Instituto de Computação
Universidade Estadual de Campinas

Validação do Fluxo Excepcional a partir do Diagrama de Atividades da UML 2.0

Jeferson Ferreira

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Jeferson Ferreira e aprovada pela Banca Examinadora.

Campinas, 22 de julho de 2011.

Profa. Dra. Eliane Martins (Orientadora)

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

ERRATA:

Desconsiderar informação no campo "Área de Concentração".

Prof. Dr. Paulo Licio de Geus
Coord. de Pós-Graduação
Instituto de Computação - Unicamp
Matrícula 10.326-8

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA – UNICAMP

Ferreira, Jeferson, 1973-
F413v Validação do fluxo excepcional a partir do diagrama de atividades da UML 2.0 / Jeferson Ferreira. – Campinas, SP: [s.n.], 2011.

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

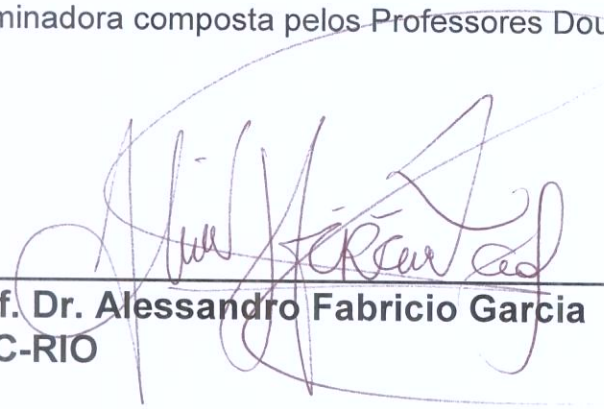
1. Tolerância à falha (Computação). 2. Tratamento de exceções. 3. Engenharia de software. 4. Software - Testes. I. Martins, Eliane, 1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

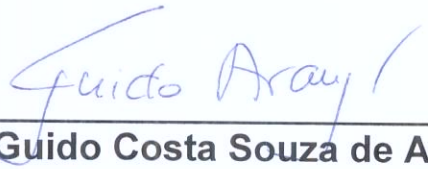
Título em Inglês: Validation of exceptional flow in UML 2.0 acitivity diagram
Palavras-chave em Inglês:
Fault-tolerant computing
Exception handling
Software engineering
Software testing
Área de concentração: Engenharia de software
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Alessandro Fabricio Garcia
Guido Costa Souza de Araújo
Data da defesa: 17-06-2011
Programa de Pós Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

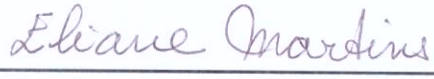
Dissertação Defendida e Aprovada em 17 de junho de 2011, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Alessandro Fabricio Garcia
PUC-RIO



Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP



Profª. Drª. Eliane Martins
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Validação do Fluxo Excepcional a partir do Diagrama de Atividades da UML 2.0

Jeferson Ferreira

Junho de 2011

Banca Examinadora:

- Profa. Dra. Eliane Martins (Orientadora)
- Prof. Dr. Alessandro Fabricio Garcia
  Departamento de Informática - PUC-Rio
- Prof. Dr. Guido Araújo
  IC-UNICAMP
- Prof. Dra. Ana Estela Antunes da Silva
  FT-UNICAMP (Suplente)
- Prof. Dr. Arnaldo Vieira Moura
  IC-UNICAMP (Suplente)

Resumo

Para a construção de sistemas robustos, devem ser utilizadas técnicas de tolerância a falhas que podem ser implementadas através de mecanismos de tratamento de exceções. Esses mecanismos possibilitam o tratamento de possíveis exceções, ou até mesmo a continuação da execução das funcionalidades do sistema mesmo na presença de uma exceção. O uso dos mecanismos de tratamento de exceções para desenvolver sistemas de software em larga escala, juntamente com o fato de ser implementado por diversas linguagens modernas, confirma a importância desta prática de desenvolvimento. Por outro lado, o uso desses mecanismos tem suas desvantagens, impactando principalmente na complexidade dos sistemas. Um problema que ocorre com muita frequência é efetuar a validação do fluxo excepcional somente na fase de implementação. A detecção de um problema de especificação nesta etapa do processo, pode acarretar em um aumento nos custos e prazos para a entrega do software.

