas ou erros [HTI97], bem como a verificação do impacto da propagação das falhas injetadas. Neste aspecto, pode-se usar injeção de falhas na realização de testes de robustez [KSD+03].

Os critérios funcional, estrutural e orientados a caminhos serão detalhados nas Seções 2.3, 2.4 e 2.5, respectivamente. Este destaque deve-se ao relacionamento que apresentam com este mestrado, que propõe uma abordagem para testes funcionais aplicando técnicas e critérios de testes estruturais.

2.3 Critério de Testes Funcional

No critério funcional de testes, também conhecido como "caixa preta", os detalhes internos do sistema não são considerados na elaboração e execução dos casos de teste [Bei90] apenas suas funcionalidades, conforme descrito na especificação do sistema. São fornecidas entradas ao sistema e suas saídas são verificadas de acordo com o comportamento especificado.

As fontes utilizadas pelos testes funcionais são as especificações de requisitos e os artefatos produzidos durante as atividades de projeto. A principal dificuldade na realização de testes funcionais é a obtenção do menor conjunto de testes que consegue revelar o maior número de defeitos, a chamada eficácia dos testes [Fre91]. Algumas abordagens para os testes funcionais são [Pre01]:

1. **Partição de Equivalência**: essa técnica é utilizada para a escolha de dados de entrada para os testes. O domínio de entrada é dividido em classes de equivalência, que representam um conjunto de valores válidos e inválidos para as condições de entrada. Por exemplo, caso os dados válidos sejam representados por uma faixa de valores, três classes de equivalência são derivadas: os números menores que a faixa, os dentro da faixa, e os maiores que a faixa. Os valores de entrada dos testes são escolhidos dentro de cada classe de equivalência.

2. **Análise de Valores Limite**: essa técnica também pode ser utilizada para a escolha de dados de entrada, combinada à partição de equivalência. Como um grande número de erros tende a ocorrer nos limites dos valores de entrada, a técnica de análise de valores limite determina que os dados de entrada escolhidos sejam valores limite de cada uma das classes de equivalência.

3. **Baseado em Modelos**: o comportamento do sistema é descrito em forma de modelos e diagramas como, por exemplo, os propostos pela UML [Gro03b]. Normalmente testes baseados em modelo (MBT) utilizam diagramas para representar o comportamento do SUT. Assim, a cobertura dos modelos é uma forma de exercitar os comportamentos e estados mostrados em um modelo como, por exemplo, percorrer todos os estados em um Diagrama de Estados, ou ainda, executar todas as possíveis atividades em Diagrama de Atividades [Gro04]. A cobertura dos modelos deve ser realizada visando encontrar defeitos na implementação.

Os critérios funcionais são bastante usados durante a fase de *Teste de Sistema* [Pre01], principalmente devido a criação dos testes se basear na especificação do sistema. Além disso, podem ser usados juntamente aos critérios de teste estrutural, para realização de cobertura de modelos, como proposto por Beizer [Bei95].

2.4 Critério de Teste Estrutural

No critério estrutural, também conhecido como "caixa-branca" ou caixa de vidro, considera-se a estrutura interna do sistema e sua implementação. Os casos de teste são derivados do código fonte do sistema e exploram a cobertura do código para que os possíveis defeitos sejam detectados. Neste critério de teste, uma abordagem bastante usada é a criação de um modelo, o Grafo de Fluxo de Controle (CFG), para visualização e seleção de caminhos de testes.

Os CFGs são vastamente utilizados para realização de testes estruturais [Bei90]. Um CFG é a representação gráfica da estrutura de controle de um programa [Bei90, Pre97] que é representado por um grafo direcionado [CLRS01]. Um grafo direcionado é uma ferramenta matemática usada para representar um conjunto de pontos, definidos como *vértices*, e um conjunto de associações orientadas entre os vértices, que são definidas como *arestas* [CLRS01]. É um modelo útil para a visualização das informações do fluxo interno de um programa e é bastante usado em Testes Estruturais.

Para representar o CFG, Beizer divide os elementos obtidos a partir do código fonte em três grupos: blocos de código, decisões e junções [Bei90]. Um bloco de código é uma seqüência de sentenças no programa sem interrupções por decisões ou junções. Uma decisão é um ponto do programa no qual o fluxo de controle pode divergir, como uma sentença *if ( ) then* { } *else*{ }, *while*, *for* e demais similares. Uma junção é um ponto no programa no qual o fluxo converge, como o fim de uma sentença *if*, *while* e demais sentenças de decisão correspondentes.

2.4.1 Grafo de Fluxo de Controle

Um CFG é geralmente usado na representação do fluxo de controle de um programa. Uma definição mais formal é a seguinte: CFG é um grafo direcionado definido como uma tupla (V, A, e, s), onde V é um conjunto de nós ou vértices. A é uma relação em V, que associa quaisquer dois vértices de V, representando as arestas direcionadas do CFG. Dois vértices e, s de G, tal que, $e, s \in V$, representam respectivamente os vértices de entrada e saída no CFG, ou seja, todos os caminhos do CFG iniciam em e e terminam em s. Assim o conjunto de vértices $V \setminus e, s$ (todos os vértices de V exceto e e s) é denominado como o conjunto de vértices internos do CFG. Um par (v_n, v_m) é uma aresta direcionada, que pertence ao conjunto de arestas de saída do vértice v_n e de entrada em v_m.

Um exemplo de CFG é mostrado na Figura 2.2 juntamente com o código fonte do procedimento que ele representa. Este procedimento calcula o máximo entre dois números e é chamado de *max*. O procedimento *max* recebe como entrada dois número inteiros positivos, retornando o maior entre os dois números quando eles são diferentes e zero quando são iguais. No CFG mostrado na Figura 2.2 os vértices (nós) tem como rótulo o número da(s) linha(s) de código correspondentes no procedimento. É importante observar que no exemplo de CFG mostrado, os vértices de junção não são diferenciados visualmente dos blocos de código, sendo diferenciados somente por possuir mais de uma aresta de entrada como nos vértices que representam as linhas 13 a 15.

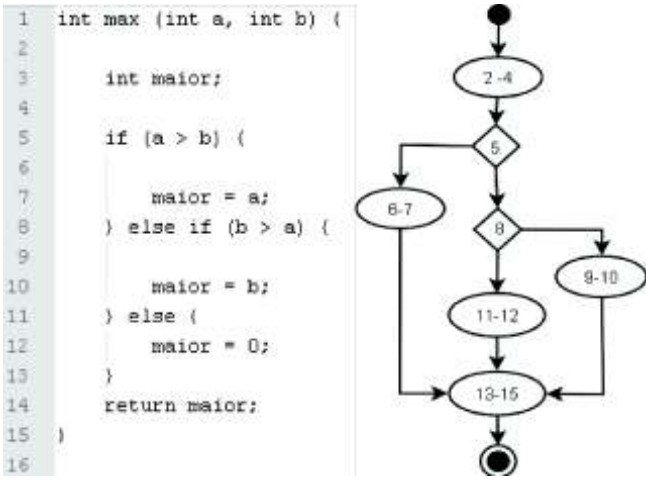

Figura 2.2: Código Fonte e CFG correspondente.

Um vértice representa seqüência ininterrupta de uma ou mais instruções, ou seja, um vértice v é executado se, e somente se, todas as instruções que ele representa forem executadas na ordem descrita por seu programa fonte. Com isso, o bloco de instruções considerado não possui desvios para nenhuma instrução externa ou interna no mesmo

bloco. A execução do bloco começa sempre na primeira instrução e termina na última instrução do bloco representado pelo vértice [Bei90, Rob99].

Em um CFG, uma seqüência de vértices $(v_0, v_1, ..., v_k)$, onde $k \geq 0$ e os vértices pertencentes ao CFG, define um caminho de tamanho k se, e somente se, existir arestas partindo de v_{i-1} e chegando em v_i para todo i, tal que $1 \leq i \leq k$. O caminho parte do vértice v_0 e chega ao vértice v_k.

Um ciclo (*loop*) é um caminho $(v_0, v_1, ..., v_k)$ onde $v_0 = v_k$. Um CFG é conectado se para cada par de vértices $v_m, v_n \in G$, existe pelo menos um caminho ligando v_m e v_n. Um caminho é dito completo se ele inicia no vértice inicial do CFG, ou seja, o vértice e, e termina no vértice final do CFG, o vértice s.

Um CFG G é um Grafo bem formado se, e somente se, para cada vértice interno $v_i \in G$ existe um caminho direcionado entre e e v_i e também um caminho entre v_i e s.

É importante ressaltar que os CFGs, através de seus caminhos, representam as dependências internas entre as instruções de determinado método ou procedimento, definida também como a ordem de execução das instruções de um software. Para realização da análise de fluxo de controle de maneira judiciosa é necessário considerar também as dependências externas, ou dependências com instruções entre métodos ou procedimentos relacionados ao CFG em questão, o que pode ser feito através da ligação entre dois ou mais CFGs.

2.4.2 Grafo de Fluxo de Controle Interprocedural

Alguns trabalhos propõem interligar os CFGs nos pontos onde existem chamadas a outros métodos ou procedimentos, sendo considerada assim as dependências entre instruções em todo o contexto de chamada e retorno, e não só CFG de um módulo isolado. A utilização dos CFGs interligados foi proposta por Landi e Ryder [LR92] para computar as dependências de controle entre as instruções em CFGs interligados. Alguns trabalhos de M. J. Harrold, G. Rothermel e S. Sinha são bastante completos e detalhados [HRS98, SHR01].

As informações de dependências entre procedimentos são importantes para realização de testes estruturais e para análise de regressão de testes durante a manutenção de sistemas. As informações de dependências são obtidas através da análise de interação entre os procedimentos, ou chamadas entre procedimentos. A Figura 2.3 mostra um programa e seu CFG. O programa tem como entrada dois números inteiros e enquanto a entrada for de inteiros positivos distintos é exibido o maior entre dois números de entrada. O algoritmo termina quando o usuário entra com dois inteiros iguais.

O procedimento *main* do programa chama, na linha 22, o procedimento *max* apresentado na Figura 2.2. Ainda na Figura 2.3, na linha 29, há outra chamada para o

procedimento *max*, que pode ser executado zero ou mais vezes dependendo do resultado da chamada da linha 22 e, posteriormente, na própria linha 29.

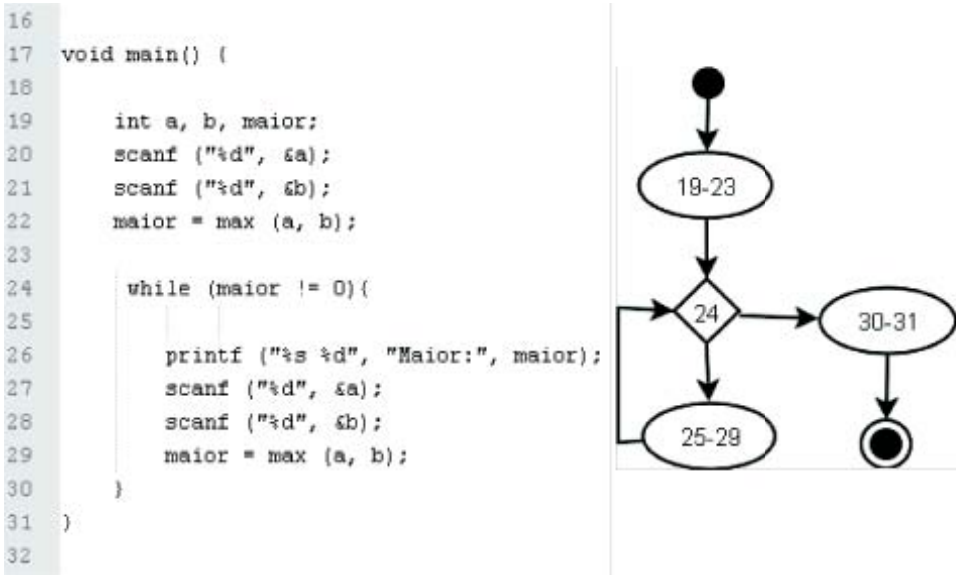

Figura 2.3: Código fonte e CFG de programa que compara dois números usando o procedimento *max*.

Para realizar análise de dependências, Harrold et al [HRS98] apresentam duas abordagens para criação de CFG interligados , nas duas abordagens são usados os conceitos de CFG já apresentados.

Na primeira abordagem é criado um CFG aninhado (em inglês, *Interprocedural Inlined Flow Graph* – IIFG) em que a cada nova chamada de um procedimento é incluída uma cópia do CFG chamado. Desta forma, se o procedimento for chamado mais de uma vez, são feitas várias cópias para cada CFG correspondente a um procedimento. Um programa P mantém um conjunto de procedimentos ρ e, cada procedimento $p_i \in \rho$ é representado por um CFG g_k. O IIFG correspondente a P mantém um conjunto de CFGs G, onde para cada procedimento $p_i \in P$ existe uma cópia $g_k^{'}$ de g_k, $g_k^{'} \in G$, onde g_k é correspondente a p_i [SHR01].

Uma definição formal para o Grafo de Fluxo de Controle Aninhado é a seguinte: o IIFG é definido pela tupla $(\eta, \varepsilon, e_s, s_s)$, em que η é um conjunto que contém a união de todos os conjuntos de vértices (V_i) de cada CFG do programa. Além disso, η contém os Vértices de Chamada (VCs) de cada CFG que realizam chamadas a outros procedimentos (correspondentes a um CFG) e para cada VC são criados Vértices de Retorno (VR), que também pertencente a η. O ε é a união de todos os conjuntos A_i de cada cópia de CFG correspondente a um procedimento do programa. Além disso, ε mantém as Arestas de Chamada (AC) que são criadas para cada chamada a um procedimento, ligando VC ao

vértice e do CFG chamado e as Arestas de Retorno (AR) que ligam o CFG chamado ao VR correspondente. Os vértices e_s e s_s são respectivamente os vértices de entrada e saída global do IIFG.

A Figura 2.4 mostra o exemplo de representação do IIFG para o programa da Figura 2.3 que usa o procedimento mostrado na Figura 2.2. Pode-se observar, na Figura 2.4, que foram criados duas cópias do procedimento *max*, uma para cada chamada (nas linhas 22 e 29 do programa). Além disso, os vértices que representavam o procedimento *max* nas linhas 22 e 29 do programa, foram substituídos por dois vértices no CFG Aninhado, o primeiro que representa o VC para o procedimento *max* e o segundo seu VR correspondente. Ainda na mesma Figura, as AC e AR foram representadas de forma pontilhada.

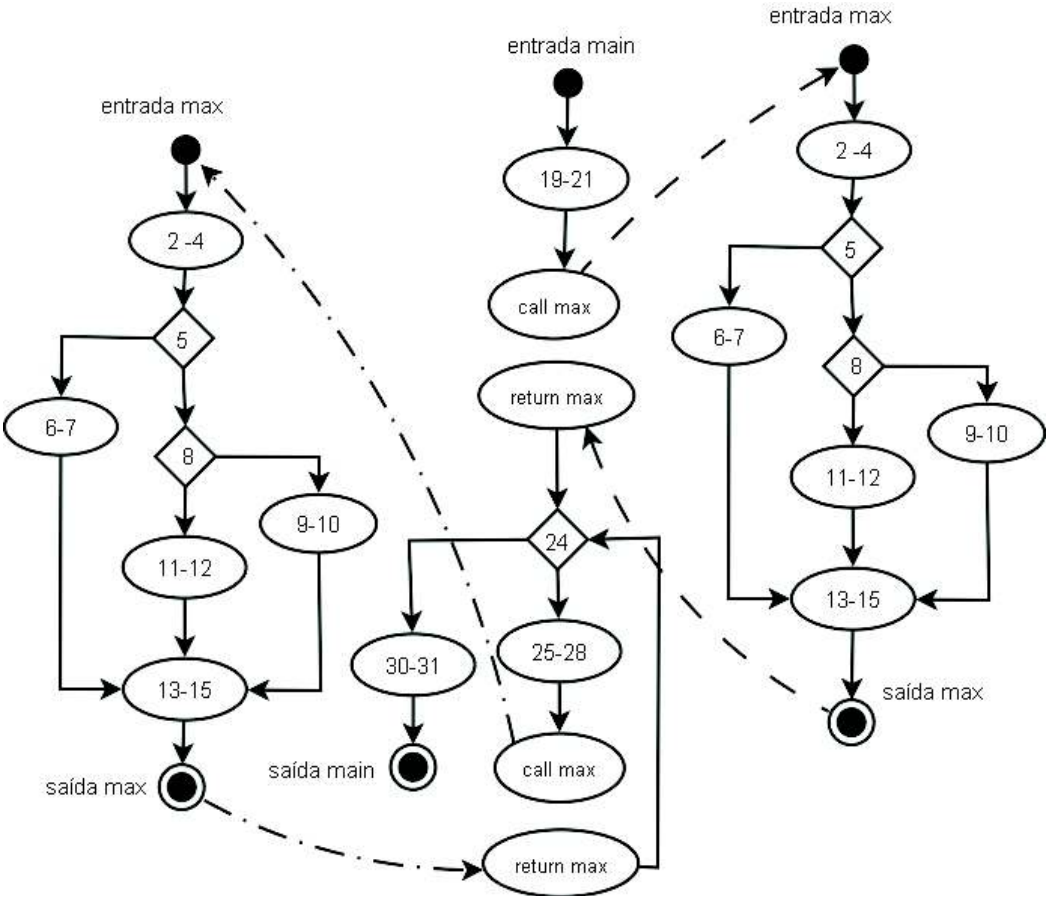

Figura 2.4: IIFG do programa que compra dois números utilizando o procedimento *max*.

Uma limitação desta abordagem é observada quando existem chamadas recursivas. Neste caso, o IIFG seria infinito. Desta forma, para contornar o problema é necessário limitar a quantidade de níveis de recursividade, mas mesmo em situações sem chamadas recursivas essa abordagem tende a criar muitas réplicas dos CFGs, mantendo muitas

informações redundantes.

A segunda abordagem apresentada por Harrold et al [HRS98] propõe a criação de um CFG Inter-procedimental (em inglês, *Interprocedural Control Flow Graph* – ICFG) que contém uma única cópia de cada procedimento presente no programa. O ICFG pode ser definido da seguinte forma: Dado um programa P que contém uma coleção de CFGs G onde $g_i \in G, i > 0$, sendo que cada g_i representa uma única cópia de CFG de procedimento $p_i, i > 0$ correspondente no programa. Cada CFG é composto por um conjunto V de vértices e A de arestas. Assim, o Grafo de Fluxo de Controle Inter-procedural é definido pela tupla $ICFG = (\eta, \varepsilon, e_s, s_s)'$, em que os elementos compondo esta tupla são os mesmos elementos do IIFG já definidos previamente.

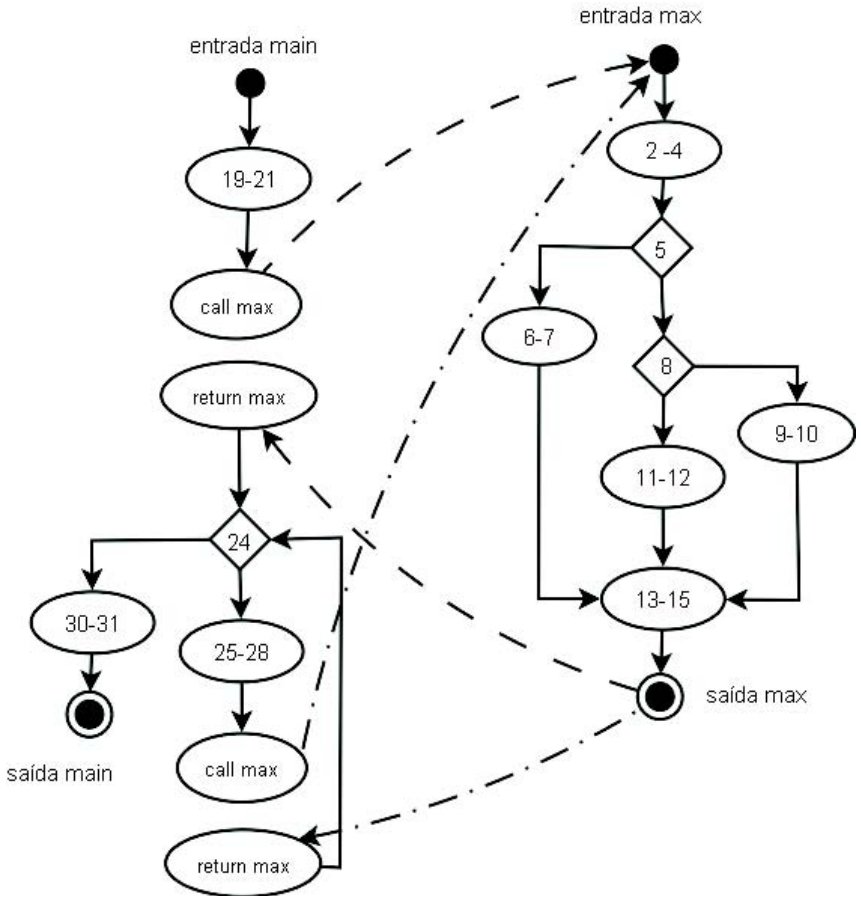

Figura 2.5: ICFG do programa que compra dois núemros utilizando o procedimento *max*.

A Figura 2.5 mostra um ICFG usando o mesmo exemplo apresentado para o IIFG da Figura 2.4. Este exemplo usa o programa da Figura 2.3 que faz chamada ao CFG (*max*) da Figura 2.2. Pode-se observar ainda na Figura 2.5, que existe uma única cópia do CFG *max* com suas Arestas de Chamada e Retorno para os dois pontos de chamada no

programa (linhas 22 e 29). Assim, as arestas inter-procedimentais ligam os CFGs ao vértice de entrada da única cópia do CFG do procedimento *max*.

Na proposta para geração de casos de teste apresentado no Capítulo 4 utilizou-se uma variação de ICFG, como passo intermediário à criação automática de casos de teste criados a partir de modelos da especificação.

2.5 Testes Orientados a Caminhos

Testes de Caminho (em inglês, *Path Testing*) é o nome dado a uma família de critérios de testes baseada na seleção de um conjunto de caminhos executáveis de teste em um programa [Bei90]. Um caminho é dito "executável" quando existem dados que possibilitem a sua execução, caso contrário o caminho é chamado de caminho "não executável". Segundo Beizer [Bei90], entre as técnicas de teste estrutural, Testes de Caminho é uma das mais antigas e ainda assim bastante empregadas atualmente. Como são selecionados um conjunto de caminhos deve-se, de alguma forma, mensurar a completude dos testes realizados através desses caminhos [Whi00]. Com este intuito existem vários trabalhos que apresentam conceitos, padrões e critérios de testes orientados à Caminho em CFGs [Pai78], [Zhu95], [Bei95] e [Bei90].

Para realização de testes orientados a caminhos, vários critérios de teste foram propostos. Os critérios de teste têm como objetivo determinar (mensurar) se uma determinada parte do software foi testada de forma adequada. Para isso, deve ser delimitado o conjunto de casos de teste que atenda ao critério de teste determinado e, preferencialmente, o menor conjunto de casos de teste possível.

Segundo Hong Zhu [Zhu95] os critérios de teste podem ser divididos em quatro principais grupos: critério de teste baseado em fluxo de controle, critério de teste baseado em fluxo de dados, critério de teste baseado em falhas e critério de testes baseado em erros. Neste mestrado foi aplicado o critério baseado em fluxo de controle. Para esse critério, Hong Zhu propôs um conjunto de axiomas que define o que um critério de teste deve conter, possibilitando assim, a verificação de quais critérios satisfazem os axiomas. O trabalho de Hong Zhu foi baseado nos propostos por Weyuker [Wey86] e Baker *et al* [BHB86].

O critério de teste baseado em fluxo de controle, por sua vez, foi sub-dividido em outros quatro principais grupos:

1. **Critério de Testes de Instruções e Decisões**: Esses critérios são os mais conhecidos em testes de caminhos e visam, respectivamente, atender a cobertura de todas as instruções e decisões de um programa. O critério de cobertura de instruções é satisfeito quando existe um conjunto finito de caminhos completos que exercita

todas as instruções presentes no CFG. Mais formalmente: Para um dado CFG G (conforme definido na Seção 2.4.1), um conjunto finito de caminhos completos C atende ao critério de cobertura de instruções se, e somente se, para todo vértice alcançável $v_a \in V, \exists c_i \in C \backslash v_a \subseteq c_i$. O critério de cobertura de decisões é similar ao de cobertura de instruções. Porém, para garantir a cobertura de todas as decisões é necessário encontrar um conjunto finito de caminhos de teste que cobre todas as arestas do CFG, que representa o programa.

2. **Critério de Teste de Caminhos**: Visa exercitar todos os caminhos de teste em um programa. O *critério de teste de caminhos*, em grande parte das situações, é considerado um critério de teste não aplicável devido à existência de infinitos caminhos nos programas. Para tornar esta abordagem de testes viável, podem-se realizar algumas restrições para considerar um número finito de caminhos. Uma primeira restrição é considerar somente os caminhos simples e caminhos elementares no CFG. Um caminho é considerado simples se não existe repetição de arestas no caminho e é dito elementar se não houver repetição de vértices. Desta forma, o critério de caminhos simples do CFG é finito. Outra abordagem que pode ser aplicada a testes de caminhos é a de delimitar o tamanho máximo dos caminhos de teste. Neste critério são considerados somente caminhos com tamanho até n. O principal problema na cobertura de caminhos simples e de caminhos de tamanho n é que as duas abordagens não garantem a cobertura de todos os vértices e arestas do CFG. Para cobrir as arestas e vértices não cobertos o critério de cobertura de nível i, proposto por Paige [Pai78], parte dos caminhos criados pelo critério de caminhos simples e são inseridos níveis em que, a cada novo nível, são inseridos vértices ou arestas ainda não cobertas no nível anterior até realizar a cobertura de todos os vértices e arestas do CFG.

3. **Critério de Teste de Loops**: Os ciclos, mais conhecidos como loops, em um CFG foram definidos na Seção 2.4.1 e representam um ponto particular para realização testes orientados a caminhos. Algumas técnicas de testes geralmente consideram que os *loops* devem ser executados pelo menos uma vez durante os testes. Assim como em testes de caminhos, existem os critérios para cobertura de ciclos que visam realizar testes de ciclos elementares. Um ciclo é elementar se não houver repetição de vértices no ciclo, exceto pelo vértice de entrada v_e e saída v_s do ciclo que são iguais $v_e = v_s$. Partindo de ciclos elementares outros critérios definidos são de cobertura de ciclos elementares k *vezes*, sendo que k é o número de interações do *loop* no caminho (grau), onde $k \geq 1$. O critério é atendido se para todo ciclo com grau menor ou igual a k existe pelo menos um caminho que contém o *loop* como sub-caminho.

4. **Critério de Testes de Caminhos Básicos**: Este critério foi proposto por Mc-

Cabe [McC76] e é baseado na seleção de caminhos completos de teste no CFG para obtenção de um conjunto de caminhos básicos. Um conjunto de caminhos é considerado conjunto de caminhos básicos se todos os outros caminhos no CFG podem ser representados através de combinações lineares entre caminhos do conjunto.

É importante ressaltar que a realização de testes de todos os caminhos é impraticável devido ao número de combinações geradas e, além disso, a cobertura de instruções e decisões são requisitos mínimos para realização de testes de unidade. Após a realização dos testes estruturais, assim como na realização das demais técnicas de teste de forma isolada, não se pode garantir que a unidade testada está livre de defeitos. Além disso, mesmo quando a unidade em teste esteja livre de defeitos, não é possível garantir a ausência de defeitos durante a realização da integração das unidades. Desta forma, as técnicas de teste devem ser utilizadas de forma complementar. Este mestrado propõe a aplicação de testes orientados a caminhos, para selecionar cenários de testes funcionais baseados na especificação do SUT, conforme será mostrado no Capítulo 4. A seguir será apresentado o processo de testes adotado para realização dos testes funcionais.

2.6 O Processo para Teste Funcional Adotado

Os testes funcionais em sistemas são realizados para verificar se as funcionalidades implementadas durante o desenvolvimento atendem aos requisitos especificados pelo cliente. Nesta fase, os testes devem ser derivados da especificação do SUT, que é criada na fase de análise de requisitos no processo de desenvolvimento [Som95] e geralmente é documentada com Diagramas de Casos de Uso UML [Gro04, Coc01b]. A especificação e detalhamento dos casos de uso, em muitos sistemas, é feita textualmente sem seguir padrões na sua descrição e de forma pouco estruturada [Gel04, Coc01b].

Esta seção descreve um processo para apoiar os testes funcionais em sistemas, usando a especificação dos casos de uso como entrada para o projeto e execução dos testes. No processo proposto, os testes são criados a partir de modelos comportamentais do SUT, que são derivados dos casos de uso durante o planejamento e especificação dos testes. O processo adotado, bem como sua estrutura e atividades propostas para sua realização baseiam-se nos trabalhos apresentadas por Filho [dPPF01] e Pressman [Pre01], que definem modelos de processo para realização de testes.

A definição de um processo é importante pois possibilita que as atividades sejam reproduzidas de forma estruturada e padronizada para realização de testes em um ou mais produtos de software. Além disso, quanto mais bem definidas as atividades e responsabilidades dentro do processo, menor será a dependência das pessoas da equipe de teste. Por exemplo, quando houver rotatividade na equipe, o conhecimento sobre as tarefas de-

sempenhadas por essas pessoas não será perdido, pois essas atividades são definidas no processo. A Figura 2.6 mostra o diagrama de blocos que representa o processo de testes funcionais adotado nesta dissertação.

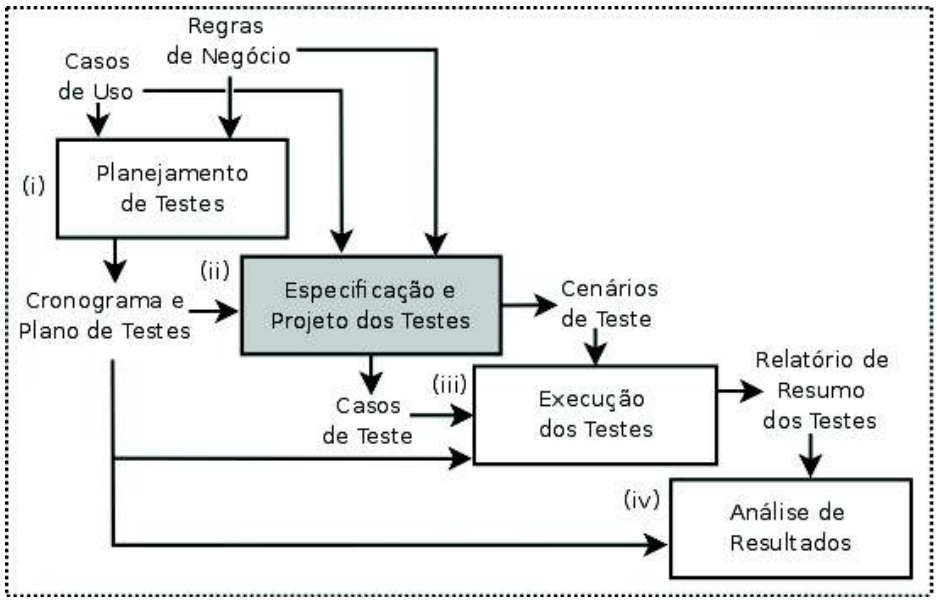

Figura 2.6: Estrutura do Processo de Testes

2.6.1 Visão Geral do Processo

Um processo pode ser definido como um conjunto de atividades ordenadas que devem ser realizadas para cumprir um objetivo [Pre01]. No caso deste trabalho, o objetivo é executar as funcionalidades do SUT para revelar a presença de falhas. As atividades do processo de testes funcionais proposto foram organizadas em duas principais fases seguindo [dPPF01]: preparação e realização de testes, que ocorrem em colaboração com outras atividades do ciclo de desenvolvimento do SUT.

A preparação dos testes pode ocorrer paralelamente com o processo de desenvolvimento, pois não requer que o sistema esteja implementado. Durante a preparação dos testes, os requisitos do sistema são os principais artefatos de entrada para realização. Uma vantagem em antecipar a fase de preparação dos testes é a redução de riscos de desenvolvimento [Pre01, Bei90] como, por exemplo, não testar de forma adequada (minimamente) o sistema e não cumprir os prazos estabelecidos para entrega. A fase de realização é iniciada após a implementação do sistema ou implementação de parte dele, quando o desenvolvimento do sistema é incremental.

Cada fase principal do processo é subdividida em outras duas fases. A fase de pre-

paração em (i) fase de planejamento e (ii) fase de especificação e projeto de testes e a fase de realização em (iii) fase de execução e (iv) fase de análise de resultados, conforme é mostrado na Figura 2.6. Além disso, a Figura 2.6 apresenta os principais artefatos de entrada e saída em cada fase do processo.

Ainda na Figura 2.6, a fase (ii), Especificação e Projeto dos Testes, está representada em destaque, pois nesta fase devem ser especificados modelos comportamentais do sistema que serão usados posteriormente para seleção de testes através do método de teste proposto nesta dissertação, que é apresentado no Capítulo 4.

O processo detalha "quais" atividades devem ser executadas durante os testes funcionais do sistema e não "como" essas atividades devem ser realizadas. Esse processo pode ser utilizado e integrado a outros processos de testes existentes, podendo ser aplicado em diversas fases de teste como, por exemplo, na seleção de cenários de teste durante a fase de Teste de Componente, como apresentado por Perez *et al* [PMV07]. Além disso, é importante ressaltar que este processo não detalha as atividades para realização de outros tipos de testes de sistema como, por exemplo, desempenho e robustez.

