**Ambiente integrado para verificação e teste
da coordenação de componentes
tolerantes a falhas**

Simone Hanazumi

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Programa: Ciência da Computação
Orientadora: Profa. Dra. Ana Cristina Vieira de Melo

Durante o desenvolvimento deste trabalho a autora recebeu auxílio financeiro da FAPESP
Processo: 2008/03235-0

São Paulo, setembro de 2010

**Ambiente integrado para verificação e teste
da coordenação de componentes
tolerantes a falhas**

Esta versão definitiva da dissertação
contém as correções e alterações sugeridas pela
Comissão Julgadora durante a defesa realizada
por Simone Hanazumi em 01/09/2010.

Comissão Julgadora:

- Profa. Dra. Ana Cristina Vieira de Melo (orientadora) - IME-USP
- Prof. Dr. Flávio Soares Corrêa da Silva - IME-USP
- Prof. Dr. Mario Jino - FEEC-UNICAMP

Agradecimentos

A **Deus**, por ter me dado forças para concretizar a realização deste trabalho.

Ao **Instituto de Matemática e Estatística da Universidade de São Paulo e seus professores**, por prover o ambiente para o desenvolvimento deste trabalho e o conhecimento necessário para sua realização.

À **Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)**, pelo financiamento da pesquisa por meio de bolsa¹.

À **Profa. Dra. Ana Cristina Vieira de Melo**, pela sua paciência, amizade, confiança e, sobretudo, pelas valiosas orientações feitas para o andamento e conclusão deste trabalho.

Aos **meus pais e minha irmã**, pelo apoio e pela confiança no sucesso deste trabalho, em todas as horas.

Aos **meus amigos e colegas**, em especial a **David Paulo Pereira, Kleber da Silva Xavier e Paulo Roberto A. F. Nunes**, por suas colaborações, críticas e sugestões que auxiliaram no desenvolvimento deste trabalho.

A **todos** aqueles que me apoiaram e acreditaram no meu sucesso.

Muito obrigada.

¹Processo: 2008/03235-0

Resumo

Hoje, diante das contínuas mudanças e do mercado competitivo, as empresas e organizações têm sempre a necessidade de adaptar suas práticas de negócios para atender às diferentes exigências de seus clientes e manter-se em vantagem com relação às suas concorrentes. Para ajudá-las a atingir esta meta, uma proposta promissora é o Desenvolvimento Baseado em Componentes (DBC), cuja ideia básica é a de que um novo software possa ser construído rapidamente a partir de componentes pré-existentes. Entretanto, a montagem de sistemas corporativos mais confiáveis e tolerantes a falhas a partir da integração de componentes tem-se mostrado uma tarefa relativamente complexa. E a necessidade de garantir que tal integração não falhe tornou-se algo imprescindível, sobretudo porque as consequências de uma falha podem ser extremamente graves.

Para que haja uma certa garantia de que o software seja tolerante a falhas, devem ser realizadas atividades de testes e verificação formal de programas. Isto porque ambas, em conjunto, procuram garantir ao desenvolvedor que o sistema resultante da integração é, de fato, confiável. Mas a viabilidade prática de execução destas atividades depende de ferramentas que auxiliem sua realização, uma vez que a execução de ambas constitui um alto custo para o desenvolvimento do software.

Tendo em vista esta necessidade de facilitar a realização de testes e verificação nos sistemas baseados em componentes (DBC), este trabalho de Mestrado se propõe a desenvolver um ambiente integrado para a verificação e teste de protocolos para a coordenação do comportamento excepcional de componentes.

Palavras-chave: Testes, Verificação Formal, Coordenação de Componentes.

Abstract

Nowadays, because of continuous changes and the competitive market, companies and organizations have the necessity to adapt their business practices in order to satisfy the different requirements of their customers and then, keep themselves in advantage among their competitors. To help them to reach this aim, a promising purpose is the Component-Based Development (CBD), whose basic idea is that a new software can be built in a fast way from preexisting components. However, mounting more reliable and fault-tolerant corporative systems from components integration is a relatively complex task. And the need to assure that such integration does not fail becomes something essential, especially because the consequences of a failure can be extremely serious.

To have a certain guarantee that the software will be fault-tolerant, testing activities and formal verification of programs should be done. This is because both, together, try to assure to developer that the resulting system of the integration is, in fact, reliable. But the practical feasibility of executing these activities depends on tools which support it, once both executions have a high cost to software development.

Having the necessity to make test and verification easier in systems based in components (CBD), this work has, as main objective, the development of an integrated environment for verification and test of protocols to the coordination of components' exceptional behaviour.

Keywords: Tests, Formal Verification, Components Orchestration.

Sumário

Lista de Figuras	xi
Lista de Tabelas	xiii
1 Introdução	1
1.1 Motivação	1
1.1.1 Componentes e Desenvolvimento Baseado em Componentes (DBC)	1
1.1.2 Testes e Verificação Formal	1
1.2 Objetivos	3
1.3 Organização dos Capítulos	3
2 Teste de Software	5
2.1 Termos Utilizados	5
2.2 Conceito e Importância dos Testes	6
2.3 Objetivos e Limitações	6
2.4 Etapas do Processo	7
2.5 Fases e Estratégias de Teste	8
2.5.1 Testes Unitários	8
2.5.2 Testes de Integração	8
2.5.3 Testes de Sistemas	10
2.6 <i>Design</i> de Casos de Testes	11
2.6.1 Testes <i>Black-Box</i>	11
2.6.2 Testes <i>White-Box</i>	12
2.7 Critérios de Testes para Comportamento Normal de Programas	14
2.7.1 Critérios de Fluxo de Objetos	14
2.7.2 Exemplo	15
2.8 Critérios de Teste para o Comportamento Excepcional de Programas	17
2.8.1 Estrutura de exceções em Java	17
2.8.2 Termos utilizados	18
2.8.3 Definição e uso de exceções	19
2.8.4 Ativação e desativação de exceções	19
2.8.5 Critérios de Teste	20
2.8.6 Grafo de Fluxo de Controle para Mecanismos de Tratamento de Exceção	20
2.8.7 Exemplo	20
2.9 Ferramentas para Testes	21

2.9.1	Ferramentas para Cobertura de Código	22
2.9.2	Ferramentas para Geração de Requisitos de Teste	23
2.9.3	Justificativa da Escolha da OConGraX	24
2.10	Conclusão	25
3	Verificação de Programas	27
3.1	Conceito de Verificação de Programas	27
3.1.1	Métodos Formais	27
3.1.2	Definição de Verificação	28
3.1.3	Limitações	28
3.2	Etapas para a Verificação de Programas	28
3.3	Abordagens para a Verificação de Programas	29
3.3.1	Sistema de Prova de Programas	29
3.3.2	Verificação de Modelos (<i>Model Checking</i>)	31
3.4	Ferramentas para Verificação	33
3.4.1	Ferramentas para <i>Model Checking</i>	33
3.4.2	Justificativa da Escolha do JPF/JPF Exceptions	34
3.5	Conclusão	35
4	Componentes e Ações Atômicas Coordenadas	37
4.1	Desenvolvimento Baseado em Componentes (DBC)	37
4.1.1	Componentes	37
4.1.2	Padrões de Componentes	38
4.2	Sistemas Tolerantes a Falhas	38
4.2.1	Tratamento de Exceções no Componente	38
4.2.2	Tratamento de Exceções na Arquitetura	39
4.3	Ações Atômicas Coordenadas (Ações CA)	39
4.3.1	Aninhamento e Composição	40
4.3.2	Tratamento de Exceções	41
4.4	Uma Implementação das Ações Atômicas Coordenadas	41
4.4.1	Arquitetura do Modelo	42
4.4.2	CoordinatedAtomicAction	43
4.4.3	ExceptionTree	44
4.4.4	Participant	44
4.4.5	Role	47
4.4.6	Handler	47
4.4.7	Transaction e External Object	48
4.4.8	Coordinator	49
4.4.9	Exemplo de Uso	55
4.5	Conclusão	60
5	Propriedades sobre Coordenação de Componentes	61
5.1	Propriedades sobre Ações Atômicas Coordenadas	61
5.1.1	Lógica Temporal	61

5.1.2	Alloy	63
5.1.3	CSP	64
5.2	Implementação das Propriedades	65
5.2.1	Programação de Propriedades Customizadas no JPF	65
5.2.2	Escolha das Propriedades	67
5.2.3	Limitações do JPF	67
5.3	Propriedades Implementadas	68
5.3.1	Propriedade 1: Desfazimento de Transações	69
5.3.2	Propriedade 2: Tratamento das Exceções	72
5.4	Exemplo de Uso	75
5.4.1	Propriedade 1: Desfazimento de Transações	76
5.4.2	Propriedade 2: Tratamento das Exceções	77
5.5	Conclusão	78
6	Integração de Teste e Verificação	79
6.1	Validação e Verificação (V&V)	79
6.1.1	Abordagem Teste + Verificação	80
6.2	OConGraX	81
6.2.1	Arquitetura da Ferramenta	81
6.2.2	Funcionalidades da OConGraX	82
6.3	Java PathFinder Exceptions	87
6.3.1	Java PathFinder	88
6.3.2	Propriedades Relativas a Tratamento de Exceção	89
6.3.3	Funcionalidades do Java PathFinder Exceptions	90
6.4	Integração das ferramentas via XML	94
6.4.1	Modelo XML	94
6.4.2	OConGraX - Geração do XML	96
6.4.3	JPF Exceptions - Leitura do XML	97
6.4.4	Exemplo	98
6.5	Integração das Ferramentas via Interface Gráfica	101
6.5.1	Interface Gráfica da Java V&V	102
6.5.2	Adaptações da Interface para Integração	103
6.5.3	Novas Funcionalidades	104
6.5.4	Exemplo	104
6.5.5	Análise da Cobertura dos Pares du	108
6.6	Conclusão	109
7	Conclusões	111
7.1	Contribuição Científica	111
7.2	Escrita de Artigos	112
7.2.1	Artigos Publicados	112
7.2.2	Artigos em Julgamento e em Processo de Escrita	113
7.3	Trabalhos Futuros	113

A	Especificação do <i>framework</i> em CSPm	115
A.1	Especificação do <i>Framework</i>	115
A.2	Especificação das Propriedades	119
B	Código do Exemplo de Instanciação do Modelo de Ações Atômicas Coordenadas	123
B.1	Pacote exception	123
B.1.1	Classe AccountAlreadyExistsException	123
B.1.2	Classe InvalidAccountNumberException	123
B.2	Pacote simulation.database	123
B.2.1	Classe BankAccount	123
B.2.2	Classe BankDatabase	124
B.2.3	Classe Transaction	125
B.3	Pacote simulation.exceptionTree	126
B.3.1	Classe ExceptionTree	126
B.4	Pacote simulation.participants	126
B.4.1	Classe Participant0	126
B.4.2	Classe Participant1	127
B.5	Pacote simulation.handlers	127
B.5.1	Classe Participant0Handler0	127
B.5.2	Classe Participant1Handler0	128
B.6	Pacote simulation.roles	128
B.6.1	Classe Participant0Role0	128
B.6.2	Classe Participant1Role0	129
B.6.3	Classe Participant1Role1	130
C	Código Fonte de Propriedade de Tratamento de Exceção	133
C.1	Propriedade: Bloco de Captura Vazio	133
C.2	Saída da Verificação Feita pela JPF Exceptions	134
C.2.1	Propriedade 1: Desfazimento de Transações	134
C.2.2	Propriedade 2: Tratamento das Exceções	135
D	Código Fonte para Integração via XML	139
D.1	Geração do XML pela OConGraX	139
D.1.1	XMLGenerator.java	139
D.2	Leitura do XML pelo Java PathFinder Exceptions	143
D.2.1	Método readOcongraXml, da classe ExecutedPath.java	143
	Referências Bibliográficas	145

Lista de Figuras

1.1	Esquema da Abordagem Teste + Verificação [XHdM08]	2
2.1	Modelo de três universos desenvolvido por Avizienis [Avi82], em que percebemos as diferenças de contexto em que estão inseridos uma “falha”, um “erro” e um “defeito”	6
2.2	Diagrama de atividades de teste de software, baseado em [Som06].	7
2.3	Fases do teste de software	8
2.4	Estruturas de controle na notação de grafo de fluxo de controle [Pre01]	13
3.1	Esquema do processo de verificação de um sistema <i>a posteriori</i> [BK08].	28
3.2	Esquema do processo de <i>model checking</i> [BK08]	31
4.1	Componente tolerante a falhas [dMF01]	38
4.2	Diagrama de representação de uma ação CA.	40
4.3	Ações CA aninhadas	40
4.4	Ações CA compostas	41
4.5	Esquema de arquitetura do modelo implementado.	43
4.6	Exemplo - Comportamento Normal	55
4.7	Exemplo - Comportamento Excepcional	59
5.1	Classe <i>PropertyListenerAdapter</i> e suas dependências	66
6.1	Esquema da Abordagem Teste + Verificação [XHdM08]	80
6.2	Esquema da arquitetura da OConGraX.	81
6.3	GUI da OConGraX	82
6.4	Visualização das definições e usos do método <i>main</i> do exemplo 6.2, em que pode-se ver as definições e usos do objeto <i>obj</i> .	83
6.5	Visualização dos pares do do objeto <i>obj</i> , do exemplo 6.2.	83
6.6	<i>Screenshots</i> mostrando algumas das funcionalidades da OConGraX.	84
6.7	Tela com as opções para geração do arquivo XML	86
6.8	Visualização do grafo de fluxo de controle para a exceção e do exemplo 6.2	86
6.9	Tela para seleção do projeto a ser verificado	91
6.10	Tela para seleção das propriedades	91
6.11	Tela para seleção dos <i>listeners</i>	92
6.12	Tela de estatísticas de cobertura	93
6.13	Tela de rastreamento de dados	93
6.14	Esquema da integração via XML entre OConGraX e JPF Exceptions	94

6.15	Execução no Java PathFinder Exceptions	100
6.16	Java V&V	102
6.17	GUI da Java V&V	102
6.18	Java V&V - Aba “Listeners” da JPF Exceptions.	103
6.19	Java V&V - Aba “Properties” da JPF Exceptions.	104
6.20	Java V&V: visualização da árvore de definições e usos e do OCFG	105
6.21	Seleção do projeto a ser verificado	106
6.22	Seleção das propriedades	106
6.23	Configuração dos <i>listeners</i>	107
6.24	Estatísticas de cobertura	107
6.25	Estatísticas da busca	108

Lista de Tabelas

2.1	Definições e usos do objeto <i>shapeObj</i> do exemplo	16
2.2	Critérios e pares du	16
2.3	Critérios baseados em definição-uso	21
2.4	Critérios baseados em ativação-desativação	21

Capítulo 1

Introdução

Neste capítulo é apresentada a motivação e os objetivos a serem alcançados neste trabalho. A motivação provém, principalmente, dos problemas enfrentados atualmente na área de Desenvolvimento Baseado em Componentes relativos à confiabilidade do software gerado; já os objetivos, por sua vez, expõem como o trabalho que está sendo desenvolvido pode ajudar na resolução destes problemas. Ao final do capítulo, o modo como este documento está organizado é apresentado.

1.1 Motivação

No mercado global de competição acirrada e mudanças constantes, as organizações precisam, continuamente, adaptar suas práticas de negócio para atender às diferentes exigências impostas por governos, clientes, fornecedores e parceiros comerciais, mantendo a vantagem competitiva sobre seus concorrentes. Uma das mais interessantes e promissoras propostas é o *Desenvolvimento Baseado em Componentes* (DBC), que promete uma alta produtividade no processo e sistemas com maior flexibilidade, confiabilidade e qualidade.

1.1.1 Componentes e Desenvolvimento Baseado em Componentes (DBC)

Com o crescimento e a popularização do mercado de componentes, muitos fornecedores de software têm se especializado no desenvolvimento e comercialização de componentes que atendam às necessidades específicas dos desenvolvedores de sistemas. Estes componentes são mais conhecidos como *COTS*¹ ou de *prateleira*.

Utilizar estes componentes pré-existentes, integrando-os e assim construir rapidamente um novo software, é a ideia básica da abordagem do DBC. Além de prover um grande ganho da produtividade no desenvolvimento de software graças à reutilização de componentes já existentes [Pre01], possíveis mudanças nos requisitos, impulsionadas pelas necessidades de negócio, podem ser facilmente aplicadas com a substituição de um pequeno número de componentes.

Entretanto, a montagem de sistemas corporativos que sejam confiáveis e tolerantes a falhas, a partir da integração de componentes comerciais, tem-se mostrado uma atividade relativamente complexa [Crn02, GAO95, Voa98]. Garantir que esta integração não falhe passou a ser algo imprescindível, sobretudo porque as consequências de uma falha poderiam acarretar, por exemplo, grandes prejuízos financeiros, perda material e de vidas humanas.

1.1.2 Testes e Verificação Formal

Com o objetivo de definir quais requisitos são necessários para estabelecer um bom teste do software, definem-se os critérios de teste [Wey88, MK94, CK99, KSH⁺95]. Mesmo considerando o direcionamento da escolha de conjuntos de testes a serem realizados a partir dos critérios, tais conjuntos são, na sua maioria, ainda muito complexos. Apesar do esforço de contemplar “todo” o

¹Do inglês *Commercial-Off-The-Shelf*

código do software e, com isso, ter uma chance maior de ter os defeitos encontrados, os testes são capazes de assegurar o funcionamento do software apenas para o conjuntos de testes a que ele foi submetido.

Uma outra abordagem para garantir o funcionamento do software como inicialmente especificado, são as técnicas de verificação formal de programas, as quais procuram assegurar que os programas funcionam de acordo com a especificação ou atendem a um conjunto de propriedades relevantes. Existe atualmente um esforço no desenvolvimento de ferramentas que auxiliem a verificação formal de programas, inclusive no paradigma de orientação a objetos – para a linguagem Java, por exemplo, explorada neste projeto [VHB⁺03, HDD⁺03, RDH04, EM04, DHJ⁺01, PDV01a, HD01]. Assim como nos testes, assegurar o bom funcionamento de programas por meio da verificação formal tem um impacto considerável no custo do desenvolvimento de software. Teste e verificação de programas são, portanto, atividades complementares no desenvolvimento de software e podem ser exploradas como tal [Xav08, HDD⁺03, RDH04, EM04, HD01, PDV01a, DHJ⁺01].

A viabilidade prática de testes de software depende de mecanismos e ferramentas que auxiliem esta atividade, de forma a automatizar a geração de casos e dados de testes a serem aplicados. Por outro lado, a aplicação prática de verificadores formais em linguagens de programação depende da predefinição de propriedades, já que os “desenvolvedores” de software, em geral, não possuem treinamento adequado para definir novas propriedades. Além disso, faz-se necessário o uso conjunto de ferramentas de ambas as atividades para que sejam exploradas suas complementaridades.

No desenvolvimento de sistemas baseados em componentes (DBC), o teste e a verificação devem ser realizados tanto para os componentes individuais como para os seus mecanismos de integração. O projeto “Formalização da Orquestração de Componentes Tolerantes a Falhas (CNPq: 551038/2007-1)” tem como objetivo principal dar suporte ao tratamento formal para os mecanismos de integração de componentes tanto para o comportamento normal como para o comportamento excepcional. Dentro deste projeto maior, este trabalho de Mestrado se propõe a desenvolver um ambiente integrado para a verificação e teste de protocolos para a coordenação do comportamento excepcional de componentes. A integração proposta no trabalho entre validação e verificação almeja a **redução do espaço de testes** por meio da utilização dos resultados da verificação para geração de um conjunto de testes, conforme ilustrado a seguir.

Figura 1.1: Esquema da Abordagem Teste + Verificação [XHdM08]

Maiores detalhes desta abordagem são fornecidos na Seção 6.1.

1.2 Objetivos

A meta deste trabalho é construir um ambiente que facilite a realização tanto de testes como de verificação de sistemas baseados em componentes tolerantes a falhas. Para isso, foi definido inicialmente um conjunto de propriedades desejáveis relativas à coordenação do comportamento excepcional de componentes; e posteriormente, foi feita a integração de ferramentas para teste e verificação de programas Java:

1. OConGraX (*Object Control-Flow Graph with eXceptions*) [HM07, NHdM09]²: esta ferramenta, escrita na linguagem Java [GJSB05a], recebe um sistema também escrito em Java e devolve ao usuário informações como: objetos e exceções contidos no software recebido como entrada, suas definições e usos [CK99, SH98], e um grafo de controle de fluxo de objetos e de exceções [CK99, SH98, SH99]. Estas informações, em conjunto, permitem realizar testes *white-box* de maneira mais rápida e eficiente.
2. JPF Exceptions (*Java Pathfinder Exceptions*) [Xav08]: utilizando-se do sistema JPF (Java Pathfinder) [VHB⁺03, Jav], o aplicativo desenvolvido nesse trabalho permite a realização de verificação de propriedades relativas ao comportamento excepcional dos componentes e testes utilizando os critérios baseados em fluxos de objetos e exceções de programas [CK99, SH99].

Dessa forma, os objetivos a serem alcançados neste trabalho são:

- Definição de propriedades relativas à coordenação do comportamento excepcional dos componentes: predefinir um conjunto de propriedades relativas à coordenação do comportamento excepcional dos componentes para que sejam utilizadas diretamente pelos desenvolvedores. Desta forma, o desenvolvedor de componentes pode verificar os seus mecanismos de coordenação, a partir de uma arquitetura padrão, sem precisar definir novas propriedades para cada sistema especificamente, tornando prático o uso de verificadores formais na construção de sistemas baseados em componentes.
- Integração dos trabalhos acima citados: permitir a comunicação entre ambos via XML. Esta comunicação consiste na geração de um arquivo XML contendo as definições e usos de objetos e exceções, mais os pares de definição e uso de exceções, por parte da OConGraX. Em seguida, o arquivo gerado deverá ser lido pela JPF Exceptions para obtenção dos dados de definições e usos oriundos da OConGraX. Estes dados são usados então pela JPF Exceptions para realizar a verificação de propriedades e o rastreamento dos pares de definições e usos cobertos durante a verificação.
- Desenvolvimento de um aplicativo em Java, com uma interface gráfica que permita ao testador utilizar facilmente todas as funcionalidades de testes e verificação proporcionadas pela integração dos dois trabalhos citados. Este aplicativo é dedicado a programas escritos na linguagem Java.

1.3 Organização dos Capítulos

Este trabalho está organizado nos seguintes capítulos:

- **Capítulo 2:** apresentação da teoria relacionada a testes de programas e de algumas ferramentas utilizadas para automatizar este processo.
- **Capítulo 3:** apresentação da teoria relacionada à verificação de programas e de algumas ferramentas utilizadas para a automação deste processo.

²A ferramenta OConGraX foi desenvolvida no IME-USP por mim, durante iniciação científica feita sob orientação da Prof^a Ana C. V. de Melo [HM07].

- **Capítulo 4:** apresentação da teoria relacionada a componentes e ações atômicas coordenadas, e da implementação do modelo de ações atômicas coordenadas a ser utilizado para a realização de verificação de suas propriedades.
- **Capítulo 5:** apresentação das propriedades sobre as ações atômicas coordenadas e sua implementação na ferramenta JPF Exceptions.
- **Capítulo 6:** apresentação da ideia de V&V (uso integrado de validação e verificação), uma descrição das ferramentas a serem utilizadas no trabalho e o resultado da integração entre elas.
- **Capítulo 7:** apresentação das conclusões finais sobre o desenvolvimento deste trabalho.

Capítulo 2

Teste de Software

Neste capítulo é apresentada a teoria relativa a testes de programas utilizada neste trabalho. Inicialmente, descrevemos os termos relativos a este tema que serão utilizados ao longo do texto. Em seguida, abordamos os tópicos relacionados à teoria de testes:

- o conceito e a importância do teste;
- seus objetivos e limitações;
- as etapas do processo de teste de um software;
- as fases e estratégias de teste existentes;
- as técnicas para projeto de casos de teste e os critérios de teste.

Ao final, apresentamos algumas das ferramentas utilizadas para automatizar parte do processo de teste de software, tornando-o mais viável para aplicação prática.

2.1 Termos Utilizados

Nesta seção, apresentamos alguns termos¹ [Bin00, Avi82, DMJ07, Int06, Xav08, Per07, WFW09] a serem utilizados no decorrer do texto:

- **Bug:** problema em um sistema de software, como falha ou defeito.
- **Falha:** ocorre quando um sistema ou componente não fornece a funcionalidade esperada, devido, por exemplo, a condições físicas anômalas. Pode ser percebida por meio de saídas incorretas, finalizações anormais ou o não cumprimento de metas estabelecidas para restrições de tempo e espaço.
- **Erro:**² manifestação de uma falha no sistema, causando modificações no estado do sistema por meio de valores diferentes dos esperados.
- **Defeito:** causado por um erro, é definido como um desvio de execução em relação à especificação original do sistema.
- **Dados de testes:** correspondem aos dados de entrada a serem utilizados no teste de um programa.

¹Embora utilizadas neste texto, as definições para os termos aqui apresentadas nem sempre são seguidas por não serem unanimidade entre os pesquisadores da área [DMJ07].

²O termo “erro” pode ser empregado também de uma maneira informal, como tendo qualquer um dos significados apresentados para “falha” e “defeito”.

Figura 2.1: Modelo de três universos desenvolvido por Avizienis [Avi82], em que percebemos as diferenças de contexto em que estão inseridos uma “falha”, um “erro” e um “defeito”

- **Casos de testes:** definem o estado anterior ao teste do elemento a ser testado, bem como de seu ambiente, as entradas ou condições de teste e os resultados esperados. Os resultados esperados definem o que o sistema deve produzir a partir das entradas de teste. Isto inclui mensagens geradas pelo sistema, exceções, valores retornados e o estado resultante do sistema e de seu ambiente.
- **Conjunto de testes (*Test suite*):** conjunto de casos de teste relacionados.
- **Execução de testes:** é a execução de um conjunto de testes.
- **Test driver:** classe ou programa utilitário que aplica casos de teste a um sistema.
- **Stub:** consiste numa implementação que simula o funcionamento de um método, módulo ou componente do sistema, cujo código ainda está incompleto. Seu uso ocorre principalmente para efeito de teste do sistema.
- **Cobertura de teste:** métrica que descreve a quantidade de requisitos de teste exercitada no código-fonte de um programa por um dado conjunto de teste. Critérios de teste baseados nessa métrica especificam o que deve ser coberto (isto é, exercitado por ao menos um teste) no sistema, limitando assim o conjunto de testes necessários para testar efetivamente o programa.

2.2 Conceito e Importância dos Testes

Segundo Bertolino [Ber03], o teste de software consiste na verificação dinâmica do comportamento do programa por meio de um conjunto finito de casos de teste, selecionados apropriadamente a partir de um domínio de execução geralmente infinito, e na comparação do comportamento obtido com o esperado de acordo com a especificação. Ele constitui uma importante fase do desenvolvimento de software, e sua importância se dá, principalmente, devido ao fato de ser por meio dele que *bugs* de software são identificados. Corrigindo estes *bugs*, a probabilidade de falha no software diminui, tornando assim o seu uso mais seguro e confiável.

No caso de sistemas críticos, como os utilizados em aeronaves e na área da saúde, a realização de testes é ainda mais importante (senão imprescindível), para se evitar eventos catastróficos que envolvam perdas de vidas e/ou de grandes somas de dinheiro. Como exemplos de eventos destas proporções, pode-se citar o caso da queda do foguete francês Ariane 5, em 1996, que ocorreu devido a um erro de conversão de um valor de ponto flutuante de 64 bits para um inteiro de 16 bits [Inq96], e o caso da perda de uma sonda da NASA em Marte devido a incompatibilidades entre dois componentes de software, onde um deles utilizava o sistema inglês e o outro o sistema métrico de medidas [MPI00].

2.3 Objetivos e Limitações

As principais metas a serem alcançadas com o teste de software [Som06, Pre01] são:

- Descobrir *bugs* no software que fazem com que ele não funcione adequadamente. Para isto, é importante que os casos de teste a serem projetados tenham alta probabilidade de encontrar erros ainda não descobertos.
- Mostrar aos desenvolvedores e ao cliente que o software funciona de acordo com sua especificação.

É importante salientar aqui que, como Dijkstra afirmou [Dij70], “Testes de programas podem ser usados para mostrar a presença de erros, mas nunca para mostrar sua ausência”. Desta forma, testes jamais podem ser usados para demonstrar que o software está completamente livre de erros, mas sim, podem ser utilizados para convencer que o software desenvolvido é bom o suficiente para uso operacional.

Outra questão relacionada aos testes é o fato de ser necessário muito tempo para sua realização. Como afirmou Pressman [Pre01], cerca de 30 a 40% de todo o esforço no desenvolvimento de um software é gasto em testes, o que torna este processo também bastante caro. Para tornar o teste de programas viável, ferramentas que automatizam etapas desta atividade estão sendo construídas. Algumas delas são apresentadas na Seção 2.9.

2.4 Etapas do Processo

Testar um software é uma tarefa dinâmica e complexa, que demanda tempo para sua realização. De um modo geral, podemos dividir o processo de teste nas seguintes etapas [Som06]:

1. **Fazer o *design* dos casos de teste:** primeira etapa a ser feita, consiste no planejamento de casos de teste com todas as suas entradas e respectivas saídas esperadas, com a finalidade de encontrar possíveis erros no código. Idealmente, deve-se escrever uma quantidade de casos de teste não muito grande, mas suficiente para que estes casos cubram todas as funcionalidades e estrutura do programa [Pre01].
2. **Preparar os dados de teste:** prover os dados de entrada, especificados nos casos de teste, para execução no programa.
3. **Executar o programa com os dados de teste:** utilizando os dados de teste, o conjunto de testes do programa é executado e os resultados retornados pelo software são colhidos.
4. **Comparar os resultados obtidos com os esperados:** fazer a comparação dos resultados obtidos na execução do programa, com os resultados esperados nos casos de teste. Caso haja divergência nos valores, é preciso averiguar no software os erros que causaram este problema, identificá-los e corrigi-los.

Na Figura 2.2, podemos ver um diagrama mostrando estas etapas de maneira esquematizada.

Figura 2.2: Diagrama de atividades de teste de software, baseado em [Som06].

2.5 Fases e Estratégias de Teste

Fases (ou níveis) de teste têm como função determinar o escopo de cobertura que o teste de um software deve ter. De acordo com Delamaro et al. [DMJ07], são três as fases principais no desenvolvimento de testes: testes unitários, testes de integração e testes de sistemas. A Figura 2.3 ilustra este conceito, apresentando por meio de um diagrama as três fases em sua ordem correta de realização.

Figura 2.3: *Fases do teste de software*

Além de escopos próprios, cada fase possui estratégias de teste que podem ser usadas durante sua realização. Estas estratégias são as responsáveis por guiar o desenvolvimento dos testes, de maneira planejada e eficaz.

Nas próximas seções, descrevemos cada uma dessas fases e suas respectivas estratégias.

2.5.1 Testes Unitários

Testes unitários [Pre01, DMJ07] possuem como foco principal o teste nas menores unidades/-módulos de um programa ou componente. No caso de programas procedimentais, poderíamos considerar como unidades suas funções; no caso de programas orientados a objetos, elas poderiam ser os métodos, ou mesmo as classes do programa.

Nesta fase, os erros a serem descobertos estão relacionados principalmente ao código do programa, sobretudo na implementação de algoritmos e na manipulação de estruturas de dados. Exemplos de *bugs* a serem encontrados, são: computações erradas, comparações incorretas e fluxos de controle impróprios.

- **Procedimento para a Realização dos Testes Unitários**

Os testes unitários são feitos, em geral, juntamente com o processo de implementação do código. Por ser testado apenas um único componente/módulo do programa, e por ele não ser um programa *stand-alone*, é preciso utilizar um *test driver* e *stubs* para a execução dos testes. Para evitar uma carga de trabalho adicional, deve-se manter estes códigos auxiliares o mais simples possível.

2.5.2 Testes de Integração

Após verificar que os módulos de um programa funcionam bem isoladamente, por meio dos testes unitários, é necessário garantir que, ao colocá-los funcionando juntos, não ocorram *bugs*. Para isso, a segunda fase dos testes deve ser feita, a fase dos testes de integração [Pre01, DMJ07].

Os testes de integração são feitos de modo incremental, de forma que a cada módulo integrado ao sistema, testes sejam realizados para garantir que esta integração não gerará erros. Assim, caso algum erro seja encontrado, ele pode ser isolado e corrigido facilmente.

Para realizar os testes de integração, diversas estratégias existem. A seguir, descrevemos três das principais estratégias [Pre01]: a de integração *top-down*, a de integração *bottom-up* e a de regressão.

- **Testes para Integração *Top-Down***

A integração *top-down* consiste em, a partir do módulo principal do programa (utilizado como *test driver*), adicionar seus módulos diretamente subordinados sucessivamente, por meio da substituição dos *stubs* construídos anteriormente. Para a incorporação destes módulos subordinados, duas abordagens podem ser adotadas: a integração em profundidade (*depth-first integration*), que integra todos os componentes do maior caminho de controle da estrutura; e a integração em largura (*breadth-first integration*), que incorpora todos os componentes de um mesmo nível hierárquico.

Uma vez integrado, o componente deve ser testado para assegurar que erros não estão sendo inseridos durante o processo, garantindo assim a qualidade do software.

Com a aplicação desta estratégia de teste, os pontos principais de decisão e controle são verificados logo no início do processo de teste, considerando que o sistema tenha uma hierarquia consistente e os níveis mais altos desta hierarquia contenham tais pontos. Desta forma, problemas nesses pontos principais do programa são detectados mais cedo, o que é essencial para o desenvolvimento do software. Ademais, se a abordagem utilizada for a de integração em profundidade, é possível ter uma funcionalidade completa do programa implementada e devidamente demonstrada. Isto dá confiança ao desenvolvedor e ao cliente para prosseguir com o desenvolvimento do sistema.

Apesar das vantagens, e de esta estratégia parecer simples, na prática muitos problemas podem surgir quando ela é aplicada. Estes problemas são causados, em geral, pela necessidade de se integrar componentes de nível hierárquico mais baixo para testar componentes que estão no alto da hierarquia. Isto ocorre porque, inicialmente, *stubs* substituem os componentes de nível hierárquico baixo, não gerando um fluxo significativo de dados para os componentes já integrados. Com isso, os testadores podem seguir três alternativas:

1. **Atrasar a realização dos testes até que os *stubs* sejam substituídos pelos módulos reais:** isto pode dificultar o acesso à causa de possíveis erros encontrados, além de violar a natureza da integração *top-down*.
2. **Desenvolver *stubs* que simulem o funcionamento do módulo real:** isto pode significar uma carga maior de trabalho adicional, uma vez que os *stubs* tornam-se mais e mais complexos.
3. **Utilizar uma outra forma de integração:** ao invés da estratégia *top-down*, utilizar por exemplo a estratégia *bottom-up*.

- **Testes para Integração *Bottom-Up***

A integração *bottom-up* começa com os módulos do nível hierárquico mais baixo, e os integra, continuamente, aos módulos que se encontram diretamente acima na hierarquia. Desta forma, a necessidade de *stubs* é eliminada, já que os componentes subordinados já estão integrados no momento do teste dos componentes de nível hierárquico mais alto.

Para a realização desta integração, primeiro os módulos de nível hierárquico mais baixos são combinados em um *cluster*, que faz uma tarefa específica do programa. Em seguida, um *test driver* é escrito, para testar o *cluster*. Uma vez que o *cluster* é verificado (e corrigido), o *driver* utilizado é removido e a integração prossegue subindo na estrutura do programa.

A principal desvantagem desta abordagem é a de que um programa não existe como uma entidade até que o seu último módulo seja adicionado [Mye79], problema que não existe na abordagem *top-down*.

- **Testes de Regressão**

Esta estratégia é utilizada nesta fase de teste para minimizar os efeitos colaterais ³ que possam ser causados pela integração dos componentes do programa. Ela consiste na re-execução de um subconjunto de testes realizados numa fase anterior, assegurando assim que as mudanças ocasionadas pela integração não geram efeitos colaterais.

O sub-conjunto de testes a ser utilizado no teste de regressão possui três classes de casos de testes:

1. Um conjunto representativo de testes que exercitam todas as funções do software;
2. Testes adicionais que focam nas funções do software que podem ser afetadas pelas mudanças causadas pela integração;
3. Testes que focam nos componentes de software que sofreram alguma alteração no processo de integração.

2.5.3 Testes de Sistemas

O teste de sistemas [Pre01, DMJ07] engloba uma série de testes, cujo principal objetivo é o de exercitar todas as funcionalidades do sistema computacional completo, garantindo assim que todos os elementos do sistema foram integrados adequadamente e realizam suas funções corretamente.

A seguir, três tipos de teste de sistemas são apresentados: o teste de carga (ou de *stress*), o teste de desempenho e o teste de recuperação.

- **Teste de Carga (ou de *Stress*)**

Esta estratégia é utilizada para verificar o comportamento do sistema em situações anormais, que demandam uma grande quantidade, volume e frequência de seus recursos. Para isso, são executados testes que forcem o programa a enfrentar tal situação, como por exemplo, a execução de casos de teste que precisam que a memória do sistema e outros recursos sejam utilizados em sua capacidade máxima. A ideia é conhecer o limite do sistema desenvolvido.

- **Teste de Desempenho (ou de Performance)**

No teste de performance, testes são projetados para verificar o desempenho em tempo de execução do sistema, dentro do contexto de sistemas integrados. Frequentemente esta estratégia é aplicada juntamente com a de teste de carga, pois ambas fazem uso tanto do hardware quanto do software. Para monitorá-la e verificar os resultados, deve ser utilizado um aparelho externo que faça a medição da utilização de recursos pelo sistema.

Verificar a performance da execução de um sistema é necessário, sobretudo no caso dos sistemas de tempo real e dos sistemas embarcados, uma vez que eles exigem rapidez na resposta do sistema.

- **Teste de Recuperação**

Consiste em fazer com que o sistema falhe de diversas maneiras, a fim de avaliar o modo como se dá sua recuperação. Caso a recuperação seja feita automaticamente pelo próprio sistema, a correção da reinicialização e/ou da recuperação dos dados é verificada. Já no caso da recuperação ser feita com intervenção humana, é avaliado o tempo necessário para que o sistema seja restabelecido.

Esta estratégia deve ser aplicada principalmente em sistemas que devem ser tolerantes a falhas (e assim devem continuar funcionando após a ativação de uma falha), e em sistemas que precisam retornar ao funcionamento rapidamente para evitar transtornos a seus usuários.

³Por efeitos colaterais, entende-se a existência de erros em funções que funcionavam corretamente antes de sua integração com outras partes do código, e a posterior propagação destes erros.

2.6 Design de Casos de Testes

Métodos para guiar o projeto de casos de teste têm sido criados com o intuito de ajudar os desenvolvedores de teste a realizar esta tarefa do modo melhor e mais completo possível. Estes métodos fornecem, por meio de uma abordagem sistemática, mecanismos que auxiliam na derivação de casos de teste que possuem alta probabilidade de encontrar erros no software.

Para fazer o *design* dos casos de teste, duas abordagens principais são utilizadas [Pre01]:

1. Assegurar que todas as funções especificadas na fase de requisitos do produto estejam implementadas e funcionando corretamente. Assim, testes devem ser feitos de modo a demonstrar que as funcionalidades do produto satisfazem sua especificação, ao mesmo tempo em que é feita uma varredura para encontrar eventuais erros no software. Esta abordagem é também conhecida como **teste *black-box***.
2. Verificar que todas as partes internas do software funcionam de acordo com a especificação fornecida nas fases anteriores de seu desenvolvimento. Assim, os testes devem ser conduzidos de modo a procurar exercitar todos os componentes do software, fazendo uma varredura na estrutura do código à procura de *bugs*. Esta abordagem é conhecida como **teste *white-box***.

Nas próximas seções, detalhamos estas abordagens, descrevendo os procedimentos e técnicas que podem ser feitos para a adoção de cada uma delas no desenvolvimento dos casos de teste, baseados na teoria apresentada por Pressman [Pre01] e Delamaro et al. [DMJ07].

2.6.1 Testes *Black-Box*

Os testes *black-box*, também conhecidos como testes funcionais, buscam validar o comportamento de um programa, avaliando se ele se ajusta às especificações fornecidas. Esta abordagem não exige conhecimento sobre a estrutura do código do programa, o que possibilita sua aplicação a programas tanto de pequeno quanto de grande porte. Ademais, os testes funcionais permitem, além de detectar *bugs*, ver como o programa reage a situações de falha, e se a consistência e integridades dos dados externos é mantida.

Os possíveis erros a serem identificados pelos testes *black-box* podem ser separados nas seguintes categorias:

- Funções incorretas ou inexistentes;
- Erros de interface;
- Erros na estrutura de dados ou no acesso a dados externos;
- Erros de performance ou comportamento;
- Erros de início e término.

Por meio da utilização dos testes funcionais, um conjunto de casos de teste que satisfazem os seguintes critérios é derivado:

- Casos de teste que reduzem significativamente o número de casos de teste adicionais que devem ser projetados para alcançar um nível de teste aceitável;
- Casos de teste que nos informam sobre a presença ou ausência de categorias de erros, ao invés de dar informações sobre um erro específico associado ao teste em questão.

Para fazer a derivação de tais casos de teste, técnicas foram desenvolvidas. Apresentamos algumas delas a seguir [Pre01]:

- **Particionamento de Equivalência**

Este método divide o domínio de entrada em classes de equivalência, das quais são derivados os casos de teste. Por classe de equivalência, entende-se um conjunto de estados válidos ou inválidos para as condições de entrada e de saída. Estas, podem ser um valor numérico específico, uma faixa de valores, um conjunto de valores relacionados ou uma condição booleana.

Obtendo-se as classes de equivalência, casos de teste para cada item do domínio de entrada e saída são desenvolvidos e executados, com o intuito de descobrir possíveis categorias de erros presentes no código. Os casos de teste são selecionados de maneira que o maior número de atributos das classes de equivalência válidas seja exercitado numa única vez.

- **Análise do Valor Limite**

A técnica da análise do valor limite foi desenvolvida com a finalidade de selecionar casos de teste que exercitem os valores limites do domínio de entrada, valores nos quais os erros tendem a se concentrar. Ela é complementar à técnica do particionamento de equivalência, e analisa tanto o domínio de entrada, quanto o domínio de saída do programa, de modo que:

1. Para um intervalo de valores do domínio de entrada, são testados seus valores mínimo e máximo;
2. Para testar os valores limites do domínio de saída, devem ser criados casos de teste cujos resultados de execução sejam os valores mínimo ou máximo permitidos para os dados de saída;
3. Caso as estruturas de dados internas também tenham limites, como por exemplo *arrays*, os testes devem exercitar cada uma delas em seus valores limites.