Este trabalho apresenta uma abordagem que utiliza as técnicas de análise estática, normalmente empregadas para detectar falhas no código fonte, para antecipar a validação do fluxo excepcional de um componente de software durante o ciclo de desenvolvimento. A solução proposta utiliza as informações do fluxo de controle e fluxo de dados obtidas a partir de um modelo comportamental. O modelo utilizado nesta abordagem é o diagrama de atividades da UML, que passa por uma série de transformações até gerar um grafo de fluxo de controle interprocedimental. Durante este processo são executadas análises de fluxo de dados para inferir com precisão quais são os tipos de exceções podem ser lançadas em dado ponto do modelo. Também faz parte deste trabalho a apresentação de uma ferramenta de apoio para o processo de validação do fluxo excepcional. Esta ferramenta, denominada *ADEX* (Activity Diagram EXceptional flow analyzer), implementa os algoritmos utilizados para a conversão do diagrama de atividades no grafo de fluxo de controle interprocedimental. A ferramenta também oferece recursos para a visualização do fluxo de controle normal e excepcional do modelo.

Abstract

In order to develop robust software, should be used fault tolerant techniques that can be implemented by exception handling mechanisms. These mechanisms allow the handling of possible exceptions or even the continued of execution of the system's functionalities, even in the presence of an exception. The use of exception handling mechanisms to develop large scale software systems together with the fact that several modern programming languages provide these mechanisms, confirm the importance of these mechanisms in practice. On the other hand, the use of these mechanisms has some disadvantages, principally impacting on the complexity of the systems. One problem that occurs very often is performing the validation of the exceptional flow only during the implementation phase. The detection of a specification problem at this stage of the process can lead the increasing of costs and delays to delivery the software.

This paper presents an approach that uses static analysis techniques, usually used to detect anomalies in the source code, to antecipate the validation of the exceptional flow of a software component in the development cycle. The proposed solution uses the information of control flow and data flow gathered from a behavioral model. The model used in this approach is the UML activity diagram, which undergoes a series of transformations to generate a interprocedural control flow graph. During this process are performed data flow analysis to inferring precisely what kind of exceptions can be thrown at a specific point of the model. The presentation of a tool to support the validation of the exceptional flow, also is part of this work. This tool, called *ADEX* (Activity Diagram EXceptional flow analyzer), implements the algorithms used to convert the activity diagram in the interprocedural control flow graph. The tool also provides features for visualization of normal and exceptional control flow of the model.

Agradecimentos

A minha família por todas as manifestações de apoio e carinho; especialmente a minha esposa e incondicional companheira Divoná; aos meus irmãos Leonilda, Leosilda, Gerson, Léa e Elisabete; a minha mãe Tereza e ao meu saudoso pai Zacheus (in memorian).

A minha professora e orientadora Eliane pela dedicação, compreensão e pela paciência na tarefa de nortear o meu caminho.

Aos demais professores da pós-graduação, Guido, Célia, Rodolfo, Beatriz, Ariadne e Cecília, pelas contribuições diretas e indiretas para o êxito deste trabalho.

A todos os meus colegas de curso, que me apoiaram e incentivaram ao longo do percurso, principalmente ao Maxwell e ao Rafael Gava.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O progresso do desenvolvimento de software nos últimos anos é facilmente percebido quando levamos em conta a complexidade dos sistemas desenvolvidos atualmente. Cada vez mais essa crescente complexidade é agravada pelos requisitos impostos pelas aplicações modernas, como portabilidade, interoperabilidade, confiabilidade, disponibilidade e segurança. Apesar das exigências em se produzir sistemas robustos e confiáveis, esse aumento da complexidade, aliado a restrições de tempo para o desenvolvimento, acabam comprometendo a qualidade final do software, uma vez que induzem a inserção de um número maior de falhas durante o processo de desenvolvimento [Lee 90b].

Além disso, a área de desenvolvimento de software também enfrenta outros desafios, por exemplo, a constante necessidade de adaptação às novas tecnologias e às mudanças em seus requisitos. A fim de reduzir essa complexidade e consequentemente reduzir o tempo e o custo do desenvolvimento, cada vez mais a indústria do software vem aperfeiçoando os seus processos de desenvolvimento. Uma das abordagens recentes utilizada para mitigar os riscos deste processo cada vez mais dinâmico é o paradigma de Desenvolvimento Dirigido por Modelos (Model-Driven Development - MDD) [Stah 06].

Com o surgimento da MDD os modelos assumiram o papel de protagonistas no processo de desenvolvimento de software, não são mais coadjuvantes utilizados apenas para documentar e facilitar a comunicação entre os projetistas e as equipes de desenvolvimento. Neste paradigma os modelos são os principais artefatos do processo de desenvolvimento, sendo refinados e transformados em outros modelos, cada vez com um menor nível de abstração até chegar na geração automática do código fonte. Os modelos mais abstratos são independentes de plataforma, possibilitando o seu reuso para atender os requisitos de portabilidade, interoperabilidade e de evolução tecnológica.