Nas seções seguintes, cada fase do processo de teste adotado nesta dissertação será apresentada em maiores detalhes.

2.6.2 Planejamento dos Testes

A fase de preparação dos testes, conforme apresentado acima, é dividida em duas fases principais: Planejamento dos Testes e Especificação e Projeto dos Testes. Na primeira parte, são definidos quais itens e aspectos serão testados, as abordagens de testes adotadas, os critérios de cobertura, os recursos necessários e cronogramas para a realização dos testes. Essas informações compõem o documento de "Plano de testes".

As atividades de planejamento são realizadas por arquitetos e projetistas de testes. Foram definidas duas atividades principais para o planejamento dos testes: **Definir os Objetivos e Critérios de Teste** e **Obter Artefatos para Preparação dos Testes**. É importante ressaltar que essas atividades não necessariamente devem ser realizadas seqüencialmente. Por exemplo, a segunda atividade pode ser realizada de forma contínua em toda a fase de preparação dos testes.

Definir os Objetivos e Critérios de Teste

Nesta atividade são definidos os objetivos, os critérios para realização dos testes e requisitos de qualidade do SUT. O tipo de teste proposto neste processo é o *Teste Funcional*, que é realizado a partir da especificação e tem como objetivo executar o SUT para verificar se o comportamento obtido corresponde ao esperado, ou seja, o resultado obtido está em conformidade com a especificação do SUT. Desta forma, deve-se definir o critério

de cobertura de testes para cobertura dos modelos da especificação do SUT. Os modelos serão criados durante na fase de Projeto e Especificação dos Testes.

Os critérios de teste "caixa-branca" [Zhu95] geralmente são aplicáveis em modelos, podendo ser utilizados para cobertura de transições no modelo, estados em Diagramas de Estado e ainda cobertura de atividades em Diagramas de Atividades.

Os critérios são usados para seleção dos caminhos (ou cenários de teste) e além deles, devem ser definidos também os critérios para obtenção dos dados de teste como, por exemplo, partição de equivalência e análise de valores limite [Pre01]. Além da cobertura da especificação, pode-se definir também o critério de cobertura de código fonte do SUT. Para isso, deve-se utilizar uma ferramenta para a análise de cobertura de código como, por exemplo, a *Clover* [Sys07].

Obter Artefatos para Preparação dos Testes

Os artefatos produzidos durante as fases de análise de requisitos e projeto no ciclo de desenvolvimento do SUT são importantes durante a fase de planejamento dos testes. Os principais artefatos utilizados no planejamento e projeto dos testes funcionais são: Diagrama de Casos de Uso e Mapa de Navegabilidade, sendo que eles são essenciais para definição dos aspectos a serem testados, a definição do esforço necessário para realização dos testes e a definição de um cronograma de teste.

O Mapa de Navegabilidade facilita o entendimento do sistema pela equipe de teste e é necessário, principalmente, em se tratando de sistemas interativos (com interface gráfica). No Mapa de Navegabilidade geralmente são representados os *links* entre as telas do SUT, ou seja, o fluxo de execução entre as telas e os campos representados nessas telas. O Mapa de Navegabilidade pode ser representado através de um protótipo interativo do sistema, o que geralmente ocorre em métodos ágeis de desenvolvimento [Coc01a], ou mesmo por um modelo que representa os *links* entre as telas do sistema e as estruturas dessas telas [Con99, CMPV05].

Os Casos de Uso devem possuir a descrição de seus cenários internos, que pode ser feita textualmente ou através de Diagramas de Atividades, Estados, Seqüência ou Colaboração da UML [Gro04] ou ainda através da combinação de duas ou mais formas propostas pela UML. Apesar das várias possibilidades para a modelagem proposta pela UML, a abordagem para descrição de casos de uso mais usada é a descrição textual [Coc01b] que geralmente é ambígua [Gel04]. Neste trabalho, na tentativa de reduzir as ambigüidades desses modelos, os casos de uso devem ser descritos através de modelos, conforme será apresentada na Seção 2.6.3.

2.6.3 Especificação e Projeto dos Testes

Nesta fase de testes é feito um refinamento do plano de testes, detalhando as funcionalidades e características do SUT que serão testadas. A partir desse refinamento é iniciada a fase de especificação e projeto dos testes, que foi definida através de quatro atividades: (1) Criar Modelos para Testes, (2) Criar Cenários de Teste, (3) Preparar Ambiente de Testes e (4) Especificar os Casos de Teste, que serão detalhadas logo abaixo.

Criar Modelos para Testes

Nesta atividade são criados os modelos necessários para apoiar o projeto dos testes funcionais, caso eles não façam parte da especificação do SUT. Os modelos são uma forma econômica de se capturar conhecimento sobre o sistema e possibilitar o reuso deste conhecimento por todos participantes diretos e indiretos no processo de testes. Além disso, as informações descritas no modelo não são perdidas quando houver mudanças na equipe de testes como, por exemplo, a troca de projetistas e arquitetos de teste [AD97].

Os modelos descrevem o comportamento do SUT e devem ser criados pelos projetistas de testes. Estes modelos fazem parte da especificação dos testes funcionais e a partir deles serão selecionados os cenários de teste. Basicamente, dois tipos de modelos devem ser criados e, para isso, foi adotada a notação dos Diagramas de Atividades da UML 2.0 [Gro04].

Os caso de uso do SUT descrevem seus cenários de uso (ou cenários internos), que são os cenários normais, alternativos e de exceção dos casos de uso e, geralmente, são descritos de forma textual. Além disso, existem as dependências de execução entre os casos de uso, ou seja, fluxo de execução entre eles. Por exemplo, em um sistema de cadastro, para realizar remoção de um item (executar caso de uso remover item) é necessária, primeiramente, a execução do caso de uso inserir item. Desta forma o caso de uso remover item só pode ser executado após o item ter sido inserido, caracterizando assim uma dependência. Assim, devem ser criados dois níveis de modelos que representam esses aspectos, sendo que, no primeiro nível devem ser representadas as dependências entre os casos de uso e no segundo nível os cenários dos casos de uso.

A abordagem para representação do primeiro nível é baseada nas propostas por Edwards [Edw00], que cria um grafo com essas dependências para componentes de software, e Briand e Labiche [BL02], que criam diagramas de atividades para representar dependências entre as funcionalidades do sistema. O Capítulo 3 detalhará como os modelos são especificados e criados a partir dos casos de uso do SUT para apoiar as atividades de testes funcionais. No segundo nível, a abordagem para criação dos modelos baseia-se na abordagem de Hartmann *et al* [HVFR04], que propõe a representação desses cenários internos através de modelos (Diagramas de Atividades) representando o fluxo de execução.

É importante destacar que como os modelos nesta atividade do processo são criados a partir de casos de uso, então estes modelos são independentes da plataforma de desenvolvimento do SUT (PIM), ou seja, os modelos não possuem aspectos relacionados a plataforma em que o SUT deve ser implementado. A separação de modelos independentes de plataforma e modelos específicos para uma determinada plataforma (PSM) tem sido bastante utilizada em abordagens de desenvolvimento com arquitetura dirigida a modelos (em inglês, *Model-Driven Architecture* –MDA) [Gro03a].

Criar Cenários de Teste

A partir dos modelos criados durante o projeto dos testes será realizada a seleção dos cenários de teste do sistema, que descrevem o fluxo de controle de execução do sistema. O uso de cenários, para toda equipe de testes, facilita o aprendizado de como o SUT deve funcionar [Kan03]. Além disso, os cenários são relacionados diretamente aos requisitos do SUT, o que facilita a rastreabilidade de cenários de teste por requisitos e, conseqüentemente, as falhas reveladas na execução de cada cenário podem ser associadas a esses requisitos.

Nessa etapa são selecionados os Cenários de Uso (ou Cenário de Teste) do sistema a partir dos modelos comportamentais do sistema. Um Cenário de Teste descreve uma seqüência de passos de execução do sistema, ou seja, uma seqüência de interações entre o sistema e seus atores. Os cenários de teste são criados a partir dos modelos para testes (Seção "Criar Modelos para Testes") e combinam cenários de execução de um ou mais casos de uso. Os cenários de teste devem ser selecionados de forma a atender aos critérios de cobertura dos modelos, que foram estabelecidos nas fases de planejamento de testes. O número de cenários de teste selecionados pode variar de acordo com o critério adotado.

Os cenários não contêm as entradas de teste para a realização efetiva desses passos, mas servem como guia para criá-los, pois o objetivo nesta abordagem é selecionar dados que possibilitem cobrir cada cenário de teste. Entre os passos devem estar descritos também os resultados esperados, que são obtidos a partir da descrição dos casos de uso. Além disso, o nível de abstração usado para especificar estes cenários depende do nível de abstração especificado nos modelos de teste. Como os modelos são derivados dos casos de uso então os cenários gerados possuem um alto nível de abstração. Desta forma, eles são descritos de forma textual e isso facilita o seu entendimento pela equipe de teste.

Como os modelos utilizados para criação dos cenários de teste são modelos PIM (criados na atividade de teste anterior), então os cenários obtidos a partir deles também são independentes da plataforma do SUT. O Capítulo 4 mostra como os modelos são transformados em um Grafo de Fluxo de Controle (de forma automática) e, a partir desse CFG, um algoritmo é proposto para automatizar a seleção de cenários de teste.

Preparar Ambiente de Testes

Nesta atividade devem ser definidos a arquitetura e o ambiente para realização dos testes. Para isso são definidas quais tecnologias, *frameworks* linguagens de programação (ou *script*) usadas para implementação dos CTs e demais ferramentas e métodos, que serão necessários para apoiar a realização dos testes. É importante ressaltar que são definidos "quais" tecnologias devem ser utilizadas no projeto e realização de testes e não "como" elas serão usadas durante os testes.

Durante a preparação do ambiente de testes devem ser definidos também os componentes de apoio aos testes como, por exemplo, os *drivers*, os *stubs* e os oráculos de testes [Bin99]. Um *driver* é uma classe principal ou um programa externo que aplica os casos de testes no SUT. Um *stub* é uma implementação falsa, ou implementação parcial de um componente (ou outro sistema) que mantém alguma interação com o SUT, recebendo requisições ou provendo serviços.

Para verificação do resultado esperado da execução dos testes é necessário especificar os oráculos e (ou) que serão utilizados durante os testes. Os oráculos de teste especificam qual o comportamento esperado como resposta da aplicação a um conjunto de entradas de testes. Por exemplo, os oráculos podem ser consultas à base de dados, avaliação de relatórios, avaliação de resultado apresentado na interface do sistema ou qualquer outra forma de avaliação que possibilite visualizar se o conjunto de entradas aplicadas por um CT gerou o comportamento esperado no sistema [Bin99, Capítulo 18]. Um cenário de teste pode possuir um ou mais oráculos, eles são especificados para um passo ou para um conjunto de passos dos cenários de teste, dependendo da quantidade de ações necessárias no SUT que irão gerar um resultado esperado.

Especificar os Casos de Teste

O CT é uma instância de um cenário de teste (Seção "Criar Cenários de Teste") com dados de entrada e saídas esperadas para sua execução. A execução dos CTs têm como objetivo avaliar o comportamento do sistema usando a seqüência de passos descritas em cada cenário de teste. Para isso devem ser especificados os dados de entrada e respectivos oráculos para execução de cada cenário de teste. Deve-se aqui combinar o critério de cobertura escolhido e o critério de escolha de dados, tais como, partição de equivalência ou análise de valores limites [Pre01].

Nesta atividade, os testes podem ser descritos de maneira mais formal, já considerando informações do ambiente de execução e arquitetura do sistema implementado. A especificação do caso de teste é uma notação intermediária entre um cenário de teste e o caso de teste implementado, que pode ser aplicado (executado) de forma automática.

Conforme já mencionado anteriormente, os cenários de teste são independentes da

implementação do sistema. Já os casos de teste estão prontos para serem executados, seja de forma automática ou manual. O que torna a especificação de cenários de teste e casos de teste similar ao padrão proposto por Rayner [Ray87], onde existem os casos de teste genéricos, que são independentes da implementação do sistema e o seu refinamento em casos de teste abstratos antes de sua implementação. Além disso, como os casos de teste possuem informações sobre o ambiente de execução e a arquitetura do sistema então eles podem ser considerados dependentes da plataforma de execução.

Para definição mais precisa dos artefatos de teste, a UML 2.0 define um Perfil UML de Testes (em inglês, *UML 2.0 Testing Profile* – U2TP) [Gro05]. A U2TP é uma linguagem para apoiar as atividades de documentação, projeto, visualização e construção de artefatos de teste [Gro05]. Desta maneira, durante a fase de especificação dos casos de teste, os artefatos produzidos podem ser especificados ou persistidos usando uma notação em conformidade com os padrões da U2TP.

Quando necessário, o Caso de Teste pode ser implementado em uma linguagem de programação (ou de *script*), essa abordagem nem sempre é realizada pois a implementação e manutenção dos testes de forma automática é uma atividade custosa em situações onde o SUT sofre constantes alterações em sua interface.

2.6.4 Execução dos Testes

Nesta fase os casos de teste são executados e para cada caso de teste executado é dado um veredicto de sua execução: *passou*, *não passou* ou foi *não conclusivo*, este último caso ocorre quando não é possível coletar informação para avaliação da execução. Havendo algum defeito no SUT no caso de teste, devido a falhas ou enganos cometidos na especificação ou implementação do caso de teste, e não no sistema, é necessária a revisão do caso de teste. Se houver defeitos no sistema, os defeitos percebidos devem ser reportados às equipes de análise e desenvolvimento responsáveis para correção da falha que ativou o defeito, seja esta falha de implementação ou de especificação (documentação do sistema).

Para apoiar a execução dos Casos de Teste (CT) nesta fase, foram identificadas quatro principais atividades de execução: Preparar o CT, executar os passos do CT, avaliar os oráculos de teste e reportar resultados. A segunda e terceira atividades podem ser realizadas de forma intercalada, para adequação da avaliação dos oráculos conforme as necessidades de cada caso de teste. A seguir essas atividades são detalhadas.

Preparar o CT

Para executar os CTs é necessário que o ambiente de execução dos testes seja controlado. Antes da execução de cada CT deve-se garantir que o sistema está em um estado inicial consistente, sendo necessário tomar alguns cuidados, como garantir que o ambiente de

testes está é um ambiente isolado e não existem execuções os testes são aplicados sequencialmente em casos em que a execução simultânea de CTs pode causar impacto no resultado dos testes. Este tipo de situação pode ser observada em sistemas que não permitem o acesso a determinados recursos ou funcionalidades de forma simultânea assim, caso dois CTs sejam executados de forma concorrente, o resultado da execução de um dos CTs pode ser afetado devido aos possíveis bloqueios de recursos ou funcionalidades. Nesta situação é importante garantir que o Assim, é importante verificar todas as pré-condições e invariantes para realização dos testes, para garantir que o CT possa ser aplicado sem interferência de fatores externos.

Executar o CT

O CT deve ser executado conforme a descrição apresentada em seus passos. A automação da execução, bem como a automação da avaliação dos resultados, é possível mas nem sempre viável, principalmente na avaliação dos resultados durante a execução dos cenários de teste [Whi00], pois os resultados esperados são obtidos através da especificação em alto nível, que muitas vezes, pode ser uma simples informação foi ou não exibida em uma tela do SUT. Além disso, a criação de *scripts* para executar os testes nem sempre é viável, pois em sistemas onde ocorrem constantes alterações de interface, as alterações acabam provocando grandes mudanças no CT automatizado e, conseqüentemente, torna-se necessária a realização de manutenção contínua desses CTs.

Neste cenário, uma abordagem interessante para automatizar os testes funcionais é a de gravação e reprodução dos testes através de ferramentas para *capture-playback* de testes. Na primeira vez em que o CT é executado manualmente, essas ferramentas podem ser utilizadas para gravar a execução dos testes. Desta forma, nas próximas vezes que o CT gravado precisar ser re-executado, a ferramenta de *capture-playback* pode ser usada para reproduzir de forma automática o CT gravado. Para automatizar a execução dos testes através de *capture-playback*, neste trabalho foi utilizada ferramenta *Selenium* [Ope06], que grava e re-executa as interações de usuários em navegadores de Internet.

Avaliar a Execução do Teste

Durante a execução do CT é necessário avaliar o comportamento do sistema. Esta avaliação é feita através da comparação das respostas do sistema com seus resultados esperados (oráculos de teste). Esta avaliação pode ser feita de forma fracionada, conforme forem executados os passos do CT. O nível de detalhamento para realizar essa avaliação deve estar de acordo com os objetivos dos testes [AD97]; esse nível de detalhamento foi definido na Seção 2.6.3.

Reportar Resultados Obtidos

Após a execução dos testes os resultados obtidos devem ser reportados, independentemente de que veredicto de execução, seja passou ou seja não passou. Esta atividade deve ser feita imediatamente após a conclusão da execução do caso de teste para possibilitar o acompanhamento parcial do "Relatório de Resumo de Teste", que contém ainda a descrição resumida das atividades de teste e tempo total para a realização destas atividades [dPPF01, Capítulo 13]. Desta forma, pode-se obter informações do total de casos de teste executados, total de defeitos encontrados e também avaliar o cronograma e detectar possíveis atrasos nas atividades de teste. Para realizar essa etapa podem ser usadas ferramentas para rastrear defeitos, conhecidas também como *defect tracking*. Duas ferramentas bastante conhecidas e usadas para isso são o *Bugzilla* [Org07] e o *ClearQuest* [Cor07].

Outra atividade realizada durante a execução dos testes é a de revisão do plano e cronograma de testes. Além disso, quando necessário, devem ser obtidos novos artefatos de desenvolvimento atualizados e, se necessário, a atualização dos modelos de testes criados nas fases de preparação dos testes.

2.6.5 Análise de Resultados

Para o acompanhamento parcial da execução dos testes, a análise de resultados deve ser realizada durante a atividade de execução dos testes. Entretanto, somente após a execução dos testes é possível consolidar os resultados dos testes aplicados em uma bateria de testes.

Os resultados dos testes aplicados ao SUT podem ser resumidos na forma de um simples demonstrativo para consultas rápidas por responsáveis do projeto. Este resumo deve conter, pelo menos, o número de defeitos encontrados, total de CTs que passaram e não passaram durante a execução, total de defeitos encontrados classificados por severidade (por exemplo, crítico, alto, médio ou baixo). Uma forma mais detalhada de classificar os defeitos e analisar os resultados dos testes foi proposta pelo grupo de pesquisas da IBM, e foi chamada de *Orthogonal Defect Classification* (ODC) [CP02].

A ODC propõe uma forma classificação de defeitos que pode ser realizada após a correção das falhas que originaram os defeitos encontrados. Na ODC os defeitos são classificados de acordo com a descrição detalhada de sua causa. Por exemplo, se for uma falha de código que causou o defeito, sua classificação deve ser feita relacionada exatamente à falha que causou este defeito como, por exemplo, o atributo não iniciado corretamente, falha nas condições (*if, while* e etc), ausência de condições, entre outras possíveis causas.

Com os resultados resumidos dos testes é possível avaliar se o sistema está de acordo com os requisitos de qualidade definidos durante o planejamento dos testes, essas informações podem ser apresentadas no "Relatório de Resumo dos Testes", que foi apre-

sentado na Seção 2.6.4. Além disso, os casos de teste devem ser revisados e os últimos artefatos de testes devem ser produzidos e concluídos. Alguns destes artefatos são: o "Relatório de Incidentes de Testes" e o "Diário de Teste". O Relatório de incidentes de teste tem como objetivo documentar qualquer evento considerado importante, ocorrido durante a execução dos testes, enquanto o Diário de testes tem por objetivo registrar todas as atividades de teste realizadas pelos membros da equipe de testes.

Após a avaliação dos defeitos pelos analistas, caso o SUT não atenda aos requisitos de qualidade desejados, devem ser realizadas as correções das falhas pela equipe de desenvolvimento. Em seguida é gerada uma nova versão do SUT com as correções dos defeitos realizadas por parte das equipes responsáveis e, desta forma, os testes devem ser re-executados para uma nova avaliação. O processo de re-execução pode ser feito através duas abordagens para seleção dos testes: retestar tudo, ou usando a abordagem seletiva, sendo essa utiliza para seleção de testes de regressão (Seção 2.2.1).

O sistema pode passar para outras etapas de teste caso seja considerado como atingido os seus requisitos de qualidade do sistema ou o sistema não apresente mais defeitos. As próximas etapas de teste podem ser ainda de testes de sistema [Pre01], tais como, desempenho, robustez, disponibilidade e até mesmo os testes de aceitação para que o sistema possa ser entregue ao cliente.

2.7 Resumo

Neste capítulo foram apresentados os fundamentos teóricos de testes de software, Grafos de Fluxo de Controle e o processo adotado para realização dos testes. É importante ressaltar que o processo de testes detalhado apresenta de forma estruturada as atividades necessárias para realização de testes funcionais e ele foi adotado, pois dá apoio ao método para seleção automática de cenários de testes, que será apresentado no Capítulo 4. A aplicação do processo foi feita utilizando estudo de caso apresentado no Capítulo 5. Além disso, este processo pode ser integrado a qualquer processo de desenvolvimento em que a especificação do sistema contenha casos de uso ou o qualquer tipo de especificação que possibilite a criação dos modelos comportamentais propostos nesta dissertação, que serão detalhados no Capítulo 3.

Capítulo 3

Modelos Utilizados para Especificar Testes Funcionais

Este capítulo apresenta uma abordagem que utiliza modelos nas atividades de projeto e criação de casos de teste funcionais, especificamente o Diagrama de Atividades (DA) da UML. Os modelos são criados a partir da especificação de casos de uso do SUT, modelos muito utilizados para o detalhamento dos requisitos de sistemas [Rum94, Gro03b, CL01, Coc01b]. A modelagem proposta nesta dissertação é feita em dois níveis. No primeiro nível devem ser representadas as dependências entre os casos de uso e, no segundo nível, os fluxos internos de cada caso de uso (CDU).

O restante deste capítulo é organizado da seguinte forma: a Seção 3.1 apresenta uma breve descrição da especificação de um SUT que foi utilizado para ilustrar a criação dos modelos. Além disso, ela também apresenta o modelo de descrição dos casos de uso que foi adotado pela equipe de desenvolvimento envolvida neste trabalho. A Seção 3.2 descreve os DAs que foram utilizados durante o projeto dos testes funcionais e detalha os aspectos que motivaram a sua escolha. Finalmente, a Seção 3.3 detalha como deve ser feita a especificação comportamental do SUT em dois níveis, representando as dependências entre os CDUs e os fluxos internos.

3.1 Descrição do Sistema e Especificação de Casos de Uso

O exemplo utilizado no restante deste capítulo é um sistema real desenvolvido por alunos do Centro Superior de Educação Tecnológica da UNICAMP (CESET) em Limeira – SP no ano de 2007 [ACM07]. Devido à grande quantidade de casos de uso do sistema, não foi possível apresentar toda a especificação criada pelos desenvolvedores. Desta forma,

esta seção apresenta uma breve descrição do sistema utilizado como exemplo para criação dos modelos de teste, além dos padrões adotados para especificação dos casos de uso do sistema.

3.1.1 Descrição do Sistema

O sistema tem por objetivo prover uma solução para cadastro de currículos para portadores de necessidades especiais (deficientes), promovendo a inclusão dos portadores de necessidades especiais no mercado de trabalho. O sistema deve ser acessível aos deficientes, entidades não governamentais (ONGs) e empresas.

Os deficientes e as entidades poderão se cadastrar e, além do cadastro, deve ser possível disponibilizar o currículo dos deficientes. Já as empresas poderão oferecer vagas de emprego e buscar currículos cadastrados no sistema pelos portadores de necessidades especiais que melhor se encaixem no perfil da vaga oferecida.

Abaixo é apresentada uma breve descrição das principais informações manipuladas no sistema:

- *Deficiente*. Deve ser possível incluir deficientes no sistema. Além disso, cada deficiente pode ainda atualizar ou excluir seu cadastro. Essas atividades podem ser realizadas pelos próprios deficientes e também pelas entidades.

- *Entidade*. Deve ser possível realizar atividades de manutenção (inclusão, atualização e exclusão) das entidades no sistema. Além disso, estas entidades poderão consultar deficientes que estejam relacionados com a entidade, ou não.

- *Empresa*. As empresas podem incluir seu cadastro no sistema e, assim como as entidades, podem também atualizar e remover seu cadastro. Outra operação permitida é a de consulta de currículos de deficientes.

- *Vaga*. As funcionalidades relacionadas às vagas são incluir e desativar vagas, que podem ser realizadas pelas empresas provedoras das vagas. Além disso, as entidades e os deficientes podem realizar consultas com o objetivo de visualizar as vagas ativas no sistema e também se candidatar à uma ou mais vagas de emprego.

- *Currículo*. Os deficientes cadastrados ou entidades cadastradas que os representam poderão incluir currículos dos deficientes no sistema. Juntamente a isso, um currículo pode estar: (i) ativo, ou seja, publicado para que empresas, entidades e outros deficientes o visualizem através de consultas; ou (ii) inativo, situação em que somente o próprio deficiente pode visualizar seu currículo. O deficiente também pode verificar o status de seu currículo (ativo ou inativo).

As funcionalidades do sistema foram detalhadas e especificadas pelos analistas através de CDU [Gro04]. Foram identificados vinte e oito (28) casos de uso que especificam as funcionalidades, bem como cinco perfis de atores no sistema: *Administrador*, *Deficiente*, *Empresa*, *Entidade* e *Visitante*. Os atores são agentes que interagem direta ou indiretamente com o sistema.

3.1.2 Detalhamento dos Casos de Uso

Os casos de uso são descritos durante a fase de Análise de Requisitos do sistema e, em muitas vezes, essa descrição é feita de forma textual [Rum94, Gel04]. Para minimizar tanto o número de problemas em projetos de sistemas, quanto as ambigüidades em sua especificação, várias abordagens propõem o uso de padrões de estruturação para a descrição textual dos casos de uso [Gel04, CL01, Rum94]. Além de utilizar padrões para detalhamento dos casos de uso é necessário delimitar quais informações na descrição do caso de uso são relevantes para o projeto e a criação de testes [Gel04].

A notação adotada neste trabalho é baseada nos padrões de notação propostos por Gelperin [Gel04] e Ferreira [dMF01]. O formato da descrição de CDUs proposto por Gelperin possibilita a redução das ambigüidades, tornando possível realizar a automação do projeto de casos de teste. Por outro lado, Ferreira propôs uma classificação dos cenários internos de exceção em dois tipos: *cenários recuperáveis* e *cenários de falha*. Nesta dissertação, a especificação dos cenários de exceção é representada de forma diferenciada nos modelos, conforme será mostrado na Seção 3.3.2.

A descrição dos casos de uso do sistema de cadastro de deficientes [ACM07] foi realizada utilizando as seguintes informações:

- *Nome ou Identificador*. Um caso de uso deve conter um nome ou identificador, sendo que este deve ser único para não haver ambigüidade.
- *Descrição*. Breve descrição do comportamento do caso de uso em seu cenário de uso principal.
- *Atores*. Quais atores podem interagir de forma direta ou indireta, com o sistema ou componente a ser desenvolvido.
- *Pré-condições*. Condições que devem ser verdadeiras antes da execução do caso de uso, ou seja, para que seja possível instanciar um caso de uso.
- *Invariantes*. Condições que devem ser sempre verdadeiras durante toda a execução do caso de uso.

- *Cenário Principal.* Exibe o cenário principal de execução de um caso de uso. Ele é descrito através de um conjunto de passos que devem ser seguidos para que este cenário principal possa ser executado. Um passo no caso de uso consiste em uma ação a ser executada por um ator ou pelo sistema, ou ainda um conjunto de possíveis escolhas que podem ser feitas por um deles. Toda vez que o caso de uso é chamado ou instanciado por um ator, sua execução deve iniciar sempre no primeiro passo de seu cenário principal. Sendo assim, este cenário deve ser obrigatoriamente descrito em todos os casos de usos.

- *Cenários Alternativos.* Um caso de uso pode conter zero ou mais cenários alternativos. Eles são usados para representar passos adicionais e tratamentos de casos especiais no decorrer do cenário principal. Este tipo de fluxo é opcional na descrição do caso de uso, podendo ser suprimido em casos onde ele não se aplica.

- *Cenários de Exceção.* Além dos cenários alternativos, há também os cenários de exceção, cujo o objetivo é descrever o comportamento esperado do sistema em situações em que podem ocorrer problemas ou situações críticas durante a execução do cenário principal ou alternativo. Os *cenários recuperáveis* devem ser detalhados com um prefixo *ER* e os *cenários de falha* com o prefixo *EF*. Estes cenários são importantes, uma vez que eles descrevem como o sistema deve se comportar quando ocorre um problema, o que muitas vezes indica o que o sistema não deve fazer quando ocorrem problemas. Este tipo de cenário também é opcional.

- *Pós-condições.* Condições que devem ser verdadeiras após a execução do caso de uso em seu cenário principal. Caso exista algum cenário alternativo onde a execução do CDU termina, a sua pós-condição também deve estar representada no cenário.

A Tabela 3.1 apresenta um exemplo de CDU descrito textualmente. O exemplo apresentado é o CDU *Consultar Deficiente* do sistema. Todos os casos de uso desse sistema foram descritos seguindo esta padronização, apresentando estrutura similar à exibida na tabela.

A descrição dos CDUs foi realizada pela equipe de desenvolvimento e, a partir dessa descrição, foram criados modelos comportamentais do sistema para auxiliar as atividades de projeto de teste, apresentadas no processo do Capítulo 2. Dois tipos de modelos comportamentais foram adotados. O primeiro descreve os cenários dos casos de uso e o segundo a ordem de execução dos casos de uso. É importante ressaltar que a criação dos modelos comportamentais não depende necessariamente que os casos de uso tenham sido escritos de forma textual, dado que estes modelos podem ser detalhados diretamente a partir dos requisitos do sistema.

Tabela 3.1: Descrição do Caso de Uso Consultar Deficiente do SUT.

Caso de Uso	Consultar Deficiente
Descrição	Consultar dados dos deficientes cadastrados
Atores	Deficiente, Entidade
Pré-condições	Usuário deve estar logado.
Invariantes	O ator deve permanecer autenticado.
Cenário Principal	**P1. Selecionar Consultar Deficiente** O *usuário* seleciona a opção consultar dados do deficiente **P2. Verificar Permissões** É incluído o *caso de uso* "Verificar Permissões", passando como parâmetro o identificador do usuário logado. **P3. Confirmar identificador do deficiente** Caso seja verificado que um deficiente foi passado como parâmetro para realização da consulta, então o *sistema* continua a execução normalmente; caso contrário, segue para o cenário alternativo A3 Caso seja solicitado pelo *usuário* a opção Cancelar, segue para o cenário alternativo A1. **P4. Exibir dados do Deficiente** O *sistema* deve disponibilizar os dados do deficiente consultado. Vai para cenário Alternativo A2, caso for selecionada a opção Atualizar dados. **P5. O caso de uso é encerrado**
Cenários Alternativos	**A1. Cancelamento da consulta** **a.** Após passo 3 do Cenário Principal o *usuário* solicita o cancelamento da operação. **b.** O caso de uso é encerrado. **A2. Alteração de dados** **a.** Após passo 3 do Cenário Principal *usuário* solicita a opção Alterar dados. **b.** É incluído o *caso de uso* "Atualizar Deficiente". **c.** O caso de uso é encerrado. **A3. Parâmetro identificador do deficiente nulo** **a.** Após passo 2 do Cenário Principal o sistema verifica que o parâmetro identificador está nulo. **b.** O *sistema* exibe os filtros para a entidade realizar a consulta. **c.** A *entidade* preenche os filtros desejados. **d.** A *entidade* confirma a pesquisa. **e.** O *sistema* retorna o resultado da consulta. **f.** Ver detalhes do cadastro O *usuário* seleciona ver detalhes. Caso desejar ver os detalhes de algum deficiente retornado pela consulta, vai para o passo P4. **g.** O caso de uso é encerrado.
Cenários de Exceção	***ER.1* Erro de Acesso ao banco de dados** **a.** No passo 4 do Cenário Principal o *sistema* não consegue acessar o banco de dados. **b.** O *sistema* exibe mensagem: "Erro ao consultar o banco de dados. Tente mais tarde". **c.** O caso de uso é encerrado sem realizar a consulta.
Pós-condições	Os dados do deficiente devem ser exibidos.

3.2 Modelos no Projeto de Teste

Dentre as inúmeras opções de modelos para derivar cenários de teste [Bin99, Edw00, Pet01, BBM02, CCD03, HN04], este trabalho optou pelo DA, tal como outros autores o fizeram [MXX06, CLL05, BLL04a, LJX+04, HVFR04, BL02]. Os DAs são utilizados aqui para apoiar as atividades de projeto de teste funcionais que é realizada a partir do detalhamento dos casos de uso do sistema. Os modelos propostos representam o comportamento do sistema através dos cenários de uso (internamente) de cada caso de uso do SUT. Além disso, representam também a ordem de execução dos casos de uso, ou seja, o cenário de execução entre casos de uso (externamente).

Alguns aspectos que favoreceram a escolha do Diagrama de Atividades durante o projeto dos testes foram:

- Possui representação gráfica de fácil entendimento;
- Permite representar o fluxo de controle entre ações de forma análoga à representação do CFG, modelos utilizados em teste "caixa branca";
- Possibilita a representação dos modelos de forma hierárquica;
- Possibilita a representação de fluxos de exceções de forma diferenciada dos demais fluxos;
- Possibilita a representação de ações concorrentes;

Em relação aos outros diagramas como, por exemplo, os diagramas de colaboração e seqüência, a representação dos DAs facilita o entendimento da especificação pelas pessoas envolvidas. Utilizando os DAs é possível adotar critérios de teste comuns e muito utilizados para realização de testes, tais como, os critérios de teste "caixa branca". A seguir serão detalhados os três últimos aspectos mencionados acima.