- **Teste de Comparação**

Em sistemas críticos, muitas vezes hardware e software redundantes são utilizados, a fim de minimizar a possibilidade de erros. Para desenvolver software redundante, diversas equipes trabalham em versões independentes da aplicação, fazendo uso de uma mesma especificação. Desta forma, para garantir a consistência do trabalho, as versões são testadas paralelamente com os mesmos dados de entrada dos casos de teste projetados utilizando outras técnicas de teste funcional (ex.: particionamento de equivalência), e a saída de cada uma das versões é então verificada. Se as saídas forem todas iguais, assumimos que todas as versões estão corretas; caso contrário, é feita uma busca em cada aplicação para se encontrar o defeito que causou o resultado diferente.

A realização dos testes *black-box* ocorre, em geral, nas etapas finais do desenvolvimento do software. Isto porque eles se concentram em achar erros no domínio da informação, e não na estrutura/código do programa, o que traz como consequência a necessidade de ter boa parte do sistema já funcionando para que suas funcionalidades sejam testadas adequadamente.

2.6.2 Testes *White-Box*

Testes *white-box*, ou testes estruturais, preocupam-se em examinar detalhadamente os procedimentos adotados no código do programa, fazendo uma varredura pelos caminhos lógicos do programa e verificando se, em determinados pontos, os resultados obtidos coincidem com os resultados esperados. Alguns dos casos de teste que podem ser derivados utilizando métodos de testes estruturais visam a garantir que:

- Todos os caminhos independentes de um módulo do software foram executados ao menos uma vez;
- Todas as decisões lógicas são exercitadas, exercitando tanto o lado verdadeiro, quanto o falso;

- Todos os laços do programa são executados em seus limites, inclusive os limites de operações internas;
- A estrutura dos dados externos é exercitada, assegurando sua validade.

A seguir, apresentamos algumas técnicas relativas ao teste *white-box* utilizadas para derivar casos de teste [Pre01]:

• Teste de Caminho Básico

Proposta por McCabe [McC76], esta técnica parte de uma medida de complexidade derivada do código, e utiliza-a como um guia para definir um conjunto básico de caminhos de execução. Os casos de teste gerados devem, necessariamente, exercitar cada instrução do programa ao menos uma vez. Eles são derivados a partir de quatro passos, mostrados abaixo:

1. **Desenhar um grafo de fluxo correspondente ao código a ser testado:** o grafo de fluxo é uma representação dos caminhos de execução do programa (Figura 2.4). Cada nó representa uma instrução do código, enquanto as arestas orientadas mostram a direção do fluxo de execução.

Figura 2.4: Estruturas de controle na notação de grafo de fluxo de controle [Pre01]

2. **Determinar a complexidade ciclomática do grafo:** a complexidade ciclomática (CC) é uma métrica de software que mede quantitativamente a complexidade lógica do programa. No contexto da técnica de teste de caminhos básicos, o valor da complexidade ciclomática determina o limite superior para o número de caminhos independentes no conjunto básico do programa, e para o número de testes que devem ser realizados para assegurar que todas as instruções foram executadas ao menos uma vez. Para calcular o valor desta métrica, utilizando o grafo, devemos efetuar um dos dois cálculos:
 - (a) $CC = A - N + 2$, onde A é o número de arestas do grafo, e N o número de nós;
 - (b) $CC = P + 1$, onde P é o número de nós predicativos contidos no grafo.
3. **Determinar um conjunto básico de caminhos linearmente independentes:** por caminhos linearmente independentes, entende-se qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição [DMJ07]. No contexto do grafo de fluxo, a cada novo caminho linearmente independente, uma aresta é incluída. Neste passo, o número de caminhos é limitado superiormente pelo valor da complexidade ciclomática calculado anteriormente.
4. **Preparar casos de teste que forcem a execução de cada caminho independente:** os dados devem ser escolhidos de modo que as condições nos nós predicativos sejam apropriadamente exercitadas a cada caminho testado.

• Teste de Estruturas de Controle

Apesar da técnica anterior ser bastante simples e eficiente, ela não é suficiente se aplicada sozinha. Para melhorar o teste *white-box*, então, outros métodos para testes de estruturas de controle foram desenvolvidos, e são apresentados a seguir.

- **Testes de Condição:** método para *design* de casos de teste que busca exercitar todas as condições existentes no programa. A vantagem deste método é a de que se o conjunto de testes obtido for efetivo para encontrar erros nas instruções de condição do programa, então provavelmente ele também o é para encontrar erros no programa [Pre01].
- **Testes de Laços:** embora desempenhem, muitas vezes, papéis fundamentais no algoritmo implementado, os laços são muitas vezes negligenciados na hora dos testes. Este método para *design* de casos de teste procura alterar este cenário, focando na validação de todos os laços existentes no programa.
- **Testes de Fluxo de Dados:** este método seleciona os caminhos para teste de um programa de acordo com a localização das definições e usos de suas variáveis. Particularmente, a variante deste método para programas orientados a objeto foi parcialmente automatizada pela ferramenta OConGraX [HM07, NHdM09], a ser utilizada neste trabalho. Maiores detalhes sobre esta variante são dadas na Seção 2.7.

Diferentemente dos testes funcionais, os testes estruturais são, em geral, aplicados já nas fases iniciais do desenvolvimento do programa. Isto ocorre porque eles consideram essencialmente a estrutura do código e sua lógica, que já ficam disponíveis para teste desde o início da implementação. Ademais, estes testes são normalmente aplicados a pequenos módulos do software, pois em módulos de maior porte o número de caminhos lógicos tende a aumentar significativamente, afetando a viabilidade de aplicação de tal abordagem de teste.

2.7 Critérios de Testes para Comportamento Normal de Programas

Critérios de testes são, grosso modo, um conjunto de regras que estabelecem certos requisitos a serem alcançados com os testes de programas. A partir de um critério específico, é feita a seleção de um *test suite* contendo casos de teste que alcancem tais requisitos da maneira mais eficiente possível.

Atualmente, diversos critérios de teste existem para serem aplicados com técnicas de teste correspondentes, aos mais diversos tipos de programas. Em especial, nosso interesse está nos critérios para testes estruturais aplicados a programas do paradigma de orientação a objetos (OO), como os programas escritos na linguagem Java. São estes programas que têm a escrita de casos de teste facilitada pela ferramenta OConGraX [HM07, NHdM09], a ser utilizada neste trabalho de Mestrado.

A seguir, apresentamos os critérios de testes proposto por Chen e Kao [CK99] para programas OO, critérios nos quais foi baseada a implementação da OConGraX [HM07, NHdM09] para a análise do comportamento normal de programas (i.e., análise de código em que exceções não são lançadas e/ou tratadas). Posteriormente, um exemplo de aplicação destes critérios é mostrado.

2.7.1 Critérios de Fluxo de Objetos

Os critérios de fluxo de objetos, propostos por Chen e Kao [CK99], são baseados em fluxo de dados, e englobam as características da programação OO. Primeiramente, vamos definir alguns conceitos necessários para o entendimento desta proposta de seleção de casos de teste, para então apresentarmos os critérios propriamente ditos:

- **Definição de Objeto:** um objeto é definido quando seu estado é inicializado ou alterado, ou seja, quando uma das seguintes condições ocorre:
 1. O construtor do objeto é chamado;
 2. O valor de um atributo de estado é definido, ou seja, é feita uma atribuição explícita a ele;
 3. Uma função que altera o valor de um atributo de estado é chamada;
- **Uso de Objeto:** um objeto é usado quando uma das seguintes condições ocorre:

1. Um de seus atributos de estado é utilizado em uma computação ou predicado;
 2. Uma de suas funções que utilizam os atributos de estado é utilizada;
 3. O objeto é passado como parâmetro para alguma função.
- **Binding:** ocorre quando um objeto pode ser associado a diferentes classes em tempo de execução, numa substituição polimórfica.
 - **Par du de Objetos:** composto por uma definição e um uso de um objeto, sendo que, a partir da definição, é possível chegar ao uso por meio de um caminho em que não ocorra redefinição.
 - **Grafo de Fluxo de Objetos (OCFG):** grafo semelhante ao do teste do caminho básico (ver Seção 2.6.2), que representa o fluxo dos objetos dos programas OO levando em consideração as instruções nas quais eles são definidos e/ou usados, e é utilizado para encontrar os pares du de objetos, sobretudo em códigos de maior porte. O grafo é composto por super-nós que representam os métodos das classes. Dentro dos super-nós encontram-se os nós que contêm o número da linha e a instrução do código na qual ocorre a definição e/ou uso do objeto. A direção do fluxo de execução do objeto dentro de um método é indicada por arestas orientadas cheias; quando o fluxo envolve um nó de outro método, a aresta que faz a ligação inter-método é orientada e tracejada.

Os critérios de fluxo de objetos propostos são definidos a seguir:

- **All-bindings:** todo possível *binding* de cada objeto deve ser exercitado ao menos uma vez quando o objeto é definido ou usado. Se uma instrução envolve múltiplos objetos, então toda combinação de um possível *binding* precisa ser testado ao menos uma vez.
- **All-du-pairs:** todos os pares du de todos os objetos devem ser exercitados ao menos uma vez.

2.7.2 Exemplo

Para exemplificar a aplicação dos critérios de fluxo de objetos, apresentamos na Listagem 2.1 um exemplo [Pru04] adaptado para linguagem Java. O código do exemplo apresenta algumas classes Java que devem ser testadas (*Frame*, *Circle*, *Shape* e *Square*) e uma classe de testes.

```

1 abstract class Shape{
2     int perimeter;
3     public void setPerimeter(int perimeter) {
4         this.perimeter = perimeter;
5     }
6     public int getPerimeter() {
7         return perimeter;
8     }
9     public void triple() {
10        setPerimeter(3 * perimeter);
11    }
12    public abstract void draw();
13 }
14 class Square extends Shape {
15     protected int x,y;
16     public Square(int initPerimeter, int x, int y) {
17         this.x = x;
18         this.y = y;
19         perimeter = initPerimeter;
20     }
21     public void draw() {
22         System.out.println("Desenhando Square com perimetro" + perimeter);
23     }
24 }

```



```

25 class Circle extends Shape {
26     protected int x,y;
27     double radius;
28     public Circle(int initPerimeter, int x, int y) {
29         this.x = x;
30         this.y = y;
31         perimeter = initPerimeter;
32         radius = initPerimeter/(2*Math.PI);
33     }
34     public void draw() {
35         System.out.println("Desenhando Circle com perimetro" + perimeter);
36     }
37 }
38 class Frame {
39     Shape shapeObj;
40     public void drawFrame() {
41         if(shapeObj.getPerimeter()>10)
42             shapeObj.triple();
43         shapeObj.draw();
44     }
45     public Shape setShapeObj(int shape, int perimeter, int x, int y) {
46         if ( shape == 1)
47             shapeObj = new Square(perimeter,x,y);
48         else
49             shapeObj = new Circle(perimeter,x,y);
50         return shapeObj;
51     }
52 }
53 class Teste {
54     public static void main(String[] args) {
55         Frame frameObj = new Frame();
56         int shape = Integer.parseInt(args[0]);
57         int perimeter = Integer.parseInt(args[1]);
58         int x = Integer.parseInt(args[2]);
59         int y = Integer.parseInt(args[3]);
60         frameObj.setShapeObj(shape,perimeter,x,y);
61         frameObj.drawFrame();
62     }
63 }

```

Listagem 2.1: *Exemplo de programa Java*

Para aplicar os critérios, devemos primeiramente obter as definições e usos dos objetos. Pegando como exemplo o objeto *shapeObj* do código em questão (Listagem 2.1), por meio de uma simples análise do código (i.e., sem a necessidade da construção do OCFG) obtemos estes dados. Na Tabela 2.1, as linhas onde ocorrem as definições e usos de *shapeObj* são apresentadas.

Tabela 2.1: *Definições e usos do objeto shapeObj do exemplo*

Objeto	Definição	Uso
<i>shapeObj</i>	42, 47, 49	41, 42, 43

Para selecionarmos os casos de testes baseados nestes critérios, o segundo passo é analisar o fluxo de objetos, identificando as ocorrências de *bindings* e pares du. A Tabela 2.2 apresenta os pares du que devem ser exercitados para atender aos critérios de fluxo de objetos.

Tabela 2.2: *Critérios e pares du*

Critério	Pares du
<i>All-bindings</i>	(42,41), (42,43), (47,41), (47,42), (47,43), (49,41), (49,42), (49,43)
<i>All-du-pairs</i>	(42,41), (42,43), (47,41), (47,42), (47,43), (49,41), (49,42), (49,43)

Neste caso em particular, os pares du a serem exercitados em ambos os critérios coincidiram. Entretanto, há casos em que o critério *all-bindings* pode exigir mais pares, como no caso de um comando utilizar mais de um objeto. Nesta situação, é necessário exercitar todas as possíveis combinações adicionais exigidas por este critério. Para exemplificar, suponha que temos no código da Listagem 2.1 a seguinte linha:

```
System.out.println("Soma dos perímetros = " + shape1.getPerimeter() + shape2.getperimeter());
```

Para aplicar o critério *all-bindings*, seria necessário testar o comando utilizando dois objetos *Circle*, dois objetos *Square* e a combinação entre um *Square* e um *Circle*.

2.8 Critérios de Teste para o Comportamento Excepcional de Programas

Atualmente, para a análise de um programa ser completa, ela deve levar em consideração os mecanismos para tratamento de exceções. Estes mecanismos, existentes em linguagens como Java e C++, passaram a ser utilizados em grande parte dos programas existentes [SH98] e, por isso, se não forem considerados durante a análise e teste estrutural de programas baseados em fluxo, os percentuais de cobertura que informam o quanto do conjunto de fluxo de controle ou dados do programa foram cobertos pelo *test suite* podem estar incorretos.

Harrold e Sinha [SH99] apresentam critérios de teste baseados em fluxo de dados, onde definem conceitos de definição e uso para exceções. Além disso, eles apresentam uma técnica para análise de fluxo de exceções [SH98] para programas Java, que facilita a identificação das definições e uso para aplicação dos critérios.

A seguir vamos apresentar estes critérios de teste. Para isso, é preciso conhecer um pouco melhor a estrutura de exceções da linguagem Java, que é utilizada nas técnicas de análise de fluxos de exceções.

2.8.1 Estrutura de exceções em Java

As exceções [Eck02, Gak07, Suna, Xav08] são estruturas utilizadas para descrever e tratar uma situação anormal (evento excepcional) que ocorre durante a execução de um programa. No momento em que ocorre uma exceção, o fluxo de execução do aplicativo é transferido do local onde surgiu a exceção (lançamento da exceção) até um ponto que pode ser definido pelo programador (captura da exceção) [GJSB05b], para que ela possa ser tratada. Esse tratamento consiste, em geral, num conjunto de instruções que permite ao programa recuperar-se da situação excepcional.

Códigos que permitem a recuperação do sistema na ocorrência de exceções são bastante frequentes, sobretudo em linguagens como Java, que possuem estruturas específicas para isso [SH98]. Isto pode ser explicado devido ao fato da utilização correta e consciente destes mecanismos fazer com que o programa seja mais robusto e tolerante a falhas, o que é essencial no mundo atual.

- **Mecanismos para Tratamento de Exceções em Java**

Em Java, há quatro elementos principais utilizados para tratar exceções: *try*, *catch*, *throw* e *finally*. Sua sintaxe para utilização é apresentada a seguir.

```
1  try {
2    // bloco de código com ocorrência de exceção.
3  }
4  catch (ExceptionType1 e1) {
5    // tratamento da exceção do tipo ExceptionType1.
6  }
7  catch (ExceptionType2 e2) {
8    // tratamento da exceção do tipo ExceptionType2.
9  }
10 catch (Exception e3) {
```

```

11  // tratamento de todos os tipos de exceção.
12  }
13  finally {
14  // código executado ao final do bloco try
15  // ou ao final da sequência de blocos try-catch,
16  // independentemente da ocorrência de exceções em try.
17  }

```

Listagem 2.2: *Sintaxe da construção de tratamento de exceções em Java [SH98].*

Try define um bloco de código em que uma ocorrência excepcional pode acontecer. Este bloco pode conter um comando *throw*, e vir acompanhado de um conjunto de blocos definidos pelo elemento *catch* e de um bloco definido por *finally*.

Throw é utilizado para lançar uma exceção de forma explícita. *Catch*, por sua vez, é um elemento que recebe como parâmetro um único tipo de exceção e define um bloco que a trata. Esta exceção é, em geral, lançada dentro de um bloco *try*, que pode estar no mesmo método ou não do bloco *catch*.

O bloco definido por *finally* é opcional, e caso exista, é o último elemento da sequência de blocos *try-catch* ou vem logo após um bloco *try*. O código descrito nele (em geral, realizando liberação de recursos adquiridos no bloco *try* correspondente) é executado obrigatoriamente, independentemente da execução do programa ter tido uma ocorrência excepcional ou não. No caso do bloco *try* ser finalizado com um comando de transferência de controle (*return*, *break* ou *continue*), o bloco *finally* (se houver algum) sempre será executado.

Além dos quatro elementos para tratamento de exceção descritos acima, Java define uma instrução *throws* que deve ser utilizada na declaração de um método quando este pode lançar exceções para o código invocador (exceções implícitas). Esta instrução não é considerada na definição dos critérios, mas é importante para a definição das propriedades baseadas em exceções implementadas no Java PathFinder Exceptions [Xav08].

2.8.2 Termos utilizados

Aqui, apresentamos algumas definições de termos a serem utilizados no critério de teste. Para exemplificá-los, usamos o código Java presente na Listagem 2.3.

- **Tipo de Exceção:** Classe da exceção. No exemplo, temos a classe *Exception*.
- **Objeto de Exceção:** Instância de uma classe de exceção. Representamos por *eobj_i*, onde *i* representa a linha onde o objeto foi instanciado. No exemplo, temos um objeto de exceção instanciado na linha 11, e o representamos por *eobj₁₁*.
- **Variável de Exceção:** variável cujo tipo é um tipo de Exceção. No exemplo, temos a variável *e*.
- **Variável temporária de exceção:** variável de exceção associada a alguma cláusula *throw*. Representamos a variável por *evan_i*, onde *i* representa a linha do respectivo *throw*. No exemplos temos uma variável temporária de exceção definida pelo comando *throw* na linha 11 (*evan₁₁*).
- **Objeto de exceção ativo:** objeto de exceção lançado através de uma cláusula *throw*. Em qualquer instante só pode existir um único objeto de exceção ativo. Representado por *evan_{active}*. Em cada instante do programa esta variável pode ter o valor *null*, quando não existir nenhuma exceção não tratada, ou algum objeto *eobj_i*. No exemplo, o objeto de exceção é inicializado na linha 11.

2.8.3 Definição e uso de exceções

Harrold e Sinha [SH99] estabeleceram os seguintes conceitos para definição e uso de exceções: Exceções são **definidas** nas seguintes condições:

1. Um valor é associado a uma variável de exceção.
EX: `Exception e = new Exception();`
2. Em um nó *catch* um valor é associado a uma variável de exceção e *evvar_{active}* é associado a `null`.
EX: `catch(Exception e)`
3. Em um nó *throw* que utiliza uma variável temporária de exceção *evvar*.
EX: `throw new Exception();`
4. Em um nó *throw* é feita uma associação a *evvar_{active}*.
EX: `throw new Exception();` OU `throw e;`

Exceções são **utilizadas** nas seguintes condições:

1. O valor de uma variável de exceção é acessado.
EX: `System.out.println(e);`
2. Em um nó *catch* o valor de *evvar_{active}* é acessado.
EX: `catch(Exception e)`
3. Em um nó *throw* com uma variável temporária de exceção.
EX: `throw new Exception();`

Um **conjunto definição-uso para uma exceção (e-du)** é uma tripla (v, i, j) , onde tem-se i a linha de definição da variável v , e j a de seu uso.

Para o exemplo da Listagem 2.3 temos as seguintes triplas e-du:

- $(e, 4, 5)$
- $(evvar_{11}, 11, 11)$
- $(evvar_{active}, 11, 4)$
- $(evvar_{11} \rightarrow e, 11, 5)$

2.8.4 Ativação e desativação de exceções

Além das definições e usos, exceções apresentam um outro conceito associado a ativação e desativação [SH99], ou seja, quando um objeto de exceção é associado ou desassociado a *evvar_{active}*.

Uma cláusula *throw* ativa uma exceção e uma cláusula *catch* desativa uma exceção. Além disso, uma exceção pode ser desativada dentro de uma cláusula *finally* com o uso de *return*, *break*, *continue* ou outra cláusula *throw*.

Um conjunto ativação-desativação para uma exceção $(e-ad)$ é uma tripla (v, i, j) onde tem-se i a linha de ativação do objeto v , e j a de sua desativação.

Para o exemplo do item anterior temos as seguintes triplas e-ad:

- $(eobj_{11}, 11, 4)$

2.8.5 Critérios de Teste

Os critérios de teste exigem que sejam realizadas operações com dados sobre o programa de forma a exercitar os conjuntos definição-uso e ativação-desativação. Ou seja, devem ser gerados casos de teste com dados que percorram as triplas definidas anteriormente.

A seguir, descrevemos os critérios definidos por Sinha e Harrold [SH99] que exercitam esses conjuntos definição-uso e ativação-desativação de exceções em programas Java:

- ***all-throw***: devemos varrer todos os nós *throw*.
- ***all-catch***: devemos varrer todos os nós *catch*.
- ***all-e-defs***: devemos varrer os caminhos de programa de forma a cobrir todas as definições de exceções e pelo menos um uso de cada uma.
- ***all-e-use***: devemos varrer todas as triplas definição-uso das exceções.
- ***all-e-act***: devemos varrer os caminhos de programa de forma a cobrir todas as ativações de exceções e pelo menos uma desativação de cada uma. É análogo ao critério *all-e-defs*, trocando as definições e usos por ativações e desativações.
- ***all-e-deact***: devemos varrer todas as triplas ativação-desativação das exceções.

2.8.6 Grafo de Fluxo de Controle para Mecanismos de Tratamento de Exceção

Como dito em seções anteriores (ver Seções 2.6.2 e 2.7), a criação de um grafo de fluxo de controle é interessante para a identificação das definições e usos de objetos, principalmente para programas maiores, em que se torna muito difícil tal identificação por meio da análise direta do código fonte. O mesmo vale para os mecanismos de tratamento de exceção: a geração de um grafo que represente adequadamente o fluxo de propagação e tratamento das exceções é um grande facilitador no momento dos testes.

Chen e Kao [CK99] descreveram o grafo de fluxo de objetos (OCFG), sucintamente descrito na Seção 2.7. Sinha e Harrold [SH98] propuseram um grafo de fluxo de controle semelhante para programas Java, mas com a capacidade de representar o fluxo das exceções apropriadamente.

Para representar o fluxo excepcional no OCFG, adaptamos a técnica apresentada por Sinha e Harrold [SH98], e adotamos os seguintes passos:

1. Blocos *try* são iniciados por um nó contendo apenas o comando “try”. Os demais nós do bloco são representados conforme o fluxo normal/excepcional do programa.
2. Para cada nó do OCFG em cuja instrução ocorre um lançamento de exceção (*throw*), uma aresta de saída é gerada.
3. Caso a exceção seja capturada por uma cláusula *catch* no mesmo método, esta aresta será cheia e será conectada ao nó correspondente; caso a captura ocorra num método distinto, a aresta a ser conectada ao nó correspondente será tracejada.
4. Blocos *finally* são tratados como se fossem métodos, possuindo nós de entrada e saída.

2.8.7 Exemplo

Na Listagem 2.3 encontra-se um exemplo simples com um trecho de código que inclui o uso de todas as instruções de tratamento de exceções em Java é ilustrado. Este código será utilizado para exemplificar a aplicação dos critérios de teste mencionados nesta seção.

```

1 public static void main(String[] args) {
2     try {
3         m1();
4     } catch ( Exception e ) {
5         System.out.println(e);
6     }
7 }
8 public static void m1() throws Exception{
9     try {
10        if (new Random().nextInt(10) > 5)
11            throw new Exception();
12        else System.out.println("Mensagem bloco try");
13    } finally {
14        System.out.println("Mensagem bloco finally");
15    }
16 }
17 }

```

Listagem 2.3: Código Java com tratamento de erros extraído de [Xav08]

Neste exemplo temos uma chamada (linha 3) a um método denominado *m1* feita dentro de um bloco protegido de código (bloco delimitado por uma instrução *try*). Este método declara que pode lançar uma exceção (linha 8) por meio da instrução *throws*. Para captura desta exceção temos o bloco *catch* que vai da linha 4 à linha 6.

Dentro do método *m1* temos a geração de um número aleatório entre 0 e 10, que pode gerar o lançamento de uma exceção (linha 11), caso seja maior do que 5. Além disso, temos um bloco de execução incondicional delimitado pela instrução *finally* (linhas 13 a 15).

Para ilustrar a aplicação dos critérios de testes para os mecanismos de tratamento de exceções, apresentamos as Tabelas 2.3 e 2.4. Elas definem as triplas a serem percorridas para cada um dos critérios de teste, com base no exemplo de código da Listagem 2.3.

Tabela 2.3: Critérios baseados em definição-uso

critério	tripla e-du
all-throw	($evar_{11}$, 11 , 11)
all-catch	(e, 4, 5)
all-e-defs	(e, 4, 5), ($evar_{11}$, 11 , 11), ($evar_{active}$, 11 , 4)
all-e-use	(e, 4, 5), ($evar_{11}$, 11 , 11), ($evar_{active}$, 11 , 4), ($evar_{11} \rightarrow e$, 11, 5)

Tabela 2.4: Critérios baseados em ativação-desativação

critério	tripla e-ad
all-e-act	($eobj_{11}$, 11, 4)
all-e-deact	($eobj_{11}$, 11, 4)

2.9 Ferramentas para Testes

A automação de algumas etapas das atividades de testes se fazem necessárias para que este processo seja viável na prática de desenvolvimento de software, como foi dito na Seção 2.3. Esta automação se faz ainda mais necessária no caso dos testes *white-box*, em que a construção de grafos de fluxo de controle e da consequente análise do código implementado, consistem em tarefas bastante complexas e demoradas quando estas devem ser aplicadas a programas de maior porte.

Por serem os testes estruturais (sobretudo os que consideram critérios de controle de fluxo aplicados a programas Java) um dos focos principais deste trabalho, descrevemos a seguir ferramentas

que existem atualmente para a automação destes testes em software. Elas são separadas em duas categorias: ferramentas para análise da cobertura de código e ferramentas para geração de requisitos de teste.

2.9.1 Ferramentas para Cobertura de Código

A cobertura de código é uma abordagem baseada nos testes estruturais, e consiste em analisar a quantidade de código coberto pelos testes. Quanto maior a cobertura, maior a probabilidade de se encontrar erros e corrigi-los. E conseqüentemente, maior a confiança construída no código implementado, uma vez que todos os testes sejam executados e não encontrem erros no programa.

A seguir, apresentamos algumas ferramentas utilizadas para automatizar a análise da cobertura de código de programa Java. Em geral, elas fazem este levantamento considerando os testes unitários feitos com o auxílio do *framework* JUnit⁴ [JUn].

- **EMMA/EclEmma** [EMM]: é um kit de ferramentas escrito em Java para a medição de cobertura de código Java. Pode ser usado tanto em códigos de pequeno porte, quanto códigos de grande porte. Suas principais funcionalidades são:
 - Instrumentação do *bytecode* do código para análise de cobertura dos testes. A instrumentação pode ser desde arquivos `.class` individuais até um arquivo `.jar` completo;
 - Apresentação da cobertura de código de classe, método, linha e blocos, feita por meio da análise da execução de um conjunto de testes unitários, em geral escritos e executados com o auxílio do JUnit. A verificação do EMMA detecta inclusive a cobertura parcial de uma única linha de código;
 - As estatísticas de cobertura são agregadas nos níveis de método, classes, pacotes e “todas as classes”;
 - Os resultados da cobertura podem ser exportados nos seguintes tipos de relatório: texto, HTML ou XML;
 - A execução do EMMA não sobrecarrega o sistema, e apresenta os resultados com bastante rapidez.

Para um uso integrado com a IDE Eclipse [Eclb], foi desenvolvido o plugin EclEmma [Ecla], que possui basicamente as mesmas propriedades do EMMA.

- **Coverlipse** [Cov]: é um plugin da IDE Eclipse, em que os resultados de cobertura são dados diretamente após a execução dos testes com o JUnit. Suas funcionalidades são:
 - Visualização da cobertura de código dos testes JUnit por meio dos critérios de blocos de instruções, e “todos os usos”;
 - Apenas uma única execução é necessária para a avaliação de todos os critérios de cobertura;
 - Fácil adição/remoção de pacotes do teste;
 - O feedback se dá diretamente no editor da IDE, junto com uma explicação dos resultados obtidos sob um ponto de vista especializado;
 - Não é necessário aprender novos métodos de configuração para a utilização do plugin. Basta configurá-lo do mesmo modo que o JUnit para sua execução normal na IDE Eclipse.

⁴O *framework* JUnit é utilizado para auxiliar a escrita de testes unitários em programas Java, e executar o *test suite* escrito. Ele é amplamente usado pelos desenvolvedores Java, e os testes unitários escritos com ele são focados na verificação das funcionalidades implementadas no código do programa (i.e., nenhum critério mais específico e rigoroso é adotado, como no caso dos testes estruturais.)

- **CodeCover** [Cod]: é uma ferramenta também integrada à IDE Eclipse. Suas funcionalidades são:
 - Visualização dos critérios de cobertura: por instrução, por ramificação, por laço e por decisão;
 - Faz a instrumentação do código de modo a obter uma medição mais precisa das estatísticas de cobertura;
 - Geração de relatório customizável no formato HTML e CSV;
 - Medida de cobertura por caso de teste;
 - Integração com o JUnit para reconhecimento automático dos casos de teste.

2.9.2 Ferramentas para Geração de Requisitos de Teste

Requisitos de teste compreendem os dados que devem ser exercitados por meio de testes, de maneira que critérios de teste sejam satisfeitos. Dentre os critérios de teste, podemos citar o de fluxo de objetos e os que englobam tratamento de exceção (ver Seções 2.7 e 2.8 respectivamente).

A seguir, descrevemos algumas ferramentas disponíveis para automação de testes estruturais que possuem, como uma de suas funcionalidades, a geração de requisitos para determinados critérios de teste.

- **POKE-TOOL (Potential Uses Criteria Tool for Program Testing)** [Cha91, MCJ89, DMJ07]: é uma ferramenta desenvolvida na FEEC-Unicamp para apoio ao teste de programas nas linguagens C, Fortran e COBOL. Suas funcionalidades são:
 - Instrumentação do código recebido, viabilizando o rastreamento do caminho de execução dos testes e uma posterior avaliação da adequação de um dado conjunto de teste;
 - Geração dos requisitos de teste (nós, arestas, caminhos ou pares du), feitos de acordo com o critério que o usuário deseja testar. Dentre os critérios disponíveis, estão: todos-nós, todas-arestas e os critérios da família Potenciais-Usos⁵ (todos-potenciais-usos, todos-potenciais-usos/du e todos-potenciais-du-caminhos), aplicados por meio de uma interface gráfica. Por linha de comando, alguns dos critérios de Rapps e Weyuker [RW82, RW85] para teste de fluxo de dados estão disponíveis, como os critérios todos-usos (em que são considerados todos os usos de variáveis), todos-c-usos (em que são considerados todos os usos computacionais de variáveis), e todos-usos (em que são considerados todos os usos predicativos de variáveis).
 - Execução e avaliação de casos de teste de acordo com o critério selecionado. O resultado da avaliação é o conjunto de elementos requeridos que restam ser executados para satisfazer o critério e o percentual de cobertura obtido pelo conjunto de teste. Também são fornecidos o conjunto de elementos que foram executados, as entradas e saídas, bem como os caminhos percorridos pelos casos de teste. O processo de execução/avaliação deve continuar até que todos os elementos restantes tenham sido satisfeitos, executados ou detectada a sua não executabilidade.
- **DaTeC (*Data-flow Testing of Classes*)** [DGP09, DGP08]: é um protótipo construído para auxiliar o teste de fluxo de dados de programas Java, por meio da instrumentação de seu *bytecode*. Desenvolvido em conjunto pela Universidade de Milão Bicocca e Universidade de Lugano, suas principais funcionalidades são:

⁵O critério Potenciais-Usos [Mal91], distingue-se dos critérios de fluxo de dados por considerarem os chamados “potenciais usos” de variáveis, que são nós nos quais é possível chegar por meio de um caminho livre de redefinição a partir de um nó de definição de uma variável, independente de haver uma referência explícita a um uso desta variável.

- Instrumentação do *bytecode* Java para identificação das associações de definição-uso contextuais. O conceito de associações de definição-uso contextuais é apresentado por Souter e Pollock [SP03], e consiste em encontrar o contexto de cada definição e uso do objeto e formar os pares du correspondentes;
 - Apresentação dos dados de cobertura do fluxo de dados contextual, mostrando desde todas as associações cobertas e não cobertas, até um resumo dos pares cobertos para classes selecionadas pelo usuário.
- **JaBUTi (*Java Bytecode Understanding and Testing*)** [VMWD05, DMJ07]: é uma ferramenta desenvolvida no ICMC-USP, que utiliza a análise de *bytecode* Java para gerar requisitos para testes estruturais intra métodos (que focam na detecção de erros dentro de cada método individualmente) que utilizam critérios de fluxo de controle de dados. Suas funcionalidades são:
 - Análise do *bytecode* para geração de requisitos de teste para os critérios: todos-nós, todas-arestas, todos-usos e todos-potenciais-usos;
 - Geração de um grafo de fluxo de dados para cada método do código;
 - Uso de um *test driver* para instrumentação das classes e carregamento e execução do programa instrumentado;
 - Visualização da cobertura, para cada método, dos testes executados;
 - Geração de estatísticas de cobertura e métricas de qualidade dos testes executados.
 - **OConGraX (*Object Control-Flow Graph with Exceptions*)** [HM07, NHdM09]: é uma ferramenta desenvolvida no IME-USP, escrita em Java para programas Java. Suas funcionalidades são:
 - Identificação das definições, usos e pares du de objetos e exceções (requisitos para os critérios de fluxo de objetos de Chen e Kao [CK99], e para os critérios que envolvem os mecanismos para tratamento de exceções Java de Sinha e Harrold [SH99]);
 - Armazenamento num arquivo XML com as informações de definições, usos e pares du de objetos e exceções;
 - Geração do OCFG com informações adicionais sobre os mecanismos de tratamento de exceções utilizados;
 - Geração de um arquivo de imagem contendo o OCFG com as informações de mecanismos de tratamento de exceções.

2.9.3 Justificativa da Escolha da OConGraX

Dentre as ferramentas descritas para testes, a **OConGraX** foi a escolhida para este trabalho. Ela se diferencia das demais por satisfazer os seguintes itens:

- **Utilização de critérios específicos para geração de dados a serem utilizados:** a implementação da geração de requisitos de teste foi baseada em critérios específicos para testes estruturais, oferecendo um nível mais formal de seleção de casos de teste em comparação às ferramentas de cobertura que fazem uso do *framework* JUnit, no qual os testes unitários escritos são testes funcionais e que não seguem necessariamente um critério.
- **Geração de requisitos considerando o critério de fluxo de objetos:** na OConGraX, as definições, usos e pares du são obtidos por meio da análise do código e do OCFG gerado, levando em conta o fluxo dos objetos no programa para a posterior aplicação dos critérios apresentados por Chen e Kao [CK99]. Isto a difere, por exemplo, da JaBUTi, na qual os requisitos de teste são gerados levando em consideração o fluxo de variáveis.

- **Geração de requisitos considerando o comportamento excepcional dos programas:** um dos principais diferenciais da OConGraX é a geração de requisitos de teste considerando não apenas o fluxo dos objetos e o comportamento normal do programa, mas também considerando o fluxo das exceções e o comportamento excepcional deles [SH98, SH99]. A geração de requisitos levando em conta as exceções é uma funcionalidade pouco oferecida pelas ferramentas existentes. O protótipo DaTeC, apesar de oferecer requisitos e dados de cobertura baseados no conceito novo de associações definição-uso contextuais, não estende esta funcionalidade para exceções.
- **Linguagem Java:** o código analisado pela OConGraX está escrito na linguagem Java, linguagem esta que usa o paradigma OO e possui estruturas adequadas para tratamento de exceções. Estas duas características são essenciais para a proposta deste trabalho, o que torna a OConGraX a melhor escolha para ferramenta de teste. A POKE-TOOL não serviria para este fim, uma vez que ela é voltada para teste de programas escritos em linguagens imperativas, como C.

Uma descrição mais minuciosa da ferramenta OConGraX e suas funcionalidades é dada no Capítulo 6 deste documento.

2.10 Conclusão

Neste capítulo, abordamos o conceito e a importância dos testes, elemento importante da abordagem de validação e verificação. Mostramos suas fases e as estratégias existentes para este fim, as técnicas para projetar casos de teste e os critérios de teste. Entender o processo de teste de software é essencial para este projeto, uma vez que um dos objetivos principais deste trabalho é o de validar código Java por meio desse processo, utilizando a ferramenta escolhida para esta finalidade, a OConGraX [HM07, NHdM09], conforme visto na Seção 2.9.3.

No próximo capítulo, apresentamos o conceito complementar ao de testes, utilizado na abordagem V&V: o da verificação de programas.

Capítulo 3

Verificação de Programas

Neste capítulo apresentamos a teoria relacionada à verificação de programas utilizada neste trabalho. Os seguintes tópicos teóricos são abordados aqui:

- o conceito de verificação de programas;
- as etapas do processo de verificação de um software;
- as técnicas para a realização da verificação de programas.

Finalizando o capítulo, uma descrição de alguns dos verificadores existentes para automação desta atividade é feita.

3.1 Conceito de Verificação de Programas

Como visto no Capítulo 2, testes são bons instrumentos para detectar erros e, portanto, constituem um processo importante para construir a confiança de um programa, tanto para os desenvolvedores quanto para os clientes finais. Entretanto, eles não são capazes de demonstrar que um software está completamente livre de erros: apesar de todo esforço para se construir e selecionar casos de teste que cubram todas as funcionalidades e código do programa, pode haver algum *bug* que tenha escapado ao processo. Isto ocorre porque os testes não são capazes de demonstrar a ausência de erros no software [Dij70], o que pode ser tornar um problema grave no caso de sistemas complexos de grande porte e sistemas críticos, uma vez que uma falha nestes sistemas pode ter efeitos desastrosos (ver Seção 2.2).

Para suprir esta necessidade de demonstrar a corretude de um programa, surgiu o conceito de verificação de programas. A verificação de programas faz uso de métodos formais para provar que um programa foi implementado corretamente.

3.1.1 Métodos Formais

De acordo com a Enciclopédia de Engenharia de Software [Mar94], métodos formais podem ser assim definidos: como técnicas baseadas matematicamente para a descrição de propriedades do sistema. Eles provêm *frameworks* nos quais as pessoas podem especificar, desenvolver e verificar sistemas de uma forma ordenada.

Ainda segundo Marciniak [Mar94], um método é formal se ele possui base matemática, dada tipicamente por uma linguagem de especificação formal. Esta base provê um meio de definir precisamente noções como consistência, completude, especificação, implementação e corretude.

Para a verificação de programas, estes precisam ser especificados formalmente utilizando metodologias e linguagens formais. Exemplos destes métodos ou linguagens são Z [WD96], B [Wor96] e VDM [Jon90] para sistemas sequenciais; para sistemas concorrentes, CCS [Mil89] e CSP [Hoa85]; e para agentes móveis, π -calculus [Mil99].

A utilização de métodos e especificações formais de programas, além de serem necessárias para a realização de sua verificação, são importantes pois evitam problemas como ambiguidade e contradições, que podem atrapalhar todo o desenvolvimento do software. Ademais, os métodos formais podem ser usados no desenvolvimento de software (ex.: abordagem de desenvolvimento de software “cleanroom” [Pre01, Som06]), além de serem altamente recomendados pelas agências espaciais como ESA e NASA para o desenvolvimento de sistemas críticos seguros [BK08].

3.1.2 Definição de Verificação

Verificação de programas consiste em conferir se o software está sendo desenvolvido corretamente [Boe81]. De um modo mais formal, podemos defini-la da seguinte maneira: dada uma especificação φ e um programa P , mostrar que P é um modelo para a especificação φ (i.e., P satisfaz φ) [Fra92, MSF06]. Esta verificação também é conhecida como verificação *a posteriori*, pois a especificação e o programa são fornecidos. Nela, devemos ter formas de comparar a especificação com o programa para que possamos provar que este último é um modelo para a especificação [MSF06].

Há duas abordagens principais para a realização da verificação de programas: a de verificação de modelos também conhecida como *model checking*; e a de sistema de prova de programas. Detalhes sobre estas abordagens são dadas na Seção 3.3.

3.1.3 Limitações

A utilização de métodos e verificação formais no desenvolvimento de software, apesar de garantir uma implementação mais segura, enfrenta uma certa resistência para sua aplicação real na prática. Isto ocorre pois muitos consideram esta metodologia demasiadamente complicada e complexa. Além disso, ela ainda está sujeita à falha humana: a especificação de um programa deve ser feita por pessoas, e não há como garantir que esta especificação realmente corresponde ao que o software deve fazer [Fra92].

3.2 Etapas para a Verificação de Programas

Para a realização da verificação de programas, de um modo geral, é preciso seguir algumas etapas [BK08] (uma visão esquemática deste processo pode ser visto na Figura 3.1). Estas etapas são discutidas a seguir:

Figura 3.1: Esquema do processo de verificação de um sistema *a posteriori* [BK08].

1. **Especificação do Sistema:** constitui a base para qualquer atividade de verificação, pois descreve de maneira formal as funcionalidades que devem estar implementadas no programa.

Para a realização desta especificação, devem ser utilizados um dos métodos ou linguagens formais disponíveis para o tipo de sistema a ser implementado, de modo a descrever o sistema da forma mais precisa e não ambígua possível [MSF06].

2. **Definição de Propriedades:** a partir da especificação, deve-se definir propriedades que o produto/protótipo desenvolvido precisa satisfazer. Um exemplo de propriedade, seria a não ocorrência de *deadlocks* no sistema.
3. **Desenvolvimento do Produto/Protótipo:** corresponde ao processo de planejamento e implementação do produto/protótipo.
4. **Verificação do Programa:** com o produto/protótipo pronto para execução, e as propriedades já definidas, é feita a verificação propriamente dita. Caso o produto/protótipo não satisfaça alguma das propriedades de especificação, significa que um erro foi encontrado e, portanto, o programa não está correto. Para fazer esta verificação, podem ser usados verificadores, que são ferramentas que automatizam esta etapa.