Outra vantagem da MDD é a possibilidade da validação do modelo antes da geração do código fonte. Esta prática permite a detecção precoce de erros e imprecisões, antes de prosseguir para as fases posteriores do ciclo de vida do software. Dessa forma evitam-

se as custosas e desnecessárias idas e vindas, que frequentemente resultam no descarte de alguns modelos, tornando-os obsoletos e inúteis. Por esse motivo, a adoção deste paradigma tem se expandido nas áreas de aplicação onde o custo da ocorrência de uma falha pode ser inaceitável. Esses sistemas são considerados críticos e consequentemente torna-se necessário desenvolver técnicas para torná-los confiáveis.

Para a construção de sistemas confiáveis, devem ser utilizadas técnicas de tolerância a falhas [Lee 90b]. Essas técnicas podem ser implementadas através de mecanismos de tratamento de exceções [Mesq 01a, Cris 89a], os quais oferecem um arcabouço para a criação de tratadores adequados para cada tipo de exceção, possibilitando o tratamento de possíveis exceções, ou até mesmo a continuação da execução das funcionalidades do sistema.

O uso dos mecanismos de tratamento de exceções para desenvolver sistemas de software em larga escala [Reim 03], juntamente com o fato de ser implementado por diversas linguagens modernas, confirma a importância desta prática de desenvolvimento. Além disso, em aplicações onde a reversão de dados para um estado anterior não é possível, os mecanismos de tratamento de exceções podem ser a única opção disponível. Por outro lado, os mecanismos de tratamento de exceções podem tem suas desvantagens relacionadas ao seu impacto na complexidade e tamanho dos sistemas. Para ilustrar este problema, no contexto de sistemas críticos, a maior parte do código do sistema é dedicada a detecção e tratamento de erros [Cris 89a, Reim 03].

Um problema que ocorre com muita frequência é efetuar a validação do fluxo excepcional somente na fase de implementação. A detecção de um problema de especificação nesta etapa do processo pode acarretar em um aumento nos custos e prazos para a entrega do software. A adoção da MDD pode mitigar estes riscos, pois esta abordagem possibilita a validação do fluxo excepcional a partir dos modelos comportamentais durante as fases iniciais do processo de desenvolvimento.

1.1 Motivação

A UML (Unified Modeling Language) [OMG 07a] é uma notação amplamente utilizada para fins de documentação e especificação durante o processo de desenvolvimento de um software. Rapidamente tornou-se um padrão no desenvolvimento orientado a objetos. A UML é dividida em duas especificações, estrutural e comportamental. A primeira modela os aspectos estáticos de um sistema, já a segunda os aspectos dinâmicos. Neste segundo grupo se enquadram os modelos comportamentais, que especificam como os aspectos estruturais de um sistema mudam através do tempo. O Diagrama de Atividades (DA) é um exemplo desse tipo de modelo. Este diagrama é bastante versátil podendo ser usado em um grande número de situações, desde a modelagem de um processo de negócio até o

comportamento detalhado de uma operação na fase de projeto. Até a versão 1.5 da UML, o DA era baseado em máquinas de estados, a partir da UML 2.0 esse diagrama tornou-se independente, possibilitando modelar, além do fluxo de controle, o fluxo de dados. Este modelo, através dos seus mecanismos de tratamento de exceções, permite a especificação do comportamento excepcional desde cedo no ciclo de desenvolvimento.

Uma das realizações de MDD que têm obtido resultados concretos é a arquitetura MDA (Model-Driven Architecture) [Klep 03]. O foco principal deste conceito é preencher a lacuna existente entre as fases de análise e projeto e de implementação. De acordo com a MDA isto deve ser feito através do aprimoramento dos artefatos de projeto (modelos UML) com toda a informação necessária para permitir que o desenvolvedor construa automaticamente uma aplicação completa a partir de tais modelos. Este paradigma trabalha basicamente com a transformação de modelos, utilizando modelos em diferentes níveis de abstração, como o modelo independente de plataforma (Platform Independent Model - PIM), modelo com alto nível de abstração, é independente de qualquer tecnologia de implementação, onde todas as funcionalidades do sistema e regras de negócio são modeladas; e o modelo específico de plataforma (Platform Specific Model - PSM), modelo associado a um sistema específico em termos de uma tecnologia de implementação, inclui informações sobre tipo de dados, banco de dados, interfaces, etc. Um modelo PIM pode ser transformado em diversos modelos PSM, e cada modelo PSM pode ser transformado em código-fonte de uma linguagem de programação específica.