A Figura 3.1 detalha os principais elementos do diagrama de atividades, que serão utilizados no decorrer do texto. Nesta dissertação não serão detalhados os elementos do Diagrama de Atividades da UML 2.0, devido a grande número de elementos presentes na especificação do Diagrama de Atividades. Assim, para obtenção de maiores detalhes dos elementos que podem ser representados com o Diagrama de Atividade da UML 2.0, pode-se consultar o manual de especificação técnica da UML [Gro04, Capítulo 12].

Decomposição hierárquica

O DA possui uma característica de representação que possibilita a descrição do comportamento de uma ou mais ações suas em outro modelo, que não precisa necessariamente

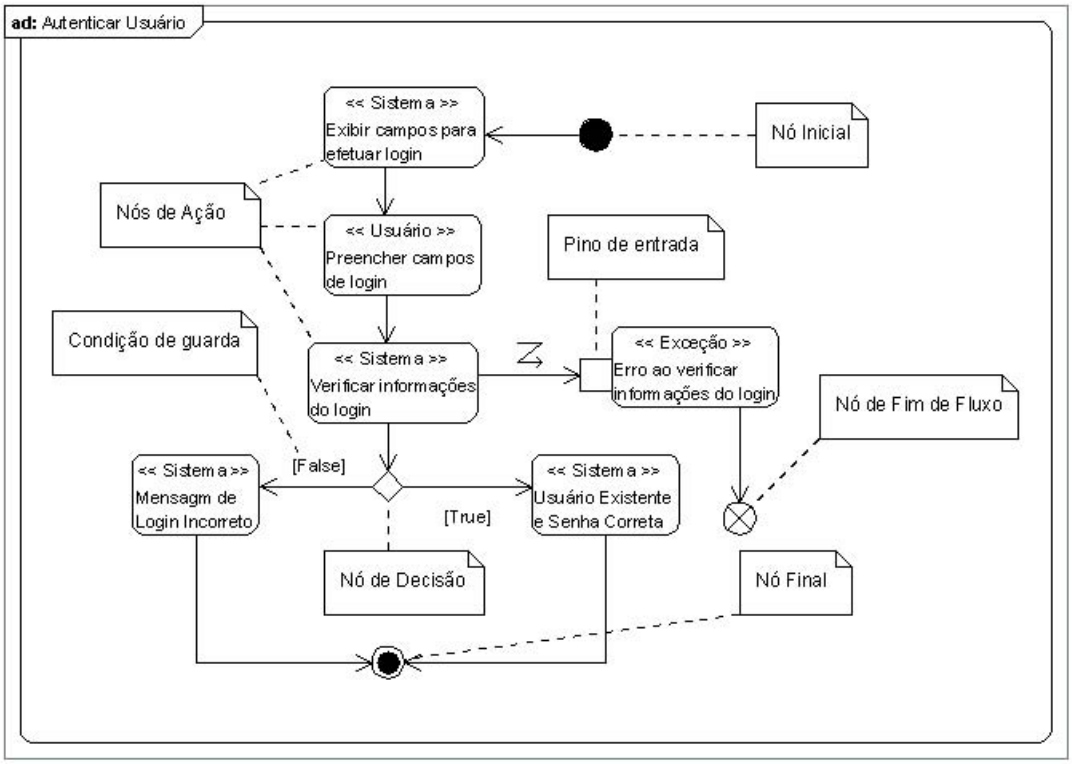

Figura 3.1: Exemplo de Diagrama de Atividades e Descrição dos principais elementos de sua representação.

ser um DA [Gro04]. Ou seja, um DA pode conter ações que, posteriormente, podem ser detalhadas em outros Diagramas UML através de ações com chamada de comportamento, chamado na UML 2.0 de *CallBehaviorAction* [Gro04].

A decomposição hierárquica das atividades é exemplificada na Figura 3.2, onde a ação *Cadastro* tem seu comportamento especificado em outro DA, mostrado na Figura 3.3, que possui as ações *Preencher Formulário*, *Enviar Formulário* e *Salvar Formulário*. Este conceito é bastante eficiente para representação de casos de uso, pelo fato de possibilitar as relações de inclusão (em inglês, *include*) e extensão (em inglês, *extend*) na modelagem utilizando os DAs [Gro04, Coc01b]. O símbolo ⫛ no canto direito na atividade *Cadastro* da Figura 3.2 é a representação visual de que essa atividade é detalhada em outro modelo através de uma chamada de comportamento.

Representação de Fluxos de Exceções

Os DAs possuem uma notação específica para representar tratadores de exceção. Esta representação é muito interessante, pois ela possibilita a representação dos fluxos de exceção

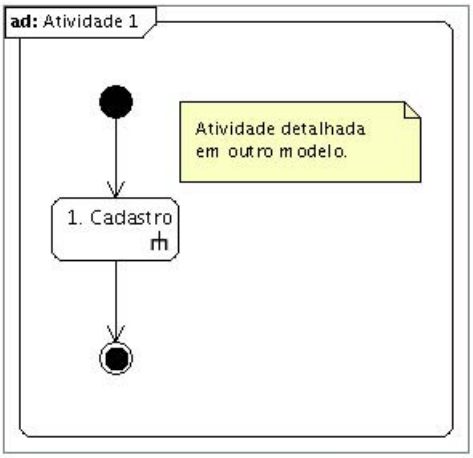

Figura 3.2: Atividade Cadastro.

de forma diferenciada aos demais fluxos dos casos de uso (apresentados na Seção 3.1.2).

A Figura 3.3 apresenta um exemplo de fluxo de exceção representado no DA. A ação *Salvar Formulário* pode encerrar sua execução normalmente ou pode lançar uma exceção para o tratador *Problema no Acesso ao Banco de Dados*. Uma transição no DA que representa o lançamento da exceção é representada pelo símbolo *"Z"* e deve conter obrigatoriamente um pino de entrada para o tratador da exceção. Este pino de entrada identifica o tipo da exceção que é transmitida ao tratador de exceção. A representação gráfica da exceção na Figura 3.3 não está em conformidade com proposta na UML 2.0, devido as restrições de modelagem da ferramenta utilizada.

Representação de Ações Concorrentes

Outra característica importante do DA é a representação de ações concorrentes, oferecendo suporte à execução de atividades paralelamente. O fluxo de execução das ações quando encontra um vértice de controle chamado de *fork* pode ser dividido em mais de um fluxo, que são executados assincronamente até que sejam sincronizados novamente quando todos os fluxos atingirem um vértice de controle chamado *join*, que possui representação gráfica similar à do *fork*. Assim, todos os fluxos de execução existentes entre o *fork* e o *join* ocorrem concorrentemente. Vale lembrar que a modelagem dos DAs utilizando atividades concorrentes não faz parte do escopo deste trabalho de mestrado, pois a especificação assíncrona agrega uma complexidade muito alta para a derivação dos testes. Para os sistemas assíncronos, que necessitam da representação de ações concorrentes, o testador precisa definir explicitamente uma ou mais ordens possíveis para execução das ações no DA sem a utilização dos *forks* e *joins*.

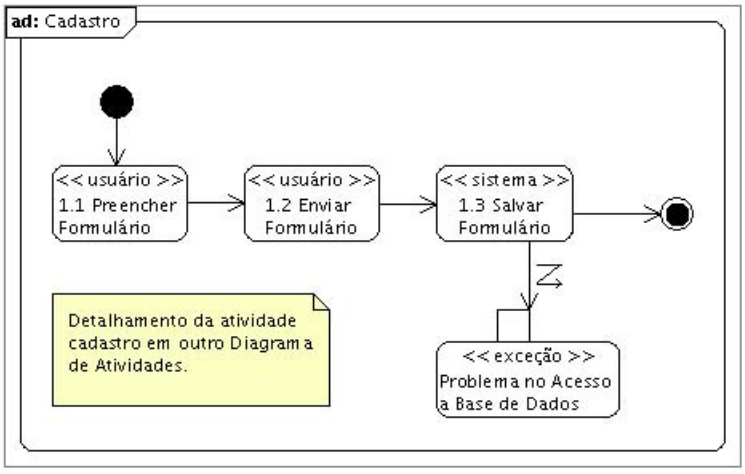

Figura 3.3: Sub-atividades para realização da atividade Cadastro.

3.3 Especificação Comportamental do Sistema

Neste trabalho, a especificação comportamental do sistema é realizada através de DAs e é baseada na abordagem proposta por Rocha [Roc05]. Rocha utilizou DAs para identificar possíveis seqüências de execução para os métodos das interfaces de componentes. Sua abordagem foi dividida em dois níveis de representação:

- o primeiro nível, chamado de nível principal, ilustra o fluxo lógico de execução entre as operações (métodos) providas pelo componente em testes.
- o segundo nível ilustra o fluxo de execução ocasionado pela invocação de um método representado no primeiro nível. Neste nível é representado a invocação dos métodos das interfaces requeridas (dependências) e os possíveis fluxos de exceção durante a interação com as interfaces requeridas.

É importante comentar que a abordagem proposta por Rocha é "caixa-preta", pois em ambos os níveis de representação são detalhadas baseando-se somente a especificação contratual do componente.

Neste trabalho, a abordagem proposta por Rocha é estendida para a representação comportamental de sistemas ao invés de componentes, mas utilizando também dois níveis de representação dos DAs. No primeiro nível os DAs são utilizados na representação das dependências externas de execução entre os CDUs, ou seja, o fluxo de controle de execução entre os casos de uso. Esse modelo é similar ao primeiro nível proposto por Rocha [Roc05], que foi baseado na proposta de Briand e Labiche [BL02].

No segundo nível são criados os diagramas que especificam os fluxos internos dos casos de uso, baseando-se na abordagem proposta por Hartmann *el al* [HVFR04]. Neste nível,

diferentemente da proposta de Rocha, os modelos são criados a partir da descrição dos passos dos fluxos (principal, alternativos e de exceção) dos casos de uso. Além disso, como os casos de uso podem realizar a inclusão de outros casos de uso, então um diagrama neste nível (segundo nível) pode ser relacionado diretamente a outro(s) diagrama(s) no mesmo nível, representando assim inclusões e extensões nos casos de uso.

Os dois níveis da modelagem comportamental são detalhados a seguir.

3.3.1 Fluxo de Controle de Execução entre Casos de Uso

Os casos de uso presentes na especificação de sistemas possuem uma relação de precedência de execução. Esta relação é geralmente especificada na forma de pré-condições nos casos de uso, requerendo que outro CDU ou que condições iniciais sejam satisfeitas para execução deste CDU. Por exemplo, suponha que existem os CDUs *Ativar Currículo* e *Cadastrar Currículo* no sistema de cadastro de deficientes. Um currículo de deficiente só pode ser ativado se este estiver cadastrado, o que caracteriza uma ordem de precedência para a execução entre esses casos de uso. Assim, os testes funcionais devem ser preparados e realizados de forma a atender a essa precedência de execução, satisfazendo as pré-condições conforme são executados os casos de uso.

O diagrama que representa o fluxo de controle entre casos de uso é necessário para a criação de casos de teste incorporando vários casos de uso na ordem de execução correta. Cada cenário de teste, obtido através desses modelos, cobrirá um ou mais casos de uso conforme forem as dependências entre eles. Os cenários de teste funcionais ou procedimentos de teste do sistema, como definido por Filho [dPPF01], descrevem a seqüência de passos necessária para executar um cenário de teste. Um cenário de teste para executar as ações do CDU *Ativar Currículo* deve realizar primeiramente os passos do CDU *Cadastrar Currículo*, tornando esse cenário independente dos demais cenários, o que não corre quando a seleção dos testes é realizada para cada CDU isoladamente.

Para apoiar o projeto dos cenários de teste, este trabalho propõe a criação de um modelo para representar a ordem de precedência entre os casos de uso. Este modelo é criado utilizando o DA e é chamado de Diagrama de Fluxo de Controle entre Casos de Uso (DAFC). O uso deste tipo de modelo representando os cenários de execução entre casos de uso, que apóia o projeto de cenários de teste funcional foi proposto inicialmente por Briand e Labiche [BL02]. Entretanto, nesta dissertação este modelo será integrado aos modelos que representam os cenários de casos de uso através da decomposição hierárquica dos DAs, ao invés de diagramas de seqüência representados por expressões regulares para derivação dos testes, conforme foi proposto por Briand e Labiche.

Cada ação no DA representa uma chamada a um caso de uso do SUT, e uma transição entre dois casos de uso somente é inserida caso exista precedência na execução desses casos

de uso. A precedência entre os casos de uso no DAFC é definida através da avaliação das pré-condições e pós-condições de cada caso de uso.

Uma transição no DAFC só existe se a pós-condição de um CDU não violar a pré-condição do CDU posterior, ou se a pré-condição de um exija que outro CDU seja executado anterior. A verificação de pré-condições e pós-condições dos casos de uso para criação desses modelos é baseada na abordagem proposta por Edwards [Edw00]. Edwards cria um modelo para representar a ordem de execução das operações providas por um componente de software. Em seu modelo são inseridas transições entre as operações do componente para a execução de cenários de teste para este componente, mas não são inseridas todas as possíveis transições no modelo para que não sejam gerados um número exponencial de possíveis cenários de teste. No DAFC proposto nesta dissertação a inserção das transições é realizada de forma manual.

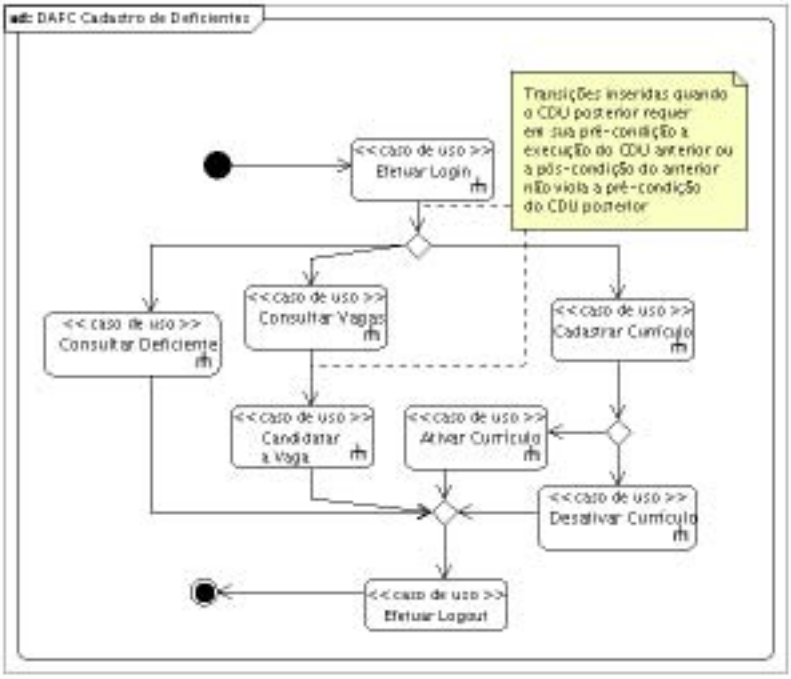

Figura 3.4: Parte do DAFC para o SUT.

A Figura 3.4 apresenta um exemplo do DAFC para uma parte do SUT. Neste diagrama as ações que correspondem aos CDUs (realizados pelos atores) possuem o estereótipo *<<caso de uso>>*. Ainda na mesma figura, as ações do diagrama que possuem o símbolo ⫛ indicam que esta ação possui uma chamada de comportamento. Além disso, o fluxo dos casos de uso deve ser detalhado para cada chamada de comportamento (segundo nível da modelagem).

3.3.2 Descrição dos Cenários dos Casos de Uso usando DAs

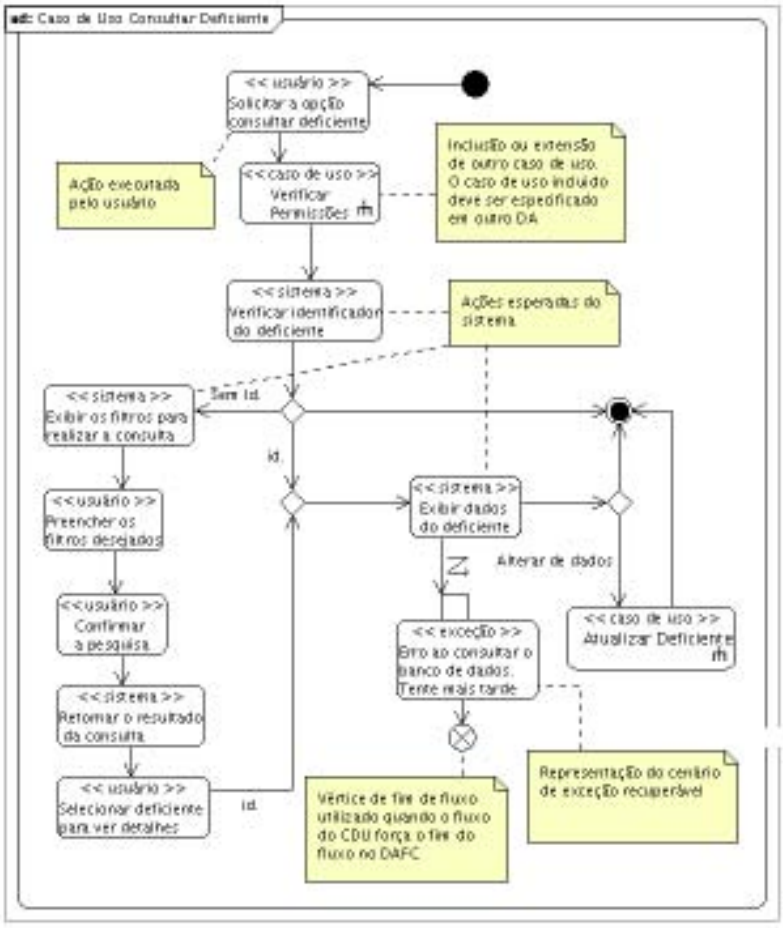

Figura 3.5: DACI do CDU *Consultar Deficiente*.

No segundo nível de representação da modelagem proposta, os fluxos internos de cada CDU da especificação do SUT devem ser representados através de um DA. Esta representação é feita por um modelo chamado de Diagrama de Atividades com Cenários Internos do Caso de Uso (DACI). O DACI representa os fluxos de controle entre os passos para a realização de cada CDU. Nele são representados explicitamente os desvios, decisões e ciclos, geralmente implícitos no caso de uso.

Os DACIs apóiam a especificação e a seleção de testes funcionais do SUT. O uso deles foi baseado na abordagem proposta por Hartmann *et al* [HVFR04], que utiliza os DAs para representar as interações dos atores do CDU no SUT. Além das interações dos atores no SUT, o DACI proposto nesta dissertação descreve também as ações esperadas do SUT, chamadas aqui de *ações do sistema*, e descreve mais precisamente os fluxos de

exceção dos casos de uso.

Um exemplo de DACI, mostrado na Figura 3.5, representa o CDU *Consultar Deficiente* (descrito na Tabela 3.1). As ações (ou atividades) presentes na figura correspondem aos passos do CDU como, por exemplo, a ação *Solicitar a opção consultar deficiente*, que corresponde ao passo **P1** do CDU. Além disso, as ações possuem estereótipos que definem a responsabilidade pela execução da ação. Os estereótipos definidos foram: <<*usuário*>>, <<*sistema*>>, <<*exceção*>> e <<*caso de uso*>>, baseados na proposta de Kösters *et al* [KSW01].

Desta forma, as ações com estereótipo *usuário* devem ser realizadas pelos usuários (ator ou atores) envolvidos; o estereótipo *sistema* indica uma ação do sistema; as ações com estereótipo *exceção* representam os fluxos de exceção do CDU; e o estereótipo *caso de uso* representa os pontos de inclusão e extensão dos CDUs, neste caso, uma chamada para outro CDU.

Representação das Inclusões

A representação de um ponto inclusão no DACI deve conter uma chamada de comportamento para o DACI incluído. Essa representação é similar à representação realizada no DAFC, apresentada na Seção 3.3.1. A Figura 3.5 mostra um exemplo de inclusão onde o CDU *Verificar Permissões* deve ser incluído pelo caso de uso *Consultar Deficiente* (passo **P2** da Tabela 3.1). O DACI do CDU *Verificar Permissões* é detalhado na Figura 3.6.

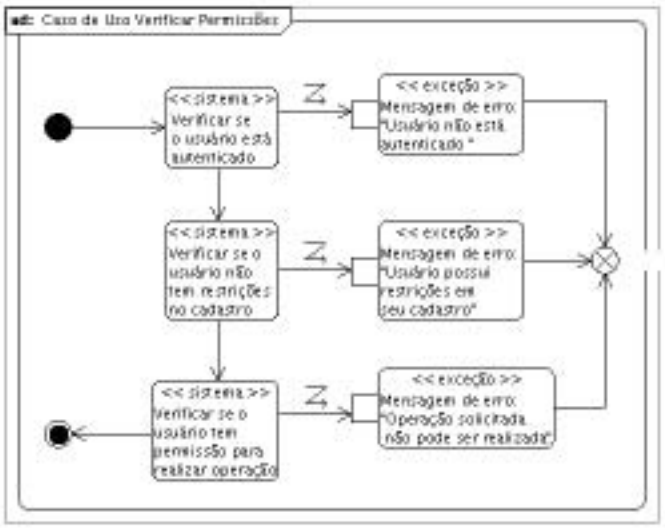

Figura 3.6: DACI do CDU *Verificar Permissões*.

Representação dos Fluxos de Exceção

A representação dos fluxos de exceção de um CDU é feita em conformidade com a classificação proposta por Ferreira [dMF01], diferenciando a representação dos *cenários recuperáveis* e dos *cenários de falha*. Na modelagem proposta, os *cenários recuperáveis* representam as situações em que a exceção é lançada e tratada no próprio caso de uso. Nesta situação, o passo do CDU que lança a exceção deve conter uma aresta de exceção para o tratador da exceção no mesmo CDU, conforme mostra a Figura 3.6. Nessa figura, os três passos do CDU lançam exceções capturadas pelos respectivos tratadores representados na mesma figura.

O *cenário de falha* ocorre quando uma exceção é lançada pelo CDU mas este CDU não possui um tratador específico para esta exceção. Neste caso, a exceção deve ser lançada e a execução deste CDU abordata. A exceção lançada pode ser tratada pelos CDUs que fazem a inclusão ou extensão do CDU que lançou a exceção. Caso a exceção não for tratada por nenhum CDU, ou o CDU que lança exceção não for incluído por nenhum outro CDU, então a execução o SUT deve ser abortada [dMF01]. A representação dessa situação é realizada no DACI através dos vértices de *parâmetros de saída*. Neste caso, o passo do CDU que lança a exceção não tratada (*cenário de falha*) deve conter uma transição para um vértice do tipo *parâmetro de saída* que representa a exceção, como mostra a Figura 3.7.

Nesta figura o CDU *Verificar Permissões* é representado com uma modificação: o passo *"Verificar se o usuário não tem restrições no cadastro"* possui uma aresta para o *parâmetro de saída* que lança a exceção para o CDU que faz a chamada, ao invés de ser conectado a um tratador no mesmo CDU. Assim, para que esta exceção seja tratada no CDU que fez a inclusão do CDU *Verificar Permissões* (neste caso, o CDU *Consultar Deficiente*), deve-se especificar um tratador de exceção no CDU que fez a chamada. Esse tratador é representado da mesma forma que no *cenário recuperável*, mas neste caso o pino de entrada do tratador deve ter o mesmo nome do parâmetro de saída que lança a exceção.

A Figura 3.8 apresenta parte do CDU onde é representado o *cenário recuperável "O usuário possui restrições em seu cadastro"* da exceção lançada pelo CDU *Verificar Permissões* representado na Figura 3.7.

É importante ressaltar que a representação dos fluxos de exceção com estereótipos *<<exceção>>* nos tratadores e com arestas de exceção, que chegam nos tratadores de exceção, é utilizada para diferenciar os fluxos de exceção dos demais cenários de teste na geração dos testes, que será apresentada no Capítulo 4.

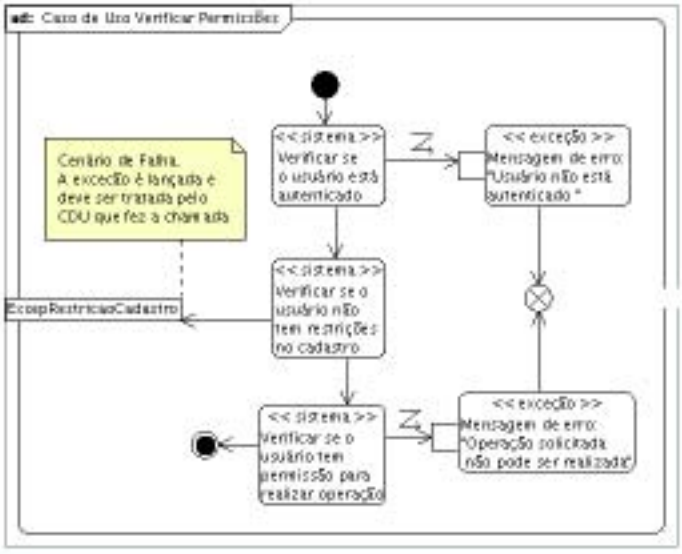

Figura 3.7: DACI do CDU *Verificar Permissões* com *cenário de falha*.

Caminhos no DACI

Um caminho no DACI representa um cenário interno que pode ser executado no CDU. O caminho é definido por uma seqüência de ações ligadas através de transições no DACI. Ele inicia-se em um *vértice inicial* do DACI e termina em um *vértice final* (do mesmo DACI), o que é equivalente a seqüência de passos do CDU. Esta seqüência de passos é chamada de *caminho de execução* no CDU. Além do *vértice final*, também foi utilizado na modelagem do DACI o *vértice de fim de fluxo* da UML 2.0 [Gro04]. Esse vértice representa o fim de um fluxo de execução do CDU e, neste caso, caso o DACI seja incluído por outro DA (DACI ou DAFC) o fluxo de execução no DA que faz a inclusão não deve continuar após o vértice de fim de fluxo. Ou seja, após a execução do CDU, que termina através de um *vértice de fim de fluxo*, o fluxo de execução nos DAs nos níveis superiores também termina.

Para exemplificar, suponha que, no DAFC da Figura 3.4 deseja-se testar o CDU *Consultar Vaga* seguido pelo CDU *Candidatar a Vaga*. O CDU *Candidatar a Vaga* só pode ser executado após o usuário realizar uma consulta a vaga (CDU *Consultar Vaga*). Entretanto, a consulta pode não ser efetuada se alguma situação inesperada acontecer durante a execução do CDU *Consultar Vaga*. Um exemplo na Figua 3.5 do CDU *Consultar Deficiente* é o fluxo de exceção *"Erro ao consultar o banco de dados"*. Após a execução deste fluxo de exceção o CDU termina sem realizar a consulta, sendo necessário um *vértice de fim de fluxo*, não possibilitando a execução do CDU *Candidatar a Vaga* após este fluxo de exceção.

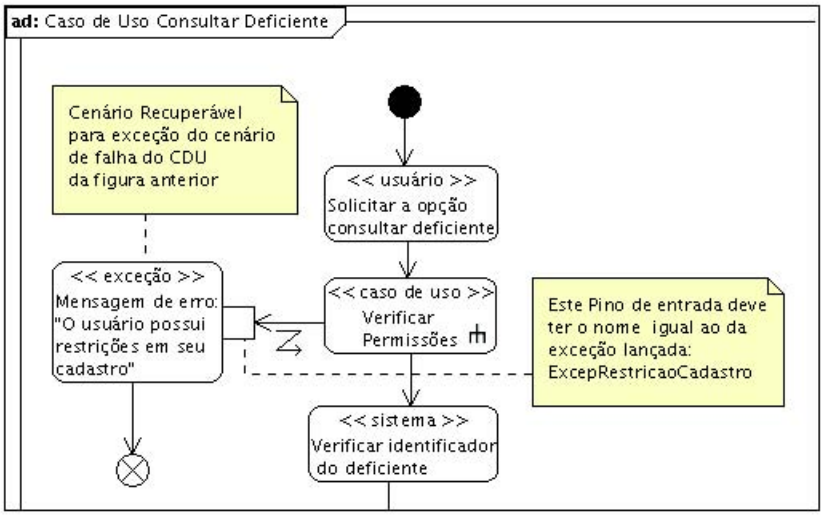

Figura 3.8: *Cenário recuperável* para o *cenário de falha* inserido no DACI *Verificar Permissões*.

Para validar a proposta apresentada neste capítulo, Patuci [Pat07] criou tanto os DACIs quanto o DAFC para o sistema de cadastro de deficientes em seu trabalho de conclusão de curso. Para cada CDU especificado foi criado um DACI, utilizando a ferramenta *Poseidon for UML* [AG06] para modelagem UML. A escolha da ferramenta se justifica pelo fato dela manter os modelos UML em conformidade com os padrões de representação e persistência de modelos proposto pela *Object Management Group* (OMG), através do *XML Metadata Interchange* (XMI) [Gro03c].

O formato XMI foi proposto pela OMG e facilita a troca de informações entre ferramentas de modelagem UML (a partir de 1.x até 2.0) [Gro03b, Gro04]. O uso do formato XMI (formato aberto) possibilitou a automação de algumas fases de especificação e projeto de teste, mais precisamente, a geração automática dos cenários de teste do SUT utilizando o método para geração automática de cenários, que será apresentado no Capítulo 4.

3.4 Resumo

Neste capítulo foi apresentado o modelo de especificação de casos de uso adotado no decorrer deste trabalho e foi detalhado um sistema utilizado como exemplo para especificação e modelagem dos testes. Além disso, neste capítulo foi apresentado também como os casos de uso podem ser detalhados de forma mais precisa utilizando o Diagrama de Atividades da UML [Gro04]. A representação de casos de uso utilizando DAs já vem sendo utilizada por outros autores [HVFR04, BLL04a, VLH+06], entretanto este trabalho agregou alguns

aspectos importantes na representação dos DAs, sendo a principal contribuição referente a modelagem utilizando DAs apresentadas neste trabalho foi a representação de fluxos de exceção, diferenciando os fluxos de falha e fluxos recuperáveis.

Capítulo 4

Um Método para Geração Automática de Cenários de Teste

Este capítulo apresenta um método para a geração automática de cenários de teste funcionais para o SUT a partir da seleção de caminhos nos modelos apresentados no capítulo anterior. Um cenário de teste descreve a seqüência de passos necessária para executar um caso de teste funcional, sendo que geralmente são baseados na seqüência de passos do caso de uso e também são conhecidos como procedimentos de teste [dPPF01].

O método para a geração de cenários de teste é dividido em cinco etapas principais: (i) criação dos modelos de teste, (ii) criação do grafo de fluxo de controle, (iii) seleção dos caminhos de teste, (iv) criação dos cenários de teste, e (v) criação dos casos de teste.

A Figura 4.1 apresenta o diagrama de blocos do método de testes. As etapas (i) e (v) são realizadas manualmente. A etapa (i) foi apresentada no Capítulo 3. As etapas (ii), (iii) e (iv) são representadas por blocos cinza na Figura 4.1 e são realizadas de forma automática, utilizando as definições e algoritmos que serão apresentados neste capítulo. A etapa (v) não faz parte do escopo deste trabalho e é apresentada brevemente neste capítulo.

As seções descritas neste capítulo detalham as etapas (ii), (iii) e (iv) do método para a geração dos cenários de teste. A Seção 4.1 descreve o CFG que interliga os modelos da especificação (os DACIs e o DAFC). Esse CFG foi baseado no CFG (interligado) proposto por Harrold *et al* [HRS98], que foi apresentado no Capítulo 2. A Seção 4.2 apresenta a etapa (iii) do método proposto, onde são selecionados caminhos da especificação. Esta etapa pode ser realizada tão logo os modelos tenham sido criados, antecipando atividades de preparação de testes funcionais. A Seção 4.3 apresenta os cenários de teste gerados a partir dos caminhos selecionados na etapa anterior. Por último, a Seção 4.4 apresenta um estudo comparativo entre a estratégia proposta neste capítulo e outras estratégias para seleção de cenários de teste.

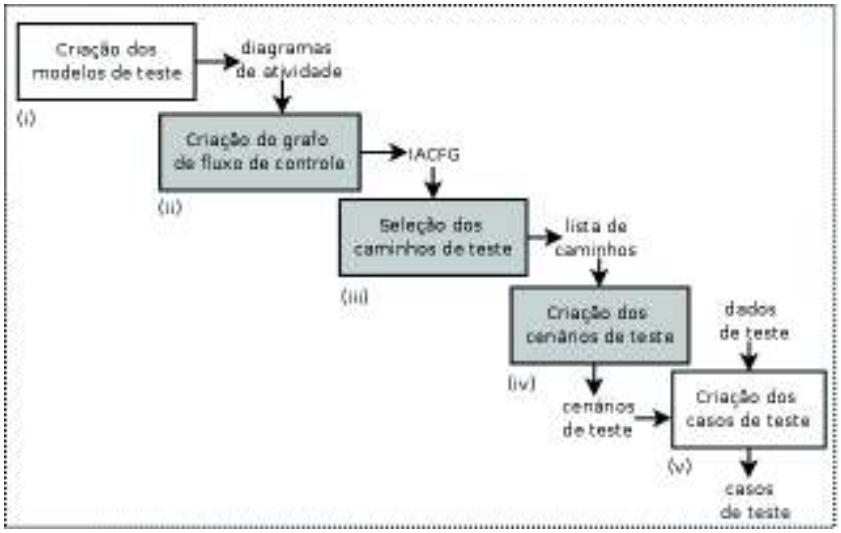

Figura 4.1: Representação do método de testes proposto.