3.3 Abordagens para a Verificação de Programas

Para fazer a verificação de programas, diversas abordagens e técnicas foram desenvolvidas ao longo dos anos. Dentre essas, duas se destacam:

- **Verificação de modelos (*model checking*) [BK08]:** técnica baseada em modelos de sistemas¹ que explora todos os possíveis estados do sistema de maneira exaustiva. O verificador de modelos (*model checker*), ferramenta que faz esta abordagem de verificação, examina todos os possíveis cenários do sistema, de uma maneira sistemática. Desta forma, pode-se mostrar que um modelo de sistema realmente satisfaz uma certa propriedade.
- **Sistema de prova de programas [MSF06]:** técnica na qual a corretude dos programas é verificada por meio da utilização de um sistema de provas de um dos métodos ou linguagens formais utilizadas para a especificação do sistema, como Z [WD96], VDM [Jon90], B [Wor96] e lógica de Hoare [Hoa69].

A seguir, descrevemos os procedimentos necessários para a aplicação de cada uma das técnicas. No caso particular da técnica de sistema de prova de programas, vamos explicar o procedimento para sua aplicação com a lógica de Hoare.

3.3.1 Sistema de Prova de Programas

De acordo com Melo et al. [MSF06], para aplicar esta técnica de verificação de programas precisamos de:

1. Uma especificação escrita em uma linguagem com fundamento matemático para que seja precisa e não ambígua;
2. O programa escrito em uma linguagem que tem o significado de seus comandos definido de forma precisa – semântica formal da linguagem de programação;
3. Uma forma de associar e comparar as asserções da especificação com os comandos do programa; as linguagens de especificação e programação devem, portanto, ser comparáveis;
4. Um sistema de prova que usamos para mostrar que um programa satisfaz uma dada especificação.

¹Os modelos descrevem o possível comportamento do sistema de uma maneira matematicamente precisa e não ambígua, e são acompanhados por algoritmos que sistematicamente exploram todos os estados dos modelos do sistema, fornecendo assim a base para diversas técnicas de verificação, dentre as quais a do *model checking* [BK08].

Aqui, vamos usar para exemplificar esta técnica a semântica da linguagem e o sistema de provas associado definidos por meio da lógica de Hoare [Hoa69]. Assim, a associação dos comandos dos programas com as asserções são feitas por meio das regras deste sistema de provas.

- **Procedimento para a Prova da Corretude de Programas** [MSF06]

Utilizando o sistema de provas da lógica de Hoare [Hoa69], queremos provar a validade da asserção:

$$\vdash \langle \varphi \rangle \text{Prog} \langle \psi \rangle$$

Nesta asserção, temos a tripla $\langle \varphi \rangle \text{Prog} \langle \psi \rangle$, também conhecida como Tripla de Hoare. Nela, φ corresponde às pré-condições, que são propriedades sobre o estado inicial do programa *Prog*. Já ψ representa as propriedades sobre o estado final do programa *Prog*, as pós condições. Desta maneira, uma interpretação para esta asserção dada por correção parcial seria:

*Para todo estado σ que satisfaz φ , se a execução de *Prog* a partir do estado σ termina produzindo o estado σ' , então σ' satisfaz ψ .*

Para mostrar a validade desta asserção, devemos utilizar provas de programas. Estas podem ser divididas em correção parcial e correção total.

- **Correção Parcial:** a tripla $\langle \varphi \rangle \text{Prog} \langle \psi \rangle$ é satisfeita sob correção parcial se, para todos os estados que satisfazem φ , o estado resultante da execução do programa *Prog* satisfaz a pós-condição ψ , se *Prog* termina. Neste caso, $\vdash_{par}$ é a relação de satisfazibilidade para a correção parcial: $\vdash_{par} \langle \varphi \rangle \text{Prog} \langle \psi \rangle$.

Na prática, correção parcial é um requisito ineficiente porque não garante a terminação do programa: qualquer programa que não termina satisfaz a sua especificação.

- **Correção Total:** a tripla $\langle \varphi \rangle \text{Prog} \langle \psi \rangle$ é satisfeita sob correção total se, para todos os estados que satisfazem φ , o estado resultante da execução do programa *Prog* satisfaz a pós-condição ψ , e *Prog* termina. Neste caso, $\vdash_{tot}$ é a relação de satisfazibilidade para a correção total: $\vdash_{tot} \langle \varphi \rangle \text{Prog} \langle \psi \rangle$.

Na correção total, a satisfazibilidade e a terminação do programa devem ser provados, o que é muito interessante na prática. Desta forma, qualquer problema que entra em um laço infinito de repetição não satisfaz sua especificação sob a relação de correção total. Ademais, pode-se dividir a tarefa de provar correção total de programas em: provar correção parcial e provar que o programa termina.

Para provar as triplas de Hoare, que foram definidas considerando o programa como um todo, sob quaisquer das duas formas de correção, devemos utilizar o sistema de provas. Este é definido sobre cada elemento sintático da linguagem, e as provas são realizadas usando indução sobre as estruturas dos programas. Isto é, as regras provam a correção de uma asserção para um comando mais complexo, pela prova de correção das asserções de seus sub-comandos. Assim, há dois elementos no sistema de provas: as regras de inferência sobre cada um dos elementos sintáticos dos programas e o mecanismo de prova utilizando as regras definidas. Ambos os elementos não serão descritos neste texto. Maiores informações sobre eles poderão ser obtidas nos livros de Hoare [Hoa69] e Melo et al. [MSF06].

- **Correção e Completude do Sistema de Provas** [MSF06]

Para provar a correção do sistema de provas, é necessária apenas a prova de correção para cada uma das regras (indução estrutural), já que o sistema é composicional. Esta prova pode ser encontrada no livro de Winskel [Win93].

Para provar a completude do sistema de provas, devemos usar um artifício: a introdução de asserções de programas como axiomas do sistema, a partir dos quais pode-se fazer uma prova de completude relativa do sistema de provas. Esta prova pode ser encontrada em no livro de Francez [Fra92].

3.3.2 Verificação de Modelos (*Model Checking*)

Model checking é uma técnica automatizada que, dado um modelo de estados finitos de um sistema e uma propriedade formal, checa sistematicamente se esta propriedade é satisfeita para um dado estado desse modelo. Esta verificação é feita utilizando força-bruta, em que o verificador varre todos os estados do modelo. Caso a propriedade não seja satisfeita, um contra-exemplo é apresentado, mostrando a localização da falha.

As propriedades a serem checadas usando esta técnica são de natureza qualitativa: elas verificam, entre outras coisas, se os resultados gerados são os esperados e se o sistema não entra em situação de deadlock. Outras propriedades que podem ser verificadas são as propriedades de tempo, que podem querer checar, por exemplo, se o tempo de resposta do sistema está dentro do período esperado.

Na Figura 3.2 é apresentado um esquema geral do processo de verificação de modelos. Este processo é discutido a seguir em maiores detalhes, uma vez que é esta a abordagem de verificação a ser utilizada neste trabalho de Mestrado.

Figura 3.2: Esquema do processo de model checking [BK08]

- **Procedimento para a Aplicação do *Model Checking*** [BK08]

Para aplicar esta técnica de verificação, as seguintes fases devem ser realizadas:

- Fase de Modelagem:
 - * Modelar o sistema em questão usando a linguagem de descrição do modelo do verificador de modelo a ser utilizado;
 - * Para uma primeira avaliação do modelo, fazer algumas simulações;
 - * Formalizar a propriedade a ser checada usando a linguagem de especificação de propriedades.
- Fase de Execução: executar o *model checker* para verificar a validade da propriedade no modelo do sistema.
- Fase de Análise:

- * Se a propriedade foi satisfeita, checar a próxima (se houver);
- * Se a propriedade foi violada, deve-se:
 1. Analisar o contra-exemplo gerado pela simulação;
 2. Refinar o modelo, *design*, ou propriedade;
 3. Repetir todo o procedimento.
- * Se a memória for insuficiente para a realização da verificação, tentar reduzir o modelo e repetir todo o procedimento.

Ademais, toda a verificação deve ser planejada, administrada e organizada.

• Verificadores de Modelos

Para trabalhar com verificadores de modelos, algumas abordagens podem ser adotadas [VHB⁺03, Xav08]:

- **Tradução:** ferramentas que funcionam exclusivamente como verificadores de modelos possuem linguagem própria para a descrição destes modelos, de modo que é necessária uma tradução entre a linguagem de programação e a de modelo. Exemplo de ferramenta que utiliza esta abordagem: SPIN [Hol97, SPI].
- **Criação de verificadores customizados:** permite a criação de seu próprio verificador de modelo, já adequado às características da linguagem de programação a ser utilizada. Esta abordagem é usada pela ferramenta Java PathFinder [VHB⁺03, Jav].

• Vantagens e Desvantagens da Verificação de Modelos

Como toda abordagem, há prós e contras para sua aplicação [BK08]. Eles devem ser analisados cuidadosamente para que esta técnica seja utilizada da melhor maneira possível, de modo a obter resultados satisfatórios com sua aplicação.

Listamos agora algumas das vantagens do *model checking*.

- É uma abordagem de verificação geral que é aplicável a uma grande variedade de aplicações, tais como sistemas embarcados, engenharia de software e *design* de hardware;
- Suporta verificação parcial, i.e., propriedades podem ser checadas individualmente, permitindo assim focar primeiro nas propriedades essenciais. Não é necessária uma especificação de requisitos completa;
- Provê informação de diagnóstico no caso da propriedade ser invalidada: isto é muito útil para o processo de *debugging*²;
- O uso de *model checking* não requer um alto grau de interação com o usuário e nem um alto grau de perícia;
- Goza de um interesse crescente pela indústria;
- Pode ser facilmente integrado a ciclos de desenvolvimento existentes;
- Tem suporte matemático: é baseado na teoria de algoritmos de grafos, estrutura de dados e lógica.

Embora as vantagens da verificação de modelos sejam significativas, deve-se considerar também suas desvantagens. Listamos a seguir alguns dos pontos fracos desta técnica de verificação.

- Sua aplicabilidade é sujeita a questões de “decisão”: para sistemas de estados infinitos, ou raciocínio sobre tipos de dados abstratos (que requer lógica indecidível ou semi-decidível), *model checking* não é eficientemente computável.

²*Debugging* é o nome que se dá ao processo de localizar a causa de um bug no software [Pre01]

- Esta técnica verifica um modelo do sistema, e não o sistema real (produto ou protótipo) em si; qualquer resultado obtido é, portanto, tão bom quanto o modelo de sistema. Técnicas complementares, como testes, são necessárias para ajudar a encontrar erros de código;
- Verifica apenas os requisitos estabelecidos, i.e., não há garantia de completude. A validade de propriedades que não foram verificadas não podem ser julgadas;
- Sofre do problema de explosão de espaço de estados, i.e., o número de estados necessários para modelar o sistema precisamente pode facilmente exceder a quantidade de memória disponível no computador. Apesar do desenvolvimento de diversos métodos efetivos para combater este problema, modelos de sistemas reais podem ser ainda muito grandes para caber na memória;
- Seu uso requer certa perícia em encontrar abstrações apropriadas para obter modelos de sistemas menores e para estabelecer propriedades no formalismo lógico utilizado;
- Não é garantido que ela renderá resultados corretos; como com qualquer ferramenta, o verificador de modelos pode conter defeitos de software, mesmo com testes.

3.4 Ferramentas para Verificação

Automatizar o processo de verificação de programas é algo extremamente útil e necessário para que sua aplicação seja realizada nos mais diversos programas e sistemas em desenvolvimento hoje. Para isso, várias ferramentas foram desenvolvidas. No nosso caso, particularmente, nos interessa as ferramentas que automatizam a verificação de modelos e, por esta razão, descrevemos a seguir algumas ferramentas usadas para este fim.

3.4.1 Ferramentas para *Model Checking*

Para realizar o *model checking*, diversos verificadores e ferramentas têm sido desenvolvidos, de modo a auxiliar na realização de tarefas como a extração de modelos de sistema, escrita das propriedades em linguagem específica, e a realização da verificação em si. Ademais, eles devem lidar adequadamente com os problemas desta abordagem, sendo o principal a explosão de estados³ do modelo a ser verificado.

Aqui, apresentamos algumas destas ferramentas e verificadores utilizados atualmente para a aplicação da verificação de modelos, mostrando suas principais características.

- **SPIN** (*Simple Promela INterpreter*) [Hol97, SPI]:

- Verificador de modelo desenvolvido pela Bell Labs que checa a consistência lógica de uma especificação, e pode ser usado para rastrear erros de design em sistemas distribuídos, tais como sistemas operacionais, protocolos de comunicação de dados e algoritmos concorrentes;
- O sistema a ser verificado deve estar descrito na linguagem de especificação PROMELA (PROCESS MEta LANGUAGE) [Hol91]; as propriedades, por sua vez, devem ser especificadas em fórmulas LTL (Lógica Temporal Linear) [Pnu77, Lam94, MP92];
- A verificação exaustiva é feita usando a teoria de redução de ordem parcial para otimizar a procura;
- Trabalha de maneira *on-the-fly*, o que significa que ele evita a necessidade de se pré-construir um grafo de estados global como pré-requisito para a verificação das propriedades do sistema.
- É escalável, tratando eficientemente problemas de grande porte;

³Segundo Hatcliff e Dwyer [HD01], explosão de estados consiste no crescimento exponencial do tamanho de um modelo de estados finito à medida que o número de componentes do sistema cresce.

- Oferece suporte ao uso de computadores com múltiplos processadores para execução da verificação de modelos.
- **FeaVer** (*Feature Verification System*) [Fea]:
 - Verificador de modelo desenvolvido pela Bell Labs, usado verificar propriedades no servidor de acesso da *Lucent'sPathStar*^(TM) [VHB⁺03];
 - O modelo do sistema é extraído do código em C e descrito em PROMELA [Hol91]; a verificação ocorre utilizando uma biblioteca de propriedades especificadas em LTL [Pnu77, Lam94, MP92];
 - Para a execução da verificação, é utilizado o SPIN [Hol97, SPI].
- **Bandera** [CDH⁺00, HD01?]:
 - Conjunto de ferramentas desenvolvido pelo Laboratório SAnToS da Universidade do estado do Kansas para a realização de *model checking* em software Java;
 - Dado o código-fonte de um programa Java, retorna o modelo do programa numa das linguagens de entrada dos vários verificadores existentes, como por exemplo a linguagem PROMELA para o verificador SPIN. Bandera também realiza o mapeamento do resultado do verificador para o código-fonte original.
 - Para fazer o processo de tradução da linguagem de programação para a linguagem do verificador, são utilizadas diversas técnicas de otimização como fatiamento, abstração de dados e restrição de componentes;
 - Permite a adição de novas propriedades. Estas devem ser definidas numa linguagem de especificação própria do Bandera, a linguagem BSL [HD01];
 - Modelos são gerados de acordo com a propriedade a ser testada;
 - Pode ser utilizada de modo integrado a IDEs Java (ex.: Eclipse), e a outras ferramentas de verificação de modelos (ex.: Java PathFinder);
 - Oferece suporte aos elementos introduzidos com Java 5.0 e ao tratamento de exceções.
- **JPF** (*Java PathFinder*) [VHB⁺03, Jav]:
 - Verificador de modelo desenvolvido pela NASA, que faz uma busca exaustiva de violações de propriedades dado um programa Java;
 - Para fazer a busca exaustiva, o JPF foi implementado de modo a funcionar de forma semelhante a uma máquina virtual Java (JVM), na qual o *bytecode* é analisado e todos os caminhos possíveis do programa são executados;
 - As propriedades são implementadas no JPF utilizando a própria linguagem Java e recursos do próprio código de implementação deste verificador;
 - É facilmente integrável com outras ferramentas e extensível, além de oferecer suporte aos elementos introduzidos com Java 5.0 e ao tratamento de exceções. Um exemplo de aplicação que estende o JPF é a **Java PathFinder Exceptions (JPF Exceptions)** [Xav08]: ela checa propriedades de tratamento de exceção num sistema Java utilizando o JPF, e estas propriedades já estão definidas e implementadas neste verificador. Ademais, ela gera estatísticas de cobertura relativas aos estados varridos durante a verificação.

3.4.2 Justificativa da Escolha do JPF/JPF Exceptions

Neste trabalho, optamos por trabalhar com o verificador **JPF** e a ferramenta **JPF Exceptions** por eles se diferenciarem das demais pelos seguintes motivos:

- **Linguagem Java:** neste trabalho de Mestrado, o uso de uma linguagem OO e que possua mecanismos para tratamento de exceção adequados e bem definidos é essencial. Por esta razão, ferramentas para programas procedimentais, como FeaVer, não foram escolhidos.
- **Suporte a tratamento de exceções:** assim como no caso da escolha da ferramenta OConGraX [HM07, NHdM09] (ver Seção 2.9.3), é importante que a ferramenta escolhida lide adequadamente com os mecanismos para tratamento de exceções, para verificar o comportamento do programa no caso de ocorrência de exceções durante a execução de algum de seus métodos. E isto é feito principalmente pela JPF Exceptions.
- **Facilidade na especificação de novas propriedades:** este é um dos pontos mais importantes que diferenciam a ferramenta e o verificador escolhidos das demais aplicações: no caso deles, as propriedades podem ser especificadas utilizando a própria linguagem Java, sem a necessidade de aprender linguagens novas e específicas para o verificador, como ocorre no caso do Bandera e do SPIN.

Uma descrição mais detalhada do verificador JPF e da ferramenta JPF Exceptions é dada no Capítulo 6 deste documento.

3.5 Conclusão

No presente capítulo, apresentamos o conceito, as etapas do processo e as técnicas da chamada verificação de programas. Abordamos suas características principais, bem como mostramos as ferramentas existentes para sua realização. Da mesma maneira que os testes de software, a verificação de programas é um conceito importante, cujo entendimento é de suma importância para a compreensão da abordagem de validação e verificação a ser utilizada neste projeto.

Para este trabalho, conforme visto na Seção 3.4.2, a ferramenta escolhida para exercer a função de verificação de código Java é a JPF Exceptions [Xav08]. Esta ferramenta, junto com a OConGraX [HM07, NHdM09], permitirá a construção do ambiente integrado de teste e verificação de coordenação de componentes tolerantes a falhas, objetivo maior deste trabalho.

No próximo capítulo, abordamos o conceito de coordenação de componentes tolerantes a falhas, que consiste no protocolo a ser testado e verificado no ambiente construído neste projeto.

Capítulo 4

Componentes e Ações Atômicas Coordenadas

Neste capítulo é abordada a teoria referente aos componentes tolerantes a falhas e sua coordenação. Os seguintes tópicos são apresentados:

- o conceito de DBC e componentes;
- tratamento de exceções nos componentes e na arquitetura;
- o conceito de ações atômicas coordenadas;
- uma implementação em Java do modelo de ações atômicas coordenadas.

4.1 Desenvolvimento Baseado em Componentes (DBC)

O DBC constitui, segundo Pressman [Pre01], um processo no qual são obtidos os requisitos do cliente, e o estilo da arquitetura do sistema é escolhido de modo a satisfazer os objetivos do produto a ser escolhido. Feito isso, os seguintes passos devem ser realizados:

1. Seleção de potenciais componentes para reuso;
2. Qualificação dos componentes para assegurar que eles se acomodam adequadamente à arquitetura do sistema;
3. Adaptação dos componentes, caso modificações precisem ser feitas para integrá-los apropriadamente;
4. Integração dos componentes para formar sub-sistemas e a aplicação como um todo.

Ademais, componentes personalizados podem ser construídos para satisfazer aspectos do sistema que não podem ser implementados utilizando os componentes existentes.

4.1.1 Componentes

Na área da computação, o termo **componente** possui várias definições. Para Szyperski [Szy02], um componente de software é definido como uma unidade de composição com interfaces¹ contratualmente² definidas e apenas dependências explícitas de contexto. São implantados de forma

¹Uma interface, segundo Bachmann et al [BBC⁺02] é uma fronteira na qual duas entidades independentes comunicam-se ou interagem entre si. Se a entidade for um componente, a interface especifica um ponto específico de acesso a determinado serviço.

²Um contrato [Mey92, LC02] descreve um conjunto de asserções para a caracterização precisa do comportamento de um componente. Ele também define, formalmente, as obrigações recíprocas de cada componente em uma determinada interação.

independente e estão sujeitos à composição por terceiros. De modo semelhante, Souza [DW99] define componente como uma parte reutilizável de software, desenvolvida independentemente e empregada na construção de unidades maiores, por meio de sua adaptação.

4.1.2 Padrões de Componentes

Devido ao potencial impacto da reutilização de componentes na indústria de software, empresas têm proposto padrões para os componentes de software. Estes padrões definem a maneira como uma aplicação pode acessar objetos reutilizáveis [Pre01], e visam a garantir aos desenvolvedores que os componentes padronizados sejam interoperáveis [Som06].

Os exemplos mais importantes destes padrões são: OMG/CORBA³, Microsoft COM⁴ e Sun JavaBeans Components⁵.

4.2 Sistemas Tolerantes a Falhas

Sistema tolerante a falhas corresponde a um software que incorpora técnicas para tratar adequadamente as exceções que podem ocorrer durante sua execução [XRR⁺95a]. No caso de sistemas concorrentes e que utilizam componentes, é essencial que este tratamento seja eficiente, de modo a manter as propriedades de concorrência e o bom funcionamento do software.

A seguir, apresentamos abordagens para o tratamento de exceções no componente, e na arquitetura do sistema.

4.2.1 Tratamento de Exceções no Componente

Nos componentes tolerantes a falhas que implementam mecanismos internos para o tratamento de exceções, as respostas são divididas em normais e excepcionais. Respostas normais são produzidas quando o componente realiza corretamente o serviço requisitado. Respostas excepcionais (exceções) são aquelas produzidas quando algum evento anormal ocorre durante a execução do serviço.

Desta forma, o processamento do componente tolerante a falhas deve ser particionado em atividade normal e atividade excepcional, conforme mostra a Figura 4.1. A atividade normal implementa os serviços especificados para o componente, enquanto a atividade excepcional implementa as rotinas de tratamento para as exceções lançadas. Assim, o tratamento de exceções é considerado uma técnica essencial para a estruturação interna do componente tolerante a falhas porque introduz uma separação clara entre a execução normal do código e o tratamento das condições anormais (ou excepcionais).

Figura 4.1: Componente tolerante a falhas [dMF01]

³<http://www.omg.org>

⁴<http://www.microsoft.com/com>

⁵<http://java.sun.com/beans>

As exceções dos componentes podem ser classificadas como:

- **Exceções Internas:** as exceções internas ocorrem quando o componente detecta uma situação inesperada durante a atividade normal. Após o lançamento de uma exceção interna, o próprio componente realiza o seu tratamento, retornando à atividade normal logo em seguida.
- **Exceções de Defeito:** caso não seja possível a recuperação da exceção interna, o componente lança uma exceção de defeito indicando que não foi possível realizar o serviço solicitado. Este tipo de exceção também é denominado de exceção externa pois as atividades relacionadas ao tratamento da mesma são externas ao componente que a lançou.
- **Exceções de Interface:** são lançadas por um componente quando ele recebe solicitações que não estão em conformidade com sua especificação. Normalmente, este tipo de exceção é provocado por uma falha na combinação entre os componentes na montagem do sistema.

4.2.2 Tratamento de Exceções na Arquitetura

Em sistemas complexos, compostos de vários componentes, é necessária uma maneira ordenada e eficiente de lidar com as exceções que podem ser lançadas pelo sistema, e com a concorrência entre os componentes. Para isso, algumas abordagens foram desenvolvidas:

- **Transações [GR92]:** ações lógicas que realizam uma sequência de operações básicas em objetos ou dados compartilhados, muito utilizadas em sistemas de banco de dados. Para proteger os dados compartilhados, as transações são feitas obedecendo-se às propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Entretanto, seu uso para sistemas concorrentes é limitado, uma vez que apenas um único processo pode entrar na transação por vez. Ademais, as transações lidam, em geral, com falhas no hardware e não com erros de software, não oferecendo portanto um mecanismo adequado para tratar exceções.
- **Conversações [Ran75]:** as conversações constituem uma técnica para realizar recuperação coordenada num conjunto de processos. Elas provêm um suporte efetivo para cooperação entre ações, tolerância a falhas e recuperação de erros. Porém, elas não evitam o problema chamado “contrabando de informação” (do inglês: *information smuggling*) [Kim82], o que pode trazer graves consequências na realização da recuperação de erros, por exemplo.
- **Ações Atômicas Coordenadas (Ações CA) [XRR⁺95a, XRR⁺95b, Rom01, RXR96]:** integra características de transações e conversações sem os problemas enfrentados por ambos, constituindo assim um modelo poderoso para a coordenação dos múltiplos componentes de um sistema e para o tratamento de exceções no nível arquitetural.

Para este trabalho de Mestrado, optamos por utilizar o modelo de ações CA para a coordenação de componentes e tratamento de exceções. Além de não ter as vulnerabilidades de transações e conversações, as ações CA têm boa performance e escalabilidade, sendo usadas na especificação de sistemas críticos reais, como o controle de trens nas estações [BRR⁺00], e o sistema de um dispositivo para administração de insulina para controle de diabetes [ACP05].

4.3 Ações Atômicas Coordenadas (Ações CA)

O modelo de Ações Atômicas Coordenadas (CA) [XRR⁺95a, XRR⁺95b, Rom01, RXR96] integra os conceitos de ações atômicas (ou conversações), transações ACID (que garantem as propriedades ACID de Atomicidade, Consistência, Isolamento e Durabilidade) e tratamento concorrente de exceções, visando a coordenar a concorrência, comunicação e recuperação de erros entre múltiplos componentes de um sistema.

Cada ação CA é constituída por um conjunto de componentes (também denominados participantes) que interagem entre si, de modo cooperativo, para realizar uma atividade. Sua estrutura

também conta com objetos externos, que funcionam como repositórios de informação, e que são compartilhados pelos participantes, ou seja, o acesso aos objetos externos dá-se de forma concorrente, por meio de transações ACID.

Na Figura 4.2, podemos ver um esquema de Ação CA que envolve dois componentes e um objeto externo. Cada componente é representado por uma seta horizontal de execução ao longo do tempo. O objeto, é representado por uma linha horizontal, e o escopo da ação CA é delimitado pelo retângulo.

Figura 4.2: Diagrama de representação de uma ação CA.

A ação CA da Figura 4.2 inicia-se com uma transação no Objeto Externo 1, sendo este início representado pela primeira seta vertical direcionada ao objeto. Em seguida, o Componente 1 se comunica com o Componente 2, conforme indica a seta bidirecional entre estes componentes. Depois, o Componente 2 efetua uma comunicação com o Objeto Externo 1, representada por uma seta vertical pontilhada. E, por fim, a transação é finalizada por meio de um *commit*, identificado por uma segunda seta vertical direcionada ao objeto externo.

4.3.1 Aninhamento e Composição

Para manter a flexibilidade e escalabilidade em sistemas complexos, as ações CA possuem as características de aninhamento e composição. Por aninhamento, entende-se que uma ação CA pode ter uma ou mais ações CA internas a ela, como pode ser visto na Figura 4.3.

Figura 4.3: Ações CA aninhadas

Desta ação interna, um sub-conjunto dos componentes da ação externa participam, e o objeto externo é o mesmo utilizado pela ação CA mais externa. Assim que a ação CA aninhada for concluída, os componentes voltam a participar da ação CA externa.

A composição de ações CA, por sua vez, permite o reuso de uma mesma atividade por várias ações CA do sistema. Na composição, uma ação CA precisa que outra ação, cujos participantes e objetos externos são próprios, seja finalizada para então prosseguir. Isto pode ser visto na Figura 4.4, em que a ação CA 1 precisa que uma ação CA 2 seja realizada para continuar.

Figura 4.4: Ações CA compostas

4.3.2 Tratamento de Exceções

No modelo de ações CA, uma ou várias exceções podem ser lançadas no sistema num dado instante. Para que elas sejam tratadas adequadamente, de uma maneira eficiente e ordenada, alguns passos devem ser seguidos [Per07, XRR⁺95a]:

1. Identificação da exceção (ou exceções) lançada por um dos participantes da ação CA;
2. Os demais participantes da ação CA são informados do lançamento, e têm suas atividades interrompidas;
3. No caso de mais de uma exceção ter sido lançada, utilizar a árvore de exceções proposta por Campbell e Randell [CR86], na qual elas são combinadas de modo a obter uma única exceção que represente todas as exceções lançadas. Esta exceção única será a exceção tratada pelos participantes da ação CA;
4. Cada um dos participantes da ação CA executa o tratamento da exceção. Caso este tratamento seja bem sucedido, as transações iniciadas nos objetos externos são confirmadas; caso contrário, as transações são abortadas e desfeitas e, se o sistema não conseguir ser devidamente recuperado ao estado anterior ao lançamento da exceção (ou exceções), ela (ou a exceção resultante da árvore de exceções) é propagada à ação CA pai.

4.4 Uma Implementação das Ações Atômicas Coordenadas

Para viabilizar a implementação e verificação de propriedades sobre as ações atômicas coordenadas, foi necessário implementar um protocolo de coordenação excepcional de componentes. Desta

forma, optamos por implementar, em Java, o modelo de coordenação segundo as ações CA, especificado formalmente na Tese de Doutorado de Pereira [Per07] e no artigo de Pereira e Melo [PdM10]. A especificação completa em CSPm encontra-se no Apêndice A.

A escolha de implementar, em Java, tal modelo de coordenação de componentes segundo as ações CA foi feita após considerarmos a utilização de outras alternativas de implementação para a coordenação de componentes. Estas alternativas, que descrevemos a seguir, mostraram-se inviáveis para o propósito deste trabalho, sobretudo no que diz respeito à implementação das propriedades de coordenação excepcional de componentes.

- **Utilização de Java EE/EJB:** pelo fato de Java EE ser a plataforma de programação utilizada para implementação de software corporativo e de componentes, sua utilização para a implementação do protocolo de coordenação excepcional de componentes foi considerada inicialmente.

Problemas encontrados: o principal problema encontrado, e que resultou na sua não utilização, foi a incompatibilidade de uso desta plataforma com o verificador escolhido para este trabalho, o Java Pathfinder [Jav]. Este verificador é compatível com a plataforma Java SE⁶, e utilizamos, portanto, esta plataforma para o desenvolvimento do trabalho.

- **Utilização do JPA:** a JPA, ou *Java Persistence API* ⁷, é bastante usada para a implementação de transações entre o programa e um ou mais objetos externos (e.g.: banco de dados), tanto em Java EE, quanto em Java SE.

Problemas encontrados: a utilização da JPA não foi possível pois ela não permite o aninhamento de transações [KS06]. Isto ocorre porque ela restringe a chamada do método que inicia a transação, de forma que uma nova chamada a esse método só seja possível caso a chamada anterior tenha sido finalizada. Para a implementação das propriedades do modelo de coordenação segundo as ações CA, esta restrição limita por demais a utilização das ações CA de modo escalável e flexível, conforme apresentado na Seção 4.3.1. Por isso, optamos por não utilizar a JPA e fazer uma implementação própria de transações que permitam o seu aninhamento.

Com tais alternativas mostrando-se inviáveis, foi necessário escolher e implementar, em Java SE, um modelo adequado de ações atômicas coordenadas. A especificação em CSP/CSPm [Per07, PdM10] mostrou-se adequada ao propósito deste trabalho por estar definida em linguagem formal e conter um modelo genérico com todos os processos e eventos necessários para a implementação do protocolo de coordenação excepcional de componentes, considerando as ações CA aninhadas. Assim como no modelo original, a implementação do modelo em questão pode ser instanciada para sistemas específicos de forma que teremos a implementação de um modelo-base de coordenação do comportamento excepcional dos componentes.

4.4.1 Arquitetura do Modelo

Para a implementação do modelo em Java, simplificamos a ideia de componente: agora, os componentes passam a ser objetos pertencentes a uma determinada classe. Desta forma, o modelo consiste em coordenar estes objetos conforme o conceito de ações CA.

Seguindo e adaptando a especificação [Per07, PdM10] em CSP, a arquitetura do modelo implementado consiste em três classes principais: *CoordinatedAtomicAction*, *Coordinator* e *Participant*. Para a comunicação com o objeto externo, foi criada uma interface *Transaction*, cujos métodos devem ser implementados para a utilização deste modelo. Analogamente, foi criada a interface *ExceptionTree*, cujo método de resolução de árvore de exceção, usado no tratamento das exceções de uma ação CA, deve ser definido pelo usuário de acordo com as exceções a serem lançadas por seu

⁶Neste trabalho, para efeito de simplificação, chamamos a plataforma Java SE apenas de Java.

⁷API é a abreviatura de *Application Programming Interface* (ou Interface de Programação de Aplicações)

sistema. As classes *Role* e *Handler* implementam, respectivamente, o comportamento normal/lançamento de exceções e o tratamento de exceções lançadas. Na Figura 4.5 é apresentado um esquema com a comunicação entre estas classes.

Figura 4.5: Esquema de arquitetura do modelo implementado.

Nas próximas seções, detalhamos os métodos implementados e sua correlação com os processos especificados em CSP.

4.4.2 CoordinatedAtomicAction

Esta classe representa uma ação atômica coordenada, possuindo em seus campos os elementos que a constituem:

```

//número que identifica a ação CA.
//obrigatoriamente, a primeira ação CA deve ter index = 0.
private int index;
//coordenador responsável por todo o processo de coordenação desta ação.
private Coordinator coordinator;
//ação CA pai
private CoordinatedAtomicAction parentCaa;
//vetor contendo todos os participantes desta ação
private Vector<Participant> participants;
// vetor contendo todas as ações CA aninhadas (ou ações CA filhas)
private Vector<CoordinatedAtomicAction> nestedActions;
//vetor contendo todas as exceções envolvidas no sistema
private Vector<Exception> exceptions;
//vetor contendo todas as exceções propagadas
private Vector<Exception> propagatedExceptions;
//vetor contendo todas as transações
private Vector<TransactionInterface> trans;
//árvore de resolução de exceções lançadas concorrentemente
private ExceptionTreeInterface exceptionTree;
//relata o status atual da ação. Pode ser: STARTED (ação iniciada),
//STOPPED (ação não iniciada ou ação finalizada) ou
//INTERRUPTED (ação suspensa).
private String status;
    
```

- **Uso da Classe**

Para instanciar uma ação CA, deve-se fornecer o número que a identificará (index). No momento da construção, o vetor de exceções propagadas é inicializado, e seu status é definido como “STOPPED”. Os demais elementos devem ser fornecidos pelo usuário e setados na instância da ação CA.

- **Diferenças em Relação à Especificação**

Devido às diferenças entre CSP e Java, foram necessárias adaptações do código. As principais alterações consistem na utilização de variáveis adicionais: o vetor de exceções propagadas, a ação CA pai e o status da ação. Estas variáveis são utilizadas para facilitar a comunicação de eventos entre ação CA, coordenador e participantes, em substituição aos canais utilizados em CSP.

4.4.3 ExceptionTree

Interface que representa a árvore de resolução de exceções de uma ação CA.

```
//Recebe um vetor que contém as exceções lançadas concorrentemente ,
//aplica o método de resolução e armazena a exceção resultante na
//primeira posição do vetor. Os demais elementos do vetor devem ser
//removidos.
public abstract void resolveException(Vector<Exception> exceptions);
```

- **Uso da Interface**

Para utilizar esta interface, o usuário deve criar uma classe que a herda, e implementar o método requisitado. Feito isso, o usuário deve criar uma instância dessa classe e colocá-la na ação CA correspondente. O método implementado será utilizado pelo coordenador da ação.

4.4.4 Participant

Para representar os participantes das ações CA, foi implementada a classe *Participant*. Esta classe representa um participante da ação como uma thread. Para tornar possível a realização de todas as atividades previstas na especificação, esta classe faz uso de outras duas threads, *Role* e *Handler*, que implementam, respectivamente, o comportamento normal e o comportamento excepcional do participante (maiores detalhes sobre *Role* e *Handler* são descritos nas Seções 4.4.5 e 4.4.6). A seguir, os campos da classe são apresentados.

```
//vetor com as thread responsáveis pelo comportamento normal
//do participante em cada ação CA.
private Vector<Role> roles;
//vetor com as thread responsáveis pelo comportamento excepcional
//do participante em cada ação CA.
private Vector<Handler> handlers;
//vetor com todas as ações CA na qual este participante realiza
//alguma atividade.
private Vector<CoordinatedAtomicAction> caaVector;
//número que identifica o participante.
private int numberId;
//número que identifica a ação CA na qual o participante terminou
//de realizar sua atividade.
private int endCaaIndex;
//número que identifica a próxima ação CA aninhada na qual este
//participante deve realizar alguma atividade.
private int nextNestedActionIndex;
//informa o tipo de término a ser requisitado pelo participante
//numa ação CA. Pode ser NORMAL, EXCEPTIONAL ou ABORT.
private String endRequest;
//informa o status do participante. Pode ser STARTED, STOPPED ou SUSPENDED.
private String status = "";
```

- **Métodos Implementados**

Para seguir o comportamento definido na especificação, foram implementados métodos na classe *Participant* que realizam esta tarefa. Eles foram divididos em três categorias: métodos

para início de atividade, de monitoramento da execução e para sinalização do término da atividade, e são descritos em detalhes a seguir.

Método para Início de Atividade: para iniciar a atividade, o participante precisa fazer primeiro a requisição de entrada na ação para o coordenador. Para isso foi implementado o método, *entryRequest*, descrito a seguir.

```
public void entryRequest(int caaIndex) {
    CoordinatedAtomicAction caaAux;
    for (int i = 0; i < caaVector.size(); i++) {
        //encontra, no vetor de ações CA, a ação CA na qual a requisição deve
        //ser feita.
        caaAux = caaVector.get(i);
        if (caaAux.getIndex() == caaIndex) {
            //inicia o coordenador da ação
            if (caaAux.getCoordinator().getCoordState().equals("STOPPED")) {
                caaAux.getCoordinator().setCoordState("STARTED");
            }
            ...
            //soma ao contador do coordenador esta requisição de entrada. Assim
            //que este contador chegar ao valor definido no coordenador, a ação
            //inicia.
            caaAux.getCoordinator().incrementRequestCount();
        }
    }
}
```

Método de Monitoramento da Execução: uma vez iniciada a ação, o participante deve iniciar sua execução. Para esta implementação, optamos por juntar os sub-processos *Execution* e *Suspension* definidos na especificação no método *beginParticipantAction*. Este método é o responsável por controlar a execução da atividade normal (representada pelo vetor de *Roles*) e por responder à requisição do coordenador da ação para abortar a atividade ou iniciar a execução do tratamento de exceção lançada no contexto da ação CA. Este método é descrito a seguir.

```
public void beginParticipantAction(int caaIndex) {
    ...
    //requisição de entrada para o coordenador
    this.entryRequest(caaIndex);
    ...
    //inicia o comportamento normal após a confirmação
    //do coordenador do início da ação.
    this.roles.elementAt(roleVindex).start();

    while (this.roles.get(roleVindex).isEndOfExecution() == false) {
        //enquanto a atividade normal não se encerra, verifica se há
        //alguma requisição feita pelo participante ou pelo coordenador.
        if (this.caaVector.elementAt(caaVindex).getStatus().equals("INTERRUPTED")) {
            //se a ação foi interrompida, a atividade normal é encerrada.
        } else if (this.getStatus().equals("SUSPENDED")) {
            //se o participante foi suspenso, significa que ele deve iniciar
            //a atividade normal de uma ação
            ...
            //inicia a ação aninhada, fazendo uma chamada recursiva.
            beginParticipantAction(nestedActionIndex);

            //após o término da ação aninhada, retoma a atividade normal
            //da ação suspensa.
        }
    }
    if (this.caaVector.elementAt(caaVindex).getStatus().equals("INTERRUPTED")) {
        ...
    }
}
```

```

//se a ação foi interrompida, é preciso verificar se houve um aborto
//da ação ou se o tratamento de uma exceção deve ser iniciado.
...
if (this.caaVector.get(caaVindex).getCoordinator().isHandlingException
    ()) {
    //inicia o tratamento excepcional
    this.handlers.get(handlerVindex).start();
    ...
    //após finalizar o tratamento, chama o método de requisição de
    //término
    //da atividade do participante para o coordenador da ação.
} else {
    if (this.caaVector.get(caaVindex).getCoordinator().
        isExceptionPropagation()) {
        //se for sinalizado que a exceção deve ser propagada,
        //o participante requisita término excepcional.
    } else {
        //se não foi sinalizado tratamento de exceção ou propagação,
        //é requisitado o término em aborto.
    }
}
}
else {
    //se não houve interrupção, é requisitado o término normal
    //ou excepcional, caso alguma exceção tenha sido lançada durante
    //a atividade normal deste participante.
}
}
}

```

Métodos para Sinalização do Término da Atividade: após a finalização da execução, o participante deve fazer a requisição de término ao coordenador da ação. Esta requisição pode ser de término normal, excepcional ou aborto. A requisição para término normal ocorre quando a atividade normal ou excepcional do participante foi concluída com êxito. A requisição para término excepcional ocorre quando alguma exceção foi lançada durante a execução da atividade do participante, ou no caso da exceção ser propagada para a ação pai. Por fim, a requisição para término em aborto ocorre quando a atividade do participante foi abortada devido à requisição do coordenador da ação.

Para cada tipo de requisição de término, foi implementado um método correspondente. A seguir, apresentamos o método para o caso de requisição de término normal.

```

public void normalEndRequest(int caaIndex) {
    //informa o número identificador da ação CA para a qual será requisitado
    //o término da
    //atividade do participante.
    this.setEndCaaIndex(caaIndex);
    //informa o tipo de término requisitado.
    this.setEndRequest("NORMAL");
}

```

Os métodos para as demais requisições de término são análogas a esta. Apenas o método para requisição de término excepcional informa ao coordenador, adicionalmente, que este participante propagou ou lançou uma exceção.

• Uso da Classe

Para implementar os participantes de seu sistema, o usuário deve implementar classes que herdam de Participant, informar um número que servirá para identificar este participante no sistema, e chamar o método beginParticipantAction. As atividades para comportamento normal e excepcional de cada participante deverá ser implementado em classes distintas (ver Seções 4.4.5 e 4.4.6).]

• Diferenças em Relação à Especificação

O processo *Execution* especificado teve sua implementação feita de modo ligeiramente diferente: ao invés de blocos de execução, as atividades foram separadas por *Roles*, sendo que no caso de uma atividade ser interrompida para execução de outra atividade numa ação aninhada, ela é retomada ao término da ação filha. O processo *Suspension*, que especifica a interrupção de todos os participantes para início do tratamento cooperativo de exceção foi implementado de acordo, assim como o processo para requisição de término em aborto.

Os processos *Query* e *Update*, referentes à consulta e atualização do estado de um objeto externo não foram implementados, uma vez que o acesso a estes métodos é possível por meio do vetor de transações, que deve ser definido na classe *Participant*.