4.1 Grafos de Fluxo de Controle Interligados

Os CFGs possibilitam a aplicação de técnicas de testes estruturais na realização dos testes funcionais, tais como as técnicas apresentadas no Capítulo 2. Além disso, eles possibilitam que a cobertura da especificação seja realizada através da seleção de caminhos de teste.

Neste trabalho, os CFGs são criados a partir dos DACIs e DAFC, que são modelos criados durante a especificação dos testes. Para cada DA da especificação (DACIs e DAFC) é criado um CFG. Posteriormente os vários CFGs são interligados criando um único CFG, que será chamado aqui de Grafo de Fluxo de Controle Inter-Atividade (em inglês, *Inter-activity Control Flow Graph* – IACFG). A criação deste grafo de fluxo de controle baseia-se na abordagem proposta por Harrold *et al* [HRS98].

Harrold *et al* criaram um CFG que mantém todos os procedimentos do programa, chamado de grafo de fluxo de controle interprocedural (ICFG). Cada CFG representa um procedimentos do programa e o ICFG interliga todos os CFGs através das chamadas entre os procedimentos. Harrold utilizou esses modelos na análise de dependência entre instruções do programa. O ICFG foi apresentado de forma mais detalhada no Capítulo 2.

De acordo com a semântica para representação de *CallBehaviorAction* da UML 2.0 [Gro04], a representação mais indicada para transformação do Diagrama de Atividades em CFG seria a representação do Grafo de Fluxo de Controle Aninhado (IIFG), apresentado no Capítulo 2. Entretanto, o ICFG foi adotado neste trabalho devido a restrições relacionadas ao tamanho do Grafo de Fluxo de Controle gerado, conforme foi apresentado no Capítulo 2.

O CFG Comportamental é semelhante ao CFG estrutural definido no Capítulo 2, com

uma característica adicional que é um vértice de saída que representa o vértice de fim de fluxo do DA. Desta forma, o CFG é representado pela tupla (V, A, e, s_n, s_h) onde V, A e e são definidos como o conjunto de vértices, o conjunto de arestas e o vértice de entrada do CFG, semelhantemente a definição apresentada no Capítulo 2. Os vértices s_n e s_h representam o vértice de saída normal e vértice de saída de fim de fluxo, respectivamente.

A Tabela 4.1 apresenta um resumo de como é feito o mapeamento entre os elementos dos DAs e os elementos do CFG. As ações (atividades) e vértices de controle são mapeados diretamente para vértices (V) do CFG e as transições representam arestas (A) do CFG. Além disso, o CFG também armazena os atributos relevantes de cada elemento. Nos vértices do CFG são mantidos o estereótipo e, caso o vértice representar uma ação de *chamada de comportamento*, o identificador do CFG chamado. Já as arestas do CFG possuem um atributo indicando se ela é ou não uma aresta de exceção e os pinos de entrada e saída, quando eles existirem. Além disso, foi necessária a utilização de um vértice interno de exceção para a representação dos parâmetros de saída dos *cenários de falha*.

Tabela 4.1: Correspondência de representação entre os elementos do DA no CFG.

Tipo	**Diagrama de Atividades**	**CFG**
Vértice	Vértice Inicial	Vértice de Entrada
	Vértice Final	Vértice de Saída Normal
	Vértice de Fluxo Final	Vértice de Saída de Fim de Fluxo
	Vértice de Ação	Vértice Interno
	Decisão / Junção	Vértice Interno
	Parâmetro de Saída	Vértice Interno (de exceção)
Aresta	Transição de controle	Aresta direcionada
	Transição de exceção	Aresta direcionada (de exceção)

A Figura 4.2 apresenta quatro CFGs criados a partir dos modelos (DACIs e o DAFC) do SUT. Estes CFGs serão utilizados no decorrer desta seção para exemplificar a criação do IACFG. Na figura, os vértices que possuem ação de *chamada de comportamento* para outros CFGs foram destacados com a cor cinza e as arestas de exceção foram representadas com um símbolo semelhante a um raio, seguindo a notação da UML 2.0 [Gro04]. Para simplificar, não serão apresentados os demais CFGs da especificação do SUT.

A criação do IACFG é realizada a partir dos CFGs Comportamentais em dois passos: (i) criação dos vértices de chamada e de retorno e (ii) criação das arestas para interligar os CFGs. Abaixo, essas atividades são descritas em maiores detalhes.

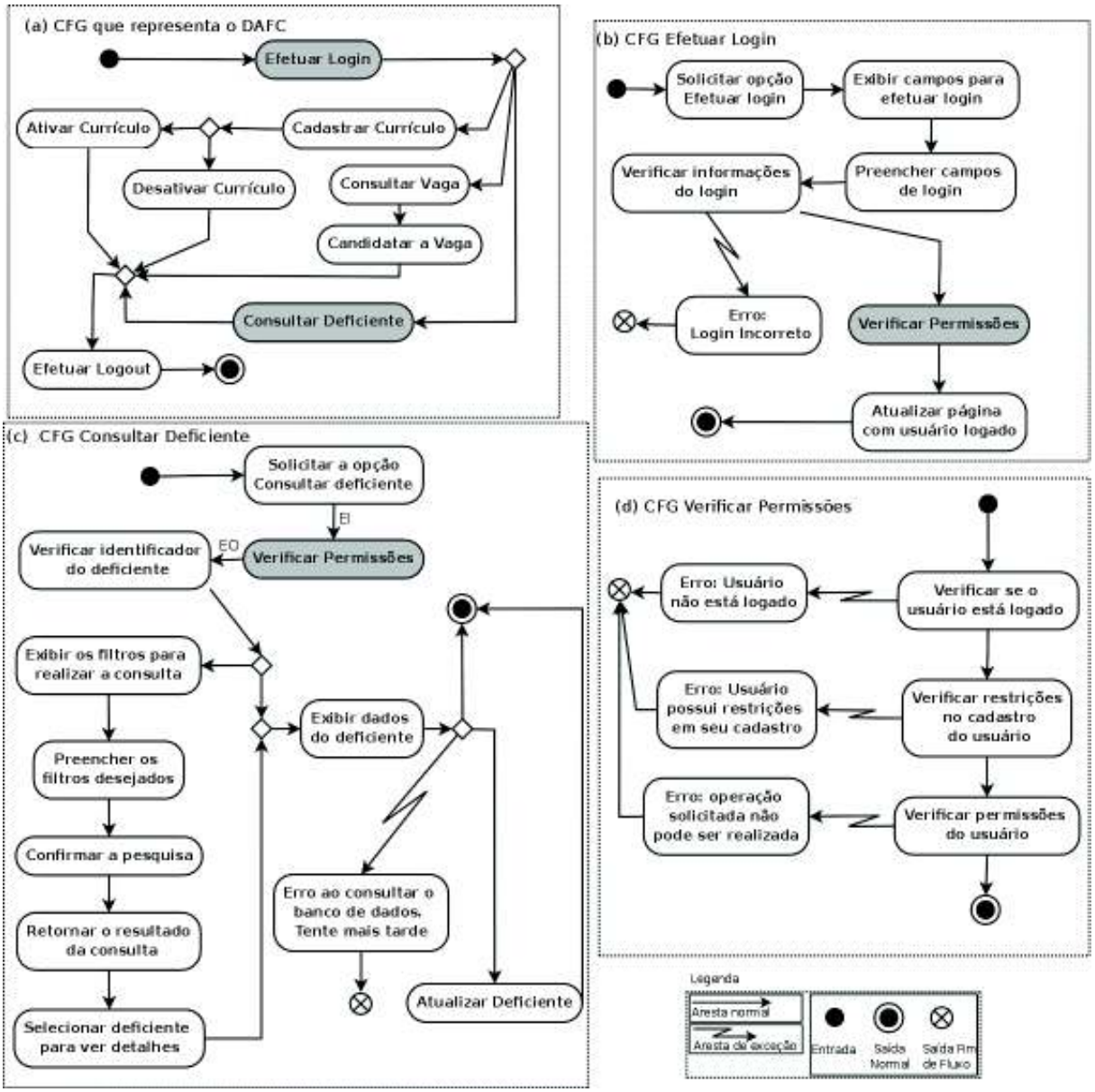

Figura 4.2: CFGs que representam o DAFC (a) e três DACIs do SUT (b), (c) e (d).

4.1.1 Criação dos Vértices de Chamada e de Retorno

Para a criação do IACFG, a primeira transformação a ser realizada nos CFGs mostrados na Figura 4.2 é a criação dos vértices de chamada e dos vértices de retorno. Os vértices de chamada foram criados de forma semelhante à apresentada por Harrold *et al* [HRS98]. Além disso, na definição do IACFG foram identificados três tipos de vértice de retorno: Retorno Normal, Retorno Fim de Fluxo e Retorno de Falha. Os vértices de Retorno Normal e Fim de Fluxo são equivalentes aos vértices de retorno apresentados por Harrold *et al*. Já o vértice de Retorno de Falha é uma representação adicional incorporada nessa dissertação, com o objetivo de representar os *cenários de falha*.

Chamada e Retorno Normal

Os vértices de chamada são criados a partir das atividades dos DAs com ação de *chamada de comportamento*. Estas, além de serem representadas pelo símbolo ⫛, possuem também o estereótipo <<*caso de uso*>>. Desta forma, os vértices que representam essas atividades devem ser substituídos pelos vértices de chamada e retorno no IACFG. O vértice de chamada é utilizado para ligar o CFG que representa o DACI (ou DAFC) ao CFG que é chamado. Assim, para cada vértice de chamada no CFG deve ser criado também um vértice de Retorno Normal, que representa a situação em que o DACI chamado retorna através de um vértice final (vértice de saída normal do CFG).

Para a criação dos vértices de chamada e de retorno normal a partir de um vértice v do CFG, que possui um conjunto de arestas de entrada e_i e de saída e_o, o vértice v deve ser substituído por dois outros: o Vértice de Chamada (v_c) e um Vértice de Retorno Normal (v_{rn}). O vértice v_c mantém o conjunto de arestas e_i do vértice v original e o vértice v_{rn} mantém o conjunto de arestas e_o de v.

A Figura 4.3 apresenta um exemplo com dois CFGs, *Consultar Deficiente* e *Verificar Permissões*, os vértices de chamada e vértices de retorno estão representados com a cor cinza e foram criados para ligar o CFG *Consultar Deficiente* ao CFG *Verificar Permissões*.

Na Figura 4.2 o vértice *Verificar Permissões* do CFG *Consultar Deficiente* possui somente a aresta *EI* em seu conjunto de arestas de entrada e *EO* em seu conjunto de arestas de saída, que foram mantidas respectivamente nos vértices de chamada e retorno, conforme mostra a Figura 4.3.

Retorno Fim de Fluxo

O Retorno Fim de Fluxo é utilizado no CFG quando o DACI chamado termina a execução através de um vértice de fluxo final (vértice de saída fim de fluxo do CFG). Para isso, deve ser criado um vértice de retorno de fim de fluxo no CFG que representa o DAFC para cada vértice de fim de fluxo dos CFGs. Por exemplo, a Figura 4.4 mostra um vértice

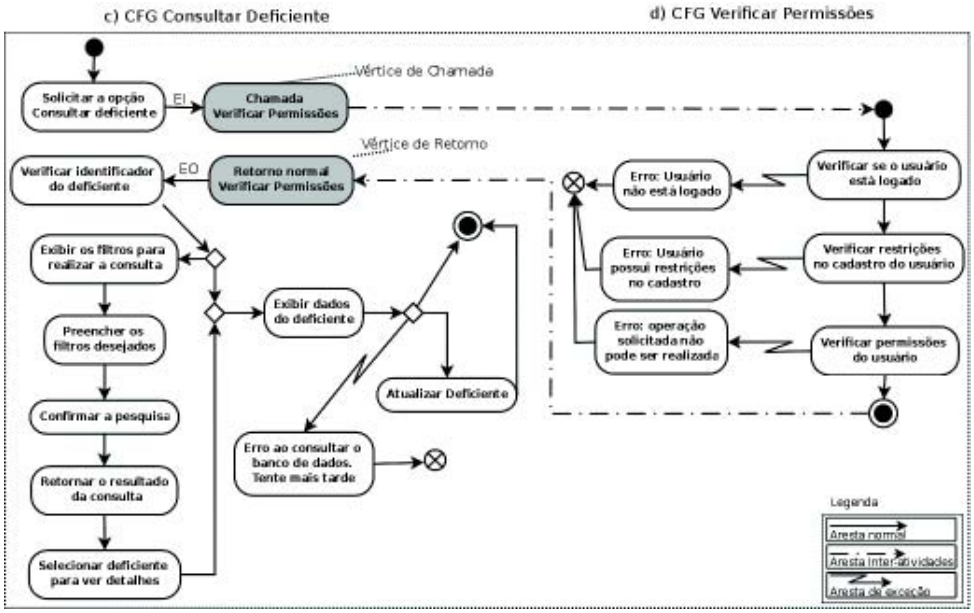

Figura 4.3: CFG *Consultar Deficiente* interligado com o CFG *Verificar Permissões*.

de fim de fluxo criado para os CFGs (a) e (c) da Figura 4.2. Neste caso, o CFG (c) possui o vértice de saída fim de fluxo **SH** que é ligado ao vértice *"Ret. fim de fluxo Consultar Deficiente"* do CFG (a).

Retorno de Falha

O Retorno de Falha é utilizado na situação em que um DACI contém *cenários de falha*, o que implica em abortar a execução do CDU. Assim, uma exceção deve ser lançada através de um parâmetro de saída, que é representado no IACFG pelo vértice de exceção. Nessa situação, o vértice de exceção deve ser ligado ao vértice de retorno de falha correspondente no CFG que faz a chamada. Para isso, deve existir no DACI que realiza a chamada um *cenário (de exceção) recuperável* para tratar a exceção do *cenário de falha* do DACI que foi chamado.

Um exemplo de vértice de retorno de falha no IACFG é apresentado na Figura 4.5. Nesta figura são representados os CFGs (c) e (d) da Figura 4.2, mas com duas modificações. A primeira modificação pode ser visualizada no CFG (d), onde o vértice interno *"Verificar restrições no cadastro do usuário"* possui uma aresta para o parâmetro de saída *"ExcepRestricaoCadastro"* (em destaque). A segunda modificação na Figura 4.5 foi realizada no *cenário recuperável "Erro: O usuário possui restrições em seu cadastro"*, este cenário era representado no CFG (d) da Figura 4.2 e agora está no CFG (c). O *cenário recuperável* do CFG (c) foi criado para tratar a exceção do *cenário de falha* do CFG (d).

Pode-se observar na Figura 4.5 que no CFG (c) foi criado o vértice *"Retorno de falha Verificar Permissões"* para tratar a exceção *"ExcepRestricaoCadastro"* do CFG (d).

Na situação em que a exceção do *cenário de falha* não é tratada em nenhum dos CFGs, deve ser criado um vértice de retorno de falha no CFG correspondente ao DAFC que é ligado diretamente ao vértice final do IACFG, tal como foi representado para o retorno de fim de fluxo.

4.1.2 Criação das Arestas para Interligar os CFGs

A próxima etapa na criação do IACFG é a criação das arestas inter-atividades para interligar os CFGs. As arestas inter-atividades são arestas direcionadas criadas para ligar cada vértice de chamada ao vértice de entrada do CFG que corresponde ao seu comportamento chamado (arestas de chamada), e também para conectar o vértice de saída do CFG chamado aos vértices de retorno correspondentes (arestas de retorno).

As Figuras 4.3, 4.4, 4.5 apresentam as arestas inter-atividades de forma tracejada. Por exemplo, a Figura 4.3 apresenta o vértice *Chamada Verificar Permissões* do CFG (c), ligado através de uma aresta inter-atividade ao vértice inicial do CFG *Verificar Permissões* (d). O mesmo ocorre com o vértice final do CFG (d), que é ligado através de uma aresta inter-atividade ao vértice *Retorno Normal Verificar Permissões* do CFG (c). Desta forma, os CFGs são interligados de maneira similar à proposta por Harrold *et al* [HRS98].

A Figura 4.6 apresenta uma parte do IACFG criado para o SUT. Neste exemplo são representados somente os CFGs da Figura 4.2 para facilitar o entendimento e não comprometer a legibilidade da figura. No IACFG, os vértices de chamada e retorno (normal e de fim de fluxo) são representados em cinza, enquanto que as arestas inter-atividades são representadas de forma tracejada.

4.1.3 Definição do IACFG

O Grafo de Fluxo de Controle Inter-Atividades criado pode ser definido de maneira mais formal através da tupla $IACFG = (G, I, C, R, e_g, s_g)$ onde G é o conjunto de CFGs que representam cada DA pertencente à especificação, e C e R representam os conjuntos de vértices de chamada e vértices de retorno pertencentes ao IACFG, respectivamente. Estes vértices substituem as ações de *chamada de comportamento* do DA (como visto na Figura 4.6). Cada vértice de retorno é associado a um único vértice de chamada, logo cada vértice de chamada forma um par com seu respectivo vértice de retorno normal, de fim de fluxo e de falha.

Os vértices e_g e s_g são, respectivamente, os vértices de entrada e saída (globais) do IACFG, sendo que o vértice de entrada global possui uma aresta que o liga ao vértice de entrada do DAFC e o vértice de saída (s_n) do DAFC possui uma aresta que o liga

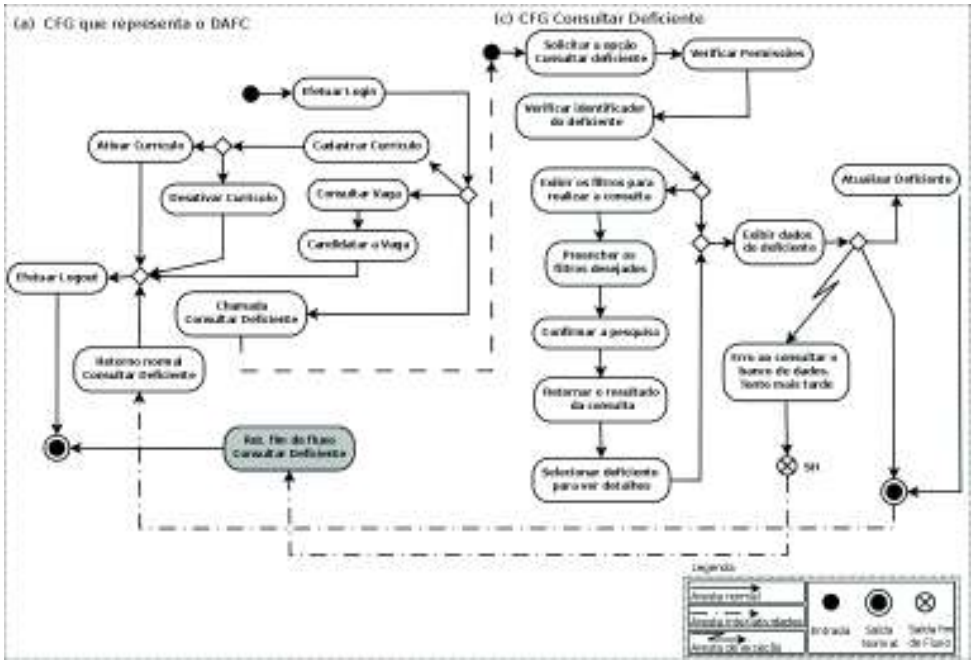

Figura 4.4: CFG do DAFC com Vértice de *Retorno de Fim de Fluxo*.

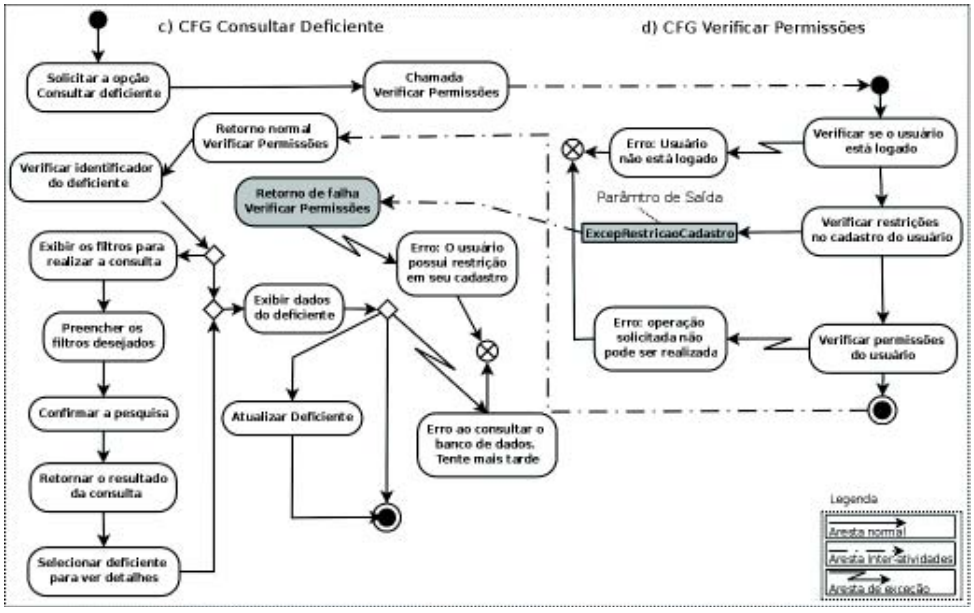

Figura 4.5: CFG *Consultar Deficiente* com *retorno de falha* do CFG *Verificar Permissões*.

ao vértice s_g do IACFG. I representa o conjunto de arestas inter-atividades do IACFG, que são arestas direcionadas criadas para ligar cada vértice de chamada de C ao vértice de entrada (e) do CFG correspondente à chamada de comportamento. As arestas de I também ligam os vértices de saída (s_n, s_h) e os vértices de exceção do CFG, correspondente à *chamada de comportamento*, aos vértices de retorno de R. A Figura 4.6 mostrou um exemplo de IACFG, entretanto para não comprometer a legibilidade, os vértices e_g e s_g não foram representados.

É importante ressaltar que o IACFG mantém apenas uma cópia do CFG para cada DA da especificação. Desta forma, o tamanho do IACFG é igual ao tamanho da união de todos os DAs da especificação, exceto pelos vértices de chamada, os vértices de retorno e as arestas inter-atividades criados no IACFG.

Nessa dissertação não foi utilizada a abordagem que propõe a criação de uma réplica para cada ponto onde o CFG é chamado. Por exemplo, na Figura 4.6 para criar o IACFG seriam necessárias duas réplicas do CFG (d), uma delas para representar a chamada de (d) realizada pelo CFG (b) e a outra para representar a chamada realizada pelo CFG (c). Entretanto, para evitar as limitações apresentadas no Capítulo 2, esta abordagem não foi adotada. Por exemplo, uma limitação seria em situações que ocorrem chamadas recursivas de um ou mais DAs. Neste caso, a representação usando réplicas tornaria o IACFG infinito e, que mesmo se fosse limitada o número de representações recursivas, essa abordagem tende a manter muitas informações redundantes.

4.2 Seleção de Caminhos de Teste no IACFG

A seleção de caminhos de testes é a terceira etapa no método testes proposto. Cada caminho selecionado nesta etapa representa um cenário de teste funcional do SUT. Um caminho de teste completo no IACFG é definido como qualquer caminho que liga o vértice de entrada global e_g ao vértice de saída global s_g do IACFG. Para selecionar caminhos de teste completos, devem ser definidos os critérios de teste para a cobertura dos modelos e as estratégias de seleção de caminhos de teste a serem utilizados.

A partir do IACFG deseja-se selecionar o conjunto de caminhos que cobrem a especificação. O IACFG será utilizado como especificação para obtenção dos testes pois possui uma estrutura similar à do CFG apresentado no Capítulo 2, porém ele é criado a partir de modelos da especificação do SUT ao invés de seu código fonte.

Para que o critério de testes seja viável, deve-se selecionar o menor conjunto de caminhos que atende ao critério, pois quanto maior o conjunto mais tempo será necessário para preparar e executar os testes. Desta forma, é necessário adotar uma estratégia para seleção de caminhos de teste, que execute em tempo computacional finito e que minimize a seleção de caminhos não executáveis.

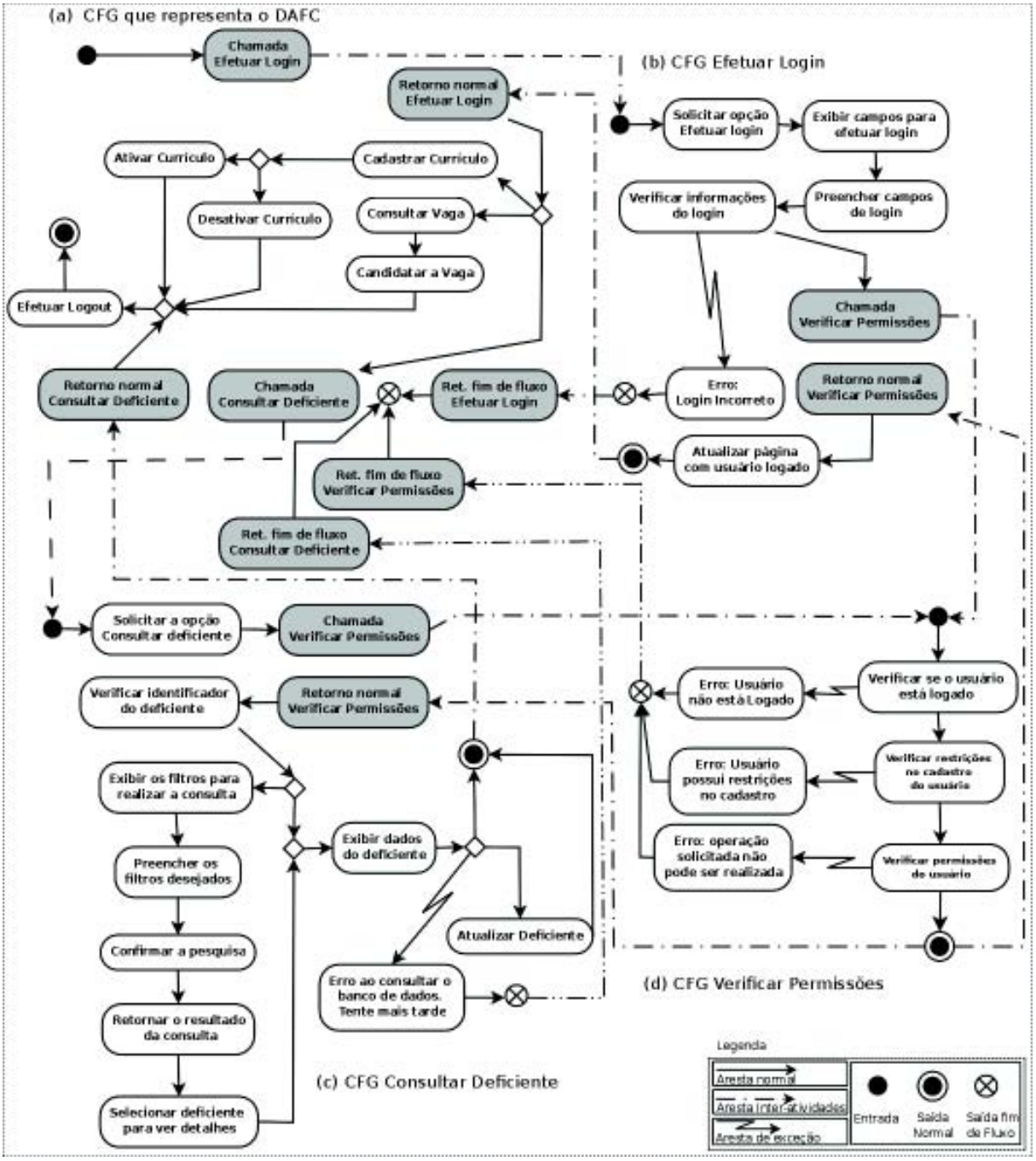

Figura 4.6: Exemplo de IACFG criado para o SUT.

Esta seção apresenta algumas estratégias para a seleção de caminhos de teste e, entre elas, a estratégia adotada para método proposto. Além disso, é apresentado o critério de teste que foi utilizado no decorrer deste trabalho.

4.2.1 Critério de Teste Adotado

Para selecionar os caminhos de teste foi adotada uma abordagem de testes orientados a caminhos [Bei90]. O critério adotado baseia-se nos critérios de cobertura de arestas do CFG. Assim, seu objetivo é realizar a seleção de um conjunto de caminhos de teste que realize a cobertura de todas as arestas do IACFG.

Existem outros critérios de cobertura em CFGs que podem ser utilizados, tais como cobertura de ciclos e caminhos básicos [Zhu95] (Capítulo 2), mas que não foram adotados neste trabalho devido a restrições no tempo para realização deste trabalho. Além disso, existem também os critérios de cobertura de fluxo de dados no CFG como, por exemplo, cobertura de definições e usos [Pre01]. Esses critérios são considerados mais fortes ou mais completos [Zhu96], mas eles não foram aplicados nesta dissertação, pois a modelagem definida para a criação do IACFG não apresenta informações suficientes sobre fluxos de dados nos modelos da especificação do SUT.

Cobertura das Arestas

Os critérios de cobertura de todos os vértices e de todas as arestas são critérios tipicamente utilizados para cobrir o fluxo de controle em CFGs [Zhu95]. A Figura 4.7 apresenta um algoritmo para cobertura de arestas no IACFG. O algoritmo seleciona um conjunto de caminhos (C) em um IACFG (G), definido na Seção 4.1, que cobre um conjunto de arestas não cobertas ($A_{NaoCobertas}$) de G, tal que $a_n(v_o, v_d) \in A_{NaoCobertas}$, onde $v_o, v_d \in V$ são os vértice de origem e destino da aresta, respectivamente. Desta forma, inicialmente nenhuma aresta do CFG é coberta e, enquanto não forem cobertas todas as arestas, o algoritmo seleciona um caminho c_1, que parte de e_g e chega ao vértice de entrada v_o de uma aresta não coberta $a_n(v_o, v_d)$, conforme apresentado no *Passo 3*.

O passo seguinte seleciona um caminho c_2 entre o vértice v_d de a_n e o vértice s_g de G. Em seguida, é feita a união dos caminhos e a aresta não coberta ($c_1 \cup a_n \cup c_2$) que irão compor um caminho de testes completo c_c (*Passo 6*). No *Passo 7*, este caminho completo é adicionado ao conjunto de caminhos C e, em seguida, são removidas da lista de arestas não cobertas ($A_{NaoCobertas}$) todas as arestas cobertas pelo caminho c_c (*Passo 8*). O algoritmo para selecionar caminhos entre dois vértices utilizados nos passos 4 e 5 é detalhado na Seção 4.2.2.

Inicialmente este algoritmo foi utilizado na cobertura de todas as arestas do modelo, conforme foi apresentado por Perez *et al* [PM07]. Posteriormente, o critério de cobertura

```
   CRITERIO-COBERTURA (G,A_NaoCobertas,e_g,s_g) //Cobertura das arestas A do IACFG G
 1 C ← Ø                                  //Atribuir vazio ao conjunto de caminhos
 2 WHILE A_NaoCobertas ≠ Ø DO              //Enq. existir arestas não cobertas
 3   a_n(v_o,v_d) ← GetArestas(A_NaoCobertas)   //Obter uma aresta ainda não coberta
 4   c_1 ← SelecionarCaminho(G,e_g,v_o)      //sel. primeira parte do caminho
 5   c_2 ← SelecionarCaminho(G,v_d,s_g)      //sel. segunda parte do caminho
 6   c_c ← c_1 + a_n + c_2                   //Concatenar os caminhos selecionados
 7   C [C.last+1] ← c_c     //Adicionar c_c à lista de caminhos selecionados
 8   RemoverArestasCobertas(A_NaoCobertas,c_c) //Remover de A_naoCobertas as arestas de c_c
 9 END
10 RETURN C                          //retornar lista de caminhos selecionados
11 END
```

Figura 4.7: Algoritmo de seleção de caminhos para cobertura das arestas no IACFG.

das arestas foi dividido em duas etapas: na primeira são selecionados caminhos para a cobertura de todas as arestas de exceção e na segunda etapa para cobertura de todas as arestas normais. A seleção de caminhos realizando a cobertura das arestas de exceção separadamente é detalhada a seguir.

Arestas Normais e Arestas de Exceção

A separação da cobertura de arestas de exceção e arestas normais foi necessária, uma vez que os caminhos com exceção têm como objetivo verificar o comportamento do sistema nos cenários de exceção que, neste mestrado, foram considerados cenários críticos. Além disso, a cobertura das arestas normais pode ser comprometida quando uma aresta normal é coberta em caminhos que possuem arestas de exceção, mas nunca é coberta por um caminho livre de exceções. Desta forma, a separação em duas etapas garantiu que todas as arestas normais do IACFG foram cobertas pelo menos uma vez por um caminho sem arestas de exceção.

Na cobertura das arestas de exceção, os caminhos com exceção são selecionados utilizando o algoritmo apresentado na Figura 4.7. Para isso, o conjunto $A_{NaoCobertas}$ de entrada do algoritmo é composto pelas arestas de exceção do IACFG (G) e pelas arestas de entrada dos vértices de exceção (vértices que representam os parâmetros de saída). O algoritmo seleciona a primeira parte do caminho c_1 (*Passo 4*), que é um sub-caminho até a exceção e, em seguida o *Passo 5* seleciona c_2, que é um sub-caminho partindo do tratador de exceções até o vértice s_g do IACFG.

As exceções presentes no IACFG representam os cenários de exceção dos casos de uso do SUT, conforme foi apresentado no Capítulo 3. Desta forma, é importante ressaltar que os cenários de exceção não representam necessariamente a implementação de tratadores

de exceção (tais como sentenças, *try catch*), mas representam os *cenários recuperáveis* e os *cenários de falha* especificados nos CDUs.

Na segunda etapa, é feita a seleção dos caminhos para a cobertura de arestas normais utilizando o algoritmo da Figura 4.7, mas considerando como arestas não cobertas ($A_{NaoCobertos}$) somente as arestas normais de G. Além disso, para garantir que nenhuma aresta de exceção seja selecionada, as arestas de exceção do IACFG G são removidas antes da execução desta etapa.

O resultado das duas etapas é um conjunto de caminhos que realizam a cobertura de 100% das arestas normais do IACFG e, também, são selecionados, um conjunto de caminhos que cobrem todos cenários de exceção especificados.

4.2.2 Algoritmos para Selecionar Caminhos

Os IACFGs são grafos direcionados com chamadas e retornos que ligam vários CFGs. Para obter os caminhos de teste nos IACFGs que atendam a um critério de cobertura, deve-se adotar uma estratégia de seleção de caminhos em grafos. Desta forma, os principais requisitos para a escolha da estratégia são: viabilidade computacional (tempo para execução), potencial para selecionar caminhos executáveis e potencial em selecionar um conjunto reduzido de caminhos.

A viabilidade computacional está diretamente relacionada à complexidade do algoritmo, pois um algoritmo com complexidade menor possibilita a sua utilização em IACFGs maiores, aumentando a escalabilidade da abordagem. A seleção de caminhos executáveis é outro fator decisivo, pois a existência de um número muito grande de caminhos não executáveis compromete a cobertura da especificação. Além disso, a quantidade de caminhos deve ter uma tendência de crescimento proporcional ao tamanho do IACFG. Um número muito grande de caminhos pode tornar inviável a preparação e execução de testes para cobrir os caminhos selecionados.

Dados estes aspectos, vários algoritmos para realizar percursos em grafos podem ser adotadas para a seleção de caminhos no IACFG. Cormen *et al* [CLRS01] apresenta de forma detalhada grande parte dessas estratégias para realizar percursos em grafos, sendo que algumas delas são apresentadas logo abaixo:

- **Todas as Combinações de Caminhos.** Uma abordagem para selecionar caminhos é a de seleção de todas as combinações de caminhos entre dois vértices. Entretanto, apesar dessa estratégia ser de fácil implementação, ela pode gerar um número exponencial de caminhos ou até nunca terminar a execução devido à presença de ciclos no IACFG, o que pode levar à existência de infinitos caminhos. Isso torna inviável a utilização desse algoritmo para selecionar caminhos em IACFGs complexos.

- **Caminho Hamiltoniano.** O caminho hamiltoniano é um caminho que percorre todos os vértices do grafo sem repetir arestas. Entretanto, nem sempre é possível selecionar caminhos hamiltonianos em grafos, para isso o grafo deve ser hamiltoniano. A verificação dessa restrição é um problema de decisão de difícil solução, por se tratar de um problema *NP-Completo* [CLRS01]. Além disso, um caminho que percorre todos os vértices do IACFG tem grandes chances de ser um caminho não executável.

- **Caminho Euleriano.** Um caminho euleriano é um caminho entre dois vértices v_o e v_d que percorre todas as arestas do grafo, passando uma única vez em cada aresta. Para que seja possível realizar este percurso, todos os vértices do grafo devem conter um número par de arestas. Se o grafo for direcionado, o grau de entrada e grau de saída de cada vértice devem ser iguais, exceto pelo grau de inicial (v_o) e final (v_d) do caminho. O grau é equivalente à quantidade de arestas de entrada e de saída. Esta estratégia também foi descartada, pois, além de selecionar um caminho muito longo, não é possível garantir a restrição no grau de entrada e saída dos vértices no IACFG.

- **Busca em Profundidade.** O algoritmo de busca em profundidade (em inglês, *Depth First Search* - DFS) é muito conhecido para realizar percursos em grafos. Ele explora o grafo a partir de um vértice raiz (v_r) com objetivo de visitar os vértices do grafo até encontrar o vértice buscado. Quando não é mais possível avançar a busca, o algoritmo retrocede para continuar a busca nos níveis anteriores (também conhecido como a operação de *backtracking*). Assim, o mesmo algoritmo é aplicado para os vértices ainda não visitados nos níveis anteriores. O algoritmo DFS é executado enquanto existirem vértices alcançáveis e não cobertos, e ele possui complexidade linear em relação ao tamanho do grafo de entrada.

- **Busca em Largura.** O algoritmo de busca em largura (em inglês, *Breadth First Search* – BFS) funciona de forma semelhante à busca em profundidade, mas visando cobrir primeiramente os vértices mais próximos ao invés de atingir a maior distância possível (tal como no algoritmo DFS) . Esse algoritmo também possui complexidade linear em relação tamanho do grafo de entrada.

- **Caminhos Mínimos.** A seleção de caminhos mínimos pode ser realizada utilizando o algoritmo BFS quando o grafo utilizado não possui pesos atribuídos às suas arestas (ou todos os pesos são idênticos). Para selecionar os caminhos mínimos nos casos em que são atribuídos pesos às arestas, pode-se utilizar o algoritmo de Dijkstra [CLRS01], um algoritmo muito difundido e utilizado para selecionar o menor

caminho entre dois vértices. A seleção de caminhos mínimos pode ser executada em tempo linear em relação ao tamanho do grafo (grafo simples) de entrada.

As abordagens apresentadas acima foram experimentadas e testadas para a seleção de caminhos, exceto pelas estratégias de seleção de caminhos hamiltonianos e caminhos eulerianos. A única estratégia testada que mostrou-se inviável foi a de seleção de todos os caminhos. Além disso, não é possível garantir que essas estratégias testadas possuem potencial em selecionar somente caminhos executáveis.

4.2.3 Caminhos Não-Executáveis

Yeates e Malevris [YM89] observaram que a existência de predicados conflitantes no caminho é uma das principais causas que tornam o caminho não executável. Um caminho não executável é um caminho em que não é possível encontrar valores para as variáveis de entrada que possibilitem a sua execução [HH85]. Desta forma, quanto maior o número de predicados em um caminho, maiores são as chances de existirem predicados conflitantes e, conseqüentemente, maiores são as chances de um caminho ser não executável. Os predicados considerados por Yeates e Malevris equivalem aos vértices de decisão do CFG.

Entretanto, na seleção de caminhos do IACFG é necessário garantir também as restrições de contexto das ações de *chamada de comportamento* do DA original quando uma aresta inter-atividade é selecionada, ou seja, se um caminho percorre uma aresta de chamada para um DA, então deve-se selecionar uma aresta de retorno correspondente à mesma *chamada de comportamento*. A Figura 4.8 apresenta um exemplo de caminho no IACFG que não mantém essa restrição de entrada e saída. A aresta **AC** da figura é a aresta que foi selecionada para ser coberta na interação do algoritmo. Neste caminho, foi selecionada a aresta **AIR 2** que seria correspondente à chamada representada pela aresta **AIE 2**, entre os CFGs (c) e (d). Para que o contexto fosse mantido, a aresta de retorno selecionada deveria ser **AIR 1**, pois esta corresponde à aresta de entrada utilizada (**AIE 1**). Desta forma, esse caminho é um caminho "não executável", pois ele não mantém as restrições de contexto no IACFG.

Uma abordagem para evitar situações onde há violação de contexto seria a replicação do CFG para cada chamada realizada. Desta forma, na Figura 4.8 o CFG (d) seria representado uma vez para cada vértice de chamada realizada. O IACFG seria representado de forma similar ao IIFG apresentado no Capítulo 2, mas com uma limitação em relação ao tamanho do IACFG gerado, pois essa abordagem proporcionaria um aumento na quantidade de caminhos para realização da sua cobertura, como foi observado por Sinha e Harrold [SHR01].

O algoritmo utilizado para selecionar o caminho apresentado em destaque na Figura 4.8 foi o de seleção de caminhos mínimos. Esse caminho foi obtido na seleção de caminhos

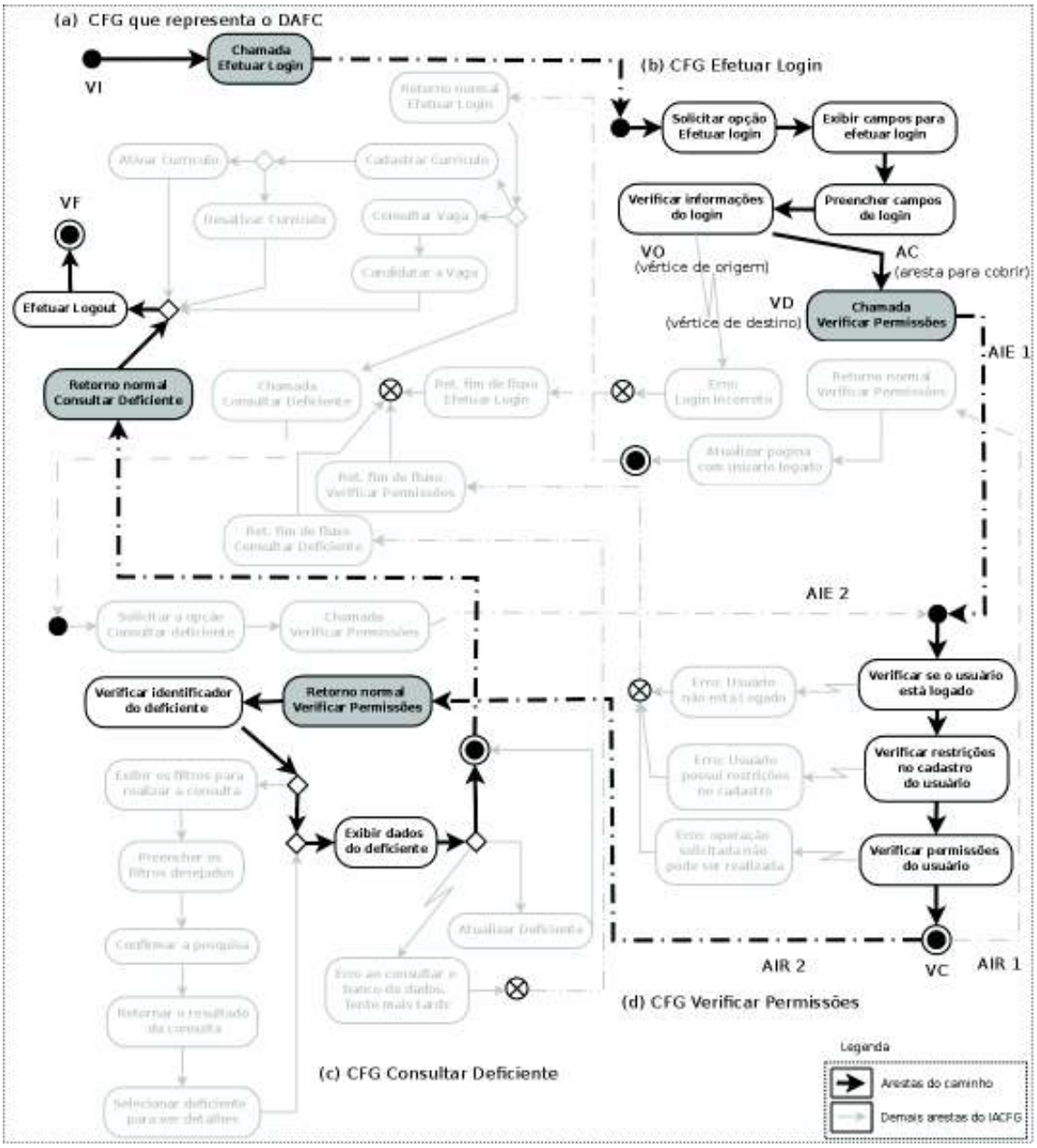

Figura 4.8: Caminho mínimo selecionado não executável.

normais do IACFG utilizando o critério de cobertura apresentado na Seção 4.2.1. Como foi mencionado acima, o caminho selecionado é não executável, pois ele não mantém as restrições de contexto de chamada no IACFG.

4.2.4 Algoritmo para Selecionar Caminhos com Contexto

Para solucionar o problema apresentado na Seção 4.2.3, este trabalho propõe um algoritmo para seleção de caminhos que mantém o contexto de chamada no IACFG, ou seja, cada aresta de chamada presente no caminho possui uma aresta de retorno associada para o mesmo contexto que fez a chamada.

O algoritmo para seleção de caminhos com contexto é uma variação do algoritmo DFS, apresentado detalhes por Cormen *et al* [CLRS01, Capítulo 22]. Para que o algoritmo DFS realize uma busca com contexto, deve-se selecionar a aresta de inter-atividade correta sempre que for encontrado um vértice de saída no CFG pecorrido. Como exemplo, na parte (d) da Figura 4.8, ao encontrar o vértice **VF** utilizando o DFS, a aresta **AIR 1** deve ser selecionada, dado que esta retorna para o contexto do CFG (b) que é o contexto de retorno correto.

A Figura 4.9 apresenta o algoritmo DFS modificado (DFS-CONTEXTO) para que seja realizada a seleção de caminhos com contexto. Para facilitar a compreensão, as variáveis globais e locais estão representadas em *itálico*, as palavras reservadas em **negrito**, as constantes em letras MAIÚSCULAS, e os procedimentos chamados possuem a primeira letra em Maiúsculo. Além disso, o número das linhas adicionadas ao DFS estão representados em negrito e sublinhado.

As estruturas de dados utilizadas pelo algoritmo DFS-CONTEXTO são:

- *vetor de cores (cor[])*, utilizado para marcar quais vértices já foram percorridos durante a busca em profundidade.
- *pilha de busca (pilha_DFS)*, que mantém o caminho que é percorrido durante a recursão.
- *pilha de contexto (pilha_contexto)*, inserida como estrutura de dados auxiliar para manter o contexto de chamada enquanto o caminho é percorrido pela busca em profundidade.
- *lista (lista_arestas)*, que mantém o conjunto de arestas de saída do vértice da busca (u).

O vetor de cores e as duas pilhas são globais, ou seja, mantêm seu estado durante toda a execução do algoritmo.

No algoritmo DFS-CONTEXTO, as *linhas 5* a *13* foram inseridas para fazer o tratamento de três casos de busca com contexto. Os dois primeiros casos são restrições adicionais para o tratamento de contexto de chamada, e o terceiro caso é a realização da busca em profundidade normalmente. Além disso, para garantir que a pilha de contexto se mantenha consistente no "retorno" da recursão, foi necessária a inserção do código auxiliar das *linhas 21* a *23*.

A primeira restrição ocorre quando o vértice corrente da busca (u) for um vértice de chamada para outro contexto como, por exemplo, os vértices de chamada representados em cinza da Figura 4.6. Neste caso, a execução da busca em profundidade deve ser realizada de forma normal, mas o vértice de chamada deve ser inserido na pilha de contexto, como mostra as *linhas 6* e *7* do algoritmo da Figura 4.9.

A segunda restrição do algoritmo é o tratamento de saída de um contexto, que ocorre quando o vértice corrente no caminho (u) é um vértice final de um CFG. Neste caso, os vértices de chamada que foram inseridos na pilha de contexto devem ser desempilhados (*linha 9*) para garantir continuidade do caminho mantendo o contexto. Em seguida, na *linha 10* são selecionadas as arestas de retorno para o contexto correto do vértice de chamada. O algoritmo para selecionar as arestas para o contexto correto é executado pelo procedimento *GetArestasParaContexto*, que obtém dentre todas as arestas de saída do vértice final a aresta para o contexto anterior. Este algoritmo é linear em relação à quantidade de arestas inter-atividades do vértice final (u) e retorna somente as arestas de saída de u que são relacionadas ao vértice de chamada desempilhado (c). No caminho não executável apresentado na Figura 4.8, quando o vértice final **VC** é encontrado no caminho exemplificado, a única aresta de retorno selecionada para continuar a recursão do algoritmo seria a **AIR 1**.

O terceiro e último caso de busca, mostrado na *linha 12* do algoritmo, equivale ao caso normal de busca em profundidade, que deve ser executado para todos os demais vértices do IACFG.

Os três casos de busca são realizados na descida da busca em profundidade. Além deles, é importante ressaltar que quando o algoritmo está na volta, ou seja, não é possível mais avançar na busca, então a busca deve retroceder (*backtracking*). Durante o retrocesso deve-se garantir também a consistência na pilha auxiliar de contexto. Para isso, é necessária a realização de ajustes na pilha quando o vértice u for um vértice de chamada, ou final, do CFG chamado, como mostra as *linhas 21 a 23* do algoritmo. Assim como na descida do algoritmo, o ajuste deve ser realizado quando o vértice u é um vértice de chamada ou um vértice final, tais como os dois casos especiais tratados nas *linhas 5 a 10*. Este ajuste é linear em relação ao tamanho da pilha de busca em profundidade, pois deve-se varrer esta pilha para mante-la consistente em relação ao caminho percorrido.

Além dos vértices de chamada, a pilha auxiliar de contexto mantém também uma

```
   DFS-CONTEXTO (u,t)     //DFS com contexto u(vértice corrente da busca)
                           t(vértice que está sendo buscado)
 1 cor[u] ← CINZA
 2 IF u = t THEN   //vértice corrente (u) da busca é igual ao vértice que
                     está sendo buscado (t)
 3   RETURN pilha_DFS
 4 END
 5 IF Tipo(u) = VERTICE_CHAMADA THEN
 6   Empilha(pilha_contexto,<u,cor>)
 7   lista_arestas ← GetArestasSaida(u)                //arestas (u, v)
 8 ELSE IF Tipo(u) = FINAL THEN
 9   <c,cor> ← Desempilha(pilha_contexto)
10   lista_arestas ← GetArestasParaContexto(u,c) //arestas ret. contexto
11 ELSE
12   lista_arestas ← GetArestasSaida(u)                //arestas (u, v)
13 END
14 FOREACH a(u,v) OF lista_arestas DO
15   IF cor[v] == BRANCO THEN
16     Empilhar(pilha_DFS,v)
17     DFS-CONTEXTO(v,t)                              //chamada recursiva
18     Desempilhar(pilha_DFS)
19   END
20 END
21 IF Tipo(u) = VERTICE_CHAMADA OR Tipo(u) = FINAL THEN
22   AjutarPilhaContexto(pilha_contexto,cor)
23 END
24 u ← PRETO
25 RETURN NULL
26 END
```

Figura 4.9: Busca em profundidade para selecionar caminhos com contexto.

cópia da lista de cores como mostrado na Figura 4.9. A lista de cores deve ser "limpa" a cada novo contexto, ou seja, todos os vértices do novo contexto devem estar marcados com a cor BRANCO. Este tratamento é necessário para as situações em que um CFG precisa ser percorrido mais de uma vez durante o mesmo caminho.

Um exemplo de caminho com a situação descrita acima é apresentado na Figura 4.10. Neste caminho deseja-se cobrir a aresta *AC*. Neste caso, a parte (d) será percorrida duas vezes através dos vértices de chamada *Chamada Verificar Permissões* presentes na parte (b) e (c), respectivamente. Nesta situação, caso não sejam mantidas na pilha ("limpas") as cores de (d), o algoritmo de busca não percorreria o CFG (d) na segunda vez em que é realizada uma chamada para ele. Isso ocorreria porque as arestas do CFG (d) já estariam marcadas com a cor CINZA e conseqüentemente, não seria possível realizar a cobertura das arestas de (d) através da chamada realizada na parte (c) do IACFG.

Cobertura de Ciclos

Em um CFG, uma seqüência de vértices $(v_0, v_1, ..., v_k)$, onde $k \geq 0$ e os vértices pertencentes ao CFG, define um caminho de tamanho k se, e somente se, existir arestas partindo de v_{i-1} e chegando em v_i para todo i, tal que $1 \leq i \leq k$. O caminho parte do vértice v_0 e chega ao vértice v_k. Um ciclo (em inglês, *loop*) é um caminho em que $v_0 = v_k$. Os ciclos podem aparecer na especificação em várias situações, mas durante a seleção de caminhos, o algoritmo de busca em profundidade não permite que sejam selecionadas arestas iguais para garantir que a busca em profundidade termine.

Entretanto, o algoritmo de busca com contexto é executado em duas etapas pelo critério de cobertura e, quando a aresta a ser coberta pelo critério é uma aresta de retorno do ciclo, o algoritmo possibilita a cobertura de uma interação no ciclo. Na primeira etapa, a busca em profundidade seleciona um caminho com uma interação no ciclo até encontrar a aresta de retorno e na segunda, é coberta a aresta de retorno e mais uma interação no ciclo. Desta forma, pode-se garantir que todas as arestas são cobertas pelo critério de cobertura usando o algoritmo DFS-CONTEXTO.

Recursão nos modelos

A recursão nos modelos ocorre quando um CFG inclui ele mesmo. Neste caso, existe um vértice de chamada com uma aresta inter-atividade para o vértice inicial do mesmo CFG. Como apresentado no algoritmo da Figura 4.9, o vetor de cores (*cor[]*) é "limpo" para que em um mesmo caminho seja possível cobrir mais de uma vez o mesmo CFG, conforme mostrado na Figura 4.10.

Para que o algoritmo de busca não efetue a chamada recursiva infinitamente, a última aresta visitada antes que seja encontrado o vértice de chamada no caminho deve ser

mantida com a cor CINZA, mesmo após a entrada no novo contexto. Este tratamento, assim como o tratamento dos ciclos, garante que uma mesma chamada recursiva irá se repetir no máximo uma vez no mesmo caminho. Logo, a aresta para a chamada recursiva estará marcada como visitada (cor CINZA) na primeira interação recursiva no CFG e não poderá ser selecionada novamente pelo algoritmo de busca em profundidade.

Complexidade do Algoritmo Proposto

A complexidade computacional do algoritmo de busca em profundidade é linear no tamanho do grafo de entrada [CLRS01], ou seja, $O(V + A)$, onde o grafo neste trabalho é o IACFG. As alterações inseridas no algoritmo tornam o algoritmo menos eficiente, pois a complexidade do algoritmo tornou-se polinomial em relação ao tamanho do grafo de entrada, entretanto ainda assim o algoritmo é viável.

As principais diferenças, neste caso, são os procedimentos das *linhas 10* e *22* do algoritmo apresentado na Figura 4.9. Na *linha 10*, o algoritmo é linear em relação à quantidade de arestas inter-atividades do vértice final em questão, conforme foi apresentado acima. No caso da *linha 22*, o algoritmo percorre todos os vértice da pilha de busca em profundidade (*pilha_DFS*), que é no máximo linear em relação ao tamanho do caminho corrente. Desta forma, a complexidade do algoritmo modificado torna-se $O(A \times (V + A))$, ou seja, $O((A \times V) + A^2)$, tornando-o quadrático. Além disso, a quantidade de cenários de teste selecionados pelo critério de cobertura é linear em relação ao tamanho do caminho.

4.3 Cenários de Teste

Um caso de teste descreve o estado inicial do sistema antes de executar o teste, o ambiente para a sua realização, as entradas, resultados e a seqüência de passos necessária para a execução do teste [Bin99, capítulo 3]. Nesta dissertação é proposta a geração automática de casos de teste a partir dos caminhos de teste. Entretanto, os casos de teste propostos aqui são chamados de casos de teste genéricos, ou ainda cenários de teste do SUT. Eles não representam os dados de entradas de teste e resultados esperados, mas representam as ações que os atores e o SUT deve realizar.

Os casos de teste genéricos foram propostos inicialmente por Rayner em [Ray87] e possuem quatro características principais:

- O refinamento de seu propósito, definindo o objetivo principal do caso de teste;
- O corpo do cenário, que contém uma seqüência de eventos de teste e pontos de verificação para a possível verificação do veredicto de testes (passou, falhou ou foi inconclusivo);

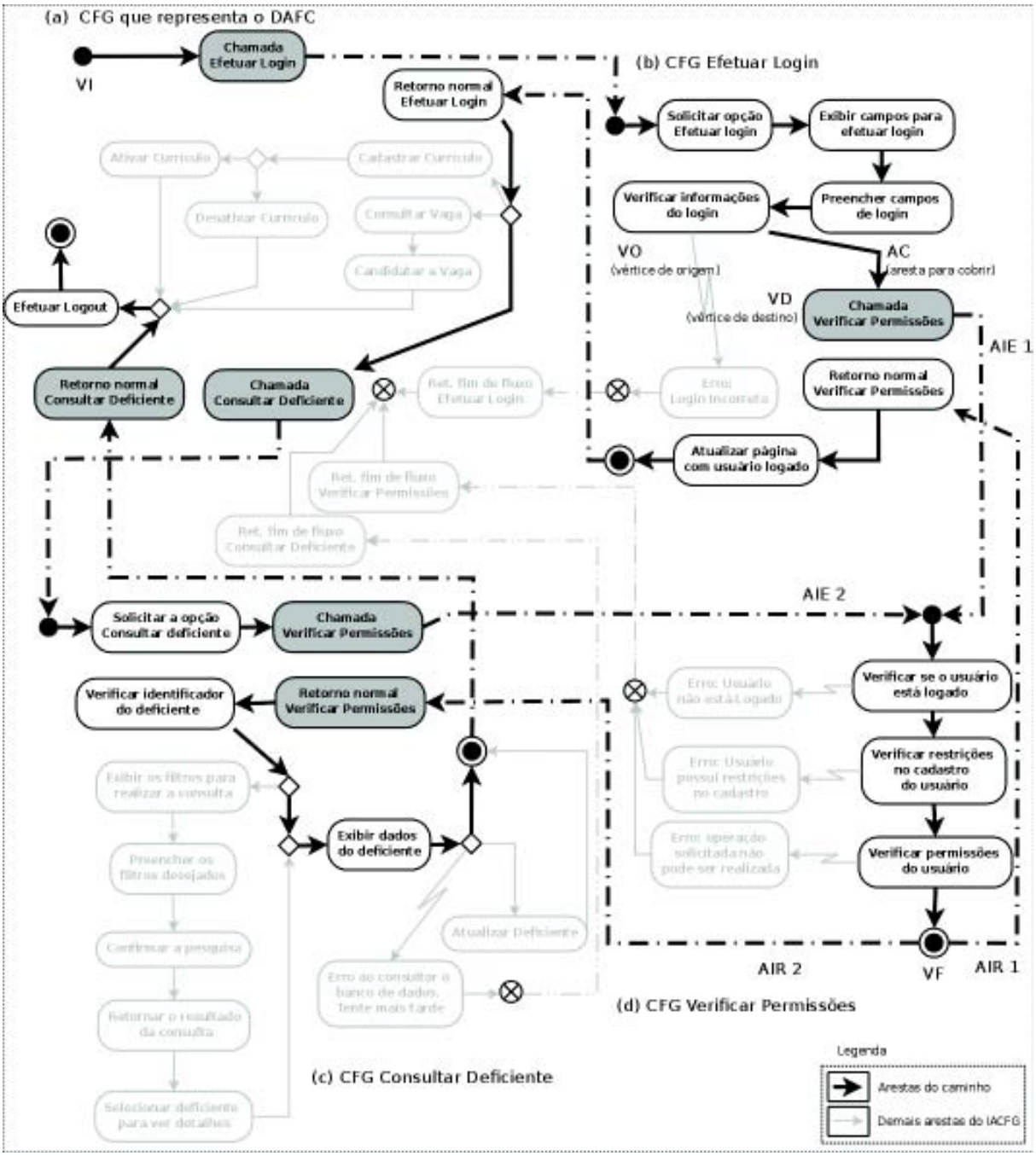

Figura 4.10: Caminho de testes com contexto completo.

- O estado inicial do sistema, para a possível execução do corpo do cenário de teste;
- As pós-condições, que são observadas após a execução do caso de teste.

Os casos de teste genéricos diferem dos casos de teste definidos por Binder [Bin99] principalmente por não apresentarem os dados de entrada e os resultados esperados. Desta forma, os cenários de teste (ou casos de teste genéricos) podem ser utilizados como padrão comum para a criação de diferentes casos de teste com dados de entrada e resultados esperados distintos, pois eles representam somente as ações necessárias para a execução do teste (ações realizadas pelos atores e pelo SUT) e não detalham as entradas e os oráculos de teste.

O corpo do cenário de teste contém um conjunto de passos para a execução de um ou mais casos de uso do SUT e é considerado completo quando é criado a partir de um caminho completo no IACFG. Portanto, um cenário de teste completo percorre uma seqüência de casos de uso no DAFC e, como cada caso de uso é detalhado em um DACI, os passos dos casos de uso estão representados e detalhados nos cenários de teste.

A Figura 4.10 destaca um caminho completo no IACFG em que as arestas da parte (d) do caminho são percorridas duas vezes sendo, a primeira na chamada realizada em (b) e a segunda na chamada realizada em (c).

Os cenários de teste são criados para atender às necessidades de uma atividade no processo de testes definido no Capítulo 2, chamada de **Criar Cenários de Teste**, e faz parte da fase de Especificação e Projeto de Teste no processo. Vale lembrar que cada cenário de teste do SUT é criado automaticamente a partir dos caminhos de teste selecionados.

A Tabela 4.2 apresenta um exemplo de cenário de teste gerado a partir do IACFG, o caminho que este cenário representa foi apresentado na Figura 4.10. O formato adotado para a representação do cenário de teste baseia-se no formato dos procedimentos de teste apresentados por Filho [dPPF01, Capítulo 18].

Uma vantagem em criar os cenários de testes anteriormente à criação dos dados e oráculos de teste é que estes cenários são independentes da arquitetura e implementação do SUT. Ou seja, não levam em conta "como" os atores ou o SUT realizam as atividades, mas somente "o quê" os atores ou o SUT fazem para que seja possível executar o conjunto de passos do cenário, assim a criação de cenários de teste é feita independentemente da implementação do SUT.

Casos de Teste com Dados de Entrada e Resultados Esperados

Vários trabalhos têm como objetivo propor meios para a geração automática de dados de teste para executar um conjunto de interações em programas ou em sistemas complexos. Entretanto, a geração de dados de teste não foi acrescentada ao escopo desta dissertação.

Tabela 4.2: Cenário de Teste que representa o caminho da Figura 4.10.

	Ações (Atividades) presentes no caminho
1	Solicitar opção efetuar login
2	Exibir campos para efetuar login
3	Preencher campos de login
4	Verificar informações do login
5	Verificar se o usuário está autenticado
6	Verificar se o usuário não tem restrições no cadastro
7	Verificar se o usuário tem permissão para realizar a operação
8	Atualizar página com usuário logado
9	Solicitar a opção consultar deficiente
10	Verificar informações do login
11	Verificar se o usuário está autenticado
12	Verificar se o usuário não tem restrições no cadastro
13	Verificar o identificador do deficiente
14	Exibir filtros para realizar a consulta
15	Preencher os filtros desejados
16	Confirmar a pesquisa
17	Retornar o resultado da consulta
18	Selecionar o deficiente para ver detalhes
19	Exibir dados do deficiente
20	Selecionar a opção efetuar logout
21	Confirmar a operação de logout

Os dados de teste servem como entrada para a etapa (v) apresentada na Figura 4.1, que também não faz parte do escopo deste trabalho.

A criação dos dados de entrada e dos resultados esperados para cada cenário de teste foi realizada manualmente pelos membros das equipes de teste que avaliaram o método de teste proposto. Desta forma, um caso de teste é um cenário de teste com um conjunto de dados de entrada e resultados esperados. Esses casos de teste podem ser considerados como instâncias de um cenário de teste, pois possuem os dados de entrada e os resultados esperados.

4.4 Comparação entre Estratégias de Seleção de Caminhos

Esta Seção apresenta uma avaliação da qualidade dos cenários de teste gerados pelo método de teste proposto comparando os cenários gerados com cenários gerados por outras estratégias de seleção de caminhos. Nesta etapa foi realizada a comparação dos cenários gerados utilizando os caminhos selecionados pelo algoritmo proposto na Seção 4.2.4 e os cenários de teste gerados a partir de outras duas estratégias de seleção de caminhos: seleção de caminhos com busca em profundidade e seleção de caminhos mínimos no grafo. Ambas as estratégias foram mencionadas na Seção 4.2.2 e são descritas em maiores detalhes por Cormen *et al* [CLRS01].

O estudo de caso apresentado nesta seção foi conduzido seguindo recomendações técnicas para avaliação de métodos e ferramentas proposta por B. Kitchenham *et al* [KPP95]. Desta forma, esta seção foi dividida em duas subseções, a primeira, Seção 4.4.1,

detalhando a preparação do estudo de caso; a segunda, Seção 4.4.2, detalhando a realização e os resultados obtidos no estudo de caso.

4.4.1 Preparação do Estudo de Caso

A preparação e realização deste estudo de caso têm por objetivo principal demonstrar que é viável a adaptação e uso de técnicas de teste "caixa branca" para geração de testes baseados em modelos da especificação do SUT. Desta forma, a criação de um grafo de fluxo de controle a partir da especificação, bem como a seleção de caminhos utilizando o algoritmo proposto na Seção 4.2.4 foram as adaptações necessárias nesta proposta para seleção de caminhos de forma similar à realizada em abordagens de teste "caixa branca".