4.4.5 Role

Para implementar o comportamento normal do participante, foi criada esta classe Role. Ela classe representa uma thread e, portanto, sua ação principal deverá ser implementada no método *run*. A seguir, a Listagem com os campos desta classe é apresentada.

```
//variável booleana que interrompe a execução desta thread
//caso tenha seu valor definido para "true".
private boolean threadSuspended;
//variável booleana que informa o término de execução da thread.
private boolean endOfExecution;
//relata o status da execução desta thread. Pode ser NORMAL (para
//término normal), EXCEPTIONAL (para término com lançamento de
//exceção) ou INTERRUPTED (para aborto ou suspensão do participante).
private String status;
//participante cujo comportamento normal é representado por esta thread.
private Participant participant;
//ação na qual esta thread é executada.
private CoordinatedAtomicAction caAction;
//vetor que contém as transações ativas nesta ação CA.
protected Vector<TransactionInterface> transactions;
```

• Uso da Classe

Para implementar o comportamento normal de um participante, o usuário deverá criar uma classe que herda de Role e programar o que ela deve fazer no contexto de uma única ação CA da qual este participante faz parte. Caso haja mais de uma ação CA na qual este participante esteja inserido, é necessário criar uma classe Role para cada uma destas ações.

4.4.6 Handler

Para implementar o comportamento excepcional do participante, que equivale ao tratamento da exceção lançada na ação CA, foi criada a classe Handler. De modo semelhante à classe Role, Handler representa uma thread que deve ser acionada no momento que o participante é requisitado a tratar uma exceção lançada no contexto de uma ação CA. Os campos desta classe são mostrados abaixo.

```
//variável booleana que informa o término de execução da thread.
private boolean endOfExecution;
//relata o status da execução desta thread. Pode ser NORMAL (para
//término normal), EXCEPTIONAL (para término excepcional).
private String status;
//participante cujo comportamento normal é representado por esta thread.
private Participant participant;
//ação na qual esta thread é executada.
private CoordinatedAtomicAction caAction;
//vetor que contém as transações ativas nesta ação CA.
protected Vector<TransactionInterface> transactions;
```


Como pode ser percebido, os campos desta classe são praticamente os mesmos da classe *Role*. A única exceção é a variável *threadSuspended* que não consta em *Handler*. Isto acontece porque esta variável é utilizada para suspender a ação da thread, e de acordo com a especificação, o tratamento de uma exceção não pode ser interrompido.

• Métodos da Classe

Além dos Getters e Setters, esta classe possui o método *throwException*, mostrado a seguir.

```
public void throwException() throws Exception {
    Vector<Exception> excVector = this.getCaAction().getExceptions();
    Exception exception = this.getCaAction().getCoordinator().
        getPropagatedExceptions().firstElement();
    for (int i = 0; i < excVector.size(); i++) {
        if (excVector.get(i).getClass().getSimpleName().equals(exception.
            getClass().getSimpleName())) {
            throw excVector.get(i);
        }
    }
}
```

Este método foi implementado para fazer o lançamento da exceção a ser tratada, que está armazenada no vetor *propagatedExceptions* do coordenador da ação CA correspondente. O laço deste método é utilizado para garantir que a exceção a ser lançada pertence ao universo de exceções do sistema, definido pelo vetor *exceptions* da ação CA.

- **Uso da Classe** Para implementar o comportamento excepcional do participante, o usuário deverá criar uma classe que herda *Handler*. Nesta classe, o método *run* a ser implementado deverá conter um bloco *try-catch* com uma chamada ao método *throwException* mostrado acima, sendo o tratamento feito, conseqüentemente, no *catch* da respectiva exceção a ser tratada.

De modo análogo ao que ocorre com a classe *Role*, é necessário implementar um *Handler* para cada ação CA na qual o participante esteja inserido, caso haja a possibilidade de ocorrer um lançamento de exceção durante a execução desta ação CA.

4.4.7 Transaction e External Object

Os objetos externos, conforme descrito na especificação, interagem com os participantes e com o coordenador da ação de modo que as propriedades ACID sejam preservadas num contexto transacional. Nesta implementação, optamos por, ao invés de implementar o processo *ExternalObject* da especificação em Java, criar uma interface *TransactionInterface* que agisse como um intermediário entre a comunicação do objeto externo com os demais elementos da ação CA. Deste modo, esta interface será a responsável por comunicar ao objeto externo as requisições de alterações em seu estado feitas pelos participantes e pelo coordenador, e transmitirá a eles se tais requisições foram efetivamente concluídas ou não. Os métodos desta interface são listados a seguir.

```
//inicia uma nova transação no objeto externo
public abstract void begin();
//confirma a transação iniciada no objeto externo
public abstract void commit();
//desfaz a transação iniciada no objeto externo
public abstract void rollback();
//retorna o estado da transação (ativa ou inativa)
public abstract boolean isActive();
//retorna o objeto externo
public abstract Object getExternalObject();
//estabelece o objeto externo
public abstract void setExternalObject(Object externalObj);
```


- **Uso da Interface**

Antes de utilizar esta interface, o usuário deve ter implementado o objeto externo de seu sistema. Para cada objeto externo do sistema, o usuário deverá implementar uma classe que herda esta interface *TransactionInterface*, e que conterá uma referência a este objeto e a implementação dos métodos requisitados. Após implementar tal classe, o usuário deve criar uma instância dela e colocá-la na ação CA correspondente. Os métodos implementados serão utilizados pelo coordenador e pelos participantes da ação.

- **Diferenças em Relação à Especificação**

Por não se tratar da interface do objeto externo em si, a interface *Transaction* possui os métodos *getExternalObject* e *setExternalObject* que retornam e definem, respectivamente um objeto externo instanciado pelo usuário em seu sistema. Os demais métodos possuem processos correspondentes na especificação. O único processo da especificação que não consta como um método desta interface é o processo *InitState*, que armazena o estado inicial do objeto externo. Isto ocorre pois consideramos tal processo implícito na construção do objeto externo e de caráter opcional nesta interface.

4.4.8 Coordinator

A classe *Coordinator* representa o coordenador da ação CA. Ele é o responsável por implementar o protocolo de coordenação e tratamento cooperativo de exceções dos participantes da ação, bem como o responsável por fazer o controle transacional entre os participantes e os objetos externos do sistema.

Nesta implementação, o coordenador é uma thread que controla as threads que representam os participantes da ação CA. Seus campos são listados a seguir.

```
//ação CA coordenada
private CoordinatedAtomicAction caa;
//ação CA pai
private CoordinatedAtomicAction parentCaa;
//vetor com as ações CA aninhadas
private Vector<CoordinatedAtomicAction> nestedActions;
//vetor com os participantes da ação CA
private Vector<Participant> participants;
//vetor contendo os participantes em execução na ação CA
private Vector<Participant> run;
//vetor contendo os participantes que irão tratar uma exceção do sistema
private Vector<Participant> pHandlers;
//vetor contendo os participantes que lançaram exceções
private Vector<Participant> particRaiseException;
//vetor contendo os tipos de exceções que devem ser propagadas pela ação CA
//para a ação pai
private Vector<Exception> fSignal;
//vetor contendo as exceções lançadas e/ou propagadas para esta ação
private Vector<Exception> propagatedExceptions;
//vetor de transações
private Vector<TransactionInterface> transactions;
//objeto contendo o método de resolução de exceções lançadas concorrentemente
private ExceptionTreeInterface exceptionTreeRes;
//variável que sinaliza o início do tratamento excepcional a ser feito pelos
//participantes da ação CA
private boolean handlingException;
//variável que sinaliza a interrupção das atividades dos participantes
private boolean interruptingParticipants;
//variável que sinaliza a propagação de uma exceção para a ação CA pai
private boolean exceptionPropagation;
//variável que sinaliza a interrupção da atividade do coordenador
private boolean coordInterrupted;
//variável que sinaliza o lançamento de uma exceção por um participante da
//ação CA
```

```

private boolean exceptionRaised;
//variável que controla as requisições de entrada na ação CA feita pelos
//participantes
private int entryRequestCount;
//variável que controla a interrupção das atividades dos participantes
//da ação CA
private int interruptedParticCount;
//informa o estado do coordenador. Pode ser STARTED ou STOPPED
private String coordState;
//informa o tipo de término da ação: NORMAL, EXCEPTIONAL ou ABORT
private String caaEnd;

```

• Métodos Implementados

Para garantir o comportamento especificado, foram implementados métodos na classe *Coordinator* que seguem os sub-processos definidos na especificação CSP. Para facilitar sua descrição, eles foram separados em seis categorias: método *run* do coordenador, métodos de início da ação, de monitoramento da execução, de tratamento cooperativo de exceções, de aborto da ação e de término da ação.

Método *Run* do Coordenador: diferentemente das outras classes apresentadas, o coordenador possui já definido o método *run*, o método principal da classe, responsável pela execução da thread. Este método é apresentado abaixo.

```

@Override
public void run() {
    try {
        //coordenador espera o início de sua atividade, que ocorre
        //quando seu estado passa de STOPPED a STARTED
        while (getCoordState().equals("STOPPED")) {
            ...
            wait();
            ...
        }
        //recebe as requisições de entrada dos participantes na ação
        this.recInitRequest();
    } catch (CoordInterruptedException ex) {
        //no caso da ação ter sido abortada, é jogada uma exceção do
        //tipo CoordInterruptedException. Como o aborto da ação CA-0
        //não pode ocorrer (pois esta é a ação CA inicial), é feita
        //a verificação do índice da ação antes de se requisitar o
        //aborto.
        if (caa.getIndex() > 0)
            recAbortRequest();
        else
            System.out.println("[ERROR]: trying to abort CAA-0");
    }
}

```

Métodos de Início da Ação: conforme visto no item anterior, o primeiro método chamado corresponde ao método responsável por receber as requisições dos participantes para entrar na ação CA. Após todos os participantes terem requisitado sua entrada na ação CA, as transações são iniciadas e a ação começa. Estes métodos são mostrados a seguir.

```

public void recInitRequest() throws CoordInterruptedException {
    //espera todos os participantes requisitarem sua entrada na ação
    while (getEntryRequestCount() < participants.size()) {
        ...
        wait();
        ...
    }
    //quando todas as ações foram requisitadas, a ação é iniciada
    beginTrans();
}

```

```

}

public void beginTrans() throws CoordInterruptedException {
    //as transações são iniciadas
    for (int i = 0; i < transactions.size(); i++) {
        transactions.get(i).begin();
    }
    //após todas as transações serem iniciadas, a ação se inicia
    beginAction();
}

public void beginAction() throws CoordInterruptedException {
    //o status da ação é alterado para STARTED, sinalizando seu início
    //ao coordenador e aos participantes
    this.caa.setStatus("STARTED");
    //início do monitoramento da execução das atividades da ação
    monitor();
}

```

Método de Monitoramento da Execução: iniciada a ação, o coordenador passa a monitorar a execução das atividades dos participantes. Todos os participantes em execução são colocados no vetor *run*, e as requisições de término de cada um dele são observadas pelo coordenador. No caso de término excepcional, o coordenador inicia o tratamento cooperativo da exceção; no caso de aborto, é lançada a exceção *CoordInterruptedException* que será capturada no método *run*; no caso de término normal, o participante é removido do vetor *run*. O método *monitor* é apresentado abaixo.

```

public void monitor() throws CoordInterruptedException {
    //adição dos participantes no vetor run, para monitorar
    //sua atividade.
    for (Participant participant : participants) {
        run.add(participant);
    }
    //enquanto o vetor com participantes em execução não ficar vazio,
    //e enquanto a interrupção da ação não é requisitada, o monitoramento
    //é feito.
    while (run.size() != 0 && !isCoordInterruptedException()) {
        for (Participant p : run) {
            String endRequest = p.getEndRequest();
            //término normal
            if (endRequest.equals("NORMAL")) {
                run.remove(p);
                break;
            }
            //término excepcional – exceção propagada de ação aninhada
            if (exceptionPropagation == true) {
                abort();
                return;
            }
            //término excepcional – exceção lançada em atividade dos
            //participantes desta ação
            if (exceptionRaised == true) {
                abort();
                return;
            }
        }
    }
    //se a interrupção foi requisitada, a exceção é lançada
    if (isCoordInterruptedException() == true) {
        throw new CoordInterruptedException();
    }
    //senão, o commit das transações pode se feito.
    else {

```

```

        commitTrans();
    }
}

```

Métodos do Tratamento Cooperativo de Exceções: no caso de término excepcional, o coordenador deve abortar as ações aninhadas e suspender os demais participantes, para que o tratamento cooperativo da exceção seja iniciado. Isto é feito pelos métodos *abort* e *suspendParticipants*.

```

public void abort() {
    for (CoordinatedAtomicAction action : nestedActions) {
        Coordinator coord = action.getCoordinator();
        if (coord.getCoordState().equals("STARTED")) {
            //ações aninhadas são abortadas e é feito rollback
            //de suas transações
            ...
            coord.rollbackTransAbort();
            ...
            //espera a confirmação do aborto da
            ...
        }
    }
    suspendParticipants();
}

public void suspendParticipants() {
    //status da ação é definido como INTERRUPTED
    this.caa.setStatus("INTERRUPTED");

    //vetor pHandler contendo os participantes que devem tratar a exceção
    //é definido. Estes participantes englobam os que estão em execução
    //(contidos no vetor run), e os que não lançaram ou propagaram
    //exceções (contidos no vetor particRaiseException).
    pHandlers = new Vector<Participant>();
    ...
    //espera todos os participantes responsáveis por tratar a exceção
    //terem sua atividade interrompida
    while (interruptedParticCount < pHandlers.size()) {
        ...
        wait();
        ...
    }
    //reset end request of participants
    for (Participant p : pHandlers) {
        p.setEndRequest("");
    }
    resolve();
}

```

Após o aborto das ações aninhadas, e a suspensão das atividades dos participantes, o tratamento cooperativo pode ser iniciado. Para isso, é preciso resolver as exceções que podem ter sido lançadas concorrentemente. Em seguida, deve-se escolher se a exceção resolvida será propagada à ação CA pai, ou se ela será tratada no contexto da ação CA atual. No caso de se escolher o tratamento cooperativo da exceção, é preciso checar se este foi bem sucedido ou não. Caso tenha sido, é feito o commit das transações. Caso contrário, o rollback das transações é feito. Os métodos responsáveis por estas atividades são mostrados a seguir.

```

public void resolve() {
    //chama o método para resolução das exceções lançadas
    exceptionTreeRes.resolveException(propagatedExceptions);
    //inicia o tratamento cooperativo da exceção resolvida
    choose();
}

```

```

public void choose() {
    //se a exceção pertence ao vetor fSignal, ela deve ser propagada
    //para a exceção pai. Isto é verificado pela função auxiliar
    //containsException()
    if (containsException()) {
        signal();
    }
    //caso contrário, a exceção é tratada cooperativamente pelos
    //participantes da ação
    else {
        handle();
    }
}

public void signal() {
    //para propagar a exceção, todos os participantes em execução
    //devem terminar excepcionalmente
    while (pHandlers.size() != 0) {
        ...
        //com a confirmação do término excepcional, o participante
        //é removido do vetor pHandlers.
    }
    //finalizadas as atividades dos participantes desta ação, é feito o
    //rollback das transações e a exceção é propagada para a ação pai.
    rollbackTransExcep();
    this.parentCaa.getPropagatedExceptions().add(propagatedExceptions.
        firstElement());
    this.parentCaa.getCoordinator().setExceptionPropagation(true);
}

public void handle() {
    //para fazer o tratamento cooperativo da exceção, o coordenador sinaliza
    //aos participantes seu início pela variável handlingException
    this.setHandlingException(true);
    //também é sinalizado aos participantes que o tratamento pode ser
    //iniciado pois a interrupção das atividades de todos os participantes
    //da ação foi concluída
    this.setInterruptingParticipants(true);

    verify();
}

public void verify() {
    //variável utilizada para sinalizar se a exceção foi tratada
    //com sucesso, ou não.
    boolean requestSignal = false;

    while (pHandlers.size() != 0) {
        //checa o tipo de término dos participantes. Caso haja
        //término excepcional, requestSignal passa a ser true.
    }
    respond(requestSignal);
}

public void respond(boolean requestSignal) {
    //se o tratamento não foi bem sucedido, é feito o rollback
    //das transações
    if (requestSignal == true) {
        rollbackTransExcep();
    }
    //caso contrário, o commit das transações é feito.
    else {
        commitTrans();
    }
}

```

```
}

```

Métodos do Aborto da Ação: caso seja requisitado o aborto da ação, é lançada a exceção *CoordInterruptedException*. Esta será capturada no método *run* da classe *Coordinator*, e o método responsável por interromper as atividades da ação é chamado. Ele é apresentado abaixo.

```
public void recAbortRequest() {
    //o aborto das ações aninhadas é feito pelo método
    //abortNested. Este método é semelhante ao método abort
    //implementado para a realização do tratamento de exceção.
    abortNested();
    //ação CA é interrompida
    caa.setStatus("INTERRUPTED");
    ...
    //é feita a verificação da interrupção dos participantes
    ...
    //ao final, o rollback das transações é feito
    rollbackTransAbort();
}
```

Métodos de Término da Ação: terminadas as atividades dos participantes, resta finalizar o coordenador e a ação CA. Para isso, deve ser feito o commit (ou rollback) das transações e o status do coordenador e da ação devem ser atualizados para “STOPPED”. A seguir, mostramos os métodos para finalização normal.

```
public void commitTrans() {
    for (int i = 0; i < transactions.size(); i++) {
        transactions.get(i).commit();
    }
    endActionNormal();
}

public void endActionNormal() {
    this.setCaaEnd("NORMAL"); //tipo de finalização da ação é NORMAL
    this.caa.setStatus("STOPPED");

    finalizeAction();
}

public void finalizeAction() {
    this.setCoordState("STOPPED");
}
```

Para a finalização excepcional ou em aborto, os métodos são semelhantes, apenas com a substituição do commit por rollback, e com a substituição do tipo de finalização da ação CA.

- **Uso da Classe**

Para utilizar esta classe, deve-se informar a ação CA durante a construção de sua instância. Em seguida, ela pode ter sua execução iniciada por meio do método *start*, da classe *Thread* de Java.

- **Diferenças em Relação à Especificação**

O que difere esta implementação da especificação é o fato da substituição do processo CSP *CoordinatorState* pela variável *coordState* na classe *Coordinator*. Outra diferença é em relação ao tratamento cooperativo das exceções: na implementação, apenas os participantes em execução e que não lançaram exceções no contexto da ação são acionados para fazer o tratamento, enquanto que na especificação, todos os participantes fazem o tratamento. Isto foi feito pois consideramos na implementação que, caso um participante decida lançar a exceção, ele o faz

por não conseguir fazer o tratamento local dela e, por isso, ele não participa do tratamento cooperativo da exceção. Este tratamento é então feito pelos demais participantes em execução (participantes que já finalizaram suas atividades não são chamados pois na implementação a thread do participante já terá sua execução finalizada) na ação.

4.4.9 Exemplo de Uso

Para mostrar como funciona esta implementação do modelo de ações CA, vamos apresentar um exemplo que consiste em duas ações CA aninhadas, com dois participantes e um objeto externo. O objeto externo consiste numa representação de um banco de dados, em que são armazenados dados de uma conta bancária. Os participantes interagem entre si e fazem operações de leitura e adição de dados no banco. O código fonte contendo a implementação de cada um dos elementos é disponibilizada no Apêndice B.

Comportamento Normal

Primeiramente, vamos mostrar o funcionamento do exemplo no caso de não ocorrer nenhum lançamento ou propagação de exceções durante a realização destas ações. Para ilustrar as operações a serem realizadas nas ações pelos participantes, mostramos o seguinte esquema da Figura 4.6.

Figura 4.6: Exemplo - Comportamento Normal

Na ação CA 0, as operações a serem feitas antes da ação CA 1 ser iniciada são:

- **Participante 0:** leitura da conta de número 1 → comunicação do número da conta a ser adicionada ao participante 1.
- **Participante 1:** obtenção do número da conta a ser adicionada ao banco de dados, informada pelo participante 0.

Na ação CA 1, as operações a serem feitas são:

- Participante 1: adição da conta no banco de dados.

Ao final da ação CA 1, as operações a serem feitas pela ação CA 0 são:

- **Participante 1:** comunicação do sucesso da operação para o participante 0.
- **Participante 0:** leitura da conta de número 1, após confirmação do sucesso da operação pelo participante 1.

Para executar este exemplo, foi criada uma classe *Simulator*, que instancia estes elementos. O código desta classe é mostrado a seguir.

```

1 package simulation;
2
3 import simulation.database.Transaction;
4 import exception.InvalidAccountNumberException;
5 import simulation.framework.CoordinatedAtomicAction;
6 import simulation.framework.Coordinator;
7 import simulation.framework.Handler;
8 import simulation.framework.Participant;
9 import simulation.framework.Role;
10 import simulation.framework.TransactionInterface;
11 import java.util.Vector;
12 import simulation.exceptionTree.ExceptionTree;
13 import simulation.participants.Participant0;
14 import simulation.participants.Participant1;
15 import simulation.participants.handlers.Participant0Handler0;
16 import simulation.participants.handlers.Participant1Handler0;
17 import simulation.participants.roles.Participant0Role0;
18 import simulation.participants.roles.Participant1Role0;
19 import simulation.participants.roles.Participant1Role1;
20
21 /**
22  *
23  * @author Simone
24  */
25 public class Simulator {
26
27     static Vector<Role> roles0;
28     static Vector<Role> roles1;
29     static Vector<Handler> handlers0;
30     static Vector<Handler> handlers1;
31     static Vector<Participant> participants0;
32     static Vector<Participant> participants1;
33     static Vector<TransactionInterface> transactions0;
34     static Vector<TransactionInterface> transactions1;
35     static Vector<Exception> exceptions;
36     static Vector<CoordinatedAtomicAction> nestedActions;
37     static Vector<CoordinatedAtomicAction> caaVector;
38
39     static CoordinatedAtomicAction caa0, caa1;
40     static Coordinator coord0, coord1;
41     static ExceptionTree excTree;
42     static Transaction t0, t1;
43     static Participant p0, p1;
44     static Role r0, r1, r2;
45     static Handler h0, h1;
46
47
48     public static void main(String args[]) {
49         //inicialização das variáveis
50         roles0 = new Vector<Role>();
51         roles1 = new Vector<Role>();
52         handlers0 = new Vector<Handler>();
53         handlers1 = new Vector<Handler>();
54         participants0 = new Vector<Participant>();
55         participants1 = new Vector<Participant>();
56         transactions0 = new Vector<TransactionInterface>();
57         transactions1 = new Vector<TransactionInterface>();
58         exceptions = new Vector<Exception>();
59         nestedActions = new Vector<CoordinatedAtomicAction>();
60         caaVector = new Vector<CoordinatedAtomicAction>();
61
62         //instanciação da árvore de resolução de exceções

```



```

63     excTree = new ExceptionTree();
64
65     //instanciação da ação CA 0, e do seu vetor de transações
66     caa0 = new CoordinatedAtomicAction(0);
67     t0 = new Transaction();
68     transactions0.add(t0);
69     //instanciação da ação CA 1, e do seu vetor de transações
70     caa1 = new CoordinatedAtomicAction(1);
71     t1 = new Transaction();
72     transactions1.add(t1);
73
74     //adição das ações CA no vetor
75     caaVector.add(caa0);
76     caaVector.add(caa1);
77
78     //instanciação do participante 0, seus roles e handlers
79     p0 = new Participant0(caaVector, 0);
80     r0 = new Participant0Role0(p0, caa0, 1); //adição no BD da conta 1
81     roles0.add(r0);
82     p0.setRoles(roles0);
83     h0 = new Participant0Handler0(p0, caa0);
84     handlers0.add(h0);
85     p0.setHandlers(handlers0);
86     //adição do participante 0 no vetor de participantes da ação CA 0
87     participants0.add(p0);
88
89     //instanciação do participante 1, seus roles e handlers
90     p1 = new Participant1(caaVector, 1);
91     r1 = new Participant1Role0(p1, caa0);
92     r2 = new Participant1Role1(p1, caa1);
93     roles1.add(r1);
94     roles1.add(r2);
95     p1.setRoles(roles1);
96     h1 = new Participant1Handler0(p1, caa0);
97     handlers1.add(h1);
98     p1.setHandlers(handlers1);
99
100    //adição do participante 1 no vetor de participantes da ação CA 0
101    participants0.add(p1);
102    //adição do participante 1 no vetor de participantes da ação CA 1
103    participants1.add(p1);
104
105    //adição da exceção InvalidAccountNumberExceptin ao vetor de
106    //exceções das ações CA. Para efeito de simplificação, apenas
107    //esta exceção será adicionada.
108    InvalidAccountNumberException exc = new InvalidAccountNumberException();
109    exceptions.add(exc);
110
111    //definição de variáveis da ação CA 0
112    caa0.setTrans(transactions0);
113    caa0.setParticipants(participants0);
114    nestedActions.add(caa1);
115    caa0.setNestedActions(nestedActions);
116    caa0.setExceptions(exceptions);
117    caa0.setExceptionTree(excTree);
118    //instanciação do coordenador da ação CA 0
119    coord0 = new Coordinator(caa0);
120    caa0.setCoordinator(coord0);
121
122    //definição de variáveis da ação CA 1
123    caa1.setTrans(transactions1);
124    caa1.setParticipants(participants1);
125    caa1.setNestedActions(new Vector<CoordinatedAtomicAction>());
126    caa1.setExceptions(exceptions);
127    caa1.setExceptionTree(excTree);

```

```

128      caa1.setParentCaa(caa0);
129      //instanciação do coordenador da ação CA 1
130      coord1 = new Coordinator(caa1);
131      coord1.setfSignal(exceptions);
132      caa1.setCoordinator(coord1);
133
134      //início da execução dos participantes
135      for (Participant partic : participants0) {
136          partic.start();
137      }
138      //início da execução dos coordenadores
139      caa0.getCoordinator().start();
140      caa1.getCoordinator().start();
141  }
142 }

```

O resultado desta execução é mostrado a seguir.

```

Participant 0: entry request – CA action 0 – coordinator started
Participant 1: entry request – CA action 0 – coordinator started

Coordinator 0: waiting.
Coordinator 0: starting.
Coordinator 0: gathering participants.
Coordinator 0: starting action.

Coordinator 1: waiting.

Participant0–Role0: starting.
Participant0–Role0: reading record 1

Account does not exist.

Participant0–Role0: setting number on Participant 1.
Participant0–Role0: number setted.

Participant1–Role0: starting.
Participant1–Role0: number setted – ok.
Participant1–Role0: nested action started.

Participant 1: entry request – CA action 1 – coordinator started

Coordinator 1: starting.
Coordinator 1: gathering participants.
Coordinator 1: starting action.

Participant1–Role1: starting
Participant1–Role1: adding record – (1, Simone, 900.00)
Participant1–Role1: record added

Participant 1: Normal Ending
commit
Coordinator 1: action finished

Participant1–Role0: nested action finished.
Participant 1: Normal Ending

Participant0–Role0: reading record 1

Account Number: 1
Name: Simone
Balance: 900.0

Participant 0: Normal Ending
commit
Coordinator 0: action finished

```

Comportamento Excepcional

Agora, vamos mostrar o funcionamento do exemplo no caso de ocorrer lançamento e/ou propagação de exceções durante a realização destas ações. As operações a serem realizadas nas ações pelos participantes são esquematizadas na Figura 4.7.

Figura 4.7: Exemplo - Comportamento Excepcional

Na ação CA 0, as operações a serem feitas antes da ação CA 1 ser iniciada são:

- **Participante 0:** leitura da conta de número 1 → comunicação do número da conta a ser adicionada ao participante 1.
- **Participante 1:** obtenção do número da conta a ser adicionada ao banco de dados, informada pelo participante 0.

Na ação CA 1, as operações a serem feitas são:

- **Participante 1:** adição da conta no banco de dados. Aqui, será lançada uma exceção porque o número da conta a ser adicionada no banco de dados será inválido (valor negativo). Como nenhum outro participante está nesta ação, esta exceção será propagada para a ação pai, a ação CA 0.

Ao final da ação CA 1, as operações a serem feitas pela ação CA 0 são:

- **Participante 0:** tratamento da exceção lançada.
- **Participante 1:** tratamento da exceção lançada.

Para executar este exemplo, é preciso alterar a linha 80 da classe Simulator para:

```
80 r0 = new Participant0Role0(p0, caa0, -1); //adição no BD da conta -1 (ERRO!)
```

O resultado da execução do exemplo com esta alteração é apresentado abaixo.

```
Participant 0: entry request - CA action 0 - coordinator started
Participant 1: entry request - CA action 0 - coordinator started

Coordinator 0: waiting.
Coordinator 0: starting.
Coordinator 0: gathering participants.
Coordinator 0: starting action.

Coordinator 1: waiting.
```

```

Participant0-Role0: starting.
Participant0-Role0: reading record 1

Account does not exist.

Participant0-Role0: setting number on Participant 1.
Participant0-Role0: number setted.

Participant1-Role0: starting.
Participant1-Role0: number setted - ok.
Participant1-Role0: nested action started.

Coordinator 1: starting.
Coordinator 1: gathering participants.

Participant 1: entry request - CA action 1 - coordinator started

Coordinator 1: starting action.

Participant1-Role1: starting
Participant1-Role1: adding record - (-1, Simone, 900.00)

Participant 1: Exceptional Ending
EXCEPTION - RAISE
rollback-excep
Coordinator 1: action finished

EXCEPTION - SIGNAL

Participant1-Handler0: starting.
Exception handling successful

Participant0-Handler0: starting.
Exception handling successful

Participant 0: Normal Ending
Participant 1: Normal Ending
commit
Coordinator 0: action finished

```

4.5 Conclusão

Neste capítulo, abordamos a teoria referente aos componentes, ao desenvolvimento baseado em componentes e ao tratamento de exceções no nível do componente e de arquitetura. Dentro desta teoria, o conceito de ações atômicas coordenadas mostrou-se o mais importante por ser o protocolo que garante a coordenação de componentes tolerantes a falhas e, por esta razão, este protocolo foi o escolhido para este trabalho.

Para a realização da implementação de propriedades para verificação de código com este protocolo, foi necessária a realização da implementação de um modelo genérico que implementasse este protocolo em Java. Por esta razão, optamos pela implementação do modelo genérico especificado em [Per07, PdM10] na linguagem formal CSP/CSPm, conforme pode ser visto na Seção 4.4. Embora existam outras implementações de coordenação de componentes via ações CA feitas em Java [CGP⁺06], uma implementação própria dá, além de um melhor entendimento do código necessário para a programação desse protocolo de coordenação, uma maior flexibilidade e segurança para programar adequadamente as propriedades referentes às ações CA na ferramenta de verificação escolhida para este trabalho, a JPF Exceptions[Xav08]. Estas propriedades e sua implementação nesta ferramenta são o tema do próximo capítulo deste documento.

Capítulo 5

Propriedades sobre Coordenação de Componentes

Neste capítulo, vamos discorrer sobre os seguintes tópicos relacionados às propriedades sobre coordenação de componentes:

- descrição de propriedades relacionadas à coordenação de componentes¹ especificadas formalmente em trabalhos da área.
- implementação das propriedades de coordenação de componentes relativas ao seu comportamento excepcional.

5.1 Propriedades sobre Ações Atômicas Coordenadas

Para modelar a coordenação de componentes, utilizamos as ações atômicas coordenadas. Por isso apresentamos, a seguir, propriedades definidas por cada um dos três trabalhos que formalizam as propriedades de ações CA. Elas foram separadas de acordo com a linguagem e método formal utilizado por cada um dos trabalhos: Lógica Temporal, Alloy e CSP.

5.1.1 Lógica Temporal

Desenvolvido por Randell et al. [RRS⁺98], este trabalho tem como objetivo formalizar as ações atômicas coordenadas, e demonstrar o seu uso na prática, por meio da implementação de protótipos e sua utilização para solução de sistemas reais. Para formalizar o conceito de ação CA, foi utilizada a Lógica Temporal Linear [Pnu77, Lam94, MP92].

As propriedades de ações CA formalizadas são divididas em quatro categorias: propriedades elementares, de interface, de *enclosure* e de tolerância a falhas. Aqui, elas são apresentadas de modo informal.

Propriedades Elementares

Caracterizam a ordenação relativa aos estados de execução de ações (aninhadas).

- Uma ação CA que termina precisa ter começado antes.
- Uma ação CA que começa deve terminar em algum tempo no futuro.

¹Conforme dito anteriormente (ver Seção 4.2.2), vamos utilizar o modelo de ações CA para a coordenação de componentes e, portanto, as propriedades a serem aqui descritas dizem respeito a este modelo.

Propriedades de Interface

Especificam entradas, saídas e possíveis resultados relacionados às pré e pós condições das ações CA.

- Se uma ação CA começa, todos os seus papéis² devem ter sido chamados antes.
- Se uma ação CA termina, todos os seus papéis devem voltar ao seu devido curso.

Os possíveis resultados provenientes da ação CA podem ser divididos em três categorias principais: normais, excepcionais e sem resultados (ocorre quando a ação é abortada). A seguir, apresentamos as propriedades relativas aos resultados e suas categorias:

- Se a ação CA começa com a pré-condição satisfeita e termina normalmente, então sua pós-condição também deve ser satisfeita.
- Se a ação CA começa com a pré-condição satisfeita e termina excepcionalmente, então uma de suas pós-condições excepcionais deve ser satisfeita.
- Se a ação CA termina com um aborto, então todas as suas modificações feitas em objetos externos devem ser desfeitas.

Propriedades de *Enclosure*

Forçam a ação CA a ter uma qualidade de membro exclusivo, ou seja: informação só pode ser transferida na entrada e saída da ação; e durante a execução de uma ação, um papel dentro da ação não pode interagir de maneira alguma com um papel ou *thread*³ que não está na ação.

Para compreender as propriedades de *enclosure*, é preciso definir alguns termos:

- **Ações estritamente serializadas:** são ações no mesmo nível de aninhamento cujas execuções não se sobrepõem em nenhum ponto no tempo;
- **Ações concorrentes:** são ações no mesmo nível de aninhamento que, num mesmo ponto no tempo, realizam suas operações;
- **Execução concorrente (intercalável) serializada:** corresponde à execução de duas ações no mesmo nível de aninhamento cujos efeitos na interseção dos objetos externos das ações são sempre equivalentes aos efeitos de execuções estritamente serializadas das duas ações.

A seguir, são enunciadas as propriedades de *enclosure* propriamente ditas:

- Para ações CA no mesmo nível de aninhamento, apenas suas execuções intercaláveis serializadas são permitidas.
- Para qualquer papel pertencente ao conjunto de papéis que estão na ação pai e não na ação CA em questão, apenas execuções intercaláveis serializadas destes papéis e da ação CA são permitidas.

Propriedades de Tolerância a Falhas

Correspondem a propriedades relevantes para o tratamento de exceções e recuperação de erros.

- Uma ação CA só finaliza normalmente se nenhuma exceção ocorre ou se a recuperação de erros é bem sucedida.

²Um *papel* especifica uma sequência de operações num conjunto de objetos

³*Thread*, neste contexto, é um agente/representante da computação.

- A finalização excepcional de uma ação CA implica na sinalização de uma exceção apropriada que especifica um resultado excepcional.
- A finalização em aborto de uma ação CA remove qualquer efeito que esta ação tivesse tido em seus objetos externos, e sinaliza uma exceção especial de aborto.
- A finalização em falha de uma ação CA ocorre se a recuperação de erros não é possível. Neste caso, uma exceção especial de falha é sinalizada.
- Se uma exceção é lançada dentro de uma ação e não pode ser tratada localmente por um papel, então todos os papéis da ação tratam a exceção cooperativamente.
- A sinalização de uma exceção numa ação aninhada faz com que todos os papéis da ação pai tratem a exceção cooperativamente.

5.1.2 Alloy

No trabalho desenvolvido por Castor Filho et al. [FRR05, CFRR09], é apresentada uma abordagem para modelar e verificar sistemas distribuídos tolerantes a falhas que utiliza o tratamento de exceção como principal mecanismo tolerante a falha. A abordagem tem como componente principal um modelo formal de ações CA, que especifica a estruturação do sistema em termos destas ações, bem como as informações relativas ao fluxo de exceção entre as ações.

Para a especificação do modelo, foi utilizada a linguagem Alloy [Jac02]. Ela é considerada uma das linguagens formais mais simples, e é baseada na lógica relacional de primeira-ordem. De modo similar a outras linguagens de especificação, Alloy suporta estruturas de dados complexas e modelos declarativos. Ademais, em Alloy, os modelos são analisados dentro de um dado escopo utilizando um analisador próprio para esta linguagem [JSS00].

A seguir, apresentamos alguns exemplos de propriedades definidas pelos autores deste trabalho como sendo as propriedades de interesse que um sistema deve satisfazer. Elas são apresentadas de modo informal, e divididas em três categorias: básicas, desejáveis, e específicas da aplicação. A formalização destas propriedades pode ser vista no trabalho de Castor Filho et al. [FRR05, CFRR09].

Propriedades Básicas

Definem as características de ações CA válidas. Eles especificam o mecanismo de tratamento de exceção coordenado e como as ações devem ser organizadas.

- Se um participante realiza um papel numa ação aninhada, ele também deve realizar um papel na ação que a contém.
- Não há ciclos no aninhamento de ações; a organização das ações no sistema é hierárquica, de modo que o grafo formado por suas definições não possua ciclos.
- O mecanismo de resolução de exceção de uma ação soluciona todas as possíveis combinações de exceções internas concorrentes, a menos que seja explicitamente declarado o contrário.

Propriedades Desejáveis

Correspondem a propriedades gerais que frequentemente são consideradas benéficas, apesar de não fazerem parte do mecanismo de ações CA. Elas descrevem requisitos importantes que muitos sistemas tolerantes a falhas devem satisfazer. Em geral, elas assumem que as propriedades básicas já foram satisfeitas.

- Ações CA do nível mais alto da hierarquia (ou seja, não são aninhadas e nem compostas) não possuem exceções externas; isto garante que o sistema é, de fato, tolerante a falhas, uma vez que todas as exceções que chegam a estas ações são tratadas.

- Todas as exceções internas de uma ação são tratadas dentro dela e, portanto, não há propagação de exceções.

Propriedades Específicas da Aplicação

Estas propriedades são regras sobre o fluxo de exceções numa aplicação baseada em ações CA específica. Elas devem ser especificadas pelo desenvolvedor de acordo com a necessidade de cada sistema.

5.1.3 CSP

Em seu trabalho, Pereira [Per07] apresenta um *framework* para a estruturação de sistemas concorrentes e tolerantes a falhas em CSP (*Communicating Sequential Processes*) [Hoa85], baseando a estruturação dos sistemas no modelo de ações CA, e permitindo a construção por aninhamento.

A linguagem formal CSP descreve o comportamento de um sistema concorrente de forma precisa, utilizando um conjunto de leis algébricas. Grosso modo, podemos dizer que CSP decompõe o sistema em sub-sistemas menores que operam de modo concorrente e interagem entre si. A unidade básica desta linguagem é o evento que indica a ocorrência de qualquer ação de interesse do sistema que afete o comportamento deste.

Para a especificação das propriedades, foi utilizado o modelo de refinamento por rastros de CSP (*traces refinement*) [RHB97]. Este modelo se baseia em sequências, ou rastros de eventos, que descrevem a execução dos processos. Um processo Q é um refinamento de P se todas as possíveis sequências de eventos em Q constituem também sequências de eventos em P.

A seguir, apresentamos o texto extraído do trabalho de Pereira que descreve informalmente as propriedades de ações CA propostas por ele. Estas propriedades foram definidas para formalização das características do modelo de aninhamento das ações CA. Sua especificação formal encontra-se no Apêndice A.2, e é explicada em detalhes no trabalho de Pereira [Per07, PdM10].

Propriedades de Sincronismo

Descrevem as condições de sincronismo exigidas no modelo de ações CA, envolvendo o momento de participação dos componentes.

- A ação CA inicia com a entrada síncrona de seus participantes.
- A ação CA termina com a saída síncrona de seus participantes.
- A ação CA só termina após ter sido iniciada anteriormente.
- A ação CA só termina após a finalização de suas ações aninhadas.

Propriedades de Condições de Término

Descrevem as possíveis condições de término de uma determinada ação CA.

- A ação CA termina normalmente se:
 1. A atividade normal da ação finaliza com sucesso, ou seja, nenhuma exceção é lançada pelos participantes.
 2. A atividade excepcional da ação finaliza com sucesso, ou seja, as exceções lançadas são completamente tratadas pelos participantes.
- A ação CA termina excepcionalmente se a atividade excepcional da ação finaliza com a propagação de uma exceção externa, ou seja, o tratamento não é finalizado adequadamente.
- A ação CA termina em aborto se a atividade da ação (normal ou excepcional) for interrompida pela ação pai.

Propriedades de Aninhamento

Descrevem as características de aninhamento das ações CA.

- Os participantes da ação CA aninhada (ou filha) são também participantes da ação pai.
- Os participantes da ação CA aninhada (ou filha) devem acessar apenas um subconjunto dos objetos externos associados à ação pai.

Propriedades de Transação

Descrevem as exigências de controle transacional sobre objetos externos a partir do início e término de uma determinada ação CA.

- Quando a ação CA inicia, transações são criadas para cada objeto externo correspondente.
- Quando a ação CA termina normalmente, todas as transações criadas nos objetos externos são confirmadas.
- Quando a ação CA termina excepcionalmente ou em aborto, todas as transações criadas nos objetos externos são desfeitas.

Propriedades de Tratamento de Exceções

Descrevem as particularidades do tratamento cooperativo e concorrente de exceções, lançadas pelos participantes de uma determinada ação CA ou propagadas de alguma ação CA aninhada.

- As exceções internas da ação CA são tratadas na própria ação.
- As exceções externas da ação CA são propagadas para a ação pai.
- Quando uma exceção é lançada, todos os participantes da ação CA correspondente devem iniciar as respectivas atividades excepcionais.
- Quando uma exceção é propagada, todos os participantes da ação CA pai devem iniciar as respectivas atividades excepcionais.
- A atividade excepcional do participante deve incluir tratamento para todas as exceções internas, incluindo as exceções resolvidas.

5.2 Implementação das Propriedades

Nesta seção, descrevemos as implementações das propriedades relativas à coordenação de componentes por meio das ações atômicas coordenadas. O código destas propriedades foi adicionado à ferramenta JPF Exceptions [Xav08] e, para sua realização, foi utilizado o modelo apresentado na Seção 4.4.

5.2.1 Programação de Propriedades Customizadas no JPF

Implementar propriedades customizadas no verificador JPF, utilizado pela ferramenta JPF Exceptions, requer basicamente a realização de dois passos:

1. Implementar a propriedade numa classe que herda a superclasse *PropertyListenerAdapter* do verificador;
2. Registrar esta propriedade para que o verificador a reconheça e faça sua checagem no código recebido como entrada.

No primeiro passo, a necessidade de se herdar de *PropertyListenerAdapter* ocorre porque esta classe engloba tanto as funções de propriedade (necessária para fazer a varredura do código em busca de violações) quanto de *listener* (necessária para fazer a observação de eventos de interesse para a propriedade). Um esquema mostrando as classes que *PropertyListenerAdapter* contém são mostrados a seguir.