A hipótese é que os caminhos de teste selecionados pelo método proposto apresentem ganhos em relação às estratégias comparadas, o que justificaria a criação de novas estratégias para seleção de teste ou adaptação nas estratégias utilizadas em testes "caixa branca" para seleção de testes a partir da especificação. A avaliação do método de geração de cenários de teste baseia-se na comparação do algoritmo proposto (DFS-CONTEXTO) com duas estratégias de seleção de caminhos: busca em profundidade (DFS) e seleção de caminhos mínimos. Estas estratégias podem ser empregadas, normalmente para seleção de caminhos em grafos, entretanto na presença grafos interligados com contextos de entrada e saída entre os grafos, que é o caso do IACFG, essas abordagens podem gerar uma quantidade muito grande de caminhos não executáveis.

Nesta avaliação tanto a estratégia de seleção de caminhos proposta quanto as estratégias utilizadas para comparação foram utilizadas para atender ao critério de cobertura de todas as arestas do IACFG, conforme apresentado na Seção 4.2.1. O estudo de caso foi monitorado e foram coletadas métricas sobre os modelos de teste criados (Diagramas de Atividades e IACFG) e métricas sobre os cenários gerados automaticamente. Desta forma, o projeto piloto para comparação dos testes gerados foi um sistema com cinqüenta e dois casos de uso e, para todos os casos de uso, foram criados os modelos de teste. Este sistema (SUT) será apresentado em maiores detalhes no Capítulo 5.

As métricas coletadas sobre os modelos gerados foram: o tempo para criação dos modelos, o tamanho do IACFG criado a partir dos modelos, o tamanho da especificação do SUT e a quantidade de modelos criados. Para cada estratégia de seleção de caminhos, foram coletadas ainda as seguintes informações: a quantidade total de caminhos selecionados, a quantidade de cenários de teste executáveis gerados a partir dos caminhos selecionados, o tamanho médio dos cenários de teste gerados e o tempo para geração dos cenários de teste.

A especificação dos modelos de teste foi realizada para o SUT, conforme foi proposto no Capítulo 3 e as atividades de planejamento de testes e projeto foram realizadas apoiadas

pelo processo de teste apresentado no Capítulo 2.

4.4.2 Realização e Resultados Obtidos

Durante a realização do estudo de caso foram obtidas informações sobre a criação dos modelos e sobre os cenários gerados a partir desses modelos. As informações coletadas são detalhadas a seguir.

Modelos criados e IACFG

A partir da especificação do SUT a equipe de testes criou os modelos necessários para o projeto dos cenários de teste, sendo eles os DACIs e o DAFC. O tempo total para criação dos cinqüenta e três modelos, um para cada caso de uso e um para o diagrama de fluxo entre os casos de uso, foi de aproximadamente setenta horas (70h). Este tempo inclui o tempo para o estudo, entendimento do sistema, revisão dos modelos criados e as interações com equipe de desenvolvimento para esclarecimento das eventuais dúvidas referentes a especificações do SUT. É importante ressaltar que a equipe de testes é independente da equipe de desenvolvimento e, desta forma, não participou das atividades de desenvolvimento e levantamento de requisitos.

A Tabela 4.3 apresenta um resumo das informações coletadas sobre os modelos criados e também do IACFG gerado automaticamente a partir dos modelos do SUT. O IACFG foi criado pelo protótipo apresentado na Seção 5.1 a partir dos modelos descritos no arquivo XMI de entrada. O arquivo XMI de entrada neste estudo de caso possui aproximadamente 14 mbytes de tamanho. Além disso, a Tabela 4.4 apresenta mais detalhes relacionados ao tamanho do IACFG criado.

Tabela 4.3: Resumo da criação dos modelos.

Modelos	**Quantidade**	**Tempo**
Casos de Uso	52	-
DACIs	52	1,23 horas
DAFC	1	5,8 horas
Geração do IACFG	1	3 segundos

A grande quantidade de Vértices de Chamada e Vértices de Retorno na especificação está relacionada ao intenso uso de inclusões nos casos de uso, mecanismo este que é representado pelas ações de *chamada de comportamento* no Diagrama de Atividades, conforme descrito no Capítulo 3. Além disso, para cada vértice de Chamada e Vértice de Retorno existe também uma aresta de inter-atividades associada.

A criação dos modelos de teste foi realizada a partir da especificação do SUT e foi equivalente a aproximadamente 24% do total de esforço empregado durante os testes

funcionais do sistema. Uma vantagem nessa especificação foi que após a geração de cenários de teste se tem a garantia de que os cenários gerados cobrem todos os passos dos casos de uso, regras de negócio e detalhamento de sub-passos dos casos de uso.

Tabela 4.4: Informações sobre o tamanho do IACFG.

Atributo	**Quantidade**
Arestas	1230
Vértices	1009
Arestas inter-atividades	256
Vértices de Chamada	115
Vértices de Retorno	141

Na seleção de caminhos, a adoção do critério de cobertura de todas as arestas do IACFG garantiu que todas as transições (ou arestas) especificadas nos Diagramas de Atividades fossem cobertas pela seleção de caminhos. Quando a criação de cenários de teste para os casos de uso é realizada de forma manual, a garantia de cobertura da especificação está sujeita a imprecisões [Gel04].

O requisito para que todas as arestas sejam cobertas é que todas elas sejam alcançáveis, ou seja, é necessário que exista pelo menos um caminho entre o vértice de entrada do IACFG e cada aresta do IACFG que deseja-se cobrir. Além disso, partindo da aresta que se deseja cobrir chegando ao vértice final do IACFG. As arestas não alcançáveis são provenientes de problemas na especificação dos modelos ou problemas na especificação dos casos de uso e, quando isso ocorre, os modelos ou a especificação dos casos de uso devem ser revisados.

Os Cenários de Teste

Os cenários de teste foram gerados utilizando três estratégias distintas de seleção de caminhos: a estratégia de DFS-CONTEXTO, a estratégia de DFS e a estratégia de seleção de caminhos mínimos. Como o IACFG não possui pesos nas arestas, então a seleção de caminhos mínimos foi realizada utilizando o algoritmo de busca em largura (BFS) [CLRS01].

A Tabela 4.5 apresenta o tempo total utilizado pelo protótipo para geração dos cenários de teste utilizando cada uma das estratégias de seleção de caminhos, bem como o número de caminhos selecionados em cada estratégia. O tempo para seleção de caminhos utilizando o algoritmo DFS-CONTEXTO foi superior ao tempo utilizado pelas demais estratégias, totalizando cinqüenta segundos, o que equivale a mais de duas vezes o tempo que a busca em profundidade simples. Apesar do desempenho inferior, esse resultado já era esperado por se tratar de um algoritmo quadrático em relação ao tamanho da entrada

que é comparado a dois algoritmos lineares: o DFS e o de seleção de caminhos mínimos utilizando busca em largura.

Em relação ao tempo necessário para seleção dos caminhos nos IACFGs, a estratégia DFS-CONTEXTO é viável apesar de ser menos eficiente que as estratégias comparadas. Isto ocorre devido a DFS-CONTEXTO ter um algoritmo quadrático sendo comparado com as demais que são lineares. Apesar da tendência de crescimento do tempo para seleção dos caminhos ser um pouco maior, a geração dos cenários de teste é realizada uma única vez durante a bateria de testes, não causando um grande impacto no tempo total dos testes.

Tabela 4.5: Geração dos cenários de teste utilizando as três estratégias.

	DFS-CONTEXTO	**DFS**	**Caminhos Mínimos (CM)**
Número de cenários normais	65	100	103
Número de cenários de exceção	96	96	96
Total de cenários	161	196	199
Tempo total para seleção de cenários	50 segundos	19 segundos	16 segundos

A quantidade de cenários de exceção selecionados não variou, pois o critério utilizado (apresentado na Seção 4.2.1) trata a cobertura de uma exceção por cenário de teste e, consequentemente, é selecionado um caminho para cada exceção representada nos DACIs que representam os casos de uso do SUT. Desta forma, só houve variação no número de cenários normais, sendo que a estratégia de seleção de caminhos mínimos apresentou o maior número de caminhos selecionados (199) seguida pela estratégia de busca em profundidade (196).

Apesar da estratégia de DFS-CONTEXTO ter sido a estratégia que apresentou o menor número de caminhos selecionados (161) essa estratégia apresentou também o maior tamanho médio nos cenários gerados, como apresentado na Tabela 4.6, sendo que em média foram sessenta e um passos para os cenários normais e vinte e sete passos para os cenários de exceção. A estratégia de DFS apresentou teve tamanho médio semelhante em relação ao DFS-CONTEXTO, por se tratarem de estratégias parecidas. Entretanto, a seleção de caminhos mínimos apresentou o menor tamanho médio, resultado este, que já era esperado por sempre buscar o menor caminho nos CFGs, apesar disso não garantiu o contexto nos caminhos.

Tabela 4.6: Quantidade média de passos por cenário de teste.

	DFS-CONTEXTO	**DFS**	**CM**
Cenários normais	61	58	43
Cenários de exceção	27	26	20
Cenários (todos)	40	45	32

Cenários de Teste Não Executáveis

Após a seleção dos cenários de teste foram analisados os cenários gerados utilizando as três estratégias de seleção de caminhos de teste. Esta análise teve como objetivo revelar quantos cenários de teste gerados pelas estratégias são não executáveis por apresentarem passos incoerentes (ou contraditórios) em sua descrição. As inconsistências consideradas aqui são quaisquer informações que tornam o cenário de teste não executável.

Esta análise é importante pois, quando são encontrados cenários inválidos é necessária a realização de adaptações para que os tornem válidos, ou quando isso é muito custoso, esses cenários devem ser desconsiderados na execução dos testes. Dessa forma, pode ser necessária a criação de novos cenários de teste para cobrir as ações dos Diagramas de Atividades que não são mais cobertas por nenhum outro cenário, e manter assim a garantia de que existem testes que cubram toda a especificação do SUT.

O número de cenários incoerentes identificados dentre os cenários gerados para cada estratégia são apresentados na Tabela 4.7. A estratégia de seleção de caminhos mínimos apresentou o maior número e também percentual de caminhos não executáveis, sessenta e três caminhos, sendo o equivalente a 31,66% do total de caminhos selecionados pela estratégia. A Estratégia DFS apresentou um resultado semelhante ao apresentado pela seleção de caminhos mínimos, sendo que 29,01% dos cenários de teste gerados apresentaram inconsistências.

Tabela 4.7: Quantidade e percentual de cenários não executáveis.

Estratégia	**Cenários Não Executáveis**	**Percentual**
DFS-CONTEXTO	18	11,18%
DFS	57	29,01%
CM	63	31,66%

A estratégia DFS-CONTEXTO proporcionou a redução do percentual de cenários de teste não executáveis, onde 11,18% dos cenários gerados foram cenários não executáveis. Mesmo com o tratamento de contexto entre os DACIs que representam os casos de uso do SUT, não foi possível eliminar a existência de cenários de teste incoerentes, pois a estratégia de seleção de caminhos não busca verificar as condições nos caminhos. Assim, pode haver inconsistências (contradições) entre dois passos, não possibilitando a seleção

de dados para cobertura deste caminho. A verificação das condições durante a seleção dos caminhos não foi considerada aqui para não criar restrições muito fortes na criação dos modelos, o que tornaria a abordagem de testes muito custosa ou até inviável, para sistemas como este testado.

A redução do número de cenários não executáveis apresenta indícios de que a seleção de caminhos com a estratégia de busca em profundidade com contexto é viável na geração de cenários de teste funcional a partir da especificação do SUT utilizando a modelagem proposta por apresentar uma baixa taxa de cenários de teste não executáveis (aproximadamente 11%).

Para que seja possível observar evidências nos ganhos proporcionados na utilização deste algoritmo, seria interessante a realização de um estudo controlado utilizando um conjunto de especificações de sistemas em domínios distintos de aplicação. Este estudo controlado não foi possível realizar durante esse trabalho por restrições de tempo e número de pessoas para realização do estudo de caso.

4.5 Resumo

Neste capítulo foi apresentado o algoritmo para seleção de caminhos de teste a partir de Diagramas de Atividades que representam a especificação do sistema em testes. No algoritmo proposto, para realizar a seleção de caminhos, os Diagramas de Atividades que representam a especificação do SUT foram transformados em CFGs, que interligados formaram um IACFG único contendo toda a especificação do SUT. A partir desse IACFG foram selecionados caminhos completos de teste que representam cenários do SUT. Além disso, foi realizado um estudo comparativo entre o algoritmo proposto neste capítulo para seleção de caminhos de teste com outras estratégias para seleção de caminhos existentes, a busca em profundidade e a mínimos.

Capítulo 5

Estudo de Caso utilizando o Método Proposto

Este Capítulo apresenta um estudo de caso para avaliação teórica do método de geração de cenários de teste apresentado no Capítulo 4. O estudo de caso em questão foi desenvolvido em duas etapas principais. Na primeira etapa, foram gerados cenários de teste utilizando método proposto (Capítulo 4). Em seguida, os cenários de teste foram utilizados para realização dos testes no SUT, os cenários de teste utilizados aqui foram os gerados no Capítulo 4. Nesta etapa o objetivo era observar indícios sobre a viabilidade de aplicação do método em sistemas reais. Na segunda etapa, os testadores envolvidos durante a modelagem e realização dos testes foram submetidos a uma pesquisa de opinião para avaliarem a utilização do método de geração de cenários de teste no projeto piloto.

Durante a realização do estudo de caso, a criação dos modelos para realização dos testes foi feita por uma equipe de testadores a partir da especificação de um sistema real, especificado e desenvolvido no âmbito do projeto Harpia. O projeto Harpia e o sistema testado serão apresentados em maiores detalhes na Seção 5.2. As atividades de preparação e realização dos testes funcionais foram efetuadas seguindo o processo apresentado no Capítulo 2. É importante ressaltar que o estudo de caso foi aplicado a um sistema (SUT) desenvolvido e testados por equipes independentes, no qual as três etapas de avaliação foram aplicadas.

Tanto a seleção de caminhos de teste quanto a geração dos cenários a partir dos caminhos selecionados foram realizadas com apoio de um protótipo que possibilitou a automação das atividades de criação do IACFG, seleção de caminhos de teste e geração dos cenários de teste, apresentadas no Capítulo 4 e que foi apresentado previamente por Perez e Martins [PM07].

O restante deste Capítulo é organizado da seguinte forma, a Seção 5.1 descreve o protótipo desenvolvido para apoiar a geração automática de cenários de teste; a Seção 5.2

apresenta o projeto Harpia e o sistema testado neste estudo de caso; a Seção 5.3 apresenta os resultados dos testes realizados em um sistema real utilizando os cenários gerados utilizando a estratégia de seleção de caminhos proposta e; por último, a Seção 5.4 apresenta a avaliação do método de testes realizada pela equipe de teste envolvida, nesta avaliação os testadores foram submetidos a um questionário.

5.1 O Protótipo para Geração dos Cenários

Para prova de conceito do método de geração de cenários de teste apresentado no Capítulo 4 foi implementando um protótipo que, futuramente, irá incorporar um ambiente integrado para geração de cenários, casos de teste e também seleção de testes para regressão que encontra-se em fase de especificação e desenvolvimento. Este ambiente integrado é chamado de Antares (A workbeNch for Testing And REgression test Selection).

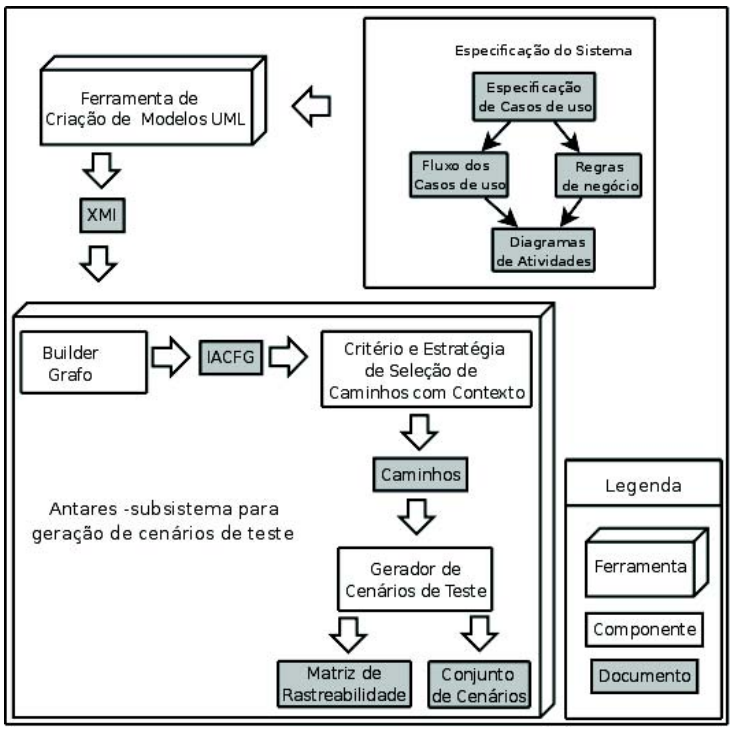

Figura 5.1: Protótipo para Apoiar a Geração dos Cenários de Teste.

A Figura 5.1 representa graficamente os módulos, os documentos de entrada para utilização do protótipo, bem como os documentos gerados como saída pelo protótipo. O protótipo recebe como entrada os Diagramas de Atividades no formato XMI [Gro03c], que é gerado por uma ferramenta de modelagem de diagramas UML. Os Diagramas de

Atividades são criados pela equipe de teste a partir da especificação do SUT, como é mostrado na Figura 5.1.

Os componentes internos do protótipo apresentados na Figura 5.1, automatizam as etapas do método da geração de cenários de teste apresentados no Capítulo 4. Dessa forma, a saída da ferramenta é o conjunto de cenários de teste e uma matriz de rastreabilidade entre modelos e cenários de teste.

Para gerar os cenários primeiramente devem ser selecionados quais DAs serão cobertos pelos cenários de teste gerados. Para fazer isso o usuário deve executar o protótipo, onde na primeira tela ele deve seleciona o arquivo XMI que será utilizado para gerar os testes e deve selecionar a opção *"Escolher DAs"*, como mostrado a Figura 5.3. Posteriormente, o protótipo apresenta a tela mostrada na Figura 5.3, onde o usuário deve selecionar quais os DAs devem ser cobertos (utilizando os *checkboxes*) e qual é o DA que representa o DAFC da especificação (*radiobutton*). Além disso, a tela apresenta ainda um campo para o usuário escolher o nome de um arquivo que deverá ser gravado com as informações selecionadas nas telas. Este arquivo gravado é o arquivo de entrada necessário para executar o protótipo de geração de cenários de teste implementado, como apresenta a Figura 5.4.

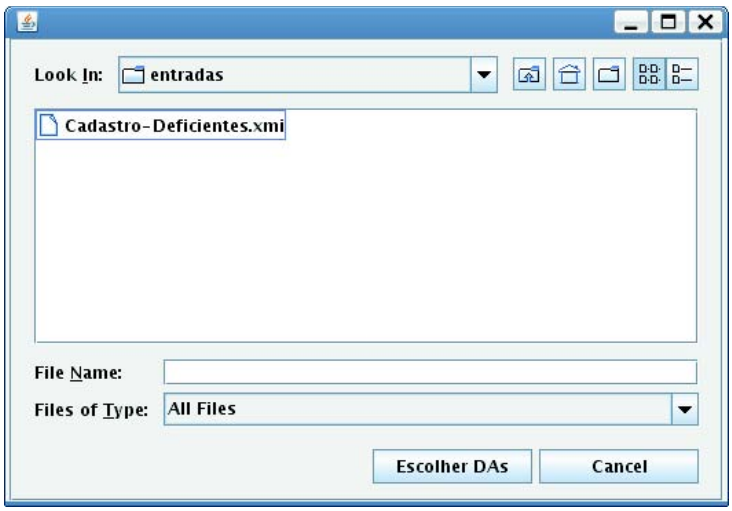

Figura 5.2: Tela para selecionar o arquivo XMI.

O protótipo implementado é compatível com a versão 1.2 de XMI e a ferramenta para criação dos modelos utilizada durante o estudo de caso foi a *Poseidon for UML* [AG06], pois ela dá suporte a criação de Diagramas de Atividades UML 2.0 [Gro04] e exporta arquivos no formato XMI.

Um exemplo de cenário de teste gerado pelo protótipo é apresentado na Tabela 5.1. Este cenário de teste tem uma formatação baseada na formação proposta por Filho

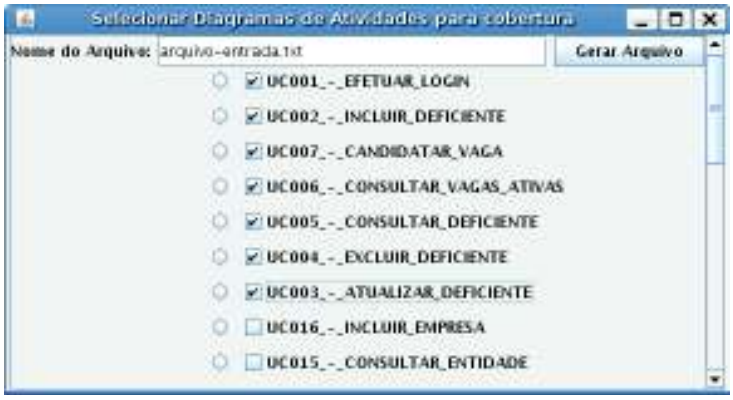

Figura 5.3: Tela para selecionar os DAs que os cenários de teste devem cobrir.

[dPPF01, Capítulo 13] e representa um caminho completo para o SUT utilizado como exemplo nos Capítulos 3 e 4. Na Tabela 5.1 o Identificador e objetivo do caso de teste deve ser atribuído pelo projetista de testes. As pré-condição e pós-condição do cenário de teste podem ser obtidas diretamente através das pré-condições e pós-condições contidas na descrição dos casos de uso do sistema. Além disso, requisitos especiais do sistema podem ser obtidos a partir dos casos de uso e documento de requisitos não funcionais do sistema.

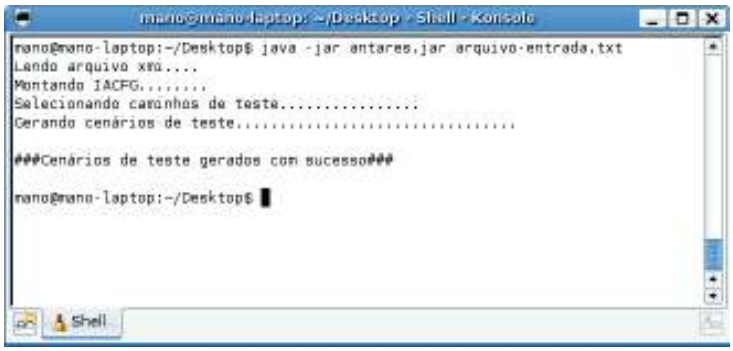

Figura 5.4: Gerar cenários de teste.

A partir dos estereótipos das Ações presentes nos modelos criados para os testes foram definidos três tipos de prefixos a serem inseridos nos passos dos cenários de teste: *O sistema deve*, *O usuário deve* e *O sistema deve exibir mensagem*. Esses prefixos são utilizados nos passos que representam ações dos DACIs que possuem os estereótipos *sistema*, *usuário* e *exceção* respectivamente, e indicam quando o evento deve ser realizado pelo sistema ou usuário.

Durante a realização do estudo de caso deste Capítulo foram coletados resultados relacionados ao desempenho do protótipo na criação do IACFG, seleção de caminhos e

geração de cenários de teste. Todo o trabalho foi realizado utilizando uma estação de trabalho do projeto Harpia, com a seguinte configuração: processador *Pentium IV* ® 2.8 ghz, com 2 gb de memória ram e com sistema operacional *Windows XP* ®.

Tabela 5.1: Cenário de Teste gerado automaticamente.

Cenário de Teste	
Identificador	CADASTRO-DEEFICIENTES-01
Objetivo	Realizar a consulta de um deficiente cadastrado no sistema de cadastro de deficientes
Pré-condição	O deficiente a ser consultado deve estar cadastro.
Pós-condição	O sistema deve exibir os dados do deficiente
Requisitos Especiais	Não se aplica.
Fluxo	
Passo	**Descrição**
1	O *usuário* deve solicitar opção efetuar login
2	O *sistema* deve exibir campos para efetuar login
3	O *usuário* deve preencher campos de login
4	O *sistema* deve verificar informações do login
5	O *sistema* deve verificar se o usuário está autenticado
6	O *sistema* deve verificar se o usuário não tem restrições no cadastro
7	O *sistema* deve verificar se o usuário tem permissão para realizar a operação
8	O *sistema* deve atualizar página com usuário logado
9	O *usuário* deve solicitar a opção consultar deficiente
10	O *sistema* deve verificar informações do login
11	O *sistema* deve verificar se o usuário está autenticado
12	O *sistema* deve verificar se o usuário não tem restrições no cadastro
13	O sistema deve verificar o identificador do deficiente
14	O sistema deve exibir filtros para realizar a consulta
15	O *usuário* deve preencher os filtros desejados
16	O *usuário* deve confirmar a pesquisa
17	O *sistema* deve retornar o resultado da consulta
18	O *usuário* deve selecionar o deficiente para ver detalhes
19	O *sistema* deve exibir dados do deficiente
20	O *usuário* deve selecionar a opção efetuar logout
21	O *usuário* deve confirmar a operação de logout
22	*Fim do cenário de teste*

5.2 O Sistema Testado

Para realização do estudo de caso foi utilizado um sistema de controle aduaneiro que foi desenvolvido no âmbito do projeto Harpia. O projeto Harpia é uma parceria firmada entre a Receita Federal do Brasil, a Universidade Estadual de Campinas - UNICAMP e o Instituto Tecnológico de Aeronáutica - ITA. Neste projeto são desenvolvidas soluções para combate a sonegação fiscal no país.

O estudo de caso foi realizado em um sistema de controle de importações que foi desenvolvido no decorrer de 2007 utilizando a linguagem de programação Java [Mic06]. Sua especificação é composta por cinqüenta e dois (52) casos de uso e sua implementação totalizou aproximadamente quarenta e quatro (44) mil linhas de código (klocs). A arquitetura do sistema é cliente servidor, sendo a interface com usuário final é Web, em que as funcionalidades são apresentadas e permitidas somente aos perfis de usuário apropriados.

O detalhamento dos casos de uso, bem como das funcionalidades presentes neste sistema não serão apresentados neste capítulo por se tratarem de informações confidenciais da Receita Federal do Brasil. Desta forma, este Capítulo irá apresentar somente as informações relacionadas aos cenários de teste gerados para este sistema, os resultados dos testes aplicados utilizando os cenários criados e o relato de experiência dos testadores envolvidos no projeto através de um questionário.

Para realizar o estudo de caso foi necessário projetar os modelos de teste através da criação dos Diagramas de Atividades (DACIs) para cada caso de uso do sistema e um diagrama de fluxo entre os casos de uso (DAFC), conforme definido no Capítulo 3. Ambos, tanto os DACIs quanto o DAFC foram projetados pela equipe de testes do projeto Harpia. A seleção de caminhos e geração de cenários de teste a partir desses modelos foi realizada utilizando o protótipo da ferramenta Antares, que foi apresentado na Seção 5.1.

5.3 Testes Funcionais do SUT Utilizando os Cenários Gerados

Os cenários de teste gerados pela estratégia DFS-CONTEXTO foram utilizados durante a realização dos testes do SUT. A utilização desses cenários de teste possibilitou avaliar se a aplicação do método de teste proposto foi viável para o estudo de caso, apresentando indícios de que este método pode ser utilizado para a realização de teste em outros sistemas reais.

Os testes foram realizados no SUT por uma equipe de testadores do projeto Harpia, que é composta por oito pessoas, sendo que três membros da equipe são alunos de mestrado, quatro são alunos de graduação e uma pessoa já concluiu o mestrado na área de testes de software, sendo que a área de pesquisa dos alunos de mestrado também a área de testes de software.

Antes da realização da bateria de testes, o código-fonte do SUT foi instrumentado para possibilitar a análise de cobertura dos teste. A ferramenta utilizada para realizar a instrumentação e análise de cobertura foi a ferramenta *Clover* [Sys07].

Os cenários de teste foram utilizados na criação dos casos de teste, que em seguida foram executados de forma manual no SUT. Para possibilitar a reaplicação dos testes de forma automática em baterias de teste futuras, foi utilizada também uma ferramenta para capturar e gravar a execução dos testes (*capture-playback*). A ferramenta de *capture-playback* utilizada foi a ferramenta *Selenium* [Ope06], que possibilita a gravação e reprodução de ações de usuários em navegadores de Internet. Os testes eram gravados em *scripts* da *Selenium* durante a sua primeira execução para possibilitar a re-execução automática deles em baterias de teste posteriores.

É importante ressaltar que só eram gravados *scripts* de teste para *Selenium* para os casos de teste quando as interfaces do SUT já estavam estáveis, pois na presença de alterações nas interfaces os *scripts* torna-se obsoletos. Desta forma, os testes que não possuem *script* foram persistidos em um formato estruturado de caso de teste baseado no padrão de caso de teste proposto por Filho [dPPF01], possibilitando a sua re-execução, mas de forma manual. Este formato de caso de teste é utilizado para persistir os dados de teste, ações dos atores e saídas esperadas na ordem que da realização das ações no sistema. Além disso, os dados de teste e as saídas esperadas foram especificados pelos próprios testadores.

Em relação geração manual de dados para os testes, foram utilizadas as técnicas de análise de valores limites e partição de equivalência tanto para os cenários de teste normais quanto para os cenários de teste com exceções. Nos cenários com exceção também foi necessário cadastrar valores inválidos em sistemas externos que interagiam com SUT e também são parte do escopo do projeto Harpia. Outras abordagens que foram necessárias para simular os cenários com exceções foram simular as falhas no banco de dados e as falhas de comunicação. No caso do banco de dados, as falhas eram simuladas com a queda (desligamento) do banco ou até remoção dos dados existentes no banco. E no caso das falhas de comunicação, além das falhas de rede, foram simuladas falhas com envios e recebimentos na Web (em inglês, *request e response*) inválidos ou falhas com dados inválidos nos envios e recebimentos Web. Para interceptar e alterar os envios e recebimentos foi necessária a utilização de plugins no navegador de internet [Ped07, Hew07, Jud07].

Uma priorização em quais cenários seria concentrado o maior esforço de testes foi feita antes da execução, dentre os 161 cenários de teste gerados (Tabela 4.5). Desta forma, foram identificados, pelos lideres das equipes envolvidas (testes e desenvolvimento), cinqüenta e quatro cenários com alta prioridade, setenta e um cenários com prioridade média e trinta e seis cenários com prioridade baixa.

A Tabela 5.2 apresenta o resumo dos resultados obtidos após a realização da bateria de testes no SUT em questão. Como mostrado na tabela, nem todos os cenários de teste projetados foram executados. Isto se deve a que treze cenários tornaram-se obsoletos devido a alterações na especificação do SUT que não foram atualizadas na documentação de casos de uso. Além disso, outros dez cenários de teste foram impedidos de serem executados por terem sido identificados outros defeitos que impediram as suas execuções.

Para cada um dos cento e trinta e oito (138) cenários de teste executados, foram criados casos de teste com objetivo de executar o SUT percorrendo as funcionalidades descritas por seus passos (corpo do cenário de teste, conforme descrito no Capítulo 4) e verificando se as situações observadas no sistema ocorriam conforme as situações esperadas representadas na especificação dos cenários de teste.

Tabela 5.2: Resumo dos resultados dos testes.

Total de cenários de teste	161
Total de cenários executados	138
Total de casos de teste	325
Casos de Teste que passaram	156
Casos de Teste que não passaram	169
Quantidade de defeitos revelados	5,6 por klocs
Defeitos com severidade Crítica	9%
Defeitos com severidade Alta	25%
Defeitos com severidade Média	30%
Defeitos com severidade Baixa	36%
Cobertura de instruções no SUT	79%
Cobertura de condições no SUT	68%
Total de tempo para executar os testes	290 horas

Foram criados trezentos e vinte e cinco (325) casos de teste associados aos cenários de teste, sendo que cento e cinqüenta e seis casos de teste passaram, ou seja, tiveram sua execução realizada sem revelar a presença de falhas no SUT. Os cento e sessenta e nove (169) casos de teste que revelaram a presença de defeitos no SUT eram associados a noventa e cinto (95) cenários de teste dos cento e trinta e oito cenários executados.

O número de casos de teste criados para cada cenário de teste variou conforme a prioridade em que foi classificado. Além disso, a possibilidade de utilização de dados de teste com valores nos limites ou valores em bordas no domínio de entrada também influiu na quantidade de casos de teste criados.

A classificação por severidade dos defeitos foi realizada pela equipe de teste de seguinte forma:

- Severidade crítica, na presença de defeito que impedem o término da execução do cenário de teste;
- severidade alta, quando o cenário termina a execução, mas os resultados obtidos eram diferentes do especificado;
- severidade média, nas situações em que o cenário de teste executava apresentando resultados diferentes do esperado, mas os resultados inválidos não eram persistidos;
- e finalmente, Severidade baixa, quando o cenário executava normalmente com resultados corretos, mas havia problemas na apresentação dos resultados (problemas de interface).

Após o término dos testes foi verificada a cobertura de código obtida. Conforme apresentado na Tabela 5.2, foram cobertas 79% das instruções do programa e 69% de suas condições. Além disso, os defeitos foram repassados para o líder da equipe de desenvolvimento e, após desconsiderar os defeitos duplicados e inválidos, a equipe de testes observou que foram identificados 5,74 defeitos por klocs do SUT.

Por último, a Tabela 5.2 apresenta a quantidade total de horas utilizadas pela equipe de testes na criação, execução e revisão dos casos de teste. O tempo total foi de duzentas e noventa (290) horas distribuídas em três semanas, ou quinze dias úteis.