Figura 5.1: Classe *PropertyListenerAdapter* e suas dependências

Os métodos de interesse utilizados para a implementação de propriedades no JPF – tanto as relativas à coordenação deste trabalho, quanto as relativas ao tratamento excepcional implementadas na JPF Exceptions – e que estão presentes nessa classe, são apresentados abaixo.

```

package gov.nasa.jpf;

...

public class PropertyListenerAdapter extends GenericProperty implements
    SearchListener, VMListener, PublisherExtension {

    //— Property interface

    //checa a violação da propriedade, indicada pelo valor booleano
    //retornado. Por padrão, o valor retornado é 'true'
    public boolean check(Search search, JVM vm) {
        return true;
    }

    //— the VMListener interface

    //método chamado a cada instrução executada pelo verificador
    public void instructionExecuted(JVM vm) {}
    //método chamado quando há a ocorrência de lançamento de exceção
    public void exceptionThrown(JVM vm) {}
    ...

    //— the SearchListener interface

    //método chamado quando o verificador faz o 'backtracking'
    public void stateBacktracked(Search search) {}
    //método chamado quando a busca por violações excede algum limite,
    //como nível de profundidade máximo da busca, e o 'heap' da JVM.
    public void searchConstraintHit(Search search) {}
    //método chamado quando a verificação é finalizada
  
```

```

public void searchFinished(Search search) {}
...
}

```

Os métodos acima são sobrescritos na implementação das propriedades customizadas, utilizando-se a anotação *@Override*. Deste modo, os códigos que compõe o corpo destes métodos são personalizados para que estes verifiquem a violação das propriedades durante a execução do JPF.

Com relação ao segundo passo para implementação de propriedades no verificador JPF, que consiste em seu registro de forma que o JPF reconheça estas implementações como propriedades a serem checadas, ele é feito automaticamente na JPF Exceptions por meio de um *listener*, que observa o evento de seleção da propriedade para checagem na interface gráfica utilizada pelo aplicativo e o registra quando o verificador é acionado. Este *listener*, foi aproveitado para a implementação da integração gráfica entre as ferramentas OConGraX e JPF Exceptions, conforme pode ser visto na Seção 6.5 e, desta maneira, para implementar as propriedades relativas à coordenação de componentes, basta fazer o passo 1 aqui descrito.

5.2.2 Escolha das Propriedades

Para este trabalho, cujo foco é a coordenação de componentes tolerantes a falhas, vamos direcionar a implementação para as propriedades relacionadas ao tratamento excepcional no contexto das ações atômicas coordenadas. Deste modo, vamos nos restringir às seguintes propriedades, baseadas nas mencionadas na Seção 5.1.

- **Propriedades de Término de Transações:**

1. Se uma ação CA termina excepcionalmente ou em aborto, todas as transações devem ser desfeitas.

- **Propriedades de Tratamento de Exceções:**

1. Se uma exceção é lançada dentro de uma ação, todos os participantes da ação CA correspondente devem tratar a exceção cooperativamente.
2. Se uma exceção é propagada de uma ação aninhada para uma ação pai, os participantes da ação pai devem tratar a exceção cooperativamente.
3. Exceções internas da ação CA são tratadas na própria ação.
4. Exceções externas da ação CA são propagadas para a ação pai.
5. A ação CA do nível mais alto da hierarquia não possui exceções externas e, portanto, não pode propagar exceções.

Embora esta seja a lista das propriedades relativas ao comportamento excepcional dos componentes no contexto das ações CA, elas não foram implementadas em sua totalidade no verificador JPF, devido a restrições deste verificador na sua aplicação no modelo de ações CA implementado neste trabalho (ver Seção 4.4), conforme descrito a seguir.

5.2.3 Limitações do JPF

A implementação das propriedades no verificador JPF implica na existência de um modelo que implemente a coordenação de componentes por meio das ações CA. Desta forma, utilizamos o modelo descrito na Seção 4.4 para este fim. Entretanto, encontramos alguns problemas para executar a verificação do JPF neste modelo:

- **Explosão de Estados:** um dos problemas a serem enfrentados pelos verificadores que utilizam *model-checking* (ver Seção 3.3.2). A explosão de estados ocorre quando o número de estados necessários para modelar um sistema excede um limite razoável, podendo acarretar

um consumo de memória maior que a quantidade física existente no hardware, resultando assim em erro. Este problema ocorre quando executamos a verificação pelo JPF no nosso modelo de ações CA implementado, devido ao fato do controle das várias threads ser feito por estruturas como laços. Isto traz como consequências os seguintes problemas:

- Tempo de execução demorado;
- Consumo de memória maior que o suportado pelo *heap* da JVM.
- **Bibliotecas Java:** bibliotecas da linguagem Java, como *java.awt*, *java.net* e *java.io* não possuem suporte adequado no verificador.

Por causa destas limitações, as implementações das propriedades só poderiam ser feitas com as seguintes restrições:

- **Interrupção da Verificação**

Para tentar evitar o problema de explosão de estados, a varredura feita pelo verificador deve ser finalizada assim que as seguintes condições sejam satisfeitas:

1. Uma exceção é lançada⁴;
2. O tratamento da exceção é realizado;
3. A ação CA-0 é finalizada.

Apesar desta restrição ser obedecida, podem ocorrer situações nas quais o consumo de memória exceda o suportado pelo *heap* da JVM. Nestas situações, a checagem das propriedades pode ter resultados incorretos e/ou incompletos.

- **Não Utilização de Bibliotecas Java Não Suportadas pelo JPF**

Para evitar problemas na verificação, é recomendável não utilizar as bibliotecas *java.awt*, *java.net* e *java.io* para instanciação do modelo de ações CA.

A restrição de interromper a verificação traz como consequência o fato de, em alguns casos, fazer com que a varredura feita pelo verificador não abranja a totalidade do programa. A restrição de não utilizar algumas bibliotecas Java não suportadas pelo JPF, por sua vez, acaba limitando o universo de programas em que a verificação das propriedades pode ser aplicada.

5.3 Propriedades Implementadas

Devido aos problemas relatados na Seção 5.2.2, as propriedades implementadas foram restritas a duas:

1. **Desfazimento de Transações (*Rollback Transactions*):** se uma ação CA termina excepcionalmente ou em aborto, todas as transações devem ser desfeitas.
2. **Tratamento das Exceções (*Exception Handling*):** se uma exceção é lançada (ou propagada) dentro de uma ação, todos os participantes responsáveis por tratá-la devem iniciar suas atividades excepcionais. Esta propriedade equivale à junção das propriedades 1 e 2 de tratamento de exceções apresentadas na Seção 5.2.2.

A implementação destas propriedades foi feita conforme descrito na Seções 6.3.2 e 5.2.1, em que são apresentadas as propriedades da JPF Exceptions. Ela é mostrada em detalhes a seguir.

⁴Caso uma exceção não seja lançada, corre-se o risco da verificação ser interrompida antes de se checar a violação da propriedade e, portanto, esta situação foi descartada

5.3.1 Propriedade 1: Desfazimento de Transações

A ideia desta propriedade é verificar se:

- caso haja propagação de exceção, a ação correspondente desfaz suas transações;
- caso haja lançamento de exceção, a ação correspondente desfaz suas transações se o tratamento não foi bem sucedido; caso contrário, as transações são confirmadas.

Para realizar esta verificação, criamos a classe *RollbackTransactionProperty*. Ela herda de *PropertyListenerAdapter* para que o JPF a reconheça como uma propriedade a ser checada, conforme consta na Seção 5.2.1. Os métodos a serem sobrescritos são:

```
//checa a violação da propriedade, indicada pelo valor booleano retornado.
public boolean check(Search search, JVM vm) {
    return true;
}
//método chamado a cada instrução executada pelo verificador
public void instructionExecuted(JVM vm) {}
//método chamado quando há a ocorrência de lançamento de exceção
public void exceptionThrown(JVM vm) {}
```

Para se certificar que a propriedade é satisfeita ou não, verifica-se a ocorrência dos seguintes eventos:

- **Propagação de exceção:** sinalizada pela chamada do método *signal* da classe *Coordinator* do framework (ver Seção 4.4), indica que a ação termina excepcionalmente e, portanto, o desfazimento das transações deve ser feito obrigatoriamente.
- **Tratamento cooperativo de exceção:** sinalizada pela chamada do método *handle* da classe *Coordinator* do framework, indica que a ação pode terminar excepcionalmente, caso o tratamento não tenha sido bem sucedido; ou normalmente, caso o tratamento tenha obtido êxito. No primeiro caso, devemos desfazer as transações obrigatoriamente; no outro, as transações devem ser confirmadas. Para observar o tipo de término da ação após a chamada do *handle*, é checado o valor da variável *requestSignal*, presente no método *respond* da mesma classe. Se o valor for *false*, o tratamento da exceção não foi bem sucedido; caso contrário, o tratamento ocorreu com sucesso.

A seguir, apresentamos o código da implementação desta propriedade com comentários que visam a auxiliar o entendimento do código.

```
1 package br.usp.jpf.property.caa;
2
3 import br.usp.jpf.listener.ListenerUtils;
4 import br.usp.jpf.property.ReflectionUtils;
5 import gov.nasa.jpf.PropertyListenerAdapter;
6 import gov.nasa.jpf.jvm.JVM;
7 import gov.nasa.jpf.jvm.MethodInfo;
8 import gov.nasa.jpf.jvm.bytecode.Instruction;
9 import gov.nasa.jpf.jvm.bytecode.InvokeInstruction;
10 import gov.nasa.jpf.search.Search;
11 import gov.nasa.jpf.jvm.ThreadInfo;
12 import java.util.Vector;
13
14 /**
15  *
16  * @author Simone Hanazumi
17  */
18 public class RollbackTransactionProperty extends PropertyListenerAdapter {
19     //vetor que contém o nome das threads que coordenam a ação CA
20     Vector<String> coordThreadNames = new Vector<String>();
```

```

21 //variável booleana que informa se a propriedade foi satisfeita ou violada
22 boolean rollbackTransProperty = true;
23 //variável que indica se uma exceção foi lançada na ação CA
24 boolean exceptionThrown = false;
25 //variável que indica se a ação termina excepcionalmente
26 boolean exceptionalEnd = false;
27 //variável que informa se a exceção foi propagada
28 boolean signal = false;
29 //variável que armazena o nome da thread coordenadora da ação CA na qual
30 //houve lançamento de exceção
31 String threadRaiseExc = "";
32 //variável que armazena o nome do método que violou a propriedade
33 String methodName = "";
34 //variável que armazena o nome da thread da primeira ação CA (ação CA-0)
35 String firstCoordinator = "";
36
37 @Override
38 //checa se a propriedade foi violada ou não
39 public boolean check(Search search, JVM vm) {
40     return rollbackTransProperty;
41 }
42
43 @Override
44 //observa o lançamento de uma exceção durante a verificação
45 public void exceptionThrown(JVM vm) {
46     exceptionThrown = true;
47 }
48
49 @Override
50 //verifica a cada instrução executada se a propriedade é satisfeita ou não
51 public void instructionExecuted(JVM vm) {
52     Instruction instruction = vm.getLastInstruction();
53     ThreadInfo ti;
54     String threadName = "";
55     //checa se a instrução pertence a uma classe de interesse
56     if (!ListenerUtils.filter(instruction)) {
57         boolean methodCalled = ReflectionUtils.isInvokeInstruction(instruction);
58         if (methodCalled) {
59             InvokeInstruction invokeInstruction = (InvokeInstruction) instruction;
60             MethodInfo methodInfo = invokeInstruction.getInvokedMethod();
61             //checa se a instrução invocada pertence a uma classe de interesse
62             if (!ListenerUtils.filter(methodInfo)) {
63                 ti = vm.getLastThreadInfo();
64                 threadName = ti.getName();
65                 //se o método inicia uma transação, então sua thread de execução
66                 //é uma thread coordenadora da ação CA
67                 if (methodInfo.getFullName().contains("beginTrans")) {
68                     //adiciona a thread ao vetor de threads coordenadoras
69                     addCoordThread(threadName);
70                 }
71                 //se uma exceção foi lançada, verifico seu tratamento
72                 if (exceptionThrown == true) {
73                     //se a thread chama o método de início do tratamento,
74                     //ela é a thread coordenadora da ação em que a exceção
75                     //foi lançada
76                     if (methodInfo.getFullName().contains("abort")) {
77                         threadRaiseExc = threadName;
78                     }
79                     //se o método chamado faz a propagação da exceção,
80                     //a finalização da ação é do tipo excepcional
81                     else if (methodInfo.getFullName().contains("signal")) {
82                         signal = true;
83                         exceptionalEnd = true;
84                     }
85                 }
86                 //se a exceção foi tratada na ação, é preciso verificar

```

```

86         //se a ação termina normalmente ou excepcionalmente
87     else if (methodInfo.getFullName().contains("respond")) {
88         if (!ti.getBooleanLocal("requestSignal")) {
89             exceptionalEnd = false;
90         } else {
91             exceptionalEnd = true;
92         }
93     }
94     //se a exceção vai ser tratada, não há propagação
95     else if (methodInfo.getFullName().contains("handle")) {
96         signal = false;
97     }
98     //se a thread em execução é a thread coordenadora da ação
99     //na qual houve lançamento de exceção, verifico sua atividade
100    //em relação às transações
101    if (threadName.equals(threadRaiseExc)) {
102        //se as transações são confirmadas, verifico se o final da
103        //ação é normal. Se não for, há violação da propriedade
104        if (methodInfo.getFullName().contains("commitTrans")) {
105            if (exceptionalEnd) {
106                methodName = methodInfo.getFullName();
107                rollbackTransProperty = false;
108                ti.printStackTrace();
109            } else {
110                coordThreadNames.remove(threadName);
111            }
112            finalizeSearch(threadRaiseExc, vm);
113        }
114        //se as transações são desfeitas, verifico se o final da
115        //ação é excepcional. Se não for, há violação da propriedade
116        else if (methodInfo.getFullName().contains("rollback")) {
117            if (exceptionalEnd) {
118                coordThreadNames.remove(threadName);
119            } else {
120                methodName = methodInfo.getFullName();
121                rollbackTransProperty = false;
122                ti.printStackTrace();
123            }
124            finalizeSearch(threadRaiseExc, vm);
125        }
126    }
127    //se a thread em execução não é a de coordenação da ação
128    //em que houve lançamento de exceção
129    else {
130        if (!signal) {
131            if (threadName.compareTo(threadRaiseExc) < 0) {
132                if (methodInfo.getFullName().contains("commitTrans")) {
133                    coordThreadNames.remove(threadName);
134                }
135            }
136            finalizeSearch(threadName, vm);
137        }
138    }
139 }
140 }
141 }
142 }
143 }
144
145 //adiciona a thread de coordenação ao vetor correspondente e define
146 //a thread da primeira ação CA
147 private void addCoordThread(String threadName) {
148     if (!coordThreadNames.contains(threadName)) {
149         coordThreadNames.add(threadName);
150         if (firstCoordinator.equals("")) {

```

```

151         firstCoordinator = threadName;
152     }
153 }
154 }
155
156 //se não há ações pendentes e se a thread em execução corresponde à
157 //thread coordenadora da primeira ação CA, a verificação é finalizada
158 private void finalizeSearch(String threadName, JVM vm) {
159     if (coordThreadNames.isEmpty() && threadName.equals(firstCoordinator)) {
160         vm.getJPF().getSearch().terminate();
161     }
162 }
163
164 @Override
165 //retorna o método que resultou na violação da propriedade
166 public String getErrorMessage() {
167     return methodName;
168 }
169 }

```

5.3.2 Propriedade 2: Tratamento das Exceções

Nesta propriedade, devemos verificar se as seguintes condições são satisfeitas quando uma exceção é lançada:

- os participantes da ação correspondente são suspensos (método *suspend* é chamado) e iniciam o tratamento da exceção (método *handle* é chamado);
- os participantes da ação correspondente são suspensos (método *suspend* é chamado) e fazem a propagação da exceção para a ação pai (método *signal* é chamado).

Para realizar esta checagem, criamos a classe *ExceptionHandlingProperty*. Ela herda de *PropertyListenerAdapter* para que o JPF a reconheça como uma propriedade a ser checada, conforme descrito na Seção 5.2.1. Os métodos a serem sobrescritos são os mesmos da propriedade anterior.

```

//checa a violação da propriedade, indicada pelo valor booleano retornado.
public boolean check(Search search, JVM vm) {
    return true;
}
//método chamado a cada instrução executada pelo verificador
public void instructionExecuted(JVM vm) {}
//método chamado quando há a ocorrência de lançamento de exceção
public void exceptionThrown(JVM vm) {}

```

Para verificar se esta propriedade é satisfeita, os seguintes eventos são observados quando uma exceção é lançada:

- **Suspensão dos Participantes:** o método *suspend* da classe *Coordinator* do framework (ver Seção 4.4) é chamado, fazendo com que as atividades dos participantes da ação sejam interrompidas. Este método marca o início efetivo do tratamento da exceção.
- **Propagação ou Tratamento Cooperativo das Exceções:** após a verificação da suspensão, é preciso decidir se a propagação da exceção será feita (chamada ao método *signal*) ou se o tratamento cooperativo será feito (chamada ao método *handle*). Se nenhum destes métodos é chamado, a exceção não foi tratada e, portanto, a propriedade é violada. Note que aqui apenas verificamos se estes métodos de tratamento de exceção são chamados, não sendo verificada a ordem de chamada destes métodos.

O código contendo a implementação desta propriedade é apresentado a seguir. Para facilitar o entendimento de seu funcionamento, foram inseridos comentários ao longo da implementação.


```

1 package br.usp.jpff.property.caa;
2
3 import br.usp.jpff.listener.ListenerUtils;
4 import br.usp.jpff.property.ReflectionUtils;
5 import gov.nasa.jpff.PropertyListenerAdapter;
6 import gov.nasa.jpff.jvm.JVM;
7 import gov.nasa.jpff.jvm.MethodInfo;
8 import gov.nasa.jpff.jvm.ThreadInfo;
9 import gov.nasa.jpff.jvm.bytecode.Instruction;
10 import gov.nasa.jpff.jvm.bytecode.InvokeInstruction;
11 import gov.nasa.jpff.search.Search;
12 import java.util.Vector;
13
14 /**
15  *
16  * @author Simone Hanazumi
17  */
18 public class HandlingExceptionProperty extends PropertyListenerAdapter {
19     //vetor que armazena as seguintes informações das threads que coordenam
20     //a ação CA na estrutura CoordThreadInfo: nome da thread e uma variável
21     //booleana que indica se na ação CA que ela coordena houve lançamento
22     //de exceção
23     Vector<CoordThreadInfo> coordThreadS = new Vector<CoordThreadInfo>();
24     //variável booleana que informa se a propriedade foi satisfeita ou violada
25     boolean handlingExcProperty = true;
26     //variável que indica se uma exceção foi lançada na ação CA
27     boolean exceptionThrown = false;
28     //variável que indica o início do tratamento excepcional
29     boolean beginningHandling = false;
30     //variável que indica a chamada aos métodos de tratamento ou propagação
31     //da exceção
32     boolean handlingExc = false;
33     //variável que armazena o nome da thread coordenadora da ação CA na qual
34     //houve lançamento de exceção
35     String threadRaiseExc = "";
36     //variável que armazena o nome da thread da primeira ação CA (ação CA=0)
37     String firstCoordinator = "";
38
39     @Override
40     //checa se a propriedade foi violada ou não
41     public boolean check(Search search, JVM vm) {
42         return handlingExcProperty;
43     }
44
45     @Override
46     //observa o lançamento de uma exceção durante a verificação
47     public void exceptionThrown(JVM vm) {
48         exceptionThrown = true;
49     }
50
51     @Override
52     //verifica a cada instrução executada se a propriedade é satisfeita ou não
53     public void instructionExecuted(JVM vm) {
54         Instruction instruction = vm.getLastInstruction();
55         ThreadInfo ti;
56         String threadName = "";
57         //checa se a instrução pertence a uma classe de interesse
58         if (!ListenerUtils.filter(instruction)) {
59             boolean methodCalled = ReflectionUtils.isInvokeInstruction(instruction);
60             if (methodCalled) {
61                 InvokeInstruction invokeInstruction = (InvokeInstruction)
62                     instruction;
63                 MethodInfo methodInfo = invokeInstruction.getInvokedMethod();
64                 //checa se a instrução invocada pertence a uma classe de interesse

```

```

64     if (!ListenerUtils.filter(methodInfo)) {
65         ti = vm.getLastThreadInfo();
66         threadName = ti.getName();
67         //se o método inicia uma transação, então sua thread de execução
68         //é uma thread coordenadora da ação
69         if (methodInfo.getFullName().contains("beginTrans")) {
70             //adiciona as informações da thread no vetor correspondente
71             addCoordThread(threadName);
72         }
73         //se uma exceção foi lançada, verifico seu tratamento
74         if (exceptionThrown == true) {
75             //se a thread inicia o aborto das ações aninhadas, então
76             //ela é a thread coordenadora da ação na qual a exceção
77             //foi lançada
78             if (methodInfo.getFullName().contains("abort")) {
79                 int k = getCoordVectorIndex(threadName);
80                 if (k != -1) {
81                     CoordThreadInfo info = coordThreads.get(k);
82                     //atualizo a informação da thread, confirmando
83                     //que ela deve tratar ou propagar uma exceção
84                     info.setExceptionThrown(true);
85                 }
86                 if (!threadRaiseExc.equals(threadName)) {
87                     threadRaiseExc = threadName;
88                     //reseto os valores das variáveis booleanas de controle
89                     handlingExc = false;
90                     beginningHandling = false;
91                 }
92             }
93             //se os participantes são suspensos, o tratamento da exceção
94             //iniciou efetivamente
95             else if (methodInfo.getFullName().contains("suspendParticipants")) {
96                 beginningHandling = true;
97             }
98             //se a thread em execução é a thread da ação na qual a exceção foi
99             //lançada, verifico se é feito algum tratamento
100            if (threadName.equals(threadRaiseExc)) {
101                if (methodInfo.getFullName().contains("signal") || methodInfo.
102                    getFullName().contains("handle")) {
103                    handlingExc = true;
104                    int k = getCoordVectorIndex(threadName);
105                    if (k != -1) {
106                        coordThreads.remove(k);
107                    }
108                }
109                //se a thread em execução está finalizando sua ação, verifico se
110                //o tratamento
111                //da exceção foi feito. Se não foi, a propriedade é violada
112                else if (methodInfo.getFullName().contains("commitTrans") ||
113                    methodInfo.getFullName().contains("rollback")) {
114                    if (beginningHandling && !handlingExc) {
115                        handlingExcProperty = false;
116                        ti.printStackTrace();
117                    }
118                    else if (beginningHandling && handlingExc) {
119                        int k = getCoordVectorIndex(threadName);
120                        if (k != -1) {
121                            coordThreads.remove(k);
122                        }
123                    }
124                    //se a propriedade não foi violada, checo se a verificação
125                    //pode ser finalizada
126                    finalizeSearch(threadName, vm);
127                }
128            }
129        }
130    }

```

```

126         //se a thread em execução não lançou nenhuma exceção, retiro ela
127         //do vetor, e checo se a verificação pode ser finalizada
128         else {
129             int k = getCoordVectorIndex(threadName);
130             if (k != -1) {
131                 CoordThreadInfo info = coordThreads.get(k);
132                 if (!info.isExceptionThrown()) {
133                     coordThreads.remove(k);
134                     finalizeSearch(threadName, vm);
135                 }
136             }
137         }
138     }
139 }
140 }
141 }
142 }
143
144 //adiciona a thread coordenadora no vetor correspondente
145 //e define a thread da primeira ação CA (ação CA-0)
146 private void addCoordThread(String threadName) {
147     if (!isInVector(threadName)) {
148         coordThreads.add(new CoordThreadInfo(threadName, false));
149         if (firstCoordinator.equals("")) {
150             firstCoordinator = threadName;
151         }
152     }
153 }
154
155 //verifica se a threadName consta no vetor de threads
156 private boolean isInVector(String threadName) {
157     for (CoordThreadInfo coordThreadInfo : coordThreads) {
158         if (coordThreadInfo.getThreadName().equals(threadName)) {
159             return true;
160         }
161     }
162     return false;
163 }
164
165 //retorna o índice da threadName no vetor
166 private int getCoordVectorIndex(String threadName) {
167     for (int i = 0; i < coordThreads.size(); i++) {
168         if (coordThreads.get(i).getThreadName().equals(threadName)) {
169             return i;
170         }
171     }
172     return -1;
173 }
174
175 //se não há ações pendentes e se a thread em execução corresponde
176 //à thread coordenadora da primeira ação CA, finaliza a verificação
177 private void finalizeSearch(String threadName, JVM vm) {
178     if (coordThreads.isEmpty() && threadName.equals(firstCoordinator)) {
179         vm.getJPF().getSearch().terminate();
180     }
181 }
182 }

```

5.4 Exemplo de Uso

Para exemplificar o uso das propriedades implementadas, vamos utilizar o exemplo da Seção 4.4.9, em que é feito o tratamento excepcional de uma exceção. Como o código do exemplo não

viola as propriedades, vamos mostrar nas próximas seções as alterações no código que acarretam na violação dessas propriedades, e o resultado parcial retornado pela ferramenta JPF Exceptions(o resultado mais completo pode ser visto no Apêndice C).

5.4.1 Propriedade 1: Desfazimento de Transações

Esta propriedade, que checa se as transações são desfeitas quando a ação termina excepcionalmente ou em aborto, é violada quando o método de confirmação (*commit*) é chamado no lugar do método de desfazimento (*rollback*), e vice-versa. Deste modo, se alterarmos o método *respond* da classe *Coordinator* do framework (ver Seção 4.4) para:

```
public void respond(boolean requestSignal) {
    if (requestSignal == true) {
        rollbackTransExcep();
    } else {
        rollbackTransExcep(); //ERRO: aqui deveria ser chamado 'commitTrans'
    }
}
```

Teríamos a violação da propriedade, e obteríamos a seguinte saída de JPF Exceptions:

```
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center

===== system under test
application: simulation.Simulator.class

===== search started: 03/08/10 13:24

simulation.framework.Coordinator.rollbackTransExcep(Coordinator.java:386)
simulation.framework.Coordinator.respond(Coordinator.java:358)
simulation.framework.Coordinator.verify(Coordinator.java:350)
simulation.framework.Coordinator.handle(Coordinator.java:331)
simulation.framework.Coordinator.choose(Coordinator.java:280)
simulation.framework.Coordinator.resolve(Coordinator.java:272)
simulation.framework.Coordinator.suspendParticipants(Coordinator.java:267)
simulation.framework.Coordinator.abort(Coordinator.java:238)
simulation.framework.Coordinator.monitor(Coordinator.java:201)
simulation.framework.Coordinator.beginAction(Coordinator.java:183)
simulation.framework.Coordinator.beginTrans(Coordinator.java:173)
simulation.framework.Coordinator.recInitRequest(Coordinator.java:163)
simulation.framework.Coordinator.run(Coordinator.java:83)

===== error #1
br.usp.jpf.property.caa.RollbackTransactionProperty
simulation.database.BankDatabase.rollback()V

===== trace #1
...
(código de rastreamento que mostra a execução que resultou na violação da propriedade)
...

===== snapshot #1
no live threads

===== results
error #1: br.usp.jpf.property.caa.RollbackTransactionProperty "simulation.database.
    BankDatabase.rollback()V"

===== search finished: 03/08/10 13:26
```

Quando a propriedade não é satisfeita, é impressa a pilha de execução da thread que violou a propriedade, para que seja possível identificar os métodos chamados que resultaram nesta violação. Em seguida, o método que causou esta situação é apresentado, seguido do código de rastreamento que apresenta a execução feita pelo verificador que resultou na violação.

5.4.2 Propriedade 2: Tratamento das Exceções

Nesta propriedade, é checado se o método *suspend* (responsável por interromper as atividades dos participantes) é chamado e, em seguida, é verificado se ou o método *handle* (responsável pelo tratamento na própria ação em que a exceção foi lançada) ou o método *signal* (responsável por propagar a exceção para a ação pai) são chamados. Com a chamada destes métodos, o tratamento cooperativo das exceções é realizado e, portanto, a propriedade é satisfeita.

Para violar esta propriedade, devemos rodar o exemplo com a seguinte alteração no método *choose*, da classe *Coordinator* do framework (ver Seção 4.4):

```
public void choose() {
    if (containsException()) {
        anyMethod(); //ERRO: não chama 'signal'
    } else {
        handle();
    }
}
```

Com esta modificação, quando a exceção deveria ser propagada, um método qualquer, que não *signal*, é chamado. Isto acarreta na violação desta propriedade, como pode ser visto na saída da ferramenta JPF Exceptions:

```
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center

===== system under test
application: simulation.Simulator.class

===== search started: 03/08/10 13:28

simulation.framework.Coordinator.rollbackTransExcep(Coordinator.java:386)
simulation.framework.Coordinator.anyMethod(Coordinator.java:321)
simulation.framework.Coordinator.choose(Coordinator.java:278)
simulation.framework.Coordinator.resolve(Coordinator.java:272)
simulation.framework.Coordinator.suspendParticipants(Coordinator.java:267)
simulation.framework.Coordinator.abort(Coordinator.java:238)
simulation.framework.Coordinator.monitor(Coordinator.java:206)
simulation.framework.Coordinator.beginAction(Coordinator.java:183)
simulation.framework.Coordinator.beginTrans(Coordinator.java:173)
simulation.framework.Coordinator.recInitRequest(Coordinator.java:163)
simulation.framework.Coordinator.run(Coordinator.java:83)

===== error #1
br.usp.jpf.property.caa.HandlingExceptionProperty

===== trace #1
...
(código de rastreamento que mostra a execução que resultou na violação da propriedade)
...

===== snapshot #1
no live threads

===== results
error #1: br.usp.jpf.property.caa.HandlingExceptionProperty

===== search finished: 03/08/10 13:29
```

De modo análogo ao que ocorreu na propriedade anterior, quando a propriedade não é satisfeita, é impressa a pilha de execução da thread que violou a propriedade, para que seja possível identificar os métodos chamados que resultaram nesta violação. No caso desta propriedade, o método que causou esta violação não é apresentado explicitamente, mas pode ser identificado ao analisar a pilha de execução da thread. Assim, após a pilha é apresentada a propriedade violada e o código de rastreamento que apresenta a execução feita pelo verificador que resultou na violação.

5.5 Conclusão

Neste capítulo apresentamos as propriedades sobre ações CA descritas formalmente em trabalhos da área [RRS⁺98, FRR05, CFRR09, Per07, PdM10], e destacamos as de real interesse neste projeto, que correspondem às propriedades que envolvem o tratamento das exceções. Apesar das limitações encontradas (ver Seção 5.2.3) para implementar as propriedades no verificador JPF [VHB⁺03, Jav], utilizado na ferramenta JPF Exceptions [Xav08], foi possível implementar duas propriedades: a propriedade de desfazimento de transações, que estabelece que no caso da ação terminar excepcionalmente ou em aborto as transações devem ser desfeitas; e a propriedade de tratamento de exceções, que define que quando há um lançamento de exceção, esta deve ser devidamente tratada ou propagada para a ação pai.

A implementação destas propriedades conta com o modelo de ações CA (ver Seção 4.4) para checar a chamada dos métodos relativos ao desfazimento e à confirmação de transações, e ao tratamento e/ou propagação de exceções nas ações CA. Estas propriedades foram adicionadas ao código da ferramenta JPF Exceptions, aumentando assim as opções de verificação da ferramenta.

No próximo capítulo, vamos mostrar a abordagem de validação e verificação, a integração das ferramentas de teste e verificação, OConGraX [HM07, NHdM09] e JPF Exceptions, e sua aplicação em código Java que utilize o protocolo de coordenação de componentes definido pelas ações CA.

Capítulo 6

Integração de Teste e Verificação

Neste capítulo apresentamos a abordagem de validação e verificação (V&V), na qual os processos de teste e verificação de software são usados de modo integrado. Resultados interessantes podem ser obtidos com esta abordagem, que pretendemos seguir neste trabalho com a integração das ferramentas OConGraX [HM07, NHdM09] e JPF Exceptions [Xav08]. Desta forma, os seguintes tópicos serão tratados aqui:

- o conceito de validação e verificação;
- a integração entre a OConGraX e a JPF Exceptions, sendo esta explicada por meio da abordagem destes itens:
 - descrição das ferramentas;
 - descrição da integração das ferramentas feita por XML e interface gráfica.

6.1 Validação e Verificação (V&V)

Os conceitos de validação e verificação são importantes no que diz respeito à qualidade do software desenvolvido. Embora pareçam iguais, estes conceitos são complementares e distintos, representando aspectos diferentes a serem assegurados pela qualidade do produto. De modo claro e sucinto, Boehm [Boe81], os definiu da seguinte maneira:

- **Validação:** certifica que o produto construído é o correto;
- **Verificação:** certifica que o produto foi construído corretamente.

Em outras palavras, a validação assegura que o produto final está de acordo com as expectativas do cliente, enquanto que a verificação assegura que a implementação do produto satisfaz suas especificações [Som06]. De uma maneira geral, pode-se afirmar que a validação é feita por meio dos testes (ver Capítulo 2), enquanto que a verificação é feita por meio da verificação formal de programas (ver Capítulo 3).

Para entender a complementaridade destes dois conceitos, devemos relembrar as definições de teste e verificação de programas. Os testes nos garantem que o software realiza suas funções e se comporta conforme o esperado, mas apenas para um certo conjunto de testes e casos de teste. A verificação formal, por sua vez, garante a corretude da implementação do programa dada sua especificação. Alguns benefícios que podem ser obtidos por meio da utilização conjunta de testes e verificação seriam, por exemplo, o uso dos resultados das atividades de teste para especificar as propriedades a serem verificadas, e o uso de especificações para guiar os casos de teste [XHdM08]. Outros resultados interessantes que podem ser obtidos fazendo uso desta característica de complementaridade seriam:

- Geração de casos de teste por meio do uso dos contra-exemplos gerados pelo *model checker* [Tra00];
- Redução do espaço de testes por meio do relatório resultante da verificação [XHdM08].

Neste trabalho, estamos interessados na **redução do espaço de testes**, por meio da integração entre as ferramentas OConGraX [HM07, NHdM09] e JPF Exceptions [Xav08]. Maiores detalhes são dados a seguir.

6.1.1 Abordagem Teste + Verificação

Devido aos benefícios da utilização conjunta de teste e verificação, muitos trabalhos foram desenvolvidos nesta área [FWA09]. Algumas abordagens de validação e verificação buscam testar a especificação formal do sistema, almejando perceber e minimizar o número de defeitos de software no início dos projetos [ABM98, CSE96, GH99, Mac00]. Outros trabalhos focam em apontar partes do programa em que um defeito é encontrado, à medida que propriedades são formalmente verificadas [DHJ⁺01, EM04, HDD⁺03, HD01, PDV01b, RRDH03].

A abordagem aqui utilizada consiste no uso dos dados de rastreamento gerados pelo verificador de modelos para reduzir o espaço de testes necessário para garantir a robustez de um programa. Ela diferencia-se das demais por focar o seu uso em sistemas tolerantes a falhas. Na Figura 6.1 um esquema ilustrando esta abordagem é mostrado.

Figura 6.1: Esquema da Abordagem Teste + Verificação [XHdM08]

Neste trabalho, pretendemos alcançar a redução de espaços de teste fazendo a integração da OConGraX com a JPF Exceptions. Esta integração permite sua aplicação a sistemas tolerantes a falhas, uma vez que ambas as ferramentas oferecem suporte aos mecanismos de tratamento de exceção Java: a OConGraX, por meio da geração do OCFG com informações adicionais de tratamento de exceção, e dos requisitos de teste para critérios que englobam tais mecanismos [SH98, SH99]; e a JPF Exceptions pela definição de propriedades relativas ao tratamento das exceções.

Para alcançar a redução do espaço de testes, a ideia é de, a partir da geração dos requisitos de teste (para certos critérios [CK99, SH99]) pela OConGraX, utilizar o verificador da JPF Exceptions para rastrear quais foram cobertos durante a execução da verificação do programa, satisfazendo as propriedades. Com isso, obtemos a quantidade de requisitos que já foram cobertos tirando a necessidade de criar casos de teste para eles e, portanto, temos o espaço de testes reduzido [XHdM08].

6.2 OConGraX

A OConGraX (*Object Control Flow Graph with eXceptions*) [NdM04, HM07], é uma ferramenta criada com o objetivo de auxiliar a realização de testes estruturais em programas Java. Suas principais características são:

- Interface Gráfica de fácil interação com o usuário;
- Leitura e Análise de um Projeto Java, de modo a devolver:
 - Definições e Usos de Objetos e Exceções;
 - Pares de definição-uso de Objetos e Exceções;
 - Grafo de Fluxo de Controle de Objetos (OCFG) com informações sobre as estruturas de tratamento de exceção (*try*, *catch*, *throw*, *finally*) utilizadas no projeto Java recebido como entrada.
- Salvar os dados de Definições e Usos de Objetos e Exceções num arquivo XML;
- Salvar o grafo gerado como arquivo de imagem.

A seguir, serão apresentados maiores detalhes sobre a arquitetura e desenvolvimento da ferramenta, além de uma descrição mais detalhada de suas funcionalidades.

6.2.1 Arquitetura da Ferramenta

A OConGraX foi desenvolvida utilizando a IDE Eclipse 3.4¹, e escrita em Java 6. Possui aproximadamente 5500 linhas de código distribuídas em 6 pacotes principais:

```
br.usp.ime.ocongrax.graph
br.usp.ime.ocongrax.program.control
br.usp.ime.ocongrax.program.data
br.usp.ime.ocongrax.program.file
br.usp.ime.ocongrax.program.file.filters
br.usp.ime.ocongrax.program.view
```

Listagem 6.1: Pacotes do código-fonte da OConGraX

Estes seis pacotes podem, ainda, ser categorizados nas quatro principais partes que compõem a arquitetura da ferramenta, esquematizada na Figura 6.2. Detalhes sobre cada parte da arquitetura são mostradas na sequência:

Figura 6.2: Esquema da arquitetura da OConGraX

¹<http://www.eclipse.org>

- **Interface Gráfica (*Graphic User Interface* ou *GUI*):** feita utilizando *Java Swing*, a interface gráfica (ver Figura 6.3) dá ao usuário acesso rápido e simples a todas as funcionalidades da OConGraX. Para facilitar a visualização das informações retornadas pela ferramenta, o painel foi dividido em duas partes, utilizando-se o componente *JSplitPane* do *Java Swing*: à esquerda, está a árvore de informações contendo todos os dados relativos às definições e usos de objetos e exceções, organizados de acordo com o método, classe e pacote a que pertencem; à direita, por meio de uma navegação por abas, é possível visualizar as definições e usos de objetos e exceções, os pares de objetos e exceções e o grafo de fluxo de controle.

Pacotes: `br.usp.ime.ocongrax.program.view`

Figura 6.3: GUI da OConGraX

- **Parser:** feito com o auxílio do *framework Recoder* [Rec], é o responsável por processar os arquivos `.java` e identificar os dados referentes a objetos e exceções (como, por exemplo, suas definições e usos). Estes dados são então armazenados em estruturas de dados criadas para facilitar seu uso na geração do grafo de fluxo de controle e na geração do arquivo XML com os dados das definições e usos de objetos e exceções.

Pacotes: `br.usp.ime.ocongrax.program.control`, `br.usp.ime.ocongrax.program.data`

- **Construtor do Grafo:** utilizando-se do *framework JGraph* [JGr], ele constrói e mostra o grafo de fluxo de controle feito por meio do processamento das informações de definições e usos de objetos e exceções obtidos anteriormente pelo *parser*.

Pacotes: `br.usp.ime.ocongrax.graph`

- **Gerador de Arquivos:** por meio de requisição feita pelo usuário via GUI, é possível gerar arquivos de imagem do grafo nos formatos `.jpg`, `.jpeg`, `.gif`, `.bmp` e `.png`, e gerar um documento XML contendo as informações de definição e uso de objetos e exceções.

Pacotes: `br.usp.ime.ocongrax.program.file`, `br.usp.ime.ocongrax.program.file.filters`

6.2.2 Funcionalidades da OConGraX

Nesta seção são descritas todas as funcionalidades da OConGraX. Para ilustrar cada uma delas, vamos utilizar o código mostrado na Listagem 6.2, adaptado de [Xav08]:

```
1 public class CatchExceptionExample {
2
3     static CatchExceptionExample obj;
4
5     public static void main(String[] args) {
```

```

6      try {
7          obj = new CatchExceptionExample();
8          obj.method();
9      } catch (Exception e) {
10         e.printStackTrace();
11     }
12 }
13 private void method() throws Exception{
14     Exception exc = new Exception();
15     throw exc;
16 }
17 }

```

Listagem 6.2: Código do exemplo

Definição, Uso e Pares do de Objetos

A implementação da definição, uso e pares do de objetos foi feita baseada nos critérios de teste descritos na Seção 2.7.1, e propostos por Chen e Kao [CK99]. A identificação de cada definição e uso é feita por meio do *Parser*: a cada definição identificada na varredura do arquivo .java, ele armazena o dado num vetor que contém todas as definições de objeto encontradas; o análogo acontece a cada uso identificado, sendo este armazenado num vetor que contém todos os usos de objeto encontrados na varredura dos arquivos. Munida com estes dados, a ferramenta verifica os pares formados, armazenando-os num vetor de pares.

Figura 6.4: Visualização das definições e usos do método *main* do exemplo 6.2, em que pode-se ver as definições e usos do objeto *obj*.

Para visualizar os dados de definições e usos de objetos, pode-se abrir a aba “Def-Use” ou visualizar na árvore de informações exibida no painel à esquerda. Para ver os pares, basta abrir a aba “Def-Use Pairs”.

Figura 6.5: Visualização dos pares do do objeto *obj*, do exemplo 6.2.

Definição, Uso e Pares du de Exceções

A implementação desta funcionalidade foi feita de acordo com o conceito apresentado na Seção 2.8, proposto por Sinha e Harrold [SH98, SH99]. A obtenção das definições e usos das exceções foi realizada de modo análogo a dos objetos: no momento que a leitura é realizada (por meio de uma varredura nas classes e métodos dos arquivos .java do sistema recebido como entrada), as definições e usos são armazenadas, respectivamente, num vetor de definições e num vetor de usos. Estes vetores armazenam todas as definições e usos de objetos e exceções do sistema.

A visualização das definições e usos das exceções pode ser feita em três locais: na aba específica para visualização de definições e usos, “Def-Use Exc”, em que apenas as definições e usos de exceções aparecem, numa notação semelhante a especificada por Sinha e Harrold [SH98, SH99]; na árvore, mostrada do lado esquerdo do *frame* da ferramenta; ou na aba “Def-Use”. Nas duas últimas, não há diferenciação nas linhas de definição e uso de objetos e exceções, ou seja, todas as linhas em que há definição e/ou uso de objetos e exceções num método (método este escolhido pelo usuário, na interface) são mostradas.

(a) Visualização das definições e usos (Aba: “Def-Use”) do método `method` do exemplo 6.2.

(b) Visualização das definições e usos (Aba: “Def-Use Exc”) do método `method` do exemplo 6.2.

(c) Visualização dos pares du da exceção `exc`, do exemplo 6.2.

Figura 6.6: Screenshots mostrando algumas das funcionalidades da *OConGraX*.

Geração do XML

Para poder compartilhar as informações de definições e usos de objetos e exceções, a ferramenta gera um arquivo com estas informações no formato XML (*eXtensible Markup Language*) [W3, Sunb].

```
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements>
3   <object>
```

```

4      <classname>...</classname>
5      <methodname>...</methodname>
6      <name>...</name>
7      <pair>
8          <def>...</def>
9          <use>...</use>
10     </pair>
11     ...
12     <def>...</def>
13     ...
14     <use>...</use>
15     ...
16 </object>
17 <exception>
18     <classname>...</classname>
19     <methodname>...</methodname>
20     <name>...</name>
21     <pair>
22         <def>...</def>
23         <use>...</use>
24     </pair>
25     ...
26     <def>...</def>
27     ...
28     <use>...</use>
29     ...
30 </exception>
31 </elements>

```

Listagem 6.3: Estrutura do arquivo XML gerado pela OConGraX

A linguagem XML foi utilizada não apenas por apresentar os dados de uma maneira estruturada, mas também por ser recomendada pela W3C (*World Wide Web Consortium*)² e ser flexível, permitindo que a estrutura a ser utilizada seja definida pelo programador.

A estrutura do arquivo XML utilizada na OConGraX é mostrada na Listagem 6.3. A versão de XML utilizada é a 1.0, e a codificação, ISO 8859-1, correspondente às línguas ocidentais.

O elemento raiz é “*elements*”. Seus filhos são “*object*” e “*exception*”:

O elemento “*object*” armazena os dados de um objeto. Possui os sub-elementos “*classname*”, “*methodname*”, “*name*”, “*pair*”, “*def*” e “*use*”. Cada sub-elemento armazena, respectivamente, o nome da classe a qual o objeto pertence, o nome do método a que ele pertence, o nome do objeto, o par du (com os dados: número da linha em que ocorre a definição, e número da linha em que ocorre o uso)³, o número da linha em que o objeto é definido, e o número da linha em que ele é usado. Estes mesmos sub-elementos são encontrados para o elemento “*exception*”, que armazena os dados relativos a uma exceção do sistema.

Na OConGraX, são oferecidas algumas opções para gerar o arquivo XML, conforme pode ser visto na Figura 6.7. Isto foi feito para dar maior flexibilidade para o usuário, permitindo que ele obtenha no XML apenas as informações relevantes necessárias para ele aplicar critérios de testes [CK99, SH99] no software que ele deseja testar.

Para o trabalho deste Mestrado, o modo como é gerado este XML foi modificado. Mais especificamente, a alteração foi feita no pacote `br.usp.ime.ocongrax.program.data`, que contém a classe `Pair.java` (responsável pelo armazenamento dos dados do par du), e no pacote `br.usp.ime.ocongrax.program.file` (responsável pela geração do arquivo XML com as informações de definições e usos de objetos e exceções), na classe `XMLGenerator.java`, para, no caso de termos um par externo (par no qual o uso de um objeto/exceção ocorre num método distinto ao da definição), podermos representar suas informações de uma maneira adequada. Maiores detalhes sobre as alterações feitas serão dadas na Seção 6.4.2.

²<http://www.w3.org>

³No caso do par (uso) pertencer a outro método, é armazenado não somente o número da linha em que ele se encontra, mas também seu nome e a classe em que ele pertence

Figura 6.7: Tela com as opções para geração do arquivo XML

Grafo de Fluxo de Controle

O grafo gerado pela ferramenta é o grafo de fluxo de controle de objetos com informações adicionais sobre os mecanismos de tratamento de exceção. Sua construção é baseada na teoria apresentada no Capítulo 2.8.

Na OConGraX, o grafo gerado é referente a um objeto de uma classe ou a uma exceção da classe. Tanto a classe quanto o objeto/exceção podem ser escolhidos pelo usuário. Além disso, o grafo pode ser editado pelo usuário⁴, e a opção “*Full Graph*”, que quando marcada faz com que sejam visualizados todos os nós dos métodos representados pelo grafo e não apenas os mais relevantes para o objeto/exceção da classe em questão, foi disponibilizada.

Figura 6.8: Visualização do grafo de fluxo de controle para a exceção *e* do exemplo 6.2

⁴Por “edição”, entende-se mover os nós do grafo

Apesar de ter se baseado no artigo de Sinha e Harrold [SH98], a implementação não obedeceu à risca a notação sugerida no texto. A seguir, explicamos alguns pontos sobre os elementos representados no grafo:

- **Representação dos Nós**

Cada nó do grafo resultante contém o número da linha percorrida e seu conteúdo. O texto contido num nó é diferenciado na ferramenta por meio de cores, da seguinte maneira:

1. Se o objeto/exceção é definido no código daquela linha, a cor do texto utilizada na ferramenta é a verde;
2. Se o objeto/exceção é usado no código daquela linha, a cor do texto utilizada na ferramenta é a azul;
3. Se o objeto/exceção é definido e usado no código daquela linha, a cor do texto utilizada na ferramenta é a roxa;
4. Se o conteúdo do nó indica o início e fim de um método, a cor do texto utilizada na ferramenta é a vermelha;
5. Se o conteúdo do nó indica o início e fim de um bloco *Finally*, a cor do texto utilizada na ferramenta é a magenta.

Ademais, para diferenciar os nós que contém linhas pertencentes a blocos definidos por mecanismos de tratamento de exceção (*try*, *catch*, *finally*), ou que contenham o comando *throw*, eles foram pintados em tom alaranjado. Os demais nós passaram a ser coloridos em tom de azul.

- **Arestas**

As **arestas de controle** (representadas por setas desenhadas com linha cheia) mostram os caminhos com os fluxos de execuções possíveis dentro de um mesmo método. No caso de ligar nós que contenham algum elemento pertencente a uma estrutura de tratamento de exceção, as arestas passam a ser laranjas, e não mais pretas.

As **arestas passa-mensagem** (representadas por setas desenhadas com linha tracejada) mostram os caminhos com os fluxos de execuções possíveis entre métodos distintos, que podem estar inclusive em classes diferentes. Com o acréscimo do caso das exceções, as arestas tracejadas passaram a ligar não apenas chamadas a métodos existentes no código recebido como entrada, mas também os nós *throw* e *catch* que formam pares de definição-uso. O uso das cores é análogo ao feito para as arestas de controle.

Além de gerar, a OConGraX permite que o usuário faça a manipulação⁵ do grafo e o salve como um arquivo de imagem. Permitir manipular o grafo dá flexibilidade ao usuário, enquanto salvar o grafo como arquivo de imagem dá a ele a chance de utilizar o grafo e as informações nele contidas em outros aplicativos de testes e verificação de programas, além de permitir que o grafo seja impresso.

6.3 Java PathFinder Exceptions

O aplicativo Java PathFinder Exceptions [Xav08] utiliza a Java PathFinder, ferramenta responsável por verificar as propriedades e gerar o rastreamento de instruções executadas. Ele possui uma interface gráfica que permite a seleção do programa e das propriedades relativas a tratamento de exceção a serem testadas, bem como exibir as estatísticas de cobertura dos critérios. Além disso, ela permite a configuração dinâmica da ferramenta JPF sem a necessidade de alteração dos arquivos de propriedades.

Maiores informações sobre o verificador Java PathFinder e das propriedades relativas a tratamento de exceção testadas são dadas a seguir.

⁵Por manipulação, entende-se mover e alterar o tamanho dos nós do grafo.

6.3.1 Java PathFinder

Java PathFinder (JPF)⁶ é um verificador de modelo desenvolvido pelo centro de pesquisas da NASA. Ele funciona como uma JVM executando um programa alvo de todas as maneiras possíveis, procurando por violações de propriedades como *deadlocks* ou exceções não tratadas. Caso um erro seja encontrado, todos os passos seguidos até a ocorrência do erro são exibidos.

Uma diferença no uso de verificação de modelos em comparação aos testes está na simulação do não determinismo. O JPF oferece duas características importantes que auxiliam nesta verificação:

- **Backtracking:** o JPF pode retornar aos passos anteriores da execução do programa, para verificar se existem estados não executados. Isto é mais eficiente do que executar novamente o programa do início.
- **State Matching:** o estado do sistema é armazenado como um par de uma amostra do *heap* e do *stack* da thread executando. Quando um estado já visitado é encontrado, o JPF executa o *backtracking* até encontrar um estado novo inexplorado.

Programas grandes podem ocasionar uma explosão de estados, que consiste na geração de um número muito grande de estados a serem verificados em busca da violação de propriedades. De forma a minimizar este problema, o JPF conta com os seguintes recursos:

1. **Estratégias de busca configuráveis:** consiste na aplicação de filtros que determinam que estados são interessantes ou não verificar.
2. **Redução do número de estados:** a ferramenta utiliza alguns mecanismos, listados a seguir:
 - **Geradores de escolha heurísticos:** consiste em restringir as escolhas de valores para teste de estado. Por exemplo, considere um valor *float* (representação de número real em Java) de entrada não-determinística, e um valor limite. Para verificar o comportamento do sistema, ao invés de gerar todos os possíveis valores do tipo *float*, bastaria comparar se ele é menor, igual ou maior do que o valor limite, reduzindo assim o número de estados a serem varridos pela verificação.
 - **Redução de ordem parcial:** é o mecanismo mais importante em programas concorrentes. Consiste em levar em consideração apenas operações que podem ter efeitos em outras threads, como instruções que alteram objetos acessíveis externamente. Para realizar esta redução dinamicamente e sem a intervenção do usuário, o JPF utiliza o *bytecode* de Java e obtém a informação de alcance dos efeitos das operações do “*Garbage Collector*” (processo de gerenciamento de memória). Com a redução de ordem parcial, operações com efeito puramente local são desconsideradas, reduzindo assim o número de estados a serem percorridos durante a verificação.
 - **Execução de VM aninhada:** o JPF executa sobre a JVM nativa. Isto permite que determinadas operações sejam delegadas para a JVM nativa ao invés de executadas pelo JPF. Desta forma, estas execuções não têm seu estado rastreado.

O JPF oferece, ainda, alguns mecanismos que permitem sua extensão:

- **Search/VMListeners:** é permitida a criação de classes Java que utilizam o padrão de projeto *Observer* para serem notificados quando ocorrem eventos como execução de determinadas instruções de *bytecode* ou avanço/retrocesso nos passos de execução. Podemos utilizar estes elementos para verificação de propriedades complexas, executar buscas, estender o modelo de estado interno do JPF ou colher estatísticas de execução do programa, como por exemplo, a sequência de instruções executada.

⁶<http://javapathfinder.sourceforge.net/>

- **Model Java Interface (MJI):** permite a separação da execução com rastreamento de estado do JPF e a execução nativa da JVM. Pode ser utilizada para construção de abstrações de APIs que reduzem a geração de estados necessária.

Apesar de seus vários pontos positivos, como o suporte a tratamentos de exceção, suporte a elementos introduzidos pelas novas versões de Java (como a “tipagem” de variáveis introduzida com o Java 5) e a extensibilidade nas propriedades testadas (além de suportar originalmente testes nas propriedades de *deadlock*, asserções e exceções não tratadas, permite a criação de novas propriedades), há alguns pontos negativos, como a inexistência de interface gráfica e o fato do JPF não permitir o uso de outros verificadores de modelo. Particularmente, a inexistência de interface gráfica é um problema que foi sanado com a JPF Exceptions.

6.3.2 Propriedades Relativas a Tratamento de Exceção

No JPF Exceptions, foram criadas algumas propriedades a serem verificadas com a ferramenta Java PathFinder. A abordagem utilizada foi selecionar algumas práticas usadas comumente no tratamento de exceções em Java e transformá-las em propriedades da ferramenta. É importante ressaltar que mesmo que não se deseje utilizar a ferramenta para diminuir o espaço de testes ou geração de casos de uso manuais, podemos utilizá-la para melhorar a qualidade do código, garantindo que certas práticas não ocorrem.

Propriedades no Java PathFinder

Um dos motivos da escolha da ferramenta JPF para utilização neste trabalho, foi a facilidade na sua customização e criação de propriedades. Todas as propriedades no JPF são classes Java que implementam uma interface comum: `gov.nasa.jpf.Property`.

Outra característica importante utilizada foi a capacidade de criar classes (*listeners*) que são notificadas pela ferramenta na ocorrência de eventos relativos a execução do código, como execução de instruções e lançamento de exceções. Para isso existem algumas interfaces que devem ser implementadas pelas classes interessadas na notificação de eventos. A criação destas classes segue o padrão de projeto *Observer*[GHJV00].

Assim, para criação de propriedades customizadas basta criar uma implementação de *Property* e registrá-la para verificação com a ferramenta. Este registro poder ser feito via customização do arquivo de inicialização padrão da ferramenta (`jpf.properties`) ou em tempo de execução via código. Normalmente, dependendo apenas da propriedade, devem também ser criadas classes *listener* para monitorar a execução da ferramenta.

No caso deste trabalho, como foi necessário criar um *listener* para cada propriedade, optou-se por utilizar uma superclasse disponível na ferramenta especificamente para estes casos denominada *PropertyListenerAdapter*. Todas as propriedades criadas são subclasses de *PropertyListenerAdapter*.

Apresentação de Práticas Relativas ao Tratamento de Exceções

Aqui apresentamos algumas práticas relativas a tratamento de exceções, que foram estudadas junto com suas respectivas propriedades. Estes estudos foram feitos, pois as propriedades de ordenação a serem definidas e implementadas neste trabalho do Mestrado serão desenvolvidas de maneira similar.

Denominamos “Tratamento de Exceções” o código criado para o tratamento das condições excepcionais. As condições excepcionais são diferentes dos *bugs*, e podem ser definidas como problemas que impedem a continuação da execução de um código [Eck02]. Exemplos de condições excepcionais envolvem a perda de uma conexão de rede, a queda de um servidor de bancos de dados, a leitura de um arquivo de entrada mal-formatado, entre outros.

Linguagens mais modernas como o Java e o C++ incluem estruturas para tratamento de exceções na própria sintaxe. Quando bem utilizadas, estas estruturas podem melhorar a legibilidade, confiabilidade e manutenibilidade de um programa[Blo01].

Para auxiliar a identificação da presença de práticas relativas ao tratamento de exceções que podem dificultar a tarefa de manutenção do sistema[McC, RS03], o seguinte conjunto de propriedades foram implementadas para o Java PathFinder:

1. **Lançar `java.lang.Exception` na assinatura do método:** ocorre quando é executada uma instrução correspondente à chamada de um método e `java.lang.Exception` pertence ao conjunto de exceções lançadas na assinatura do método. O problema é que `java.lang.Exception` é muito genérica, não fornecendo informações úteis sobre os erros que podem ocorrer no processamento do método.
2. **Lançar um grande número de exceções na assinatura do método:** ocorre quando é executada uma instrução correspondente à chamada de um método e o número de elementos no conjunto de exceções lançadas pelo método é maior do que `n` e este conjunto não está contido no conjunto de tratadores de exceção presentes no código que o chama. O problema é que isto só é válido se cada exceção necessitar de um tratamento diferenciado. Em caso contrário, apenas polui o código.
3. **Declarar uma exceção na assinatura do método que nunca é lançada:** ocorre quando é executada uma instrução correspondente à chamada de um método e existem exceções pertencentes ao conjunto de exceções lançadas pelo método, que não possuem um correspondente comando *throw* contido no corpo do método, direta ou indiretamente. Não é necessário verificar se o método é abstrato, pois é necessária uma implementação concreta para ser executada. O problema é que isto só é válido na definição de métodos abstratos, além de obrigar o desenvolvedor a capturar exceções que nunca serão lançadas.
4. **Capturar e ignorar a exceção:** ocorre quando uma exceção é lançada e seu tratamento consiste em um bloco de código vazio, isto é, ocorre quando uma exceção é capturada por um bloco *catch* vazio. O problema é que isto esconde que ocorreram erros na execução.
5. **Capturar `java.lang.Exception`:** ocorre quando é executada uma instrução dentro de um método que possui um bloco de captura de `java.lang.Exception` em seu código. O ruim é que isto pode gerar problemas se o método que está sendo tratado for modificado para lançar novas exceções que devem ser manipuladas de modo diferenciado, pois o código tratador não ficará ciente de que existe esta nova exceção.

Para exemplificar o modo de implementação de uma propriedade, apresentamos no Apêndice C o código fonte da propriedade referente ao bloco de captura vazio (identificada acima como a má prática de capturar e ignorar a exceção).

6.3.3 Funcionalidades do Java PathFinder Exceptions

Nesta seção são apresentadas as funcionalidades do Java PathFinder Exceptions. Elas permitem que o testador configure o Java PathFinder para executar o programa desejado, além de permitir a verificação das propriedades citadas anteriormente e dar estatísticas relacionadas à cobertura do código verificado.

Seleção do projeto e configuração do JPF Exceptions

A seleção do projeto a ser analisado é feita na aba `JPFConfig` (Figura 6.9). Conforme pode ser visto na Figura 6.9, há vários campos que o testador deve preencher para que a ferramenta possa localizar o projeto adequadamente:

- **Base Dir:** diretório raiz do projeto a ser analisado, com o seu código fonte (arquivos `.java`);
- **ClassPath:** diretório onde se encontram os arquivos `.class` do projeto;

Figura 6.9: Tela para seleção do projeto a ser verificado

- **Main Class:** arquivo `.java` que contém o método `main` de execução do projeto;
- **Max Depth:** configura a restrição de profundidade máxima de busca da ferramenta, durante a verificação, de modo a limitar a verificação em programas muito grandes ou que possam entrar em loops infinitos.

Seleção das propriedades

Na aba `Properties` (Figura 6.10), pode-se selecionar as propriedades a serem verificadas (ver Seção 6.3.2), tanto as de tratamento de exceção quanto as nativas, presentes inicialmente no verificador JPF.

Figura 6.10: Tela para seleção das propriedades

Seleção dos Listeners

Na aba `Listeners` (Figura 6.11), pode-se habilitar o listener `ExceptionChoiceGenerator`, que para cada método executado que declara lançar exceções, o código é executado simulando o lançamento de cada uma das exceções declaradas, bem como a execução sem lançar exceções. Por padrão, somente serão gerados lançamentos de exceção para métodos desenvolvidos pelo programador, mas é possível habilitar esta opção também para classes do próprio Java.

Figura 6.11: Tela para seleção dos listeners

Outro listener que pode ser habilitado é o do rastreamento da execução do programa. Pode-se marcar para gerar um arquivo texto com toda a execução e, ainda, fazer a verificação dos pares cobertos: basta informar o caminho para o arquivo XML com as informações de definições e usos de objetos e exceções, e ainda os locais para salvar os arquivos `ok.xml` e `notOk.xml`, que imprimem, respectivamente, as linhas de critérios percorridas e as não percorridas.

Uma observação importante a ser feita é a de que, inicialmente, estava prevista uma integração com a ferramenta OConGraX, como pode ser notado pelo campo OcongraXML. Entretanto, esta integração não foi realizada satisfatoriamente pois os modelos XML implementados em cada um era diferente. O modelo XML esperado pelo JPF Exceptions era como o mostrado na Listagem 6.4.

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements>
3   <object>
4     <classname>...</classname>
5     <name>...</name>
6     <pair>
7       <def>...</def>
8       <use>...</use>
9     </pair>
10    ...
11  </object>
12  <exception>
13    <classname>...</classname>
14    <name>...</name>
15    <pair>
16      <def>...</def>
17      <use>...</use>
18    </pair>
19    ...
20  </exception>
21 </elements>

```

Listagem 6.4: Modelo XML lido pelo JPF Exceptions

Como pode ser visto, este modelo é diferente do da OConGraX, apresentado na Listagem 6.3. Além disso, o JPF Exceptions não suportava a leitura da informação de pares externos como feita na OConGraX, na qual aparecia na tag `use` não apenas o número da linha do uso, mas também o nome da variável no método externo no qual ocorria o uso e o nome da classe a qual este uso pertencia. Desta forma, em [Xav08], foram geradas entradas XML manualmente, e não geradas pela OConGraX, para efeitos de teste do aplicativo JPF Exceptions.

Por esta razão, neste trabalho de mestrado, definimos um novo modelo XML que permite uma integração real entre essas duas ferramentas. Informações mais detalhadas serão dadas na Seção 6.4.

Estatísticas de cobertura

Na aba *Results* (Figura 6.12), aparecem as estatísticas de cobertura após a verificação. As estatísticas aparecem ao se clicar no botão *Process*. Para limpar a tela com os resultados, basta clicar em *Reset*.

Figura 6.12: Tela de estatísticas de cobertura

Rastreamento de dados

A aba *State* (Figura 6.13) exibe as estatísticas relativas aos estados gerados pela ferramenta, como número de estados visitados, por exemplo. Para que as estatísticas sejam geradas e exibidas pela ferramenta, é necessário habilitá-las selecionando o *checkbox*.

Figura 6.13: Tela de rastreamento de dados

Botão “Run”

Localizado no inferior da tela do aplicativo, é o responsável por executar o aplicativo JPF Exceptions e iniciar a verificação de propriedades. Ao final da verificação, pode exibir as seguintes mensagens:

- *No property violated*: nenhuma propriedade foi violada.

- **Property X violated:** a propriedade de nome X foi violada.
- **Constraint Hit:** a verificação é interrompida devido a uma restrição de busca, como por exemplo a restrição de profundidade configurada na primeira aba do JPF Exceptions (ver Figura 6.9).

6.4 Integração das ferramentas via XML

Conforme visto na Seção 1.2, um dos objetivos deste trabalho é a realização de uma integração das ferramentas via um documento XML, conforme pode ser visto na Figura 6.14.

Figura 6.14: Esquema da integração via XML entre OConGraX e JPF Exceptions

Para esta integração, é necessária a definição de um modelo XML que permita tal integração, a geração deste XML pela OConGraX e a leitura deste XML pela JPF Exceptions. Estas três atividades foram realizadas, e detalhes sobre sua implementação são dadas nas próximas subseções.

6.4.1 Modelo XML

Para permitir uma integração total entre as ferramentas OConGraX e JPF Exceptions, foi preciso definir um modelo XML que representasse de um modo estruturado os seguintes dados de cada elemento (objeto ou exceção) do sistema analisado:

- Sua identificação: classe e método a que pertence, e nome;
- Número das linhas de código em que ocorre uma definição e/ou um uso do elemento;
- Pares de definição-uso formados ao longo do código. Estes pares podem ser:
 - Formados no mesmo método, isto é, a definição e o uso ocorrem no mesmo método;
 - Formados em métodos diferentes, isto é, a definição ocorre num método e o uso, em outro.

Para tornar a estruturação mais simples de ser lida e entendida, foi definido um modelo XML, apresentado na Listagem 6.5:

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements>
3   <object>
4     <classname>...</classname>
5     <methodname>...</methodname>
6     <name>...</name>
7     <innerpairs>
8       <pair>
9         <def>...</def>
10        <use>...</use>
11      </pair>
12    </innerpairs>
13    <externalpairs>
14      <pair>
15        <def>...</def>
16        <externaluse>
17          <classname>...</classname>
18          <methodname>...</methodname>
19          <name>...</name>
20          <linenumber>...</linenumber>
21        </externaluse>
22      </pair>
23    </externalpairs>
24    <def>...</def>
25    <def>...</def>
26    <use>...</use>
27  </object>
28  <object>
29    ...
30  </object>
31  .
32  .
33  <exception>
34    ...
35  </exception>
36  .
37  .
38  .
39  <exception>
40    ...
41  </exception>
42 </elements>

```

Listagem 6.5: Modelo XML para integração entre OConGraX e JPF Exceptions

Neste modelo, a tag `elements` engloba todos os elementos (objetos ou exceções) do sistema que foram definidos e usados. Inicialmente, todos os objetos (marcados pela tag `object`) são mostrados para, em seguida, todas as exceções (marcadas pela tag `exception`) serem apresentadas. Dentro de cada objeto (ou exceção), aparecem as tags que identificam os valores obtidos, na seguinte ordem:

- `classname`: nome da classe (no formato: `<nome do pacote>.<nome da classe>`) a que o elemento pertence;
- `methodname`: nome do método a que o elemento pertence;
- `name`: nome do elemento;
- `innerpairs`: pares, nos quais tanto a definição quanto o uso do objeto ocorrem no mesmo método. As informações do par são fornecidas pelas tags:
 - `pair`: identifica o par;
 - * `def`: número da linha em que ocorre a definição;
 - * `use`: número da linha em que ocorre o uso;
- `externalpairs`: pares, nos quais a definição do objeto ocorre no método identificado pela tag `methodname`, e o uso ocorre em outro método. As informações do par são fornecidas pelas tags:

- pair: identifica o par;
 - * def: número da linha em que ocorre a definição;
 - * externaluse: identifica o uso externo pelas tags:
 - classname: nome da classe (no formato: <nome do pacote>.<nome da classe>) em que ocorre o uso “externo”;
 - methodname: nome do método em que ocorre o uso “externo”;
 - name: nome do elemento neste método/classe “externo”;
 - use: número da linha em que ocorre o uso neste método/classe “externo”.
- def: cada tag identifica um número de linha em que ocorre uma definição do elemento no método identificado pela tag methodname;
- use: cada tag identifica um número de linha em que ocorre um uso do elemento no método identificado pela tag methodname.

Assim, com este modelo XML, a geração do arquivo e a organização de seus dados pela OConGraX fica melhor e mais organizado (antes, na OConGraX, a identificação dos pares def-uso externos/internos não era bem estruturada e continha apenas a informação da classe “externa” e da linha do uso, omitindo a informação do método “externo”). E os dados completos são passados ao JPF Exceptions, permitindo uma análise completa do sistema por parte dele (antes, a leitura do arquivo não previa as tags com valores referentes a pares def-uso externos, o que impediu uma verdadeira integração com a OConGraX).

6.4.2 OConGraX - Geração do XML

Para gerar o modelo XML descrito anteriormente em 6.4.1, foi preciso alterar, basicamente, dois pontos no código:

- **Estrutura para armazenamento do par du**

A estrutura foi modificada de modo que as informações do uso ocorrido num método externo fossem guardadas adequadamente e recuperadas posteriormente para geração do arquivo XML.

Inicialmente, a classe que representava e armazenava as informações dos pares possuía apenas dois campos, referentes à definição e uso de objetos e exceções, como pode ser visto na Listagem 6.6. No campo uso, quando este ocorria num método distinto a da definição, era armazenada uma String contendo o número da linha em que ocorria o uso, o nome da variável no método em que o uso ocorria, e o nome da classe a que este uso pertencia.

```
public class Pair {
    String lineDef;
    String lineUse;
    ...
}
```

Listagem 6.6: Campos da classe Pair antes das alterações feitas para geração do novo XML.

Com a necessidade de se armazenar de uma maneira mais adequada as informações quando o uso ocorria fora do método em que a definição ocorria, foi preciso criar novos campos, e dois construtores, como apresentados na Listagem 6.7:

```
public class Pair {
    String lineDef;
    String lineUse;

    boolean isInnerPair;
```



```
String externalClassName;
String externalMethodName;
String externalVariableName;

public Pair(String lineDef, String lineUse, boolean isInnerPair, String
    externalClassName, String externalMethodName, String
    externalVariableName) {
    this.setLineDef(lineDef);
    this.setLineUse(lineUse);
    this.setInnerPair(isInnerPair);
    this.setExternalClassName(externalClassName);
    this.setExternalMethodName(externalMethodName);
    this.setExternalVariableName(externalVariableName);
}

public Pair(String lineDef, String lineUse) {
    this(lineDef, lineUse, true, "", "", "");
}
...
}
```

Listagem 6.7: Campos da classe *Pair* após as alterações feitas para geração do novo XML.

Com as criações dos novos campos, as informações podem ser acessadas separadamente, facilitando a recuperação delas para a geração do XML; além disso, é possível diferenciar agora a natureza do par, se ele é externo (em que o uso de um objeto/exceção ocorre num método diferente da sua respectiva definição) ou interno (definição e uso de um objeto/exceção ocorrem num mesmo método).

A criação dos construtores facilitou o armazenamento dos dados, sendo o primeiro construtor utilizado para armazenar os dados de um par externo, e o segundo, o utilizado para armazenar os dados de um par interno.

• Geração do arquivo XML

Para alterar a geração do arquivo XML, de forma que ele ficasse como o modelo proposto na Seção 6.4.1, foi preciso modificar a classe `XMLGenerator.java`, do pacote `br.usp.ime.ocongrax.program.file`. As seguintes alterações foram feitas:

- Acréscimo das tags `innerpairs` e `externalpairs`, identificando, respectivamente, os pares internos e externos de um objeto ou uma exceção;
- Criação de dois métodos utilizados para incluir os pares internos e externos, apresentando os dados de cada um de modo organizado e estruturado.

A classe `XMLGenerator.java`, com as modificações citadas acima, pode ser vista no Apêndice D.1.1.

6.4.3 JPF Exceptions - Leitura do XML

Para que fosse possível ler este novo XML, foi preciso alterar, essencialmente, dois pontos no código do JPF Exceptions:

• Estrutura de `OcongraElement`

Na classe `OcongraElement`, foi adicionado o campo `isInnerPair` que passará a ser utilizado para verificar se um par é interno ou externo. Esta informação é colhida durante a leitura do arquivo XML gerado pela `OConGraX`, e será utilizada na geração dos arquivos XML de rastreamento (`ok.xml`, `notOk.xml`).

```

public class OcongraElement {
    private String className;
    private String name;
    private boolean isInnerPair; //new field
    private boolean object;
    private int lineDef;
    private String threadDef;
    private int lineUse;
    private String threadUse;
    private boolean ok;
    ...
}

```

Listagem 6.8: Campos da classe *OcongraElement*.

- **Método `readOcongraXML`**

Método responsável pela leitura do arquivo XML da OConGraX, foi modificado para reconhecer corretamente os pares du externos e internos representados no arquivo. Este método encontra-se na classe `ExecutedPath.java`, responsável pela verificação da cobertura do código, e está disponibilizado no Apêndice D.2.1.

- **Método `generateXmlFile`**

Método encontrado na classe `ExecutedPath.java`, ele é responsável pela geração dos arquivos `ok.xml` e `notOk.xml`. Foi alterado para exibir, ao invés da tag `pair`, as tags `innerpair` quando o par a ser impresso é interno, e `externalpair` quando o par a ser impresso é externo. Para isso, a verificação apresentada na Listagem 6.9 foi acrescentada ao método, onde `myElement` corresponde a um `OcongraElement`.

```

if (myElement.isInnerPair())
    p = doc.createElementNS(null, "innerpair");
else
    p = doc.createElementNS(null, "externalpair");

```

Listagem 6.9: Verificação feita para a identificação mais específica dos pares du nos arquivos `ok.xml` e `notOk.xml`.

Os demais dados referentes ao uso num par externo não foram adicionados nos arquivos XML gerados pelo JPF Exceptions porque tais informações podem ser encontradas no XML gerado pela OConGraX.

6.4.4 Exemplo

Para exemplificar a integração, geramos com a OConGraX o arquivo `input.xml` com todas as informações de definição e uso de objetos e exceções do exemplo da Listagem 6.2. O XML gerado é apresentado na Listagem 6.10.

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements>
3   <object>
4     <classname>(default package).CatchExceptionExample</classname>
5     <methodname>main(String)</methodname>
6     <name>obj</name>
7     <innerpairs>
8       <pair>
9         <def>7</def>
10        <use>8</use>
11      </pair>

```

```

12      <pair>
13          <def>8</def>
14          <use>8</use>
15      </pair>
16  </innerpairs>
17  <def>7</def>
18  <def>8</def>
19  <use>8</use>
20 </object>
21 <exception>
22     <classname>( default package ). CatchExceptionExample</classname>
23     <methodname>main (String)</methodname>
24     <name>e</name>
25     <innerpairs>
26         <pair>
27             <def>9</def>
28             <use>10</use>
29         </pair>
30     </innerpairs>
31     <def>9</def>
32     <use>9</use>
33     <use>10</use>
34 </exception>
35 <exception>
36     <classname>( default package ). CatchExceptionExample</classname>
37     <methodname>method ()</methodname>
38     <name>exc</name>
39     <innerpairs>
40         <pair>
41             <def>14</def>
42             <use>15</use>
43         </pair>
44     </innerpairs>
45     <externalpairs>
46         <pair>
47             <def>15</def>
48             <externaluse>
49                 <classname>( default package ). CatchExceptionExample</classname>
50                 <methodname>main (String)</methodname>
51                 <name>e</name>
52                 <linenumber>9</linenumber>
53             </externaluse>
54         </pair>
55     </externalpairs>
56     <def>14</def>
57     <def>15</def>
58     <use>15</use>
59 </exception>
60 </elements>

```

Listagem 6.10: Arquivo *input.xml* gerado pela *OConGraX*.

Em seguida, utilizando o JPF Exceptions, fazemos a verificação de algumas das propriedades referentes ao *catch*: bloco *catch* vazio e exceção não capturada, sendo que nenhuma é violada pelo exemplo (ver Figura 6.15). Ademais, utilizamos como entrada o arquivo *input.xml* gerado pela *OConGraX* para gerar as estatísticas de cobertura de código e os arquivos de rastreamento *ok.xml* e *notOk.xml* (ver Listagens 6.11 e 6.12). No arquivo *ok.xml*, todos os pares aparecem pois não ocorre violação alguma de propriedades no exemplo e, portanto, nenhum par é mostrado no arquivo *notOk.xml*. Além disso, em *ok.xml*, aparecem as tags *threaddef* e *threaduse*, que identificam as *threads* que executaram, respectivamente, a definição e o uso de um determinado par.

(a) Configurando o JPF Exceptions.

(b) Seleção das propriedades a serem verificadas.

(c) Configuração dos listeners.

(d) Estatísticas de cobertura.

(e) Dados do rastreamento.

Figura 6.15: Execução no Java Pathfinder Exceptions

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements>
3   <object>
4     <classname>(default package).CatchExceptionExample</classname>
5     <methodname>main(String)</methodname>
6     <name>obj</name>
7     <innerpair>
8       <def>7</def>
9       <threaddef>main</threaddef>
10      <use>8</use>
11      <threaduse>main</threaduse>
12    </innerpair>
13    <innerpair>
14      <def>8</def>
15      <threaddef>main</threaddef>
16      <use>8</use>
17      <threaduse>main</threaduse>
18    </innerpair>
19  </object>
20  <exception>
21    <classname>(default package).CatchExceptionExample</classname>
22    <methodname>main(String)</methodname>
23    <name>e</name>
24    <innerpair>
25      <def>9</def>
26      <threaddef>main</threaddef>
27      <use>10</use>
28      <threaduse>main</threaduse>
29    </innerpair>
30  </exception>
31  <exception>
32    <classname>(default package).CatchExceptionExample</classname>
33    <methodname>method()</methodname>
34    <name>exc</name>
35    <innerpair>
36      <def>14</def>
37      <threaddef>main</threaddef>
38      <use>15</use>
39      <threaduse>main</threaduse>
40    </innerpair>
41    <externalpair>
42      <def>15</def>
43      <threaddef>main</threaddef>
44      <use>9</use>
45      <threaduse>main</threaduse>
46    </externalpair>
47  </exception>
48 </elements>

```

Listagem 6.11: Arquivo *ok.xml* gerado pelo JPF Exceptions.

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <elements/>

```

Listagem 6.12: Arquivo *notOk.xml* gerado pelo JPF Exceptions.

6.5 Integração das Ferramentas via Interface Gráfica

A integração das ferramentas via XML (ver Seção 6.4) permitiu a comunicação entre a OConGraX e a JPF Exceptions, de modo que o usuário possa usufruir das funcionalidades providas pelo uso conjunto de validação e verificação (ver Seção 6.1).

Embora a comunicação por arquivos XML seja útil, ela ainda exigia que o usuário acessasse as ferramentas separadamente, o que interfere na praticidade de se utilizar tal abordagem. Por esta razão, foi implementada uma interface gráfica que integrasse sob um único aplicativo as ferramentas OConGraX e JPF Exceptions (conforme previsto na Seção 1.2). Este aplicativo, nomeado Java V&V, disponibiliza ao usuário todas as funcionalidades de ambas as ferramentas, facilitando assim a aplicação da abordagem de validação e verificação num dado sistema Java.

Figura 6.16: *Java V&V*

6.5.1 Interface Gráfica da Java V&V

Na Figura 6.17, podemos observar como as ferramentas OConGraX e JPF Exceptions foram integradas na interface deste aplicativo.

Figura 6.17: *GUI da Java V&V*

À esquerda, o painel contendo a árvore com os dados do sistema (pacotes, classes, campos, métodos, parâmetros, definições e usos) pode ser visualizada. Esta árvore pertence à OConGraX e, para sua visualização é necessário carregar um projeto por meio do menu “File”. À direita, na parte superior, encontram-se as abas da ferramenta OConGraX; já na parte inferior, encontram-se as abas referentes à JPF Exceptions. A divisão por meio dos componentes de *Java Swing*, *JSplitPanes*,

permite que o usuário ajuste a visualização conforme sua necessidade, recolhendo o painel com a árvore de dados e/ou recolhendo o painel com uma das ferramentas.

Apesar de integradas sob este aplicativo, foi mantida uma certa independência no funcionamento de cada uma das ferramentas, de modo que o usuário possa utilizar as funcionalidades de apenas uma ou outra. Assim, caso o usuário deseje utilizar apenas a JPF Exceptions, ele pode assim fazê-lo, sem ter que, para isso, utilizar a OConGraX para gerar um arquivo XML contendo informações de definições e usos de um pacote do sistema a ser testado.

6.5.2 Adaptações da Interface para Integração

Para facilitar o acesso do usuário às funcionalidades das ferramentas, foram feitas algumas adaptações. Elas são relatadas a seguir, separadas por ferramenta.

OConGraX

Para a Java V&V, foram feitas as seguintes adaptações para utilização da OConGraX:

- **Menus:** o menu “Main” foi renomeado para “File”, mantendo os mesmos itens. O menu “View”, embora tenha o nome mantido, não possui os itens para visualização das abas da OConGraX. Ao invés disso, ele possui as opções para visualização de arquivos XML e de imagem (ver Seção 6.5.3).
- **Abas:** com a retirada das opções de visualização das abas do menu “View”, as quatro abas da ferramenta foram colocadas no painel da OConGraX, de modo semelhante à ferramenta JPF Exceptions (ver Figura 6.17). Não foram feitas alterações adicionais ao conteúdo de cada aba.

JPF Exceptions

As adaptações feitas na JPF Exceptions são referentes à integração com a OConGraX e à inserção de novas propriedades para verificação com o JPF. Por isso, as alterações foram feitas apenas em duas abas: “Propriedades” e “Listeners”.

Na aba “Listeners”, foi adicionada a opção de utilizar o último arquivo XML gerado pela OConGraX, como pode ser visto na Figura 6.18. Caso o usuário não deseje esta opção, ele pode escolher um arquivo XML salvo em seu computador.

Figura 6.18: Java V&V - Aba “Listeners” da JPF Exceptions.

Na aba “Propriedades” (ver Figura 6.19), por sua vez, foram adicionadas as propriedades relacionadas às ações CA que foram implementadas neste trabalho (ver Seção 5.2).

Figura 6.19: Java V&V - Aba “Properties” da JPF Exceptions.

6.5.3 Novas Funcionalidades

Para melhorar a utilização das funcionalidades de validação e verificação proporcionadas pela ferramenta, foram adicionadas duas novas funcionalidades:

- **Visualização de arquivo XML:** ao clicar neste item, é solicitado ao usuário a seleção de um arquivo XML. Este arquivo será então aberto numa nova janela do aplicativo Java V&V para que o usuário possa visualizá-lo. Esta funcionalidade está disponível no menu “View”, e também pode ser selecionada para que, logo após a geração do arquivo XML pela OConGraX, este seja visualizado pelo usuário.
- **Visualização de arquivo de imagem:** análogo ao item anterior, para visualização de arquivos de imagem do grafo gerado pela OConGraX.

6.5.4 Exemplo

Para exemplificar o uso da Java V&V, vamos utilizar o código do exemplo da Seção 4.4.9. Neste exemplo, temos duas ações CA (ver Seção 4.3) aninhadas, e a ocorrência de propagação e tratamento de exceções.

Ao abrir o projeto, a ferramenta OConGraX insere os dados referentes às definições e usos dos objetos e exceções do exemplo. Na Figura 6.20, podemos observar a árvore gerada e o OCFG do objeto *coord0*, que corresponde ao coordenador da primeira ação CA do exemplo, na classe *Simulator*.

Figura 6.20: *Java V&V: visualização da árvore de definições e usos e do OCFG*

Após a abertura do projeto, podemos gerar o XML com todas as informações de definições, usos, e pares du de objetos e exceções. Para este exemplo, optamos por gerar o arquivo com os dados colhidos da classe *Simulator* do exemplo. Este arquivo é apresentado parcialmente a seguir, contendo as informações dos objetos *coord0* e *coord1*, que correspondem às threads coordenadoras das ações CA.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<elements>
...
<object>
  <classname>simulation.Simulator</classname>
  <methodname>main(String)</methodname>
  <name>coord0</name>
  <innerpairs>
    <pair>
      <def>119</def>
      <use>120</use>
    </pair>
  </innerpairs>
  <def>119</def>
</object>
<object>
  <classname>simulation.Simulator</classname>
  <methodname>main(String)</methodname>
  <name>coord1</name>
  <innerpairs>
    <pair>
      <def>130</def>
      <use>131</use>
    </pair>
    <pair>
      <def>131</def>
      <use>132</use>
    </pair>
  </innerpairs>
  <def>130</def>
  <def>131</def>
  <use>131</use>
</object>
...
</elements>
```

Feito isso, vamos acionar a JPF Exceptions para verificar não apenas as propriedades, mas

também os pares que são cobertos durante sua execução. Primeiramente, fazemos a seleção do projeto Java a ser verificado, que consiste no exemplo aqui utilizado.

Figura 6.21: Seleção do projeto a ser verificado

Em seguida, selecionamos a(s) propriedade(s). Neste caso, devemos necessariamente selecionar uma das propriedades relativas às ações CA implementadas neste trabalho, para que a busca termine em tempo razoável (ver Seção 5.2.3). Também é necessário que a propriedade selecionada não seja violada pois, caso contrário, o rastreamento dos pares do não ocorrerá.

Figura 6.22: Seleção das propriedades

Antes de executarmos a verificação, devemos selecionar o arquivo XML gerado pela OConGraX e o caminho dos arquivos *ok.xml* e *notOk.xml*, que conterão, respectivamente, os pares cobertos e não cobertos durante a busca por violações das propriedades. Em seguida, clicamos no botão *Run* para iniciar a verificação.