Deve-se destacar que não existem evidências que indicam que a qualidade do conjunto de casos de teste associado aos cenários de teste gerados pelo método de teste proposto contribuiu com melhorias em relação a abordagem de testes manual de criação de cenários e casos de teste. Entretanto, pode se observar que foi possível aplicar o método para geração de cenários de teste em um SUT real.

Além deste SUT, foram criados os modelos para gerar e executar os cenários de teste em outros sistemas do projeto Harpia que estão em fase de desenvolvimento. Também foram projetados e gerados os cenários de teste para o sistema de cadastro de deficientes, que foi utilizado como exemplo nos Capítulos 3 e 4. Tanto a especificação dos testes (modelos e cenários gerados) quanto os resultados dos testes para o sistema de cadastro de deficientes foram apresentados no trabalho de conclusão de curso de Patuci [Pat07].

5.4 Avaliação do Método pela Equipe de Testes

Para avaliar se a utilização dos cenários de teste facilitou a realização dos testes no SUT, após o término da bateria de testes, os testadores envolvidos foram submetidos a um questionário. O questionário que foi utilizado é apresentado no Apêndice A e foi elaborado seguindo as recomendações para elaboração de pesquisa de opinião propostas por B. Kitchenham e S. L. Pfleeger [PK01, KP02c, KP02a, KP02d, KP03].

A organização teve seu formato de apresentação semelhante ao utilizado por P. Runeson [Run06]. Desta forma, o questionário foi organizado em quatro principais assuntos: experiência do testador, criação dos modelos para realização dos testes, avaliação dos cenários de teste e sugestões para melhoria do método de teste proposto. Cada assunto contém perguntas relacionadas a ele e, além disso, para cada pergunta são apresentadas questões (afirmações) relacionadas a essa pergunta. Estas questões foram respondidas de forma objetiva pelos testadores utilizando as opções proposta para cada grupo de questões, conforme é apresentado no Apêndice A.

A coleta e avaliação dos resultados da aplicação do questionário tiveram por objetivo principal observar indícios de melhorias e problemas no método de teste proposto. Esta avaliação foi realizada pela equipe de testes envolvida no estudo de caso, que foi o projeto

piloto para aplicação deste questionário.

O Apêndice B apresenta os gráficos com os totais e percentuais das respostas obtidas para cada uma das questões do questionário. É importante ressaltar que, para observar resultados com evidências estatísticas, seria necessária a aplicação deste questionário em um conjunto maior de pessoas, uma população alvo bem definida e a escolha de um sub-conjunto representativo desta população para responder ao questionário [KP02b].

A primeira parte do questionário teve como objetivo coletar a experiência dos testadores envolvidos. Neste aspecto foi observado que apenas dois dos sete testadores haviam tido experiência em testes de projetos com características diferentes, dois deles já haviam realizado testes em sistemas similares, e quatro dos sete disseram ter conhecimentos teóricos de testes. Em relação à execução de testes, todos afirmaram já terem executado testes manualmente. Além disso, aproximadamente metade das pessoas envolvidas afirmou que já automatizaram a execução de testes utilizando ferramentas, *frameworks* de teste ou ferramentas de *capture/ playback*.

Na criação dos casos de teste, os testadores envolvidos consideraram que os cenários de teste gerados pelo método proposto facilitaram a criação de casos de teste e consideraram também que os cenários facilitaram o entendimento do sistema.

Na segunda parte da pesquisa de opinião os testadores responderam questões relacionadas a criação dos modelos necessários para o projeto dos cenários de teste.

Neste aspecto, os testadores consideraram que a modelagem proposta (Capítulo 3) apresentou as informações necessárias para o projeto dos casos de teste. Além disso, na maior parte das vezes, não foi necessário recorrer a outros modelos da especificação para criar os casos de teste do SUT, pois os cenários de teste gerados já apresentavam informações da especificação suficientes para a obtenção dos cenários de teste.

Ainda segundo os testadores, foi possível realizar a atualização dos modelos quando a especificação dos casos de uso era atualizada, mantendo assim a consistência da especificação dos cenários de teste. Quando aplicável, foram feitas atualizações dos modelos a partir de suas versões anteriores. Ou seja, não houve dificuldades para realizar a evolução nas versões dos modelos para acompanhar as atualizações da especificação do SUT.

Os cenários de teste gerados, de forma geral, facilitaram a realização dos testes. Isto se deve ao fato dos testadores, na maior parte das vezes, não necessitarem de outros documentos da especificação para realização dos testes e, quando foi necessário utilizar outros modelos, os mais utilizados foram os modelos criados pela própria equipe de teste, os Diagramas de Atividades.

O nível de detalhamento dos cenários de teste também foi satisfatório, do ponto de vista dos testadores, sendo que, na maior parte das situações, os cenários apresentaram informações relevantes para a criação dos casos de teste e para obtenção dos dados de testes. Além disso, as informações relacionadas às situações esperadas (ou ações do SUT)

que foram representadas nos cenários de teste foram úteis para revelar a presença de defeitos no sistema. Entretanto, em algumas situações os testadores ainda precisaram recorrer a outros documentos da especificação para verificar se a situação observada durante a execução do SUT foi diferente da situação esperada (especificada).

A existência de inconsistências nos cenários gerados, conforme apresentado na Seção 4.4, dificultou a criação dos casos de teste e, em algumas situações, revelaram até defeitos inválidos. Ao se deparar com cenários incoerentes, os testadores realizaram adaptações nos cenários de teste para que fosse possível a criação dos casos de teste.

Em relação as situações que mais impactaram na criação dos casos de teste a partir dos cenários gerados, os testadores concordaram que a criação dos casos de teste é dificultada na presença de inconsistências ou de poucos detalhes nos cenários de teste. Desta forma, para melhorar a abordagem proposta é interessante identificar os tipos de passos apresentados nos cenários de teste que podem ser apresentados com uma menor ou maior ênfase ou até omitidos, para facilitar a criação dos casos de teste. Os testadores consideraram ainda que a criação de *templates* de casos de teste para cada cenário facilitaria a criação dos casos de teste. Também segundo os testadores, a utilização de uma abordagem para regressão de testes facilitaria a realização dos testes.

A partir desta avaliação preliminar realizada após a realização do estudo de caso no projeto piloto, é possível observar neste estudo de caso que foi possível criar os casos de teste a partir dos cenários de teste gerado, sendo que poucos cenários apresentaram inconsistências que os tornaram inválidos. Além disso, o nível de detalhamento dos cenários de teste foi satisfatório, facilitando a criação e avaliação dos resultados da execução dos casos de teste a partir dos cenários.

5.5 Resumo

Neste capítulo foi apresentado o estudo de caso em que foram realizados testes utilizando os cenários de teste gerados pelo método proposto. O sistema testado foi especificado e desenvolvido pela equipe do projeto Harpia, parceria entre a Receita Federal do Brasil, UNICAMP e ITA. Os cenários de teste foram gerados com a ferramenta desenvolvida neste mestrado, a Antares, que tem como objetivo gerar cenários de teste a partir de modelos da especificação do SUT. Os testes foram realizados com sucesso e os testadores envolvidos foram submetidos à um questionário para avaliação do método proposto.

Capítulo 6

Trabalhos Relacionados

As abordagens de testes baseados em modelos (MBT) têm sido bastante aceitas e também utilizadas atualmente. Em MBT, os modelos criados durante o processo de desenvolvimento são utilizados para derivação automática dos casos de teste. Desta forma, uma questão decorrente em MBT é: como criar modelos que possuem informações necessárias para geração de testes de forma eficiente [Ber07]?

Várias pesquisas têm sido realizadas buscando responder essa questão relacionada à modelagem para testes. Esse conceito também é conhecido como "*Design for Testability*" [Bin94]. Os modelos são utilizados nas mais diversas fases de teste (unidade, integração, sistemas e etc.) e, dependendo da necessidade em cada fase ou projeto em particular, eles podem ser obtidos a partir da especificação ou do código-fonte do SUT. Particularmente, os principais trabalhos relacionados a esta dissertação são os que utilizam modelos para apoiar a automação de testes funcionais de sistema ou componentes de software.

Este Capítulo detalha trabalhos relacionados a geração de testes a partir de modelos. Os aspectos mais relevantes observados nos trabalhos são os modelos para apoiar as atividades de teste e a geração automática (casos ou cenários) de teste a partir desses modelos. A criação dos modelos para realização dos testes foi apresentada nessa dissertação nos Capítulos 3 e 4, que apresentaram respectivamente os Diagramas de Atividades utilizados para a criação dos testes e os Grafos de Fluxo de Controle criados a partir dos Diagramas de Atividades. Desta forma, foi dada maior ênfase em trabalhos relacionados que utilizam Diagramas de Atividades UML e Grafos de Fluxo de Controle criados a partir da especificação do SUT. A geração automática de cenários de teste é relacionada à seleção de caminhos de teste e geração de teste, que também foi proposta no Capítulo 4.

O restante do texto deste capítulo está organizado da seguinte maneira: a Seção 6.1 apresenta os trabalhos relacionados que propõe a geração de testes a partir de Diagramas de Atividades; a Seção 6.2 apresenta os trabalhos relacionados que propõem a geração automática de testes a partir Grafos de Fluxo de Controle criados a partir da especificação

de sistemas ou componentes; a Seção 6.3 apresenta trabalhos relacionados que utilizam outros tipos de modelos na geração dos casos de teste e; a Seção 6.4 apresenta um resumo das características de cada abordagem apresentada.

6.1 Diagramas de Atividade

Alguns trabalhos têm sido desenvolvidos para criação de testes a partir de Diagramas de Atividades DAs UML. Neste âmbito, nesta seção foram selecionados seis (6) trabalhos relevantes que serão detalhados abaixo.

Para a geração de testes a partir de modelos, L. Briand e Y. Labiche propuseram uma metodologia de teste, chamada *TOTEM* [BL02]. Esta metodologia propõe a especificação de testes a partir de modelos de Casos de Uso, onde os DAs são utilizados para descrever as dependências seqüenciais entre os casos de uso do SUT. Desta forma, Briand e Labiche criaram um DA que representa as dependências entre os casos de usos para cada ator do SUT. A partir desses DAs são derivados caminhos que representam seqüências de casos de uso do sistema, ou seja, cenários de uso do sistema.

Cada caso de uso é detalhado através de diagramas de seqüência e/ ou colaboração, que representam os cenários internos dos casos de uso, já associados com os diagramas de classe do sistema em testes. Os diagramas de seqüência devem ser obtidos das fases de análise e projeto do SUT, já os diagramas de atividades devem ser criados pela equipe de teste.

Um problema encontrado na abordagem de Briand e Labiche é o grande número de combinações de cenários de teste gerados. Isso pode tornar a execução de todos os casos de teste inviável em sistemas complexos. Além disso, outro problema na *TOTEM*, a seleção de caminhos nos diagramas de seqüência (que representam os cenários dos casos de uso) é feita a partir de expressões regulares criadas manualmente para representar esses diagramas, o que pode tornar o uso dessa metodologia inviável para sistemas com muitos casos de uso. Para geração de testes a *TOTEM* depende de informações estruturais do SUT, dificultando a manutenção dos casos de teste dos SUTs que sofrem muitas mudanças estruturais durante o desenvolvimento.

Conforme apresentando no Capítulo 3, o uso de DAs para representação das dependências seqüenciais entre os casos de uso do SUT foi baseado na *TOTEM* [BL02]. Entretanto, diferentemente da *TOTEM*, a descrição dos cenários internos dos casos de uso são feitas também através de DAs e não utiliza informações estruturais do SUT.

Linzhang *et al* propuseram um método para gerar casos de teste a partir de DAs chamado de *UMLTGF* [LJX+04]. Os DAs utilizados contém informações da especificação comportamental ("caixa-preta") e informações estruturais da implementação do SUT ("caixa-branca"). As abordagens de teste nas quais os modelos possuem informações

da especificação e informações da implementação são conhecidas como abordagens de teste "caixa-cinza". Os testes são gerados pelo método proposto utilizando um algoritmo de busca em profundidade visando cobrir todos os caminhos básicos do DA, considerando a cobertura de uma interação de cada ciclo. Os caminhos selecionados representam os casos de teste, que podem ser executados de forma automática no SUT. Entretanto, neste trabalho não foi detalhado como são obtidos os dados necessários para os casos de teste gerados.

Para demonstrar a viabilidade do método, Linzhang *et al* apresentaram a especificação de um SUT exemplo utilizando os modelos propostos e o formato de um caso de teste gerado a partir desses modelos. Apesar disso, este trabalho não entra em detalhes sobre "quais" informações e "como" são detalhadas as informações presentes no DA utilizado para realização dos testes. Além disso, o método não possibilita a geração de casos de teste em situações onde existe representação hierárquica de DAs, que detalham uma atividade em sub-atividades (Capítulo 3).

Um trabalho que propõe a geração de cenários de testes a partir de DAs e que suporta a representação hierárquica das atividades foi proposto por Bai *et al* [BLL04a]. Diferentemente do método proposto por Linzhang *et al*, Bai *et al* propuseram um método de testes que é "caixa-preta", onde os DAs utilizados na geração dos cenários de teste possuem somente informações da especificação do SUT.

Nesta abordagem, Bai *et al* propuseram a geração dos cenários de teste a partir de um CFG (chamado de hiper-grafo) que agrega todos os DAs. Como resultado, Bai *et al* geram três árvores que representam os chamados "*thin-thread*" que representam os cenários de teste, as condições de guarda e os tipos de dados dos cenários. Para avaliar a viabilidade do método foram gerados e executados os cenários de teste em um sistema ATM.

Um problema na abordagem de Bai *et al* é a necessidade de uma etapa de pré-processamento dos modelos realizada manualmente e, somente após ela, é possível a criação do CFG para derivação dos cenários de teste. Durante o pré-processamento deve ser criado de forma manual um DA que agrega todos os DAs da especificação. Esse DA é chamado de diagrama "inflado", pois contém todos os DAs da especificação e os DAs que representam as sub-atividades são expandidos.

Um problema no pré-processamento, é a possível introdução de falhas no DA "inflado", já que esse procedimento é realizado manualmente. Além disso, para especificações de sistemas complexos (com muitos DAs) esse pré-processamento torna-se inviável, pois pode-se tomar muito tempo para ser realizado manualmente. A criação do Grafo de Fluxo de Controle expandido foi feita de forma automática nessa dissertação, esse CFG expandido equivale ao IACFG e foi apresentado no Capítulo 4.

Uma situação que também não foi considerada por Bai *et al* é quando um DA pode ser utilizado como sub-atividade por várias atividades dos DAs especificados, ou seja, o

mesmo DA é reutilizado (chamado) como sub-atividade para representar o mesmo comportamento em vários pontos da especificação. Nesta situação, as sub-atividades do DA serão repetidas em vários pontos do DA "inflado". Consequentemente, o tamanho do DA "inflado" nem sempre é linear em relação ao tamanho dos DAs da especificação, e isso implicaria também na geração de um número muito grande de cenários de teste. Para solucionar esse problema, nessa dissertação o IACFG, apresentado no Capítulo 4, mantém apenas um CFG por DA representado na especificação, baseando-se na proposta de Harrold *et al* [HRS98] que foi melhor detalhada no Capítulo 2. Além disso, o IACFG proposto nesta dissertação é criado automaticamente a partir dos modelos, diferentemente da abordagem de Bai *et al* que necessita de uma etapa de pré-processamento dos modelos que é manual.

Chandler *et al* [CLL05] propuseram uma abordagem para geração de cenários de teste a partir de DAs que representam os cenários dos casos de uso do SUT, sendo que esta abordagem foi chamada de *AD2US*. O método *AD2US* seleciona caminhos nos DAs utilizando um algoritmo de busca em profundidade e cada caminho selecionado representa um cenário de uso do SUT.

Uma vantagem mencionada por Chandler *et al*, em relação ao método proposto por Bai *et al*, é que este método não necessita da etapa de pré-processamento dos modelos antes da geração dos cenários de uso. Entretanto, assim como Linzhang *et al*, o método *AD2US* também não considera as representações hierárquicas dos DAs durante a geração dos cenários de uso, o que limita a geração de cenários de uso para detalhar informações sobre inclusões e extensões de casos de uso. Em relação à abordagem de modelagem proposta nesta dissertação (Capítulo 3), o método *AD2US* não distingue as ações dos casos de uso que são realizadas pelos atores das ações esperadas do SUT.

Mingsong *et al* propuseram uma ferramenta, chamada de *AGTCG*, para geração automática de casos de teste para programas em Java [MXX06]. Os critérios de cobertura adotados nesta ferramenta foram: cobertura de todas as transições do modelo e cobertura de todos os caminhos básicos. Além disso, a ferramenta proposta por Mingsong *et al* possui uma interface para criação dos diagramas de atividades, onde cada diagrama representa uma classe em Java e as partições nesses diagramas representam os seus métodos. A partir desse diagrama de atividades a *AGTCG* instrumenta o programa fonte e através da instrumentação no programa é realizada de forma automática a verificação do resultado da execução do caso de teste.

Apesar de esse trabalho tratar da especificação das classes, os DAs criados consideram informações internas das classes, ou seja, relacionadas ao código fonte das classes em testes, o que pode dificultar a manutenção dos DAs. Além disso, um problema nesta abordagem de geração de teste é que ela só suporta entradas que sejam números inteiros e também não apresenta como são tratadas as interações necessárias com outras classes,

que são possíveis *stubs*.

Outro trabalho que utiliza DAs para apoiar a geração de casos de teste foi proposto por Hartmann *et al*, [HVFR04]. Neste trabalho, os autores propuseram a utilização dos DAs para descrição detalhada dos casos de uso do sistema. Assim, cada caso de uso deve ser representado por um diagrama de atividades e a partir dos caminhos nestes diagramas são criados os casos de teste.

Uma extensão a este trabalho, que foi realizada pelo mesmo grupo de pesquisa, propõe a identificação das dependências seqüenciais entre os casos de uso, baseada na metodologia *TOTEM* proposta por L. Briand e Y. Labiche. Vieira *et al* [VLH+06] propuseram a utilização dos diagramas de atividade para representar os passos dos usuários e as variáveis necessárias nas interações dos usuários em sistemas com interações com interface gráfica. As variáveis são definidas através de anotações no DA e são usadas nas condições das transições do modelos e, desta forma, tanto para o fluxo de dados quanto para o fluxo de controle.

Nesta abordagem é necessária a criação de modelos que contenham as condições de guarda de forma concisa para evitar que seja gerado um número grande de combinações de caminhos. Um problema nessa abordagem é que a definição das anotações e condições de guarda nos modelos é o ponto que toma maior tempo durante os testes [VLH+06], sugerindo que a criação dos modelos com as anotações e realização dos testes pode levar tanto tempo quanto a realização dos testes manualmente. Apesar disso, na discussão dos resultados, Vieira *et al* observaram que o esforço para manutenção do conjunto de testes criado a partir dos modelos foi reduzido, o que acaba compensando a utilização desta abordagem de testes.

Diferentemente do trabalho por Vieira *et al*, a modelagem proposta no Capítulo 3 não utiliza anotações e, como apresentado no Capítulo 5, a criação dos modelos não foi a atividade que mais consumiu tempo durante a realização do estudo de caso. Além da representação dos passos do usuário, a especificação proposta nesta dissertação representa as ações esperadas do SUT e dos cenários de exceção no SUT, conforme foi mostrado no Capitulo 3. Como os cenários de teste são descritos em alto nível, essas ações esperadas acabam facilitando a verificação se o sistema se comportou como esperado. Além disso, a abordagem proposta nesta dissertação propõe a criação (de forma automática) de um único modelo em um passo intermediário da abordagem de testes. Esse modelo é o IACFG apresentado no Capítulo 4.

6.2 Grafos de Fluxo de Controle em Testes "Caixa Preta"

Em testes "caixa preta", alguns trabalhos propõem a criação de Grafos de Fluxo de Controle Comportamental (ou CFG Comportamental) a partir da especificação do sistema. Dentre estes, foram identificados dois trabalhos relevantes relacionados a essa dissertação. Eles foram propostos por Parrish *et al* [PBC93] e por Edwards [Edw00], os quais serão descritos a seguir.

Parrish *et al* propõem a criação automática de cenários de teste para classes. O CFG Comportamental proposto neste trabalho é formado por vértices que representam as operações da interface da classe em teste, ou seja, as possíveis mensagens; e as arestas ligando quaisquer dois vértices do CFG Comportamental, representam seqüências válidas de chamada dessas operações da interface. Esse grafo foi denominado "Grafo de Fluxo da Classe" (em inglês, *Class Flow Graph*). Parrish *et al* propõem ainda um *framework* para apoiar a aplicação de diversos critérios de testes a partir desse modelo.

Edwards [Edw00] propôs uma abordagem para testes semelhante a de Parrish *et al*, mas em seu trabalho a criação do CFG Comportamental é feita para testes "caixa preta" em Componentes de software. O CFG Comportamental é criado de maneira semi-automática, se o componente for especificado na linguagem Resolve [SW94], que é uma linguagem utilizada para especificação formal de contratos (pré, pós-condições e invariantes). O CFG Comportamental criado por Edwards representa as interações na interface provida do componente em teste.

Tanto Parrish *et al* quanto Edwards referem como principal trabalho prévio Zweben *et al* [ZHK92], que propôs o uso de CFG para representar o comportamento de classes e também propôs o uso de técnicas de testes estruturais para realização de testes a partir da especificação.

Uma limitação dessas abordagens é o fato da equipe de testes ter que especificar os CFG Comportamentais diretamente. Nessa dissertação, é proposta uma abordagem em que os CFGs e o IACFG é criado de forma automática a partir dos modelos especificados pelos projetistas de teste. A utilização de modelos da UML é interessante pois estes modelos possuem grande aceitação na indústria de desenvolvimento e também na academia e, além disso, diversas metodologias de teste têm sido propostas com base nestes modelos, bem como ferramentas para apoiar essas metodologias.

Os trabalhos relacionados que utilizam CFGs interligados foram detalhados no Capítulo 2. Esses trabalhos utilizam os CFGs interligados realização de testes "caixa branca" [HRS98, SHR01].

6.3 Outros Modelos

Além dos diagramas de atividades e grafos de fluxo de controle comportamental, modelos que foram utilizados nessa dissertação, a modelagem e a geração cenários de testes podem ser realizadas através de outros tipos de modelos. Muitos trabalhos têm sido propostos para geração de testes utilizando outros modelos UML, sendo que os diagramas mais utilizados são os diagramas de seqüência [BBM02, LS06, DTGF06, PAK+07] e diagramas de estado [OA99, CCD03, HN04, MAMS06].

A metodologia *TOTEM*, conforme mencionado na Seção 6.1, além de utilizar diagramas de atividades para representar as interações entre os casos de uso, também utiliza diagramas de seqüência para geração de testes. Uma outra metodologia de testes que utiliza diagramas de seqüência para na geração de casos de teste foi proposta por Basanieri *et al* [BBM02]. No método de teste proposto por Basanieri *et al* foi desenvolvida uma ferramenta chamada de *Cow_Suite*, que automatiza a geração dos testes. Uma outra ferramenta para geração de testes que também se baseia em modelos UML, mas que gera os testes a partir de Diagramas de Estado, é a ferramenta *AGEDIS*, que foi proposta por A. Hartman e K. Nagin [HN04].

Além dos modelos UML e CFGs, existem diversos modelos para realização de testes que são detalhados em vários livros de testes de software [Bei90, Bei95, Bin99]. Desses modelos, um modelo conhecido que é bastante utilizado em testes de software e testes de protocolos é a máquina de estado finito (em inglês, *Finite-State Machine* – FSM). Petrenko apresenta uma bibliografia comentada que relaciona os principais trabalhos com objetivo de gerar testes a partir da especificação de FSMs [Pet01].

6.4 Resumo

O método para modelagem e geração dos testes que foi proposto nesta dissertação contém algumas contribuições e limitações em relação aos trabalhos apresentados no decorrer deste capítulo. Portanto, esta seção apresenta um resumo que contém algumas das características relevantes para geração de testes a partir de diagramas de atividades. Dentre as características, a Tabela 6.1 contém uma linha para cada abordagem apresentada, sendo a última linha para abordagem proposta aqui.

As colunas da tabela representam as características consideradas relevantes, nelas as opções Sim ou Não indicam se a abordagem de uma determinada linha trata ou não uma característica de modelagem e/ ou geração dos testes. Em relação à automação da geração dos testes, foram consideradas as opções de seleção de caminhos de teste e a geração dos modelos intermediários para obtenção dos testes. Estes dois aspectos foram considerados por serem fatores comuns a todas as abordagens estudadas, sendo que em

Tabela 6.1: Características de cada abordagem de teste.

Método de Teste	**Tipo de Teste**	**Concorrência**	**Fluxo de Dados**	**Exceções**	**Fluxo entre CDUs**	**Automação**
TOTEM	"caixa preta"	Sim	Não	Não	Sim	seleção dos caminhos
UMLTGF	"caixa cinza"	Sim	Não	Não	Não	modelos intermediários e seleção de caminhos
Bai *et al*	"caixa preta"	Sim	Sim	Não	Sim	seleção dos caminhos
AD2US	"caixa preta"	Sim	Não	Não	Não	modelos intermediários e seleção de caminhos
AGTCG	"caixa branca"	Sim	Não	Não	Não	seleção dos caminhos
Vieira *et al*	"caixa preta"	Não	Sim	Não	Sim	modelos intermediários e seleção de caminhos
Edwards	"caixa preta"	Não	Não	Não	Sim	seleção dos caminhos
Parrish	"caixa preta"	Não	Não	Não	Sim	seleção dos caminhos
Método Proposto	"caixa preta"	Não	Não	Sim	Sim	modelos intermediários e seleção de caminhos

todas as abordagens essas atividades são necessárias entretanto nem sempre são essas atividades são realizadas de forma automática. A automação é parcial nos casos em que é necessária a realização de algumas atividades intermediárias (manualmente) antes da geração dos testes. Por exemplo, a realização de um pré-processamento dos modelos para que seja possível gerar os testes. É importante ressaltar que a geração automática de testes considerada aqui não leva em consideração a geração dos modelos e dos dados de teste.

Capítulo 7

Conclusões e Trabalhos Futuros

Esta dissertação apresentou um método para geração automática de cenários de teste e a partir de modelos comportamentais, obtidos a partir dos casos de uso do SUT. O método proposto foi dividido em cinco etapas: (i) criação dos modelos de teste, (ii) criação do Grafo de Fluxo de Controle que mantém todos os modelos; (iii) seleção dos caminhos de teste; (iv) geração dos cenários de teste e; (v) criação dos casos de teste. A primeira e última etapa são realizadas de forma manual e as demais etapas são realizadas de forma automática.

Na etapa (i) devem ser criados os modelos utilizados na geração automática dos testes, para isso foram definidos dois níveis para criação dos modelos. No primeiro nível foi criado um modelo que representa o fluxo de controle de execução entre CDUs, que é chamado de DAFC (Diagrama de Fluxo de Controle entre Casos de Uso). E no segundo nível, para cada CDU da especificação do SUT foi criado um diagrama para representar os cenários internos do CDU, esse modelo foi chamado de DACI (Diagrama de Atividades com Cenários Internos do Caso de Uso). Após a criação dos modelos, as etapas (ii, iii e iv) são realizadas sem intervenção do testador, conforme foi apresentado no Capítulo 4.

Durante este mestrado foi implementada uma ferramenta para prova de conceito da geração dos cenários de teste, que faz parte da ferramenta Antares, que foi apresentada no Capítulo 5. A ferramenta Antares foi utilizada para a geração cenários de teste para os sistemas que estão sendo desenvolvidos no projeto Harpia e para o sistema de cadastro de deficientes [ACM07, Pat07], que foi utilizado como exemplo nessa dissertação e foi apresentado no Capítulo 3. A experiência obtida apresenta indícios que o método é viável para realização de testes para sistemas interativos que possuem especificação detalhada através de casos de uso UML.

Na realização do estudo de caso (apresentado no Capítulo 5), o tempo necessário para criação dos modelos em relação ao esforço realizado nos testes foi satisfatório e, de acordo com a avaliação realizada pelos testadores, as informações presentes nos modelos

foram úteis durante o projeto, obtenção dos dados e avaliação dos resultados dos testes. A utilização de modelos possibilitou também a antecipação das atividades de planejamento e projeto de testes e os cenários de teste facilitaram o entendimento do SUT pelos testadores envolvidos.

Durante os testes, grande parte dos cenários de teste gerados utilizando os algoritmos propostos no Capítulo 4 eram executáveis. Quando eles eram não executáveis, os testadores realizaram adaptações que possibilitaram a execução destes e, conseqüentemente, foi mantido assim o conjunto de cenários de teste que cobre toda a especificação do SUT.

Por outro lado, o refinamento necessário para criação dos casos de teste executáveis foi feito à mão pelos testadores, pois os cenários de teste foram detalhados em um alto nível de abstração. Assim, a automação da execução dos testes foi feita através de ferramentas de *capture-playback*. Na primeira vez que um caso de teste era executado o testador a ferramenta de *capture-playback* para gravar todas as interações realizadas durante os testes, tornado possível a re-execução automática do mesmo caso de teste.

7.1 Contribuições

A criação de modelos para especificação dos casos de uso, mantendo os relacionamentos entre os casos de uso, possibilitou a geração automática de cenários completos de teste. Isso foi feito através da representação de sub-atividades dos Diagramas de Atividades UML 2.0 [Gro04], que detalham os DACIs e o DAFC. A especificação dos cenários de exceção nos DAs possibilitou a geração de testes de forma diferenciada para cenários de exceção e cenários normais. A classificação para os cenários de exceção foi proposta por Ferreira, sendo que os dois tipos de cenários de exceção propostos por ela foram mapeados na especificação dos DAs: (*cenários recuperáveis* e *cenários de falha*). Além disso, o nível de detalhamento utilizado na especificação dos modelos não tornou a abordagem custosa, como ocorre em outras abordagens existentes, que foram apresentadas no Capítulo 6.

Uma contribuição importante do método de teste proposto foi a geração dos cenários de teste utilizando um modelo intermediário, o IACFG (em inglês, *Inter-activity Control Flow Graph*), que é criado de forma totalmente automática a partir dos modelos da especificação (os DACIs e o DAFC). Diferentemente de abordagens que propõem a seleção de testes para cada caso de uso isoladamente e (ou) de abordagens para geração (cenários) de testes utilizando modelos intermediários, mas que parte do processo de criação dos modelos intermediários deve ser feito de forma manual e que pode torná-las inviável para geração de testes em sistemas com uma especificação grande.

Outra contribuição foi a utilização de técnicas de teste para a definição do IACFG, onde foram interligados todos os modelos da especificação de maneira semelhante a abordagens de para realização de análise de dependência entre instrução de programas, pro-

posta por Harrold *et al*. O IACFG proposto para geração dos cenários de teste mantém somente uma cópia de cada modelo criado pelo testador (DACIs e DAFC), assim o tamanho desse modelo intermediário é linear em relação ao somatório dos tamanhos dos modelos da especificação de entrada.

A seleção de caminhos no IACFG foi realizada através de um algoritmo proposto nessa dissertação, que é uma variação do algoritmo de busca em profundidade, mas que mantém as restrições de contexto entre os modelos da especificação (DFS-CONTEXTO). A utilização do critério de cobertura de todas as arestas separando a cobertura de cenários de teste normais e de exceção tornou mais simples e claro os objetivos dos cenários de teste gerados, já que os cenários de exceção representam as situações críticas de funcionamento do SUT. As transformações no modelo e algoritmos para geração dos testes apresentados no Capítulo 4 foram automatizados através da ferramenta Antares.

Além das contribuições apresentadas acima, durante a realização desse trabalho foram realizadas três publicações, que são apresentadas a seguir:

- **Automação em Projeto de Testes Usando Modelos UML**[1]. Apresentado no *"Brazilian Workshop on Systematic and Automated Software Testing"* em 2007 [PM07]. Este artigo apresenta de forma resumida o método de testes proposto nesta dissertação. No artigo é apresentado um exemplo de especificação utilizando os Diagramas de Atividades, o IACFG e um caso de teste gerado. Também são mostrados resultados preliminares obtidos na realização de testes em outro sistema, mas que também desenvolvido no projeto Harpia.

- **Uso de Modelos da UML em Testes de Componentes**[2]. Apresentado no *"Workshop de Testes e Tolerância a Falhas"* em 2007 [PMV07]. Neste artigo o método de teste proposto foi utilizado para geração de testes a partir da especificação comportamental de componentes, que foi proposta por C. R. Rocha [Roc05]. Os casos de teste gerados são no formato de entrada suportado por uma ferramenta de execução de testes de componentes chamada *CBDUnit* [GAM05]. Entretanto, assim como nessa dissertação, esse artigo não abordou a geração de dados de teste.

- **Um processo para testes de sistemas com reuso de componentes**[3]. Relatório técnico (IC-06-21) do Instituto de Computação da Unicamp concluído em 2006 [PRM+06]. Neste relatório foi definido um processo de testes para Desenvolvimento Baseado em Componentes (DBC), que é integrado a um processo de DBC genérico que foi proposto por Brito *et al* [BBdCGR05]. Este processo apresenta

[1]Este trabalho foi financiado pelo projeto Harpia.
[2]Este trabalho foi financiado pelo projeto Harpia e pelo Projeto CompGov, FINEP, no. 1843/04.
[3]Projeto CompGov, FINEP, no. 1843/04.

de forma detalhada as atividades necessárias nas seguintes fases de teste: unidades, componentes, integração e sistemas. Este processo foi definido como parte do projeto CompGov.

7.2 Trabalhos Futuros

Em relação à criação dos modelos, podem ser estudadas algumas formas de integração da modelagem proposta nesta dissertação com outras abordagens para modelagem a partir da especificação do SUT, tais como os diagramas de seqüência e de estado. Desta forma, o testador poderá escolher o melhor modelo para detalhar cada parte do SUT no projeto dos testes do SUT, utilizando o mesmo método para geração dos testes. Pode-se pesquisar também soluções para representação de fluxo de dados nos modelos, mas com o menor impacto possível, para não tornar custosa as atividades de modelagem.

Uma abordagem de pesquisa que pode ser realizada futuramente é a criação de "modelos executáveis" a partir dos modelos propostos. Os "modelos executáveis" considerados aqui, são modelos para simular a execução dos fluxos especificados utilizando um conjunto de dados de entrada. A partir dessa execução deve-se observar qual fluxo foi executado no modelo e verificar se os dados utilizados violaram ou não as pré-condições e pós-condições no caminho executado. A transformação dos modelos em modelos executáveis possibilita a integração do método proposto nesta dissertação com abordagens que de geração automática de dados de teste. Como, por exemplo, a abordagem proposta por Abreu [Abr06].

Em relação à geração automática de testes, uma abordagem interessante é incorporar ao método proposto a geração de testes para sistemas distribuídos e sistemas concorrentes. Para que isso seja feito, a abordagem deve possibilitar a representação de ações concorrentes no IACFG, assim como em outras abordagens [LJX+04, BLL04a]. A geração de um número reduzido de combinações de possíveis testes, mas com grande potencial para revelar defeitos em sistemas concorrentes ainda é uma área de pesquisa pouco explorada.

Outro trabalho futuro interessante, é a aplicação de métodos para priorização de cenários de teste, como a proposta por Bai *et al* [BLL04b] e a identificação de testes obsoletos e re-testáveis na presença de alterações na especificação, aplicando técnicas de seleção de testes para regressão [OSH04, RH93]. Além disso, a realização de estudos de caso em sistemas de diferentes domínios de aplicação como, por exemplo, sistemas embarcados em dispositivos móveis. Outra atividade futura interessante é realização de um experimento controlado para comparação desta abordagem com outras abordagens de teste como, por exemplo, utilizar duas equipes diferentes realizando testes no mesmo sistema, mas uma equipe deve realizar os testes manualmente e a outra equipe realizar os testes utilizando a abordagem de testes proposta.

Em relação ao protótipo da ferramenta Antares, é interessante avaliar os métodos ou as ferramentas existentes para verificação e validação de modelos UML e, caso seja viável, pode-se incorporar ou implementar um método para validação de modelos na Antares. Um outro trabalho futuro interessante seria a criação de uma interface amigável para possibilitar a disponibilização dessa ferramenta para ser utilizada por terceiros.

Apêndice A

Questionário para Avaliação do Método de Teste

Objetivo

Avaliar o método de geração de cenários de teste proposto nesta dissertação. Os participantes do projeto Harpia que realizaram teste utilizando o método proposto responderam este questionário. Neste questionário as questões foram agrupadas em 4 assuntos diferentes: experiência do testador, modelos de teste utilizados, avaliação dos cenários de teste e sugestões para melhoria do método.

Glossário com Termos Utilizados no Questionário

- **Modelos de Teste**. Diagramas de Atividades contendo fluxo de negócio do sistema, detalhamento dos casos de uso e regras de negócio do sistema durante o projeto dos testes.

- **Cenário de Teste**. Um cenário de teste descreve de forma textual as interações dos atores com o sistema em testes. Os cenários possuem pré e pós-condições e o conjunto de passos (corpo) para realização de uma determinada atividade no sistema.

- **Passos**. Os passos são descritos no corpo do cenário de teste com interações dos atores, sujeito (Sistema em Teste) ou situações de exceção no sistema.

- **Cenário Incoerente**. É um cenário que não pode ser executado no sistema ("Não Executável") pois apresenta passos contraditórios.

- **Casos de Teste**. Para cada cenário de teste foram documentados (e associados aos cenários) os casos de teste e seus dados e consultas[1] (oráculos) para verificação dos resultados.

A.1 Experiência do testador

Opções de resposta para as questões 1 até a questão 11 são: **"SIM"**, **"NÃO"** e **"NÃO APLICÁVEL"**.

a) Antes de realizar testes utilizando os cenários gerados, qual era sua experiência com teste?

1) Trabalhei em vários projetos de teste com características diferentes.

2) Trabalhei em vários projetos de teste com características similares.

3) Trabalhei em poucos projetos de teste.

4) Tinha conhecimento teórico em testes, nunca havia realizado testes em projetos reais.

5) Nenhum conhecimento.

b) Como executou os testes?

6) Utilizando ferramentas para automação/ execução de testes.

7) Codificando testes com frameworks de teste (Junit, JWebUnit, HttpUnit, etc).

8) Utilizando ferramentas de capture/ playback de ações do usuário.

9) Manualmente.

[1] As consultas foram utilizadas na avaliação dos oráculos de teste.

c) Caso já tenha preparado e realizado testes com uma abordagem manual, qual é sua avaliação em relação a abordagem com geração de cenários de teste?

10) Os cenários de teste facilitaram o entendimento do sistema.

11) Os cenários de teste facilitaram a criação dos casos de teste.

Comentários

A.2 Criação dos modelos para realização dos teste

As questões abaixo (questão 12 até a questão 47) devem ser respondidas com as seguintes opções de resposta: **"SEMPRE"**, **"QUASE SEMPRE"**, **"ALGUMAS VEZES (MÉDIA)"**, **"QUASE NUNCA"**, **"NUNCA"** e **"NÃO APLICÁVEL"**.

d) Os modelos criados durante os testes facilitaram o entendimento do sistema?

12) Houve dificuldades na criação dos modelos de teste a partir da especificação.

13) Os modelos criados durante os testes apresentavam as informações para o projeto dos testes.

14) Os modelos criados durante os testes continham muitas informações desnecessárias.

15) Os modelos não proporcionaram nenhum beneficio adicional em relação aos casos de uso.

16) Os modelos estavam incompletos, foi necessário recorrer aos casos de uso durante o projeto dos testes.

17) Os modelos dificultaram o projeto de testes, eles apresentavam muitas inconsistências em relação a especificação.

e) Quando haviam alterações na especificação como foi realizado o re-projeto dos modelos?

18) Foi possível aplicar as alterações da especificação nos modelos criados anteriormente.

19) Os modelos desatualizados foram desconsiderados e foram criados novos modelos.

20) Os modelos foram mantidos sem aplicar as alterações.

21) Os modelos de teste foram abandonados (não foram mais utilizados e nem foram criados novos modelos).

Comentários

A.3 Avaliação dos cenários de teste gerados

f) Os cenários de teste facilitaram os testes?

22) Os cenários apresentavam os passos necessários para realizar testes.

23) Durante a realização dos testes foi necessário recorrer aos modelos de teste (para o melhor entendimento do sistema).

24) Durante a realização dos testes foi necessário recorrer a outros documentos da especificação (tais como, casos de uso e regras de negócio do sistema).

25) Os cenários de teste não foram utilizados, os casos de teste foram criados sem utilizá-los.

g) Qual a sua opinião sobre o nível de detalhamento (abstração) dos cenários de teste?

26) Os cenários de teste criados apresentavam os detalhes necessários para realização dos testes.

27) Os cenários de teste criados apresentavam poucos detalhes para criação dos testes.

28) Os cenários de testes apresentavam muitas informações desnecessárias durante os testes.

29) Os cenários de teste foram relevantes durante a obtenção dos dados de teste.

h) Qual é o impacto nos testes quando os cenários de teste apresentavam incoerências? (passos inválidos ou passos fora de ordem na descrição dos cenários de teste)

30) As incoerências nos cenários de teste dificultaram a criação de casos de teste.

31) As incoerências nos cenários de teste não atrapalharam a criação dos casos de teste.

32) As incoerências dificultaram o entendimento do cenário de teste.

33) Foram realizadas adaptações nos cenários de teste para criar os casos de teste.

34) Os cenários de teste com incoerências revelaram a presença de defeitos inválidos.

35) Os cenários de teste com incoerências foram desconsiderados.

i) Foi possível identificar defeitos utilizando as informações descritas nos passos do cenário de teste?

36) Os passos do cenário de teste facilitaram a identificação de defeitos no Sistema em Teste.

37) Foi necessário consultar outros documentos da especificação para avaliar se as situações observadas eram defeitos.

38) Foram identificados defeitos inválidos devido a existência de incoerências nos cenários de teste.

39) Foi necessário consultar os modelos de testes para avaliar se as situações observa-

das apresentavam defeitos.

Comentários

A.4 Sugestões para melhoria do método

As questões abaixo devem ser respondidas com as seguintes opções de resposta: **"CONCORDO FORTEMENTE"**, **"CONCORDO"**, **"NEUTRO"**, **"DISCORDO"**, **"DISCORDO FORTEMENTE"** e **"NÃO APLICÁVEL"**.

j) As situações apresentadas dificultam a criação dos casos de teste a partir dos cenários de teste propostos?

40) A presença de passos inválidos.

41) A presença de passos fora de ordem.

42) Descrição pouco detalhada do cenário.

43) Descrição muito detalhada do cenário.

k) As seguintes melhorias facilitariam a utilização da abordagem de testes baseados nos cenários propostos?

44) Tornar a descrição dos cenários mais completa (tais como, incluir mais informações nos cenários de teste).

45) Tornar a descrição dos cenários mais simplificada (tais como, identificar e remover passos desnecessários nos cenários de teste).

46) Criar templates de casos de teste para cada cenário com interfaces e campos.

47) Utilizar uma abordagem para regressão de teste nos modelos e cenários gerados.

Apêndice B

Respostas Obtidas na Aplicação do Questionário

Neste apêndice são apresentados os gráficos com os valores totais e percentuais de respostas obtidos na aplicação do questionário apresentado no Apêndice A. O questionário foi respondido por sete testadores participantes projeto Harpia. As questões que totalizam menos de sete resposta no somatório das respostas obtidas foram respondidas por um ou mais testadores com a opção "Não aplicável".

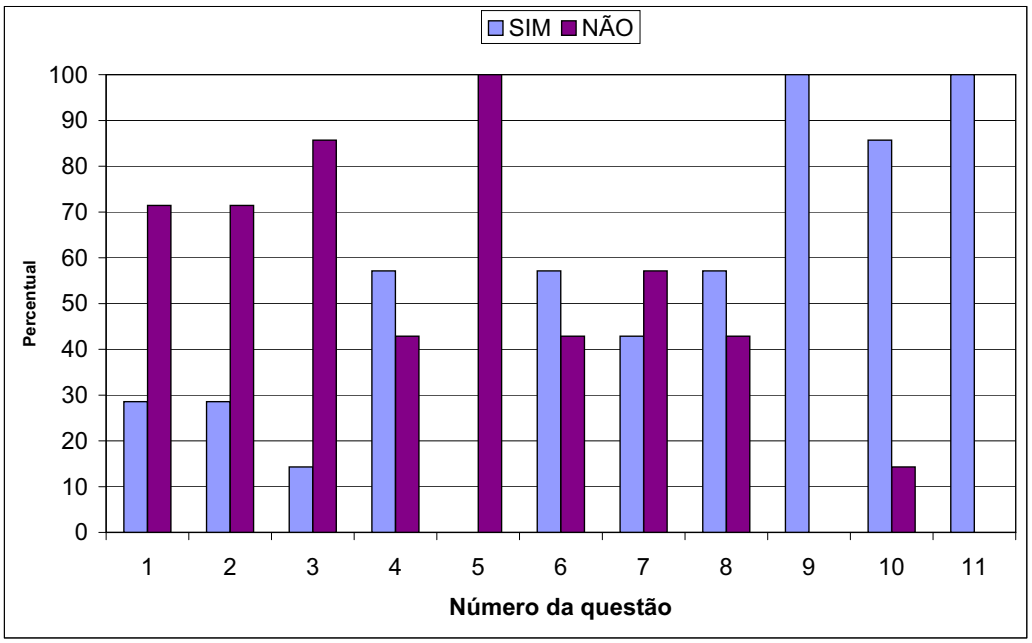

Figura B.1: Distribuição percentual das respostas para as questões 1 a 11 (Experiência do testador).

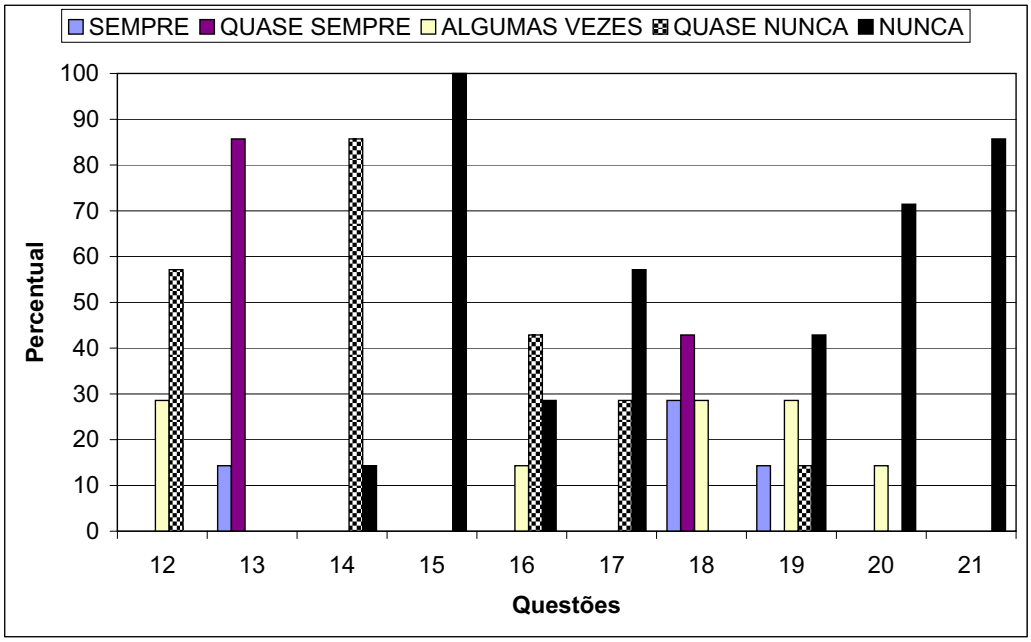

Figura B.2: Distribuição percentual das respostas para as questões 12 a 21 (Criação dos modelos para realização dos teste).

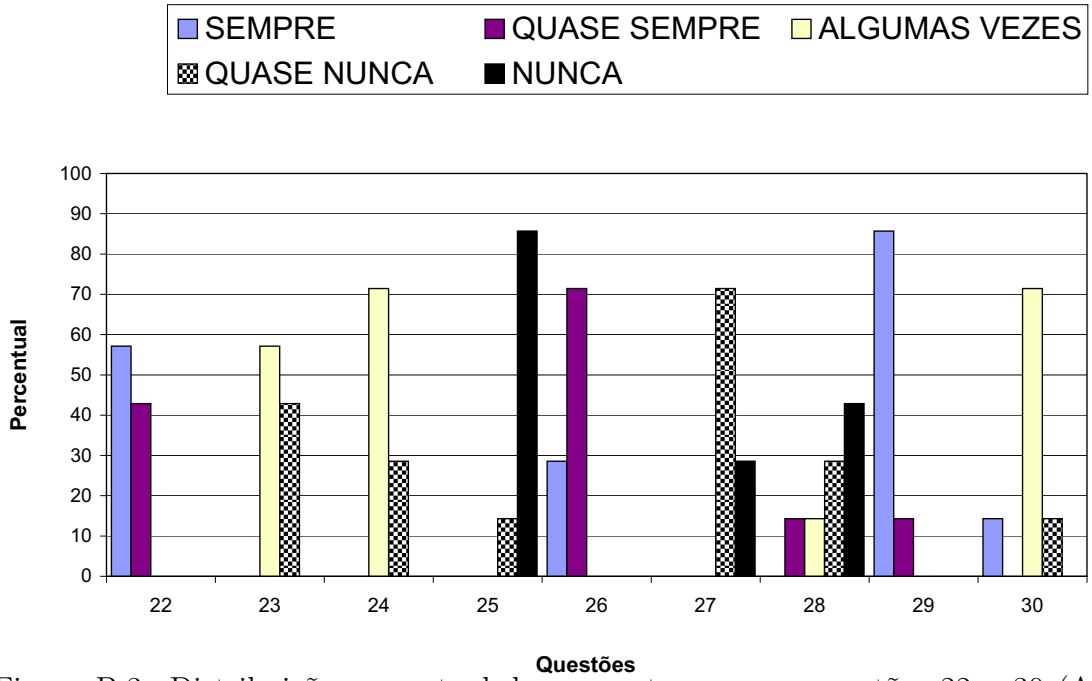

Figura B.3: Distribuição percentual das respostas para as questões 22 a 30 (Avaliação dos cenários de teste gerados).

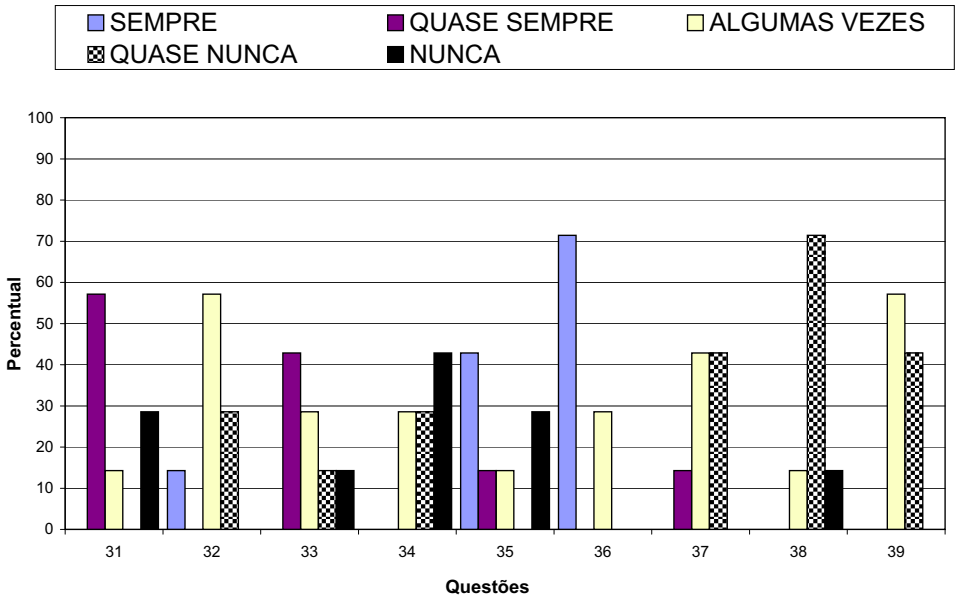

Figura B.4: Distribuição percentual das respostas para as questões 31 a 39 (Avaliação dos cenários de teste gerados).

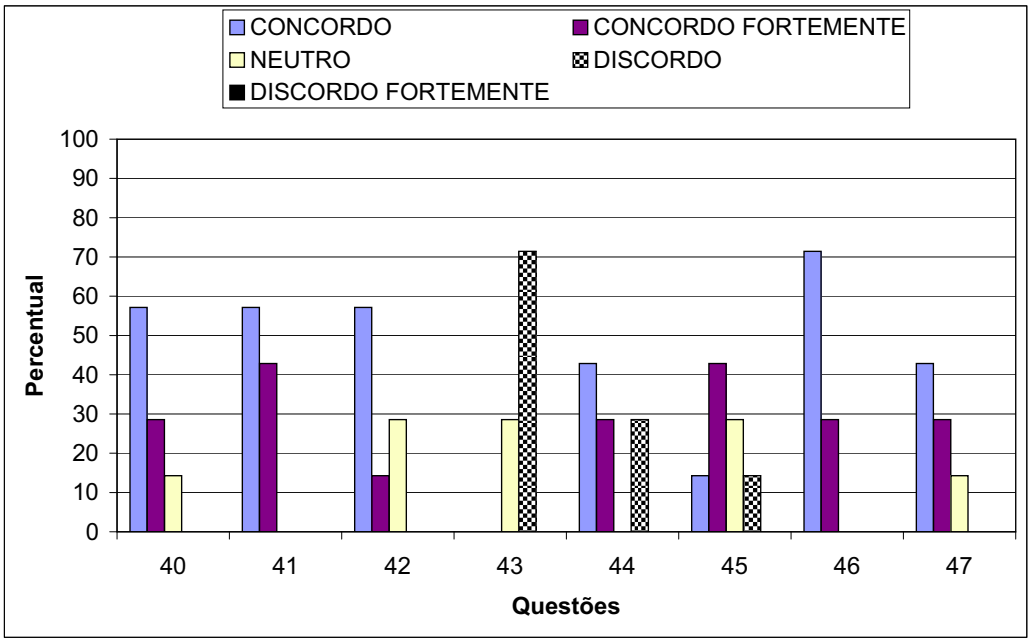

Figura B.5: Distribuição percentual das respostas para as questões 40 a 47 (Sugestões para melhoria do método).

Referências Bibliográficas

[Abr06] B. T. Abreu. Uma abordagem evolutiva para geração automática de dados de teste. Master's thesis, Instituto de Computação – Unicamp, Agosto 2006.

[ACM07] E. Avelar, F. Cruz, and M. Maranha. Desenvolvimento de um software web com foco na inclusão de portadores de deficiências no mercado de trabalho. Trabalho de Graduação Interdisciplinar, Novembro 2007.

[AD97] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, May 1997.

[AG06] Gentleware AG. Poseidon for uml. http://www.gentleware.com, 2006.

[BBdCGR05] Patrick Henrique Da Silva Brito, Maria Antonia Martins Barbosa, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira. Um processo para o desenvolvimento baseado em componentes com reuso de componentes. Technical Report IC-05-22, September 2005.

[BBM02] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow_suite approach to planning and deriving test suites in uml projects. In *UML*, pages 383–397, 2002.

[Bei90] Boris Beizer. *Sofware Testing Techniques*. International Thomson Computer Press, 1 edition, 1990.

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

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[BHB86] A. L. Baker, J. W. Howatt, and J. M. Bieman. Criteria for finite sets of paths that characterize control flow. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences (HICSS-19)*, pages 158–163, 1986.

[Bin94] Robert V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, 1994.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[BL02] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. *Software and System Modeling*, 1(1):10–42, 2002.

[BLL04a] Xiaoqing Bai, Chiou Peng Lam, and Huaizhong Li. An approach to generate the thin-threads from the uml diagrams. In *COMPSAC*, pages 546–552, 2004.

[BLL04b] Xiaoqing Bai, Huaizhong Li, and Chiou Peng Lam. A risk analysis approach to prioritize uml-based software testing. In *SNPD*, pages 112–119, 2004.

[CCD03] Alessandra Cavarra, Charles Crichton, and Jim Davies. A method for the automatic generation of test suites from object models. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1104–1109, New York, NY, USA, 2003. ACM.

[CL01] L. Constantine and L. Lockwood. Structure and style in use cases for user interface design, 2001.

[CLL05] Robert Chandler, Chiou Peng Lam, and Huaizhong Li. Ad2us: An automated approach to generating usage scenarios from uml activity diagrams. In *APSEC*, pages 9–16, 2005.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMPV05] Ana R. Cavalli, Stéphane Maag, Sofia Papagiannaki, and Georgios Verigakis. From uml models to automatic generated tests for the dotlrn e-learning platform. *Electr. Notes Theor. Comput. Sci.*, 116:133–144, 2005.

[Coc01a] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

[Coc01b] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[Con99] Jim Conallen. Modeling web application architectures with uml. *Commun. ACM*, 42(10):63–70, 1999.

[Cor07] IBM International Business Machines Corp. Rational clearquest. http://www-306.ibm.com/software/awdtools/clearquest/, 2007.

[CP02] Ram Chillarege and Kothanda Ram Prasad. Test and development process retrospective - a case study using odc triggers. In *DSN*, pages 669–678, 2002.

[DLS78] R.A. DeMillo, R. J. Lipton, and F.G Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[dMF01] Gisele Rodrigues de Mesquita Ferreira. Tratamento de exceção no desenvolvimento de sistemas confiáveis baseados em componentes. Master's thesis, Instituto de Computação – Unicamp, Dezembro 2001.

[dPPF01] Wilson de Pádua Paula Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. Livros Técnicos e Científicos, 1ª edition, 2001.

[DTGF06] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test uml design models. In *ISSRE*, pages 95–104, 2006.

[Edw00] Stephen H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Softw. Test., Verif. Reliab.*, 10(4):249–262, 2000.

[Fre91] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[GAM05] Paulo Guerra, Carolina Araújo, and Camila C. Rocha; Eliane Martins. Cbdunit - uma ferramenta para testes unitários de componentes. In *Proc. of the 19th Brazilian Symp. on Software Engineering*, volume 19, Uberlândia, Brazil, 2005.

[Gel04] David Gelperin. Precise use cases. http:// www.livespecs.com/downloads/ ClearSpecs06V05_PreciseUseCases.pdf, 2004. Methods & Tools Spring.

[GGGS02] Jerry Z. Gao, Kamal K. Gupta, Shalini Gupta, and Simon S. Y. Shim. On building testable software components. In *Proc. First International*

Conference on COTS-Based Software Systems, pages 108–121, Fevereiro 2002.

[Gro03a] Object Management Group. *MDA Guide Version 1.0.1*, 2003.

[Gro03b] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5*, 2003.

[Gro03c] Object Management Group. *XML Metadata Interchange (XMI) Specification*, 2003.

[Gro04] Object Management Group. *UML 2.0 Superstructure Final Adopted Specification*, 2004.

[Gro05] Object Management Group. *UML Testing Profile*, 2005.

[Hew07] Joe Hewitt. Firebug - web development evolved. http://getfirebug.com/, 2007.

[HH85] David Hedley and Michael A. Hennell. The causes and effects of infeasible paths in computer programs. In *ICSE*, pages 259–267, 1985.

[HN04] A. Hartman and K. Nagin. The agedis tools for model based testing. In *ISSTA*, pages 129–132, 2004.

[HRS98] Mary Jean Harrold, Gregg Rothermel, and Saurabh Sinha. Computation of interprocedural control dependence. In *ISSTA*, pages 11–20, 1998.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[HVFR04] Jean Hartmann, Marlon Vieira, Herb Foster, and Axel Ruder. Prasentation auf der TAV21, 2004.

[IEE90] IEEE. Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990. Technical report, IEEE Computer Society Press, 1990.

[Jud07] Adam Judson. Tamper data mozdev.org. http://tamperdata.mozdev.org/, 2007.

[Kan03] Cem Kaner. An introduction to scenario testing. http://www.kaner.com /pdfs/ ScenarioIntroVer4.pdf, 2003.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KP02a] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 4: questionnaire evaluation. *SIGSOFT Softw. Eng. Notes*, 27(3):20–23, 2002.

[KP02b] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 5: populations and samples. *SIGSOFT Softw. Eng. Notes*, 27(5):17–20, 2002.

[KP02c] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *SIGSOFT Softw. Eng. Notes*, 27(1):18–20, 2002.

[KP02d] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research: part 3: constructing a survey instrument. *SIGSOFT Softw. Eng. Notes*, 27(2):20–24, 2002.

[KP03] Barbara Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 6: data analysis. *SIGSOFT Softw. Eng. Notes*, 28(2):24–27, 2003.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

[KR03] Vinay Kulkarni and Sreedhar Reddy. Separation of concerns in model-driven development. *IEEE Software*, 20(5):64–69, 2003.

[KSD+03] P. Koopman, D. Siewiorek, K. DeVale, J. DeVale, K. Fernsler, D. Guttendorf, N. Kropp, J. Pan, C. Shelton, and Y. Shi. Cots software robustness testing. http://www.ece.cmu.edu/ koopman/ballista, 2003.

[KSW01] G. Kösters, H. Six, and M. Winter. Coupling use cases and class models as a means for validation and verification of requirements specifications, 2001.

[Lai07] Oliver Laitenberger. A survey of software inspection technologies, 2007.

[LJX+04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 284–291, Washington, DC, USA, 2004. IEEE Computer Society.

[LL87] Julius C.B Leite and Orlando G. Loques. Introdução à tolerância a falhas. In *Mini-curso apresentado durante o II Simpósio de Software Tolerante a Falhas (SCTF)*, 1987.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 235–248, New York, NY, 1992. ACM Press.

[LS06] Mass Soldal Lund and Ketil Stolen. Deriving tests from uml 2.0 sequence diagrams with neg and assert. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 22–28, New York, NY, USA, 2006. ACM.

[MAMS06] P. V. R. Murthy, P. C. Anitha, M. Mahesh, and Rajesh Subramanyan. Test ready uml statechart models. In *SCESM*, pages 75–82, 2006.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest editors' introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.

[Mes04] G. Meszaros. Pattern language for automated testing of indirect inputs and outputs using xunit. In *The 11th Conference on Pattern Languages of Programs*, 2004.

[Mic06] Sun Microsystems. The source for java developers. http://java.sun.com, 2006.

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for uml activity diagrams. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 2–8, New York, NY, USA, 2006. ACM.

[Mye79] Glenford Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[OA99] A. Jefferson Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *UML*, pages 416–429, 1999.

[Ope06] OpenQA. Openqa: Selenium. http://www.openqa.org/selenium/, 2006.

[Org07] The Mozilla Organization. Bugzilla. http://www.bugzilla.org/, 2007.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of theACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, Newport Beach, CA, November 2004.

[Pai78] M. R. Paige. An analytic approach to software testing. In *Computer Software and Applications Conference*, pages 527–532, 1978.

[PAK+07] Orest Pilskalns, Anneliese Amschler Andrews, Andrew Knight, Sudipto Ghosh, and Robert B. France. Testing uml designs. *Information & Software Technology*, 49(8):892–912, 2007.

[Pat07] Gabriela O. Patuci. Aplicação de uma metodologia para testes sistêmicos. Trabalho de Final de Curso - Estágio Supervisionado, Novembro 2007.

[PBC93] Allen S. Parrish, Richard B. Borie, and David W. Cordes. Automated flow graph-based testing of object-oriented software modules. *Journal of Systems and Software*, 23(2):95–109, 1993.

[Ped07] Chris Pederick. Web developer. http://getfirebug.com/, 2007.

[Pet01] Alexandre Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *MOVEP '00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 196–205, London, UK, 2001. Springer-Verlag.

[PK01] Shari Lawrence Pfleeger and Barbara A. Kitchenham. Principles of survey research: part 1: turning lemons into lemonade. *SIGSOFT Softw. Eng. Notes*, 26(6):16–18, 2001.

[PM07] Ivan R. D. C. Perez and Eliane Martins. Automação em projeto de testes usando modelos uml. In *1st Brazilian Workshop on Systematic and Automated Software Testing, junto ao XXI Simpósio de Engenharia de Software*, 2007.

[PMV07] Ivan R. D. C. Perez, Eliane Martins, and Júlio E. Viégas. Uso de modelos da uml em testes de componentes. In *VIII Workshop de Teste e Tolerância a Falhas, junto ao XXV Simpósio de Redes de Computadores e Sistemas Distribuídos*, 2007.

[Pre97] Roger S. Pressman. *Software Engineering.* McGraw-Hill, 5 edition, 1997.

[Pre01] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 5ª edition, 2001.

[PRM+06] Ivan Rodolfo Duran Cruz Perez, Camila Rocha, Regina Moraes, Josiane Aparecida Cardoso, Ruth Fabiana Soliani, and Eliane Martins. Um processo para testes de sistemas com reuso de componentes. Technical Report IC-06-21, October 2006.

[Ray87] D. Rayner. Osi conformance testing. *Computer Networks*, 14:79–98, 1987.

[RH93] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of theInternational Conference on Software Maintenance (ICSM 1993)*, pages 358–361, Montreal, Quebec, Canada, September 1993.

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting regression tests for object-oriented software. In *ICSM*, pages 14–25, 1994.

[Rob99] Harry Robinson. Graph theory techniques in model-based testing. http:// www.chillarege.com/ fastabstracts/issre99/99119.pdf, 1999.

[Roc05] Camila Ribeiro Rocha. Um método de testes para componentes tolerantes a falhas. Master's thesis, Instituto de Computação – Unicamp, Dezembro 2005.

[Rum94] James E. Rumbaugh. Getting started: Using use cases to capture requirements. *JOOP*, 7(5):8–12, 23, 1994.

[Run06] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[SHR01] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10(2):209–254, 2001.

[Som95] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6ª edition, 1995.

[SW94] R. M. Sitaraman and B. W. Weide. Component-based software engineering using resolve. *ACM SIGSOFT Software Enigineering Notes*, 19(4):21–67, 1994.

[Sys07] Atlassian Software Systems. Clover - test with confidence. http://www.atlassian.com/software/clover/, 2007.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[VLH+06] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of gui testing using a model-driven approach. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14, New York, NY, USA, 2006. ACM.

[Wey86] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Software Eng.*, 12(12):1128–1138, 1986.

[Wey98] Elaine J. Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

[Whi00] James A. Whittaker. What is software testing? and why is it so hard? *IEEE Softw.*, 17(1):70–79, 2000.

[YM89] Derek Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *Symposium on Testing, Analysis, and Verification*, pages 48–54, 1989.

[ZHK92] Stuart H. Zweben, Wayne D. Heym, and Jon Kimmich. Systematic testing of data abstractions based on software specifications. *Softw. Test., Verif. Reliab.*, 1(4):39–55, 1992.

[Zhu95] Hong Zhu. Axiomatic assessment of control flow-based software test adequacy criteria. *Software Engineering Journal*, 10(5):194–204, 1995.

[Zhu96] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Trans. Softw. Eng.*, 22(4):248–255, 1996.