Figura 6.23: Configuração dos listeners

Terminada a verificação, observamos que o resultado retornado foi *ConstraintHit: SIZE*. Isto ocorreu porque a verificação e o rastreamento dos pares du foram interrompidos devido ao fato do consumo de memória exceder a quantidade disponível no *heap* da JVM. No caso de executarmos apenas a checagem da propriedade, sem fazer o rastreamento dos pares du cobertos, o resultado retornado seria: *No property violated*. Apesar disso, como a propriedade não chegou a ser violada, os arquivos *ok.xml* e *notOk.xml* foram gerados, e é possível ver a porcentagem de cobertura dos pares du da classe *Simulator* do exemplo.

Figura 6.24: Estatísticas de cobertura

Após ver as estatísticas de cobertura, podemos ver os dados da própria busca na aba “State”.

Figura 6.25: Estatísticas da busca

6.5.5 Análise da Cobertura dos Pares du

Pelo fato da execução da classe *Simulator* ser uma das primeiras a ocorrer durante a verificação, os dados de cobertura de seus pares du estão completos apesar da interrupção da verificação devido a problemas de estouro de memória. Assim sendo, vamos analisar os pares du não cobertos. O arquivo *notOk.xml* que contém tais pares é mostrado a seguir.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<elements>
  <object>
    <classname>simulation.Simulator</classname>
    <methodname>main(String)</methodname>
    <name>caa0</name>
    <innerpair>
      <def>128</def>
      <threaddef>main</threaddef>
      <use>139</use>
      <threaduse>Thread-3</threaduse>
    </innerpair>
  </object>
  <object>
    <classname>simulation.Simulator</classname>
    <methodname>main(String)</methodname>
    <name>caa1</name>
    <innerpair>
      <def>123</def>
      <threaddef>main</threaddef>
      <use>124</use>
      <threaduse>main</threaduse>
    </innerpair>
  </object>
  <object>
    <classname>simulation.Simulator</classname>
    <methodname>main(String)</methodname>
    <name>excTree</name>
    <innerpair>
      <def>117</def>
      <threaddef>Thread-3</threaddef>
      <use>127</use>
      <threaduse>main</threaduse>
    </innerpair>
  </object>
  <object>
    <classname>simulation.Simulator</classname>
    <methodname>main(String)</methodname>
```

```

    <name>exceptions</name>
    <innerpair>
      <def>116</def>
      <threaddef>Thread-3</threaddef>
      <use>126</use>
      <threaduse>main</threaduse>
    </innerpair>
    <innerpair>
      <def>126</def>
      <threaddef>main</threaddef>
      <use>131</use>
      <threaduse>main</threaduse>
    </innerpair>
  </object>
</elements>

```

Observando os dados deste arquivo, obtemos os seguintes pares do no código:

```

//Objeto caa0
caa1.setParentCaa(caa0);
caa0.getCoordinator().start();

//Objeto caa1
caa1.setTrans(transactions1);
caa1.setParticipants(participants1);

//Objeto excTree
caa0.setExceptionTree(excTree);
caa1.setExceptionTree(excTree);

//Objeto exceptions
caa0.setExceptions(exceptions);
caa1.setExceptions(exceptions);
//_____
caa1.setExceptions(exceptions);
coord1.setfSignal(exceptions);

```

Como podemos ver, estes pares não são pares du de verdade, e são compostos ou por duas definições, ou por dois usos. Isto ocorre porque o algoritmo da ferramenta OConGraX para obtenção destes pares, quando não consegue garantir se uma linha é de definição ou de uso, considera esta linha como sendo tanto de definição quanto de uso, criando assim pares du a mais e que na verdade não existem. E são justamente estes pares que não foram cobertos durante a execução da verificação no código de exemplo aqui utilizado.

6.6 Conclusão

Neste presente capítulo, apresentamos o conceito de validação e verificação (V&V) e suas vantagens, juntamente com a proposta de integração das ferramentas OConGraX e JPF Exceptions para esta finalidade. Estas ferramentas foram descritas de modo a detalhar seu funcionamento e como esta integração se daria: via XML e via uma interface gráfica que originaria o aplicativo Java V&V.

Desta forma, mostramos neste capítulo o resultado da integração e a aplicação da abordagem de validação e verificação em código Java, tanto um código simples (ver Seção 6.4.4) quanto um código que envolve a coordenação de componentes juntamente com as ações CA (ver Seção 6.5.4). Com isso, alcançamos o objetivo maior deste trabalho, a completa integração das ferramentas OConGraX e JPF Exceptions, e a aplicação prática do conceito de validação e verificação.

Capítulo 7

Conclusões

Neste capítulo apresentamos as conclusões finais relativas ao trabalho desenvolvido. Abordamos os seguintes tópicos:

- contribuição científica do trabalho;
- trabalhos futuros.

7.1 Contribuição Científica

Os resultados obtidos com o desenvolvimento deste trabalho compõem sua contribuição científica. Eles são descritos a seguir:

- **Framework Java para ações CA**

Baseado na especificação em CSP/CSPm da coordenação de componentes por meio das ações atômicas coordenadas [Per07, PdM10], este *framework* foi desenvolvido para tornar possível a implementação de propriedades relativas a este protocolo na ferramenta JPF Exceptions [Xav08]. Ele corresponde a uma implementação genérica, podendo ser utilizado conforme a vontade do usuário.

O código do modelo implementado neste *framework* foi escrito em **Java (SE)**. Tal decisão foi feita após uma análise das possíveis alternativas, descritas a seguir:

- **Java EE/EJB:** plataforma de desenvolvimento específica para sistemas corporativos desta linguagem, não foi utilizada devido à incompatibilidade de uso desta plataforma com o verificador escolhido para a realização deste trabalho.
- **JPA (Java Persistence API):** utilizado para implementação de transações entre o sistema escrito em Java (SE ou EE) e objetos externos, como banco de dados, não foi usado neste trabalho pelo fato de não permitir o aninhamento de transações [KS06]. Desta forma, se utilizássemos tal API, não conseguiríamos implementar o *framework* baseado na especificação CSP/CSPm, uma vez que esta especificação prevê o uso da característica de aninhamento de ações CA e, consequentemente, prevê o uso de transações aninhadas.

Ademais, pelo fato da implementação estar embasada numa especificação formal, o usuário poderá verificar propriedades relativas tanto à especificação, quanto à implementação do modelo especificado. E isso diferencia este *framework* de outros baseados no protocolo de coordenação de componentes via ações CA [CGP⁺06].

- **Implementação de propriedades na ferramenta JPF Exceptions**

A implementação de propriedades relativas à coordenação de componentes tolerantes a falhas, permite garantir que, caso uma exceção seja lançada no sistema, ela seja efetivamente tratada

e as transações sejam desfeitas ou confirmadas de acordo com o término da ação CA. Para realizar estas implementações, foi necessário considerar as seguintes limitações do verificador JPF [VHB⁺03, Jav] utilizado a JPF Exceptions:

- **Explosão de Estados:** pelo fato do *framework* implementado neste trabalho utilizar várias threads para realizar a coordenação de componentes via ações CA, e pelo fato do controle ser feito por estruturas como laços, ocorre uma explosão de estados durante a busca por violações de propriedades feita pelo JPF. Com isso, o tempo para execução de uma busca numa simulação simples que utilize o *framework* fica muito demorado, além de poder acarretar num uso de memória que ultrapasse o limite disponível no *heap* da JVM.

Solução: para tentar evitar este problema, optamos por interromper a busca assim que observarmos os eventos: (1) lançamento de exceção; (2) tratamento da exceção; (3) finalização da ação CA-0 é finalizada.

- **Bibliotecas *java.awt*, *java.net* e *java.io*:** o verificador JPF não oferece suporte adequado para os métodos nativos destas bibliotecas Java.

Solução: neste caso, optamos por evitar o uso destas bibliotecas nos códigos dos programas e dos exemplos implementados para este trabalho. Também é recomendado que o usuário evite o uso de qualquer dessas bibliotecas quando for utilizar a verificação feita na JPF Exceptions.

- **Integração da OConGraX [HM07, NHdM09] e JPF Exceptions**

Integrar estas ferramentas de teste e verificação, respectivamente, permitiu a utilização efetiva da abordagem de validação e verificação em código Java. Esta integração, feita via XML e via interface gráfica, originou o aplicativo Java V&V, que engloba todas as funções de ambas as ferramentas, garantindo, por meio da OConGraX, a geração do grafo de fluxo de controle de objetos e exceções e dos pares du para aplicação dos critérios de definições e usos [SH99]; e por meio da JPF Exceptions, a verificação de propriedades relativas a *deadlock*, tratamento de exceções e coordenação de componentes tolerantes a falhas via ações CA.

7.2 Escrita de Artigos

Nesta seção é mostrada a relação de artigos publicados durante este curso de Mestrado, bem como a relação de artigos em processo de escrita neste momento, cada um seguido de uma breve descrição.

7.2.1 Artigos Publicados

SEFM'08

O artigo “*Using Formal Verification to Reduce Test Space of Fault-Tolerant Programs*” [XHdM08] foi aceito e publicado no SEFM'2008 (*6th IEEE International Conferences on Software Engineering and Formal Methods*), evento ocorrido em Novembro de 2008 na Cidade do Cabo, África do Sul. Ele foi escrito por Kleber S. Xavier, autor da ferramenta JPF Exceptions, por mim, Simone Hanazumi, e por minha orientadora, Prof^a Ana C. V. de Melo.

Neste artigo, em anexo no apêndice 7.2.1, foi tratada a questão da integração entre testes e verificação de sistemas tolerantes a falhas escritos em Java, utilizando a OConGraX e o Java PathFinder Exceptions. Com esta integração, é possível reduzir o espaço de testes a serem feitos segundo os critérios de teste de definição/uso de exceção, reduzindo assim os custos na execução de tarefas relacionadas a validação e verificação de tais programas.

TESTCOM'09

O artigo “*OConGraX - automatically generating data-flow test cases for fault-tolerant systems*” foi escrito no primeiro semestre de 2009 por Paulo R. F. Nunes, autor da ferramenta OConGra (versão que precedeu a OConGraX), por mim, Simone Hanazumi, e por minha orientadora, Prof^a Ana C. V. de Melo. Ele está em anexo no apêndice 7.2.1, e foi publicado no TESTCOM/FATES 2009 (*Joint Conference of the 21st IFIP Int. Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software*), evento realizado em Novembro de 2009 em Eindhoven, Holanda.

O assunto tratado no artigo é a ferramenta OConGraX e sua utilização para gerar automaticamente, do código-fonte de programas escritos em Java, casos de teste de fluxo de dados de objetos e mecanismos de tratamento de exceção, além de gerar dados (como grafo de fluxo de controle) para uma análise mais detalhada do programa Java recebido.

7.2.2 Artigos em Julgamento e em Processo de Escrita

Junto com Paulo R. F. Nunes e minha orientadora, Prof^a Ana C. V. de Melo, finalizamos um artigo detalhado sobre a OConGraX, o qual trata desde sua arquitetura até sua utilização para geração de requisitos de teste de fluxo de dados de objetos e mecanismos de exceção, e análise de programas orientados a objetos. Intitulado “*OConGraX - A tool for data-flow test requirements*”, este está sendo submetido a um periódico especializado na área de ferramentas para desenvolvimento de software (em fase final de formatação).

Relativo à segunda fase do projeto, no qual foram desenvolvidos implementação de um modelo de coordenação de componentes tolerantes a falhas juntamente com um conjunto de propriedades formais para o JPF (*Java PathFinder*), eu e a minha orientadora, Prof^a Ana C. V. de Melo, estamos escrevendo um artigo para um periódico. Este artigo descreve a implementação do modelo de coordenação em detalhes, as propriedades genéricas no JPF, e como ambos podem ser instanciados para sistemas baseados em componentes de forma que tenhamos a implementação da coordenação e possamos provar as propriedades sobre tal coordenação. Este artigo será submetido a um periódico sobre desenvolvimento formal de software.

7.3 Trabalhos Futuros

A realização deste trabalho proporcionou uma experiência única de pesquisa na área de teste e verificação de software. E embora os resultados obtidos sejam interessantes, muitas coisas podem ser feitas para aprimorá-los e permitir seu uso por um número maior de desenvolvedores. Por esta razão, citamos os seguintes trabalhos que podem ser feitos futuramente:

- **Estudo da performance das ferramentas:** fazer um estudo estatístico mostrando a performance de OConGraX e JPF Exceptions em código Java de aplicativos reais. Deste modo, conseguiremos determinar as classes de programas em que elas poderiam ser utilizadas;
- **Estudo do ganho efetivo com a aplicação da abordagem V&V:** fazer um estudo estatístico que comprove a eficiência do uso desta abordagem nos testes de programas;
- **Implementar *plugin* para IDEs de Java:** criar um plugin do aplicativo Java V&V, que engloba a OConGraX e a JPF Exceptions, para sua utilização no ambiente de desenvolvimento de código Java, como Netbeans e Eclipse. Isto permitiria que mais usuários utilizassem esta ferramenta, e facilitaria a criação de testes e o processo de verificação de programas.
- **Implementar novas funcionalidades na Java V&V, OConGraX e JPF Exceptions:** implementar novas funcionalidades na OConGraX e na JPF Exceptions, e agregá-las posteriormente à Java V&V permitiria fornecer uma gama mais completa de cobertura de testes e propriedades a serem verificadas no código. Mais especificamente, poderíamos acrescentar nas ferramentas:

- **OConGraX:** além de auxiliar a geração de casos de teste em testes baseado nos critérios de definições e usos, acrescentar funcionalidades que auxiliem na geração de casos de testes baseados em outros critérios;
- **JPF Exceptions:** acrescentar novas propriedades a serem verificadas com o JPF e tornar a geração de propriedades a serem verificadas com a JPF Exceptions (semi) automática.

Apêndice A

Especificação do *framework* em CSPm

Neste apêndice é mostrada a especificação em CSPm do *framework* e das propriedades de verificação desenvolvidos em [Per07, PdM10]. A especificação do *framework*, que descreve o protocolo de coordenação excepcional de componentes, foi tomada como base para a implementação do modelo de ações atômicas coordenadas desenvolvida neste trabalho (ver Seção 4.4).

A.1 Especificação do *Framework*

```
-- *****
-- * Processo Execution(parj,aci,k) *
-- *****

Execution(parj,aci,SINGLE) = RequestInit(parj,aci);
                           (Normal(parj,aci,SINGLE) /\ Raise(parj,aci));
                           RequestNormalEnd(parj,aci)

Execution(parj,aci,FIRST)  = RequestInit(parj,aci);
                           (Normal(parj,aci,FIRST) /\ Raise(parj,aci))

Execution(parj,aci,LAST)   = (Normal(parj,aci,LAST) /\ Raise(parj,aci));
                           RequestNormalEnd(parj,aci)

Execution(parj,aci,index)  = Normal(parj,aci,index) /\ Raise(parj,aci)

RequestInit(parj,aci) = let aipj = fCoordCh(parj,aci)
                        within
                          aipj!requestInit -> Init(aci)

Init(aci) = let aci_syn = fSynCh(aci)
            within
              aci_syn?init -> SKIP

RequestNormalEnd(parj,aci) = let aipj = fCoordCh(parj,aci)
                             within
                               aipj!requestOk -> (EndNormal(aci) [] EndExceptional(aci))

EndNormal(aci) = let aci_syn = fSynCh(aci)
                 within
                   aci_syn?ok -> SKIP

EndExceptional(aci) = let aci_syn = fSynCh(aci)
                      within
                        aci_syn?signal.exp -> SKIP

Raise(parj,aci) = let aipj = fCoordCh(parj,aci)
                  Exp = fRaise(parj,aci)
```

```

        within
        []exp:Exp @ aipj!raise.exp -> STOP

-- *****
-- * Processo Suspension(parj,aci) *
-- *****

Suspension(parj,aci) = let aipj = fCoordCh(parj,aci)
        within
        (aipj?suspend -> ExcepExecution(parj,aci) [] ExcepExecution(parj,
        aci)) /\ EndAbort(aci)

ExcepExecution(parj,aci) = let aci_syn = fSynCh(aci)
        within
        aci_syn?handle.exp -> Exceptional(parj,aci,exp) []
        RequestExceptionalEnd(parj,aci)

EndAbort(aci) = let aci_syn = fSynCh(aci)
        within
        aci_syn?abort -> SKIP

RequestExceptionalEnd(parj,aci) = let aipj = fCoordCh(parj,aci)
        within
        aipj!requestSignal -> EndExceptional(aci)

-- *****
-- * Processos Update e Query *
-- *****

Update(parj,extj,state) = let piej = fStateCh(parj,extj)
        within
        piej!update.state -> SKIP

Query(parj,extj,aci) = let piej = fStateCh(parj,extj)
        within
        piej?query.state -> ResultQuery(parj,extj,aci,state)

-- *****
-- * Processo ExternalObject(extj) *
-- *****

ExternalObject(extj) = initState(extj) /\ End

initState(extj) = let initial = fInitialState(extj) within ControlState(initial,initial,extj)

ControlState(saved,current,extj) = BeginTransExt(saved,current,extj) []
        ReadWrite(saved,current,extj) []
        CommitTransExt(saved,current,extj) []
        RollbackTransExt(saved,current,extj)

ReadWrite(saved,current,extj) = []piej:StateCh(extj) @
        (piej!query.current -> ControlState(saved,current,extj) []
        piej!update.new -> ControlState(saved,new,extj))

BeginTransExt(saved,current,extj) = []aiej:fTransCh(extj) @ aiej?begin ->
        ControlState(saved,current,extj)

CommitTransExt(saved,current,extj) = []aiej:fTransCh(extj) @ aiej?commit ->
        ControlState(current,current,extj)

RollbackTransExt(saved,current,extj) = []aiej:fTransCh(extj) @ aiej?rollback ->
        ControlState(saved,saved,extj)

```

```

-- *****
-- *   Processo Coordinator(aci)   *
-- *****

Coordinator(aci) = if aci == ac1 then
    RecInitRequest(CoordCh(aci),aci)
  else
    RecInitRequest(CoordCh(aci),aci) /\ RecAbortRequest(aci) /\ End

RecInitRequest(Ch,aci) = if empty(Ch) then
    BeginTrans(TransCh(aci),aci)
  else
    []aipj:Ch @ aipj:requestInit -> RecInitRequest(diff(Ch,{aipj}),aci)

BeginTrans(Ch,aci) = if empty(Ch) then
    BeginAction(aci)
  else
    []aiej:Ch @ aiej!begin -> BeginTrans(diff(Ch,{aiej}),aci)

BeginAction(aci) = let aci_syn = fSynCh(aci)
    within aci_syn!init -> Monitor(CoordCh(aci),aci)

Monitor(Run,aci) = if empty(Run) then
    CommitTrans(TransCh(aci),aci)
  else
    ([aipj:Run @ aipj:requestOk -> Monitor(diff(Run,{aipj}),aci)) []
    ([aipj:Run @ aipj:raise.exp ->
      Abort(AbortCh(aci), diff(CoordCh(aci),{aipj}),{exp},aci)) []
    ([acj_syn:SynCh(aci) @ acj_syn?signal.exp ->
      Abort(AbortCh(aci),CoordCh(aci),{exp},aci)) []
    ([acj_syn:SynCh(aci) @ acj_syn?init -> Monitor(Run,aci)) []
    ([acj_syn:SynCh(aci) @ acj_syn?ok -> Monitor(Run,aci)) []
    ([acj_syn:SynCh(aci) @ acj_syn?handle.exp -> Monitor(Run,aci))

Abort(AbortCh,CoordCh,Exp,aci) = if empty(AbortCh) then
    Suspend(CoordCh,Exp,aci)
  else
    ([acj_ab:AbortCh @ acj_ab!requestAbort ->
      ResponseAbort(AbortCh,CoordCh,
        Exp,acj_ab,aci)) []
    ([aipj:CoordCh @ aipj:raise.exp ->
      Abort(AbortCh,diff(CoordCh,{aipj}),
        union(Exp,{exp}),aci))

ResponseAbort(AbortCh,CoordCh,Exp,acj_ab,aci) = let acj = fActionName(acj_ab)
    acj_syn = fSynCh(acj)
    within
      acj_syn?abort -> Abort(diff(AbortCh,{acj_ab}),
        CoordCh,Exp,aci) []
      acj_ab?stopped -> Abort(diff(AbortCh,{acj_ab}),
        CoordCh,Exp,aci) []
      ([aipj:CoordCh @ aipj:raise.exp ->
        ResponseAbort(AbortCh,diff(CoordCh,{aipj}),
          union(Exp,{exp}),acj_ab,aci))

Suspend(Ch,Exp,aci) = if empty(Ch) then
    Resolve(Exp,aci)
  else
    ([aipj:Ch @ aipj!suspend -> Suspend(diff(Ch,{aipj}),Exp,aci)) []
    ([aipj:Ch @ aipj:raise.exp -> Suspend(diff(Ch,{aipj}),union(Exp,{exp}),aci))

Resolve(Exp,aci) = let exp = fExceptionTree(Exp,aci)
    within
      Choose(exp,aci)

```

```

Choose(exp, aci) = if member(exp, fSignal(aci)) then
    Signal(CoordCh(aci), exp, aci)
else
    Handle(exp, aci)

Signal(Ch, exp, aci) = if empty(Ch) then
    RollbackTransExcep(TransCh(aci), exp, aci)
else
    []aipj:Ch @ aipj!requestSignal -> Signal(diff(Ch, {aipj}), exp, aci)

Handle(exp, aci) = let aci_syn = fSynCh(aci)
    within
        aci_syn!handle.exp -> Verify(CoordCh(aci), {}, aci)

Verify(Ch, Resp, aci) = if empty(Ch) then
    Respond(Resp, aci)
else
    []aipj:Ch @ aipj?request.resp -> Verify(diff(Ch, {aipj}),
        union(Resp, {resp}), aci)

Respond(Resp, aci) = if member(requestSignal, Resp) then
    RollbackTransExcep(TransCh(aci), FAILURE, aci)
else
    CommitTrans(TransCh(aci), aci)

CommitTrans(Ch, aci) = if empty(Ch) then
    EndActionNormal(aci)
else
    []aiej:Ch @ aiej!commit -> CommitTrans(diff(Ch, {aiej}), aci)

EndActionNormal(aci) = let aci_syn = fSynCh(aci)
    within
        aci_syn!ok -> Finalize(aci)

RollbackTransExcep(Ch, exp, aci) = if empty(Ch) then
    EndActionExceptional(exp, aci)
else
    []aiej:Ch @ aiej!rollback ->
        RollbackTransExcep(diff(Ch, {aiej}), exp, aci)

EndActionExceptional(exp, aci) = let aci_syn = fSynCh(aci)
    within
        aci_syn!signal.exp -> Finalize(aci)

RollbackTransAbort(Ch, aci) = if empty(Ch) then
    EndActionAbort(aci)
else
    []aiej:Ch @ aiej!rollback -> RollbackTransAbort(diff(Ch, {aiej}), aci)

EndActionAbort(aci) = let aci_syn = fSynCh(aci)
    within
        aci_syn!abort -> Finalize(aci)

Finalize(aci) = if aci == acl then
    End
else
    STOP

End = end -> SKIP

RecAbortRequest(aci) = let aci_ab = fAbortCh(aci)
    aci_st = fStateCoordCh(aci)

```

```

        within
          aci_ab?requestAbort -> (aci_st?get.state ->
                                VerifyAbort(aci,state) /\ RecAbortRequest(aci))

VerifyAbort(aci,state) = if state == STOPPED then
  (let aci_ab = fAbortCh(aci)
   within
     aci_ab!stopped -> RecAbortRequest(aci))
else
  AbortNested(AbortCh(aci),aci)

AbortNested(AbortCh,aci) = if empty(AbortCh) then
  RollbackTransAbort(TransCh(aci),aci)
else
  []acj_ab:AbortCh @ acj_ab!requestAbort ->
    ResponseAbortNested(AbortCh,acj_ab,aci)

ResponseAbortNested(AbortCh,acj_ab,aci) = let acj = fActionName(acj_ab)
                                           acj_syn = fSynCh(acj)
                                           within
                                             acj_syn?abort ->
                                               AbortNested(diff(AbortCh,{acj_ab}),aci) []
                                             acj_ab?stopped ->
                                               AbortNested(diff(AbortCh,{acj_ab}),aci)

-- *****
-- * Processo CoordinatorState(aci,state) *
-- *****

CoordinatorState(aci,state) = let aci_syn = fSynCh(aci)
                              aci_st = fStateCoordCh(aci)
                              within
                                aci_syn?init      -> CoordinatorState(aci,STARTED) []
                                aci_syn?ok        -> CoordinatorState(aci,STOPPED) []
                                aci_syn?signal.exp -> CoordinatorState(aci,STOPPED) []
                                aci_syn?abort     -> CoordinatorState(aci,STOPPED) []
                                aci_syn?handle.exp -> CoordinatorState(aci,state) []
                                aci_st!get.state  -> CoordinatorState(aci,state) /\ End

```

A.2 Especificação das Propriedades

```

-- *****
-- * Propriedades para Verificacao *
-- *****

Prop_End(aci) = let aci_syn = fSynCh(aci)
                within
                  (aci_syn.ok      -> SKIP) []
                  (aci_syn.abort -> SKIP) []
                  ([exp:fSignal(aci) @ aci_syn.signal.exp -> SKIP)

Prop_Abort(aci) = if aci == ac1 then
  SKIP
else
  let aci_ab = fAbortCh(aci)
  within
    aci_ab.requestAbort -> SKIP

-- *****
-- * Propriedade 1 - Demarcacao da Acao *
-- *****

alpha_Prop1(aci,aci_syn) = {aci_syn.init,

```



```

                                aipj.requestOk ->
                                    Prop2_RequestOkSig(aci,
                                                diff(CoordCh,
                                                    {aipj})))

Prop2_RequestOkSig2(aci,CoordCh) = if empty(CoordCh) then
    SKIP
    else
        SKIP [] ([aipj:CoordCh @ (aipj.requestSignal ->
                                Prop2_RequestOkSig2(aci,
                                                diff(CoordCh,
                                                    {aipj})))
                                []
                                aipj.requestOk ->
                                    Prop2_RequestOkSig2(aci,
                                                diff(CoordCh,
                                                    {aipj}))))))

-- *****
-- * Propriedade 3 - Criação das Transações *
-- *****

alpha_Prop3_acl = {acl_syn.init,
                  aiej.begin | aiej <- TransCh(acl)}
alpha_Prop3(aci,aci_syn,aci_ab) = {aci_syn.init,
                                   aiej.begin,
                                   aci_ab.requestAbort | aiej <- TransCh(aci)}

Prop3(aci) = (Prop3_Def(aci,TransCh(aci)) /\ Prop_Abort(aci));Prop3(aci)
Prop3_Def(aci,TransCh) = if empty(TransCh) then
    let aci_syn = fSynCh(aci)
    within
        aci_syn.init -> SKIP
    else
        []aiej:TransCh @ aiej.begin -> Prop3_Def(aci,diff(TransCh,{aiej}))

-- *****
-- * Propriedade 4 - Confirmação das Transações *
-- *****

alpha_Prop4_acl = {acl_syn.ok,
                  aiej.commit | aiej <- TransCh(acl)}
alpha_Prop4(aci,aci_syn,aci_ab) = {aci_syn.ok,
                                   aiej.commit,
                                   aci_ab.requestAbort | aiej <- TransCh(aci)}

Prop4(aci) = (Prop4_Def(aci,TransCh(aci)) /\ Prop_Abort(aci));Prop4(aci)
Prop4_Def(aci,TransCh) = if empty(TransCh) then
    let aci_syn = fSynCh(aci)
    within
        aci_syn.ok -> SKIP
    else
        []aiej:TransCh @ aiej.commit -> Prop4_Def(aci,diff(TransCh,{aiej}))

-- *****
-- * Propriedade 5 - Desfazimento das Transações *
-- *****

alpha_Prop5_acl = {acl_syn.signal.exp,
                  acl_syn.abort,
                  aiej.rollback | exp <- fSignal(acl), aiej <- TransCh(acl)}
alpha_Prop5(aci,aci_syn,aci_ab) = {aci_syn.signal.exp,
                                   aci_syn.abort,
                                   aiej.rollback,
                                   aci_ab.requestAbort | exp <- fSignal(aci), aiej <- TransCh(aci)}

Prop5(aci) = (Prop5_Def(aci,TransCh(aci)) /\ Prop_Abort(aci));Prop5(aci)

```

```

Prop5_Def(aci,TransCh) = if empty(TransCh) then
    let aci_syn = fSynCh(aci)
    within
        ([]exp:fSignal(aci) @ aci_syn.signal.exp -> SKIP) []
        aci_syn.abort -> SKIP
    else
        []aiej:TransCh @ aiej.rollback -> Prop5_Def(aci,diff(TransCh,{aiej}))

-- *****
-- * Propriedade 6 - Tratamento de Exceções *
-- *****

alpha_Prop6_ac1 = {acj_syn.signal.exp,
    aipj.raise.exp,
    aipj.suspend | exp <- Exp, aipj <- CoordCh(ac1), acj_syn <- SynCh(ac1)}
alpha_Prop6(aci,aci_ab) = {acj_syn.signal.exp,
    aipj.raise.exp,
    aipj.suspend,
    aci_ab.requestAbort | exp <- Exp, aipj <- CoordCh(aci),
        acj_syn <- SynCh(aci)}

Prop6(aci) = (Prop6_Def(aci) /\ Prop_Abort(aci));Prop6(aci)
Prop6_Def(aci) = Prop6_Raise(aci) [] Prop6_Signal(aci)

Prop6_Raise(aci) = []aipj:CoordCh(aci) @ []exp:Exp @ aipj.raise.exp -> (Prop6_Suspend(aci,
                                                                    diff(CoordCh(aci),
                                                                    {aipj}))) []
                                                                    Prop6_Raise(aci))

Prop6_Signal(aci) = []acj_syn:SynCh(aci) @ []exp:Exp @ acj_syn.signal.exp -> Prop6_Suspend(aci,
                                                                    CoordCh(aci))

Prop6_Suspend(aci,CoordCh) = if empty(CoordCh) then
    SKIP
    else
        ([]aipj:CoordCh @ aipj.suspend -> Prop6_Suspend(aci,
                                                                    diff(CoordCh,
                                                                    {aipj}))) []
        ([]aipj:CoordCh @ []exp:Exp @ aipj.raise.exp ->
            Prop6_Suspend(aci,diff(CoordCh,{aipj})))

```

Apêndice B

Código do Exemplo de Instanciação do Modelo de Ações Atômicas Coordenadas

Neste capítulo vamos apresentar o código fonte do exemplo apresentado na Seção [4.4.9](#).

B.1 Pacote exception

Contém as exceções criadas para este exemplo.

B.1.1 Classe AccountAlreadyExistsException

```
1 package exception;  
2  
3 @SuppressWarnings("serial")  
4 public class AccountAlreadyExistsException extends Exception {  
5  
6 }
```

B.1.2 Classe InvalidAccountNumberException

```
1 package exception;  
2  
3 @SuppressWarnings("serial")  
4 public class InvalidAccountNumberException extends Exception {  
5  
6 }
```

B.2 Pacote simulation.database

Contém as classes relacionadas à representação de um banco de dados. Este banco de dados armazena dados de uma conta bancária e realiza duas operações: adição de conta e leitura de conta.

B.2.1 Classe BankAccount

```
1 package simulation.database;  
2  
3 class BankAccount {  
4  
5     int accountNumber;  
6     String name;  
7     double balance;  
8  
9     public BankAccount(int accountNumber, String name, double balance) {  
10         this.accountNumber = accountNumber;  
11     }  
12 }
```

```

11         this.name = name;
12         this.balance = balance;
13     }
14
15     public int getAccountNumber() {
16         return accountNumber;
17     }
18
19     public void setAccountNumber(int accountNumber) {
20         this.accountNumber = accountNumber;
21     }
22
23     public double getBalance() {
24         return balance;
25     }
26
27     public void setBalance(double balance) {
28         this.balance = balance;
29     }
30
31     public String getName() {
32         return name;
33     }
34
35     public void setName(String name) {
36         this.name = name;
37     }
38
39     @Override
40     public String toString() {
41         String str = new String("Account Number: " + this.getAccountNumber()
42             + "\nName: " + this.getName()
43             + "\nBalance: " + this.getBalance() + "\n");
44         return str;
45     }
46 }

```

B.2.2 Classe BankDatabase

```

1 package simulation.database;
2
3 import exception.AccountAlreadyExistsException;
4 import exception.InvalidAccountNumberException;
5 import java.util.Vector;
6
7 public class BankDatabase {
8
9     private static BankDatabase instance = null;
10    Vector<BankAccount> accountData;
11    Vector<BankAccount> tempAccountData;
12    int index;
13
14    public BankDatabase() {
15        accountData = new Vector<BankAccount>();
16        tempAccountData = new Vector<BankAccount>();
17        index = 0;
18    }
19
20    public static BankDatabase getInstance() {
21        if (instance == null) {
22            instance = new BankDatabase();
23        }
24        return instance;
25    }
26
27    public void addBankRecord(int accountNumber, String name, double balance) throws
28        InvalidAccountNumberException, AccountAlreadyExistsException {
29        if (accountNumber <= 0 || accountNumber > 100) {
30            InvalidAccountNumberException exc = new InvalidAccountNumberException();
31            throw exc;
32        } else {
33            if (checkNewAccount(accountNumber)) {

```

```

33         tempAccountData.add(new BankAccount(accountNumber, name, balance));
34     }
35 }
36 }
37
38 public void readBankRecord(int accountNumber) throws InvalidAccountNumberException
39 {
40     if (accountNumber <= 0 || accountNumber > 100) {
41         InvalidAccountNumberException exc = new InvalidAccountNumberException();
42         throw exc;
43     } else {
44         if (checkAccount(accountNumber)) {
45             System.out.println(tempAccountData.get(index).toString());
46         } else {
47             System.out.println("Account does not exist.");
48         }
49     }
50 }
51 private boolean checkNewAccount(int accountNumber) {
52     for (int i = 0; i < accountData.size(); i++) {
53         if (accountData.get(i).getAccountNumber() == accountNumber) {
54             return false;
55         }
56     }
57     for (int i = 0; i < tempAccountData.size(); i++) {
58         if (tempAccountData.get(i).getAccountNumber() == accountNumber) {
59             return false;
60         }
61     }
62     return true;
63 }
64
65 private boolean checkAccount(int accountNumber) {
66     for (int i = 0; i < accountData.size(); i++) {
67         if (accountData.get(i).getAccountNumber() == accountNumber) {
68             index = i;
69             return true;
70         }
71     }
72     return false;
73 }
74
75 public void commit() {
76     for (int i = 0; i < tempAccountData.size(); i++) {
77         accountData.add(tempAccountData.get(i));
78     }
79 }
80
81 public void rollback() {
82     tempAccountData.removeAllElements();
83 }
84 }

```

B.2.3 Classe Transaction

```

1 package simulation.database;
2
3 import simulation.framework.TransactionInterface;
4
5 public class Transaction implements TransactionInterface {
6
7     BankDatabase bankDB;
8     boolean active;
9
10    public void begin() {
11        bankDB = BankDatabase.getInstance();
12        active = true;
13    }
14
15    public void commit() {
16        bankDB.commit();

```

```

17         active = false;
18     }
19
20     public void rollback() {
21         bankDB.rollback();
22         active = false;
23     }
24
25     public boolean isActive() {
26         return active;
27     }
28
29     public BankDatabase getExternalObject() {
30         return bankDB;
31     }
32
33     public void setExternalObject(Object externalObj) {
34         this.bankDB = (BankDatabase) externalObj;
35     }
36 }

```

B.3 Pacote simulation.exceptionTree

Contém a classe com o método para resolução de exceções lançadas concorrentemente.

B.3.1 Classe ExceptionTree

```

1 package simulation.exceptionTree;
2
3 import java.util.Vector;
4
5 import exception.InvalidAccountNumberException;
6 import simulation.framework.ExceptionTreeInterface;
7
8 public class ExceptionTree implements ExceptionTreeInterface {
9
10     @Override
11     public void resolveException(Vector<Exception> exceptionTree) {
12         //In this simulation, the only exception to be resolved is an
13         //InvalidAccountNumberException type.
14         InvalidAccountNumberException e = new InvalidAccountNumberException();
15         exceptionTree.removeAllElements();
16         exceptionTree.add(e);
17     }
18 }

```

B.4 Pacote simulation.participants

Contém os participantes implementados para este exemplo.

B.4.1 Classe Participant0

```

1 package simulation.participants;
2
3 import java.util.Vector;
4
5 import simulation.framework.CoordinatedAtomicAction;
6 import simulation.framework.Participant;
7
8 public class Participant0 extends Participant {
9
10     public Participant0(Vector<CoordinatedAtomicAction> caa, int id) {
11         super(caa, id);
12     }
13 }

```

```

14     @Override
15     public void run() {
16         beginParticipantAction(0);
17     }
18 }

```

B.4.2 Classe Participant1

```

1 package simulation.participants;
2
3 import java.util.Vector;
4
5 import simulation.framework.CoordinatedAtomicAction;
6 import simulation.framework.Participant;
7
8 public class Participant1 extends Participant {
9
10     int number;
11     boolean addOk = false;
12
13     public Participant1(Vector<CoordinatedAtomicAction> caa, int id) {
14         super(caa, id);
15     }
16
17     @Override
18     public void run() {
19         beginParticipantAction(0);
20     }
21
22     public int getNumber() {
23         return number;
24     }
25
26     public void setNumber(int number) {
27         this.number = number;
28     }
29
30     public boolean isAddOk() {
31         return addOk;
32     }
33
34     public void setAddOk(boolean addOk) {
35         this.addOk = addOk;
36     }
37
38
39 }

```

B.5 Pacote simulation.handlers

Contém as threads responsáveis por fazer o tratamento excepcional dos participantes deste exemplo.

B.5.1 Classe Participant0Handler0

```

1 package simulation.participants.handlers;
2
3 import exception.InvalidAccountNumberException;
4 import simulation.framework.CoordinatedAtomicAction;
5 import simulation.framework.Handler;
6 import simulation.framework.Participant;
7
8 public class Participant0Handler0 extends Handler {
9
10     public Participant0Handler0(Participant participant,
11         CoordinatedAtomicAction caAction) {
12         super(participant, caAction);

```

```

13 }
14
15 public void run() {
16     try {
17         System.out.println("\nParticipant0-Handler0: starting.");
18         this.throwException();
19     } catch (InvalidAccountNumberException e) {
20         System.out.println("Exception handling successful");
21         this.setStatus("NORMAL");
22         this.setEndOfExecution(true);
23     } catch (Exception e) {
24         e.printStackTrace();
25     }
26 }
27 }

```

B.5.2 Classe Participant1Handler0

```

1 package simulation.participants.handlers;
2
3 import exception.InvalidAccountNumberException;
4 import simulation.framework.CoordinatedAtomicAction;
5 import simulation.framework.Handler;
6 import simulation.framework.Participant;
7
8 public class Participant1Handler0 extends Handler {
9
10     public Participant1Handler0(Participant participant,
11                                CoordinatedAtomicAction caAction) {
12         super(participant, caAction);
13     }
14
15     @Override
16     public void run() {
17         try {
18             System.out.println("\nParticipant1-Handler0: starting.");
19             this.throwException();
20         } catch (InvalidAccountNumberException e) {
21             System.out.println("Exception handling successful");
22             this.setStatus("NORMAL");
23             this.setEndOfExecution(true);
24         } catch (Exception e) {
25             e.printStackTrace();
26         }
27     }
28 }

```

B.6 Pacote simulation.roles

Contém as threads responsáveis pelas atividades do comportamento normal dos participantes deste exemplo.

B.6.1 Classe Participant0Role0

```

1 package simulation.participants.roles;
2
3 import simulation.database.BankDatabase;
4 import exception.InvalidAccountNumberException;
5 import simulation.framework.CoordinatedAtomicAction;
6 import simulation.framework.Participant;
7 import simulation.framework.Role;
8 import java.util.logging.Level;
9 import java.util.logging.Logger;
10 import simulation.participants.Participant1;
11
12 public class Participant0Role0 extends Role {
13     int number;

```



```

14  boolean numberOk = false;
15  boolean readOk = false;
16
17  public Participant0Role0(Participant participant ,
18      CoordinatedAtomicAction caAction, int number) {
19      super(participant , caAction);
20      this.number = number;
21  }
22
23  @Override
24  public void run() {
25      try {
26          System.out.println("Participant0-Role0: starting.");
27          this.transactions = getParticipant().getCaaVector().get(0).getTrans();
28          System.out.println("\nParticipant0-Role0: reading record 1");
29          ((BankDatabase) this.transactions.get(0).getExternalObject()).
30              readBankRecord(1);
31
32          Participant1 partic2 = (Participant1) this.getCaAction().getParticipants().
33              get(1);
34          Participant1Role0 role2 = (Participant1Role0) partic2.getRoles().get(0);
35
36          System.out.println("Participant0-Role0: setting number on Participant 2.");
37          partic2.setNumber(number);
38          this.setNumberOk(true);
39          System.out.println("Participant0-Role0: number setted.");
40
41          while (!role2.isEndOfExecution()) {
42              try {
43                  Thread.sleep(1000);
44              } catch (InterruptedException e) {
45                  e.printStackTrace();
46              }
47              if (isReadOk()) {
48                  System.out.println("\nParticipant1-Role0: reading record 1");
49                  ((BankDatabase) this.transactions.get(0).getExternalObject()).
50                      readBankRecord(1);
51                  this.setEndOfExecution(true);
52              } catch (InvalidAccountNumberException ex) {
53                  Logger.getLogger(Participant0Role0.class.getName()).log(Level.SEVERE, null,
54                      ex);
55              }
56          }
57
58          public boolean isNumberOk() {
59              return numberOk;
60          }
61
62          public void setNumberOk(boolean numberOk) {
63              this.numberOk = numberOk;
64          }
65
66          public boolean isReadOk() {
67              return readOk;
68          }
69
70          public void setReadOk(boolean readOk) {
71              this.readOk = readOk;
72      }

```

B.6.2 Classe Participant1Role0

```

1  package simulation.participants.roles;
2
3  import simulation.framework.CoordinatedAtomicAction;
4  import simulation.framework.Participant;
5  import simulation.framework.Role;
6  import java.util.logging.Level;

```

```

7 import java.util.logging.Logger;
8 import simulation.participants.Participant1;
9
10 public class Participant1Role0 extends Role {
11
12     public Participant1Role0(Participant participant,
13                             CoordinatedAtomicAction caAction) {
14         super(participant, caAction);
15     }
16
17     @Override
18     public void run() {
19
20         try {
21             System.out.println("\nParticipant1-Role0: starting.");
22             Participant partic1 = this.getCaAction().getParticipants().get(0);
23             Participant0Role0 role0 = (Participant0Role0) partic1.getRoles().get(0);
24             while (!role0.isNumberOk()) {
25                 Thread.sleep(1000);
26             }
27             System.out.println("\nParticipant1-Role0: number setted - ok.");
28
29             System.out.println("\nParticipant1-Role0: nested action started.");
30             //begin nested action
31             this.getParticipant().setNextNestedActionIndex(1);
32             this.getParticipant().setStatus("SUSPENDED");
33             this.setThreadSuspended(true);
34             while (isThreadSuspended()) {
35                 synchronized (this) {
36                     try {
37                         this.wait();
38                     } catch (InterruptedException ex) {
39                         Logger.getLogger(Participant1Role0.class.getName()).log(Level.
40                             SEVERE, null, ex);
41                     }
42                 }
43                 Thread.sleep(1000);
44                 System.out.println("\nParticipant1-Role0: nested action finished.");
45
46                 Participant1 p = (Participant1) this.getParticipant();
47                 if (p.isAddOk()) {
48                     role0.setReadOk(true);
49                 }
50                 this.setEndOfExecution(true);
51
52             } catch (InterruptedException ex) {
53                 Logger.getLogger(Participant1Role0.class.getName()).log(Level.SEVERE, null,
54                     ex);
55             }
56         }
57     }

```

B.6.3 Classe Participant1Role1

```

1 package simulation.participants.roles;
2
3 import simulation.database.BankDatabase;
4 import exception.AccountAlreadyExistsException;
5 import exception.InvalidAccountNumberException;
6 import simulation.framework.CoordinatedAtomicAction;
7 import simulation.framework.Participant;
8 import simulation.framework.Role;
9 import simulation.participants.Participant1;
10
11 public class Participant1Role1 extends Role {
12
13     public Participant1Role1(Participant participant,
14                             CoordinatedAtomicAction caAction) {
15         super(participant, caAction);
16     }

```

```

17
18  @Override
19  public void run() {
20      System.out.println("Participant1-Role1: starting");
21      this.transactions = getParticipant().getCaaVector().get(0).getTrans();
22      try {
23          System.out.println("Participant1-Role1: adding record - (" + accountNumber
24              () + ", Simone, 900.00)");
25          ((BankDatabase) this.transactions.get(0).getExternalObject()).addBankRecord
26              (accountNumber(), "Simone", 900.00);
27          System.out.println("Participant1-Role1: record added");
28          additionOk();
29          this.setStatus("NORMAL");
30          this.setEndOfExecution(true);
31      } catch (InvalidAccountNumberException e) {
32          this.setStatus("EXCEPTIONAL");
33          this.getCaAction().getCoordinator().getPropagatedExceptions().add(e);
34          this.setEndOfExecution(true);
35      } catch (AccountAlreadyExistsException e) {
36          e.printStackTrace();
37      }
38  }
39
40  public int accountNumber() {
41      Participant1 p = (Participant1) this.getParticipant();
42      return p.getNumber();
43  }
44
45  public void additionOk() {
46      Participant1 p = (Participant1) this.getParticipant();
47      p.setAddOk(true);
48  }

```


Apêndice C

Código Fonte de Propriedade de Tratamento de Exceção

Neste apêndice é mostrado o código fonte da propriedade “Bloco de Captura Vazio”, descrita em 6.3.2. Ela foi implementada no aplicativo JPF Exceptions [Xav08].

C.1 Propriedade: Bloco de Captura Vazio

```
1 package br.usp.jpff.property;
2
3 import gov.nasa.jpff.PropertyListenerAdapter;
4 import gov.nasa.jpff.jvm.BCELUtils;
5 import gov.nasa.jpff.jvm.ExceptionHandler;
6 import gov.nasa.jpff.jvm.JVM;
7 import gov.nasa.jpff.jvm.MethodInfo;
8 import gov.nasa.jpff.jvm.ThreadInfo;
9 import gov.nasa.jpff.jvm.bytecode.ASTORE;
10 import gov.nasa.jpff.jvm.bytecode.Instruction;
11 import gov.nasa.jpff.search.Search;
12
13 /**
14  * This class is used as a Property and Listener. It checks for a thrown
15  * exception if the exception handler is empty.
16  *
17  * @author Kleber
18  *
19  */
20 public class EmptyCatchProperty extends PropertyListenerAdapter {
21
22     public boolean emptyCatch;
23
24
25     @Override
26     public void exceptionThrown(JVM vm) {
27         ThreadInfo ti = vm.getCurrentThread();
28         String exceptionName = ti.getPendingException().getExceptionClassname();
29         MethodInfo mi = ti.getMethod();
30         ExceptionHandler[] exceptionHandlers = mi.getExceptions();
31         if(exceptionHandlers != null) {
32             int startPC = -1;
33             for(ExceptionHandler handler: exceptionHandlers) {
34                 String handlerType = handler.getName();
35                 //skips if this is a finally handler (exceptionType == null)
36                 if(handlerType != null && ReflectionUtils.isAssignable(
37                     exceptionName, handlerType)) {
38                     startPC = handler.getHandler();
39                     System.out.println("Handler found - " + startPC);
40                 }
41             }
42         }
43     }
44 }
```

```

39             break;
40         }
41     }
42     //Checks if a handler was found
43     if (startPC != -1) {
44         emptyCatch = !BCELUtils.findExceptionLocalVariable(mi, startPC);
45     }
46     if(emptyCatch) {
47         ti.breakTransition();
48     }
49 } //end if exceptionHandlers != null
50 }
51
52 @Override
53 public boolean check(Search search, JVM vm) {
54     return !emptyCatch;
55 }
56
57 }

```

C.2 Saída da Verificação Feita pela JPF Exceptions

Nesta seção, apresentamos a saída obtida com a execução, na JPF Exceptions, dos exemplos mostrados na Seção 5.4. As saídas aqui apresentadas mostram o rastreamento resultante do verificador de modo parcial, e são separadas de acordo com a propriedade.

C.2.1 Propriedade 1: Desfazimento de Transações

JavaPathfinder v4.1 – (C) 1999–2007 RIACS/NASA Ames Research Center

===== system under test

application: simulation.Simulator.class

===== search started: 03/08/10 13:24

```

simulation.framework.Coordinator.rollbackTransExcep(Coordinator.java:386)
simulation.framework.Coordinator.respond(Coordinator.java:358)
simulation.framework.Coordinator.verify(Coordinator.java:350)
simulation.framework.Coordinator.handle(Coordinator.java:331)
simulation.framework.Coordinator.choose(Coordinator.java:280)
simulation.framework.Coordinator.resolve(Coordinator.java:272)
simulation.framework.Coordinator.suspendParticipants(Coordinator.java:267)
simulation.framework.Coordinator.abort(Coordinator.java:238)
simulation.framework.Coordinator.monitor(Coordinator.java:201)
simulation.framework.Coordinator.beginAction(Coordinator.java:183)
simulation.framework.Coordinator.beginTrans(Coordinator.java:173)
simulation.framework.Coordinator.recInitRequest(Coordinator.java:163)
simulation.framework.Coordinator.run(Coordinator.java:83)

```

===== error #1

```

br.usp.jpf.property.caa.RollbackTransactionProperty
simulation.database.BankDatabase.rollback()V

```

===== trace #1

transition #0 thread: 0

```

gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  [282 insn w/o sources]
    simulation\Simulator.java:50 : roles0 = new Vector<Role>();
    [15 insn w/o sources]
    simulation\Simulator.java:50 : roles0 = new Vector<Role>();
    simulation\Simulator.java:51 : roles1 = new Vector<Role>();
    [15 insn w/o sources]
    simulation\Simulator.java:51 : roles1 = new Vector<Role>();
    simulation\Simulator.java:52 : handlers0 = new Vector<Handler>();
    [15 insn w/o sources]

```

```

simulation\Simulator.java:52 : handlers0 = new Vector<Handler>();
simulation\Simulator.java:53 : handlers1 = new Vector<Handler>();
[15 insn w/o sources]
simulation\Simulator.java:53 : handlers1 = new Vector<Handler>();
simulation\Simulator.java:54 : participants0 = new Vector<Participant>();
[15 insn w/o sources]
simulation\Simulator.java:54 : participants0 = new Vector<Participant>();
simulation\Simulator.java:55 : participants1 = new Vector<Participant>();
[15 insn w/o sources]

...

===== transition #1 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
simulation\Simulator.java:136 : partic.start();
simulation\Simulator.java:137 : }
simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #2 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
[3 insn w/o sources]
simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #3 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
[6 insn w/o sources]
simulation\Simulator.java:135 : for (Participant partic : participants0) {
simulation\Simulator.java:136 : partic.start();
===== transition #4 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0,Thread-3}
simulation\Simulator.java:136 : partic.start();
simulation\Simulator.java:137 : }
simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #5 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0,Thread-3}
[3 insn w/o sources]
simulation\Simulator.java:135 : for (Participant partic : participants0) {

....

===== transition #5772 thread: 8
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-7,Thread-8}
simulation\framework\CoordinatedAtomicAction.java:39 : return index;
simulation\framework\Coordinator.java:379 : System.out.println("Coordinator " + caa.
getIndex() + ": action finished");
simulation\framework\Coordinator.java:380 : }
simulation\framework\Coordinator.java:375 : }
simulation\framework\Coordinator.java:368 : }
simulation\framework\Coordinator.java:217 : }

...
simulation\framework\Coordinator.java:207 : return;
simulation\framework\Coordinator.java:184 : }
simulation\framework\Coordinator.java:174 : }
simulation\framework\Coordinator.java:164 : }
simulation\framework\Coordinator.java:89 : }
simulation\framework\Coordinator.java:90 : }

===== snapshot #1
no live threads

===== results
error #1: br.usp.jpf.property.caa.RollbackTransactionProperty "simulation.database.
BankDatabase.rollback()V"

===== search finished: 03/08/10 13:26

```

C.2.2 Propriedade 2: Tratamento das Exceções

JavaPathfinder v4.1 – (C) 1999–2007 RIACS/NASA Ames Research Center

===== system under test
application: simulation.Simulator.class

```

===== search started: 03/08/10 13:28

simulation.framework.Coordinator.rollbackTransExcep (Coordinator.java:386)
simulation.framework.Coordinator.anyMethod (Coordinator.java:321)
simulation.framework.Coordinator.choose (Coordinator.java:278)
simulation.framework.Coordinator.resolve (Coordinator.java:272)
simulation.framework.Coordinator.suspendParticipants (Coordinator.java:267)
simulation.framework.Coordinator.abort (Coordinator.java:238)
simulation.framework.Coordinator.monitor (Coordinator.java:206)
simulation.framework.Coordinator.beginAction (Coordinator.java:183)
simulation.framework.Coordinator.beginTrans (Coordinator.java:173)
simulation.framework.Coordinator.recInitRequest (Coordinator.java:163)
simulation.framework.Coordinator.run (Coordinator.java:83)

===== error #1
br.usp.jpf.property.caa.HandlingExceptionProperty

===== trace #1
===== transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  [282 insn w/o sources]
    simulation\Simulator.java:50 : roles0 = new Vector<Role>();
    [15 insn w/o sources]
    simulation\Simulator.java:50 : roles0 = new Vector<Role>();
    simulation\Simulator.java:51 : roles1 = new Vector<Role>();
    [15 insn w/o sources]
    simulation\Simulator.java:51 : roles1 = new Vector<Role>();
    simulation\Simulator.java:52 : handlers0 = new Vector<Handler>();
    [15 insn w/o sources]
    simulation\Simulator.java:52 : handlers0 = new Vector<Handler>();
    simulation\Simulator.java:53 : handlers1 = new Vector<Handler>();
    [15 insn w/o sources]
    simulation\Simulator.java:53 : handlers1 = new Vector<Handler>();
    simulation\Simulator.java:54 : participants0 = new Vector<Participant>();
    [15 insn w/o sources]
  ...

===== transition #1 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
  simulation\Simulator.java:136 : partic.start();
  simulation\Simulator.java:137 : }
  simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #2 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
  [3 insn w/o sources]
  simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #3 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
  [6 insn w/o sources]
  simulation\Simulator.java:135 : for (Participant partic : participants0) {
  simulation\Simulator.java:136 : partic.start();
===== transition #4 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0,Thread-3}
  simulation\Simulator.java:136 : partic.start();
  simulation\Simulator.java:137 : }
  simulation\Simulator.java:135 : for (Participant partic : participants0) {
===== transition #5 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0,Thread-3}
  [3 insn w/o sources]
  simulation\Simulator.java:135 : for (Participant partic : participants0) {
  ...

===== transition #5773 thread: 9
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>Thread-8}
  simulation\framework\Coordinator.java:199 : if (exceptionPropagation == true) {
  simulation\framework\Coordinator.java:204 : if (exceptionRaised == true) {
  simulation\framework\Coordinator.java:205 : System.out.println("EXCEPTION - RAISE");
  simulation\framework\Coordinator.java:206 : abort();
  simulation\framework\Coordinator.java:220 : for (CoordinatedAtomicAction action :
    nestedActions) {
  [8 insn w/o sources]

```



```

simulation\framework\Coordinator.java:220 : for (CoordinatedAtomicAction action :
    nestedActions) {
    [3 insns w/o sources]
simulation\framework\Coordinator.java:220 : for (CoordinatedAtomicAction action :
    nestedActions) {
simulation\framework\Coordinator.java:238 : suspendParticipants();
simulation\framework\Coordinator.java:242 : this.caa.setStatus("INTERRUPTED");
simulation\framework\CoordinatedAtomicAction.java:99 : this.status = status;
simulation\framework\CoordinatedAtomicAction.java:100 : this.notifyAll();
simulation\framework\CoordinatedAtomicAction.java:101 : }
simulation\framework\Coordinator.java:244 : pHandlers = new Vector<Participant>();
    [15 insns w/o sources]

```

...

```

simulation\framework\Coordinator.java:282 : }
simulation\framework\Coordinator.java:273 : }
simulation\framework\Coordinator.java:268 : }
simulation\framework\Coordinator.java:239 : }
simulation\framework\Coordinator.java:207 : return;
simulation\framework\Coordinator.java:184 : }
simulation\framework\Coordinator.java:174 : }
simulation\framework\Coordinator.java:164 : }
simulation\framework\Coordinator.java:89 : }
simulation\framework\Coordinator.java:90 : }

```

no live threads

error #1: br.usp.jpj.property.caa.HandlingExceptionProperty

search finished: 03/08/10 13:29

Apêndice D

Código Fonte para Integração via XML

D.1 Geração do XML pela OConGraX

D.1.1 XMLGenerator.java

```
1 package br.usp.ime.ocongrax.program.file;
2
3 /**
4  * @author Simone Hanazumi
5  *
6  */
7
8 import java.io.*;
9 import java.util.*;
10
11 import javax.xml.parsers.*;
12
13 import org.w3c.dom.*;
14 import org.xml.sax.SAXException;
15
16 import br.usp.ime.ocongrax.program.data.DefUseStructure;
17 import br.usp.ime.ocongrax.program.data.Pair;
18
19 import com.sun.org.apache.xml.internal.serialize.*;
20
21 public class XMLGenerator {
22
23     /**
24      * Generate the XML
25      *
26      * @param fileXML
27      *         file path and name
28      * @param hashmap
29      *         data structure which contains all the defs and uses of objects
30      * @param hashmap2
31      *         data structure which contains all the defs and uses of
32      *         exceptions
33      * @param defUsePairExc
34      *         if true, print to file the def-use pairs of exceptions
35      *
36      * @throws IOException
37      * @throws ParserConfigurationException
38      * @throws SAXException
39      */
40     public void generate(File fileXML,
41         HashMap<String, DefUseStructure> hashmap,
42         HashMap<String, DefUseStructure> hashmap2, boolean obj,
```

```

43     boolean exc, boolean defUsePairObj, boolean defUsePairExc)
44     throws IOException, ParserConfigurationException, SAXException {
45     DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
46     DocumentBuilder docBuilder = dbf.newDocumentBuilder();
47
48     Document doc = docBuilder.newDocument();
49     Element elements = doc.createElementNS(null, "elements");
50
51     if (obj) {
52         Vector<String> orderedKeys = new Vector<String>();
53         orderedKeys.addAll(hashmap.keySet());
54         Collections.sort(orderedKeys);
55         buildXML(hashmap, doc, elements, orderedKeys, true, defUsePairObj,
56                 defUsePairExc);
57     }
58
59     if (exc) {
60         Vector<String> orderedKeys2 = new Vector<String>();
61         orderedKeys2.addAll(hashmap2.keySet());
62         Collections.sort(orderedKeys2);
63         buildXML(hashmap2, doc, elements, orderedKeys2, false,
64                 defUsePairObj, defUsePairExc);
65     }
66
67     doc.appendChild(elements);
68
69     FileOutputStream out = new FileOutputStream(fileXML);
70     XMLSerializer serializer = new XMLSerializer(out, new OutputFormat(doc,
71         "iso-8859-1", true));
72     serializer.serialize(doc);
73     out.close();
74 }
75
76 private void buildXML(HashMap<String, DefUseStructure> hashmap,
77     Document doc, Element elements, Vector<String> keys,
78     boolean isObjectElement, boolean defUsePairObj,
79     boolean defUsePairExc) {
80     Element e, tag = null;
81     Node n = null;
82
83     for (int i = 0; i < keys.size(); i++) {
84         if (isObjectElement)
85             tag = doc.createElementNS(null, "object");
86         else
87             tag = doc.createElementNS(null, "exception");
88
89         DefUseStructure du;
90         Vector<Integer> vDefs, vUses = new Vector<Integer>();
91         Vector<Pair> vInnerPair = new Vector<Pair>();
92         Vector<Pair> vExternalPair = new Vector<Pair>();
93         du = hashmap.get(keys.elementAt(i));
94         vDefs = du.getDefinitions();
95         vUses = du.getUses();
96         vInnerPair = du.getInnerDefUses();
97         vExternalPair = du.getExternalDefUses();
98
99         e = doc.createElementNS(null, "classname");
100        n = doc.createTextNode(du.getClassName());
101        e.appendChild(n);
102        tag.appendChild(e);
103
104        e = doc.createElementNS(null, "methodname");
105        n = doc.createTextNode(du.getMethodName());
106        e.appendChild(n);
107        tag.appendChild(e);

```

```

108
109 e = doc.createElementNS(null, "name");
110 n = doc.createTextNode(keys.elementAt(i).replace(
111     du.getClassName() + " " + du.getMethodName() + " ", ""));
112 e.appendChild(n);
113 tag.appendChild(e);
114
115 if (defUsePairExc && !isObjectElement) {
116     printDUPairsXML(doc, tag, vInnerPair, vExternalPair);
117 }
118 if (defUsePairObj && isObjectElement) {
119     printDUPairsXML(doc, tag, vInnerPair, vExternalPair);
120 }
121
122 for (int j = 0; j < vDefs.size(); j++) {
123     int aux = vDefs.elementAt(j);
124     e = doc.createElementNS(null, "def");
125     n = doc.createTextNode("'" + aux);
126     e.appendChild(n);
127     tag.appendChild(e);
128 }
129 for (int k = 0; k < vUses.size(); k++) {
130     int aux = vUses.elementAt(k);
131
132     e = doc.createElementNS(null, "use");
133     n = doc.createTextNode("'" + aux);
134     e.appendChild(n);
135     tag.appendChild(e);
136 }
137 elements.appendChild(tag);
138 }
139 }
140
141 private void printDUPairsXML(Document doc, Element tag, Vector<Pair> vPair,
142     Vector<Pair> vExternalPair) {
143     Element iPairTag;
144     Element ePairTag;
145
146     if (vPair.size() != 0) {
147         iPairTag = doc.createElementNS(null, "innerpairs");
148         printInnerDUPairsXML(doc, tag, iPairTag, vPair);
149     }
150
151     if (vExternalPair.size() != 0) {
152         ePairTag = doc.createElementNS(null, "externalpairs");
153         printExternalDUPairsXML(doc, tag, ePairTag, vExternalPair);
154     }
155 }
156 }
157
158 private void printInnerDUPairsXML(Document doc, Element tag,
159     Element iPairTag, Vector<Pair> vPair) {
160     Element p;
161     Element e;
162     Node n;
163
164     for (int l = 0; l < vPair.size(); l++) {
165         Pair pair = vPair.elementAt(l);
166         p = doc.createElementNS(null, "pair");
167         e = doc.createElementNS(null, "def");
168         n = doc.createTextNode("'" + pair.getLineDef());
169         e.appendChild(n);
170         p.appendChild(e);
171         iPairTag.appendChild(p);
172         tag.appendChild(iPairTag);

```

```

173     e = doc.createElementNS(null, "use");
174     n = doc.createTextNode(" " + pair.getLineUse());
175     e.appendChild(n);
176     p.appendChild(e);
177     iPairTag.appendChild(p);
178     tag.appendChild(iPairTag);
179 }
180 }
181
182 private void printExternalDUPairsXML(Document doc, Element tag,
183     Element ePairTag, Vector<Pair> vExternalPair) {
184     Element p;
185     Element e;
186     Element t;
187     Node n;
188
189     for (int l = 0; l < vExternalPair.size(); l++) {
190         Pair pair = vExternalPair.elementAt(l);
191         p = doc.createElementNS(null, "pair");
192         e = doc.createElementNS(null, "def");
193         n = doc.createTextNode(" " + pair.getLineDef());
194         e.appendChild(n);
195         p.appendChild(e);
196         ePairTag.appendChild(p);
197         tag.appendChild(ePairTag);
198
199         t = doc.createElementNS(null, "externaluse");
200
201         e = doc.createElementNS(null, "classname");
202         n = doc.createTextNode(" " + pair.getExternalClassName());
203         e.appendChild(n);
204         t.appendChild(e);
205         p.appendChild(t);
206         ePairTag.appendChild(p);
207         tag.appendChild(ePairTag);
208
209         e = doc.createElementNS(null, "methodname");
210         n = doc.createTextNode(" " + pair.getExternalMethodName());
211         e.appendChild(n);
212         t.appendChild(e);
213         p.appendChild(t);
214         ePairTag.appendChild(p);
215         tag.appendChild(ePairTag);
216
217         e = doc.createElementNS(null, "name");
218         n = doc.createTextNode(" " + pair.getExternalVariableName());
219         e.appendChild(n);
220         t.appendChild(e);
221         p.appendChild(t);
222         ePairTag.appendChild(p);
223         tag.appendChild(ePairTag);
224
225         e = doc.createElementNS(null, "linenumber");
226         n = doc.createTextNode(" " + pair.getLineUse());
227         e.appendChild(n);
228         t.appendChild(e);
229         p.appendChild(t);
230         ePairTag.appendChild(p);
231         tag.appendChild(ePairTag);
232     }
233 }
234 }

```

D.2 Leitura do XML pelo Java PathFinder Exceptions

D.2.1 Método `readOcongraXml`, da classe `ExecutedPath.java`

```

1  /**
2   *
3   * This method reads an Xml file that contains the output of the Ocongra
4   * application and creates a list with the OcongraElements described in the
5   * file.
6   *
7   *
8   */
9  public List readOcongraXml(String filename) throws IOException,
10     ParserConfigurationException, SAXException {
11     String classname = "";
12     String methodname = "";
13     String name = "";
14     int defline = 0;
15     int useline = 0;
16     int index = 0;
17     int indexpair = 0;
18
19     boolean isInnerPair;
20
21     List<OcongraElement> ocongraElements = new ArrayList<OcongraElement>();
22
23     File aux = new File(filename);
24
25     if (aux.exists()) {
26         DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
27         DocumentBuilder docBuilder = dbf.newDocumentBuilder();
28         Document doc = docBuilder.parse(new File(filename));
29
30         Element elementsTag = doc.getDocumentElement();
31         Element objectTag = null;
32         Node classnameTag = null;
33         Node methodnameTag = null;
34         Node nameTag = null;
35         Element pairTag = null;
36         Node defTag = null;
37         Node useTag = null;
38
39         for (int i = 0; i < 2; i++) {
40             index = 0;
41             if (i == 0) {
42                 objectTag = (Element) elementsTag.getElementsByTagName(
43                     "object").item(index);
44             } else {
45                 objectTag = (Element) elementsTag.getElementsByTagName(
46                     "exception").item(index);
47             }
48             while (!(objectTag == null)) {
49                 classnameTag = (Node) objectTag.getElementsByTagName(
50                     "classname").item(0);
51                 classname = classnameTag.getTextContent();
52                 methodnameTag = (Node) objectTag.getElementsByTagName(
53                     "methodname").item(0);
54                 methodname = methodnameTag.getTextContent();
55                 nameTag = (Node) objectTag.getElementsByTagName("name")
56                     .item(0);
57                 name = nameTag.getTextContent();
58
59                 pairTag = (Element) objectTag.getElementsByTagName("pair")

```

```

60     .item(indexpair);
61     while (!(pairTag == null)) {
62         defTag = (Node) pairTag.getElementsByTagName("def")
63             .item(0);
64         defline = Integer.parseInt(defTag.getTextContent());
65         useTag = (Node) pairTag.getElementsByTagName("use")
66             .item(0);
67         if (!(useTag == null)) {
68             useline = Integer.parseInt(useTag.getTextContent());
69             isInnerPair = true;
70         } else {
71             useTag = (Node) pairTag.getElementsByTagName(
72                 "linenumber").item(0);
73             useline = Integer.parseInt(useTag.getTextContent());
74             isInnerPair = false;
75         }
76         if (i == 0) {
77             ocongraElements.add(ocongraElements.size(),
78                 new OcongraElement(classname, methodname,
79                     name, isInnerPair, true, defline,
80                     useline, false));
81         } else {
82             ocongraElements.add(ocongraElements.size(),
83                 new OcongraElement(classname, methodname,
84                     name, isInnerPair, false, defline,
85                     useline, false));
86         }
87         indexpair = indexpair + 1;
88         pairTag = (Element) objectTag.getElementsByTagName(
89             "pair").item(indexpair);
90     }
91
92     index = index + 1;
93     indexpair = 0;
94     if (i == 0) {
95         objectTag = (Element) elementsTag.getElementsByTagName(
96             "object").item(index);
97     } else {
98         objectTag = (Element) elementsTag.getElementsByTagName(
99             "exception").item(index);
100     }
101 }
102 }
103 }
104 return ocongraElements;
105 }

```


Referências Bibliográficas

- [ABM98] Paul E. Ammann, Paul E. Black, e William Majurski. Using model checking to generate tests from specifications. Em *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, páginas 46–54. IEEE Computer Society, 1998. [80](#)
- [ACP05] N. Guelfi A. Capozucca e P. Pelliccione. The fault-tolerant insulin pump therapy. Em *Proceedings of FM'2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems*, páginas 33–42, 2005. [39](#)
- [Avi82] A. Avizienis. The four-universe information system model for the study of fault-tolerance. Em *Digest 12th Int. Symposium on Fault-Tolerance Computing*, páginas 6–13, 1982. [xi](#), [5](#), [6](#)
- [BBC⁺02] Len Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, e Judith Stafford. Documenting software architecture: Documenting interfaces. Relatório técnico, Carnegie Mellon University, 2002. [37](#)
- [Ber03] Antonia Bertolino. Software testing research and practice. Em *10th International Workshop on Abstract State Machines*, páginas 1–21, 2003. [6](#)
- [Bin00] Robert Binder. *Testing Object-Oriented Systems: Models, patterns and tools*. Addison-Wesley, 2000. [5](#)
- [BK08] Christel Baier e Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. [xi](#), [28](#), [29](#), [31](#), [32](#)
- [Blo01] Joshua Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001. [89](#)
- [Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. [28](#), [79](#)
- [BRR⁺00] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, e R. J. Stroud. An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *SIGOPS Oper. Syst. Rev.*, 34(4):21–31, 2000. [39](#)
- [CDH⁺00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, e Hongjun Zheng. Bandera : Extracting finite-state models from java source code. Em *Proceedings of the 22nd International Conference on Software Engineering*, June 2000. [34](#)
- [CFRR09] Fernando Castor Filho, Alexander Romanovsky, e Cecília Mary F. Rubira. Improving reliability of cooperative concurrent systems with exception flow analysis. *J. Syst. Softw.*, 82(5):874–890, 2009. [63](#), [78](#)
- [CGP⁺06] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, e A. Zorzo. Caa-drip: a framework for implementing coordinated atomic actions. Em *ISSRE '06: Proceedings*

- of the 17th International Symposium on Software Reliability Engineering*, páginas 385–394, Washington, DC, USA, 2006. IEEE Computer Society. 60, 111
- [Cha91] Marcos Lordello Chaim. Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, Brasil, 1991. 23
- [CK99] Mei-Hwa Chen e Howard M. Kao. Testing object-oriented programs – an integrated approach. Em *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, página 73, Washington, DC, USA, 1999. IEEE Computer Society. 1, 3, 14, 20, 24, 80, 83, 85
- [Cod] CodeCover. Disponível em: <http://www.codecover.org/>. Acesso em: Jan. 2010. 23
- [Cov] Coverlipse. Disponível em: <http://coverlipse.sourceforge.net/>. Acesso em: Jan. 2010. 22
- [CR86] Roy H. Campbell e Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986. 41
- [Crn02] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002. 1
- [CSE96] John Callahan, Francis Schneider, e Steve Easterbrook. Automated software testing using model-checking, 1996. 80
- [DGP08] Giovanni Denaro, Alessandra Gorla, e Mauro Pezzè. Contextual integration testing of classes. Em *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, páginas 246–260, 2008. 23
- [DGP09] Giovanni Denaro, Alessandra Gorla, e Mauro Pezzè. Datec: Contextual data flow testing of java classes. Em *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, páginas 421–422, 2009. 23
- [DHJ⁺01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, e Hongjun Zheng. Tool-supported program abstraction for finite-state verification. Em *In Proceedings of the 23rd International Conference on Software Engineering*, páginas 177–187, 2001. 2, 80
- [Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, Abril 1970. 7, 27
- [dMF01] Gisele Rodrigues de Mesquita Ferreira. Tratamento de exceções no desenvolvimento de sistemas confiáveis baseados em componentes. Dissertação de Mestrado, Unicamp, Dezembro 2001. xi, 38
- [DMJ07] Márcio Eduardo Delamaro, José Carlos Maldonado, e Mario Jino. *Introdução ao Teste de Software*. Elsevier, 2007. 5, 8, 10, 11, 13, 23, 24
- [DW99] Desmond F. D’Souza e Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 38
- [Eck02] Bruce Eckel. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002. 17, 89

- [Ecla] EcEmma. Disponível em: <http://www.eclemma.org/>. Acesso em: Jan. 2010. 22
- [Eclb] Eclipse. Disponível em: <http://www.eclipse.org/>. Acesso em: Jan. 2010. 22
- [EM04] Dawson Engler e Madanlal Musuvathi. Static analysis versus software model checking for bug finding. páginas 405–427, 2004. 2, 80
- [EMM] EMMA. Disponível em: <http://emma.sourceforge.net/>. Acesso em: Jan. 2010. 22
- [Fea] FeaVer. Disponível em: <http://cm.bell-labs.com/cm/cs/what/feaver/index.html>. Acesso em: Jan. 2010. 34
- [Fra92] Nissim Francez. *Program Verification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992. 28, 31
- [FRR05] Fernando Castor Filho, Alexander Romanovsky, e Cecília Mary F. Rubira. Verification of coordinated exception handling. Em *Technical Report CS-TR-927*. School of Computing Science, University of Newcastle upon Tyne, 2005. 63, 78
- [FWA09] Gordon Fraser, Franz Wotawa, e Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, 2009. 80
- [Gak07] Luciana Setsuko Gakiya. Classificação e busca de componentes com tratamento de exceções. Projeto de Dissertação de Mestrado, 2007. 17
- [GAO95] David Garlan, Robert Allen, e John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. Em *ICSE '95: Proceedings of the 17th international conference on Software engineering*, páginas 179–185, New York, NY, USA, 1995. ACM. 1
- [GH99] Angelo Gargantini e Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, 1999. 80
- [GHJV00] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos*. Bookman, 2000. 89
- [GJSB05a] James Gosling, Bill Joy, Guy Steele, e Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edição, 2005. 3
- [GJSB05b] James Gosling, Bill Joy, Guy Steele, e Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005. 17
- [GR92] Jim Gray e Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 39
- [HD01] John Hatcliff e Matthew B. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. Em *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, páginas 39–58, London, UK, 2001. Springer-Verlag. 2, 33, 34, 80
- [HDD⁺03] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, e Venkatesh Prasad Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. *Software Engineering, International Conference on*, 0:160, 2003. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/ICSE.2003.1201197>. Acesso em 08 out. 2009. 2, 80

- [HM07] Simone Hanazumi e Ana Cristina Vieira de Melo. Ferramenta para análise do tratamento excepcional de objetos. Monografia, 2007. 3, 14, 24, 25, 35, 78, 79, 80, 81, 112
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 29, 30
- [Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. 27, 64
- [Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. 33, 34
- [Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997. 32, 33, 34
- [Inq96] Inquiry Board. Ariane 5 - flight 501 failure - full report, 1996. Disponível em: <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>. Acesso em: Jan. 2010. 6
- [Int06] International Software Testing Qualifications Board. *Standard Glossary of terms used in Software Testing*, 2006. 5
- [Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. 63
- [Jav] Java PathFinder Team. Java pathfinder. Disponível em: <http://javapathfinder.sourceforge.net/>. Acesso em: Jan. 2010. 3, 32, 34, 42, 78, 112
- [JGr] JGraph. Disponível em: <http://www.jgraph.com/>. Acesso em: Jan. 2010. 82
- [Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. 27, 29
- [JSS00] Daniel Jackson, Ian Schechter, e Hya Shlyachter. Alcoa: the alloy constraint analyzer. Em *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, páginas 730–733, New York, NY, USA, 2000. ACM. 63
- [JUn] JUnit. Disponível em: <http://www.junit.org/>. Acesso em: Jan. 2010. 22
- [Kim82] K.H. Kim. Approaches to mechanization of the conversation scheme based on monitors. *IEEE Transactions on Software Engineering*, 8:189–197, 1982. 39
- [KS06] Mike Keith e Merrick Schincariol. *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006. 42, 111
- [KSH⁺95] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, e C. Chen. Object state testing for object-oriented programs. Em *COMPSAC '95: Proceedings of the 19th International Computer Software and Applications Conference*, página 232, Washington, DC, USA, 1995. IEEE Computer Society. 1
- [Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. 33, 34, 61
- [LC02] Yi Liu e H. Conrad Cunningham. Software component specification using design by contract. Em *Proceeding of the SouthEast Software Engineering Conference, Tennessee Valley Chapter, National Defense Industry Association*, 2002. 37
- [Mac00] Patrícia D. L. Machado. Testing from structured algebraic specifications. Em *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, páginas 529–544, London, UK, 2000. Springer-Verlag. 80

- [Mal91] José Carlos Maldonado. *Crítérios Potencias Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese de Doutorado, DCA/FEEC/UNICAMP, Campinas, Brasil, 1991. 23
- [Mar94] John J. Marciniak, editor. *Encyclopedia of software engineering*. Wiley-Interscience, New York, NY, USA, 1994. 27
- [McC] Tim McCune. Exception-handling antipatterns. <http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html>, Jun. 2006. 90
- [McC76] Thomas McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, 1976. 13
- [MCJ89] José Carlos Maldonado, Marcos Lordello Chaim, e Mario Jino. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. Em *XXII Congresso Nacional de Informática*, São Paulo, Brasil, 1989. 23
- [Mey92] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, 1992. 37
- [Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 27
- [Mil99] Robin Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, New York, NY, USA, 1999. 27
- [MK94] John D. McGregor e Timothy D. Korson. Integrated object-oriented testing and development processes. *Commun. ACM*, 37(9):59–77, 1994. 1
- [MP92] Zohar Manna e Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. 33, 34, 61
- [MPI00] MPIAT. Mars program independent assessment team summary report, Mar 2000. 6
- [MSF06] Ana Cristina Vieira De Melo, Flavio Soares Correa Da Silva, e Marcelo Finger. *Lógica para Computação*. Thomson Learning, 2006. 28, 29, 30
- [Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. 9
- [NdM04] Paulo Roberto de Araújo França Nunes e Ana Cristina Vieira de Melo. Ocongra - uma ferramenta para geração de grafos de controle de fluxo de objetos. Em *Anais do 18º Simpósio Brasileiro de Engenharia de Software*, Brasília, Brasil, 2004. 18º SBES. 81
- [NHdM09] Paulo R. F. Nunes, Simone Hanazumi, e Ana C. V. de Melo. Ocongrax - automatically generating data-flow test cases for fault-tolerant systems. Em Manuel Núñez, Paul Baker, e Mercedes G. Merayo, editors, *TESTCOM/FATES’09: Joint Conference of the 21st IFIP Int. Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software*, Lecture Notes in Computer Science, páginas 229–234. Springer, 2009. 3, 14, 24, 25, 35, 78, 79, 80, 112
- [PdM10] David P. Pereira e Ana C.V. de Melo. Formalization of an architectural model for exception handling coordination based on ca action concepts. *Science of Computer Programming*, 75(5):333 – 349, 2010. Coordination Models, Languages and Applications (SAC’08). 42, 60, 64, 78, 111, 115
- [PDV01a] Corina S. Pasareanu, Matthew B. Dwyer, e Willem Visser. Finding feasible counterexamples when model checking abstracted java programs. Em *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 284–298, London, UK, 2001. Springer-Verlag. 2

- [PDV01b] Corina S. Pasareanu, Matthew B. Dwyer, e Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. Em *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 284–298, London, UK, 2001. Springer-Verlag. 80
- [Per07] David Paulo Pereira. Um *framework* para coordenação do tratamento de exceções em sistemas tolerantes a falhas. Dissertação de Mestrado, Universidade de São Paulo, São Paulo, Brasil, 2007. 5, 41, 42, 60, 64, 78, 111, 115
- [Pnu77] Amir Pnueli. The temporal logic of programs. Em *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, páginas 46–57, Washington, DC, USA, 1977. IEEE Computer Society. 33, 34, 61
- [Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001. xi, 1, 6, 7, 8, 10, 11, 13, 14, 28, 32, 37, 38
- [Pru04] Leandro Cesar Prudente. Um estudo sobre teste versus verificação formal de programas java. Dissertação de Mestrado, IME-USP, Março 2004. 15
- [Ran75] B. Randell. System structure for software fault tolerance. Em *Proceedings of the international conference on Reliable software*, páginas 437–449, New York, NY, USA, 1975. ACM. 39
- [RDH04] Edwin Rodríguez, Matthew B. Dwyer, e John Hatcliff. Checking strong specifications using an extensible software model checking framework. Em *In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 404–420. Springer, 2004. 2
- [Rec] Recoder. Disponível em: <http://recoder.sourceforge.net/>. Acesso em: Jan. 2010. 82
- [RHB97] A. W. Roscoe, C. A. R. Hoare, e Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. 64
- [Rom01] Alexander Romanovsky. Coordinated atomic actions: how to remain acid in the modern world. *SIGSOFT Softw. Eng. Notes*, 26(2):66–68, 2001. 39
- [RRDH03] Robby, Edwin Rodríguez, Matthew B. Dwyer, e John Hatcliff. Checking strong specifications using an extensible software model checking framework. Em *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 404–420. Springer, 2003. 80
- [RRS⁺98] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, A. F. Zorzo, D. Schwier, e F. von Henke. Coordinated atomic actions: Formal model, case study and system implementation. Relatório técnico, 1998. 61, 78
- [RS03] Darrel Reimer e Harini Srinivasan. Analyzing exception usage in large java applications. Em *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-oriented Systems*, 2003. 90
- [RW82] Sandra Rapps e Elaine J. Weyuker. Data flow analysis techniques for test data selection. Em *ICSE '82: Proceedings of the 6th international conference on Software engineering*, páginas 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. 23
- [RW85] Sandra Rapps e Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985. 23

- [RXR96] A. Romanovsky, Jie Xu, e B. Randell. Exception handling and resolution in distributed object-oriented systems. Em *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, página 545, Washington, DC, USA, 1996. IEEE Computer Society. 39
- [SH98] Saurabh Sinha e Mary Jean Harrold. Analysis of programs with exception-handling constructs. Em *ICSM '98: Proceedings of the International Conference on Software Maintenance*, página 348, Washington, DC, USA, 1998. IEEE Computer Society. 3, 17, 18, 20, 25, 80, 84, 87
- [SH99] Saurabh Sinha e Mary Jean Harrold. Criteria for testing exception-handling constructs in java programs. Em *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, página 265, Washington, DC, USA, 1999. IEEE Computer Society. 3, 17, 19, 20, 24, 25, 80, 84, 85, 112
- [Som06] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8th edição, 2006. xi, 6, 7, 28, 38, 79
- [SP03] Amie L. Souter e Lori L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Softw. Eng.*, 29(11):1005–1018, 2003. 24
- [SPI] SPIN. Disponível em: <http://spinroot.com>. Acesso em: Jan. 2010. 32, 33, 34
- [Suna] Sun Microsystems. The java tutorials. Disponível em: <http://java.sun.com/docs/books/tutorial/>. Acesso em: Jan. 2010. 17
- [Sunb] Sun Microsystems. Jdc tech tips: June 27, 2000. Disponível em: <http://java.sun.com/developer/TechTips/2000/tt0627.html>. Acesso em: Jan. 2010. 84
- [Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 37
- [Tra00] Hung Tran. Test generation using model checking. Em *Automated Verification, Fall 2000: A Collection of Reports*. University of Toronto, 2000. 80
- [VHB⁺03] W. Visser, K. Havelund, G. Brat, S. Park, e F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003. 2, 3, 32, 34, 78, 112
- [VMWD05] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, e M. E. Delamaro. Coverage testing of java programs and components. *Sci. Comput. Program.*, 56(1-2):211–230, 2005. 24
- [Voa98] Jeffrey M. Voas. The challenges of using cots software in component-based development. *Computer*, 31(6):44–45, 1998. 1
- [W3] W3 Schools. Xml tutorial. Disponível em: <http://www.w3schools.com/xml/default.asp>. Acesso em: Jan. 2010. 84
- [WD96] Jim Woodcock e Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 27, 29
- [Wey88] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988. 1
- [WFW09] Martin Weiglhofer, Gordon Fraser, e Franz Wotawa. Using coverage to automate and improve test purpose based testing. *Inf. Softw. Technol.*, 51(11):1601–1617, 2009. 5
- [Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993. 30

- [Wor96] J. B. Wordsworth. *Software engineering with B*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. 27, 29
- [Xav08] Kleber da Silva Xavier. Estudo sobre redução do custo de testes através da utilização de verificação de componentes java com tratamento de exceções. Dissertação de Mestrado, Universidade de São Paulo, São Paulo, Brasil, 2008. 2, 3, 5, 17, 18, 21, 32, 34, 35, 60, 65, 78, 79, 80, 82, 87, 92, 111, 133
- [XHdM08] Kleber S. Xavier, Simone Hanazumi, e Ana C. V. de Melo. Using formal verification to reduce test space of fault-tolerant programs. Em *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, páginas 181–190, Washington, DC, USA, 2008. IEEE Computer Society. xi, 2, 79, 80, 112
- [XRR⁺95a] Jie Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, e Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 0:0499, 1995. 38, 39, 41
- [XRR⁺95b] Jie Xu, Brian Randell, Alexander Romanovsky, C. M.F. Rubira-Calsavara, R. J. Stroud, e Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. Relatório técnico, 1995. 39