Um exemplo de abordagem MDD que utiliza o Diagrama de Atividades da UML 2.0 foi proposto por Sarstedt et al. [Sars 05]. Nesta abordagem o DA é utilizado para modelar o fluxo de controle durante a fase de análise e projeto e reutilizado na fase de implementação para gerar o código-fonte.

Para concretizar a ideia da MDD fica óbvio que somente a informação gráfica não é suficiente para descrever adequadamente as funcionalidades de um sistema. Portanto, outros recursos devem ser utilizados para alcançar este objetivo; um desses recursos é a semântica das ações UML. A semântica das ações define um conjunto de ações que oferece a maioria dos recursos de uma linguagem de programação orientada a objetos. Assim, é possível identificar, por exemplo, quais ações definem ou usam variáveis especificadas no modelo e que tipo de ação é capaz de efetuar a chamada de um método. Além disso, a semântica das ações da UML oferece o formalismo suficiente para a execução do modelo. A Pópulo [Fuen 08] é um exemplo de ferramenta que executa um diagrama de atividades com base nesta semântica, permitindo dessa forma uma validação prévia do modelo durante a fase de análise e projeto. Outra abordagem proposta por Knieke et al. [Knie 08] estende a semântica do DA e apresenta uma metodologia baseada em modelos para especificação de requisitos chamada LADs (Live Activity Diagrams), permitindo a simulação antecipada e a validação de modelos de requisitos.

Apesar da sua importância na construção de sistemas confiáveis, os mecanismos de tratamento de exceções não são muito bem especificados, geralmente esta preocupação só ocorre durante a sua implementação. Uma vantagem de especificar os tratadores de exceções nas fases iniciais do desenvolvimento é a possibilidade de validar o seu comportamento antes da sua implementação.

Para lidar com a complexidade inerente aos tratadores de exceções, muitos trabalhos propõem que o comportamento excepcional deve ser incorporado o mais cedo possível no processo de desenvolvimento de software, especialmente durante as fases de especificação dos requisitos e análise e projeto [Rubi 05, Brit 05]. A arquitetura do software representa explicitamente a estrutura de um sistema, e este é um dos primeiros artefatos que permitem a análise dos atributos de qualidade do sistema, bem como, confiabilidade, disponibilidade e segurança [Bass 97].

Devido a importância da arquitetura para o projeto do comportamento excepcional de sistemas, torna-se necessário efetuar um refinamento com informações mais detalhadas, com o objetivo de permitir a geração automática do código-fonte. A utilização do diagrama de atividades da UML 2.0 para a especificação detalhada do comportamento normal e excepcional, é uma alternativa para suprir esta necessidade. A utilização deste recurso pode reduzir o esforço e o custo de sistema de software tolerante a falhas, bem como melhorar de maneira geral a confiabilidade do projeto do comportamento excepcional.

1.2 Objetivos e Solução Proposta

Para validar os mecanismos de tratamento de exceções modelados no diagrama de atividades, deve-se levar em consideração as informações de fluxo de dados do modelo. Estas informações são obtidas através das definições e usos das referências dos objetos de exceção. Neste trabalho, propomos a utilização de uma análise de fluxo de dados como forma de validar os tratadores de exceção modelados em um diagrama de atividades da UML 2.0. A semântica das ações UML é usada para determinar as definições e os usos de variáveis, bem como identificar os pontos onde as exceções são lançadas e capturadas. Uma análise de fluxo de dados é utilizada para determinar quais os tipos de exceções que podem ser lançadas em cada ponto do modelo.

O propósito deste trabalho é apresentar uma técnica para validação de um modelo comportamental, permitindo que o modelo seja corrigido antes da implementação. Para validar o fluxo excepcional é utilizada uma análise de fluxo de dados para o cômputo das definições incidentes [Aho 86]. Esta análise é utilizada para inferir quais são os tipos de exceções que alcançam o nó responsável pelo seu lançamento. O resultada da análise de fluxo de dados determina quais são os tipos de exceções que podem ser lançadas por este

nó.

O objetivo da abordagem apresentada neste trabalho é fazer com que as técnicas de análise estática, normalmente empregadas para detectar falhas no código fonte, possam ser empregadas mais cedo no ciclo de desenvolvimento do software. A solução proposta apresenta uma combinação destas técnicas para validar o fluxo excepcional de um componente de software. São utilizadas as informações do fluxo de controle e fluxo de dados obtidas a partir do modelo. A abordagem proposta transforma um diagrama atividades em um grafo de fluxo de controle interprocedimental. Este modelo é amplamente utilizado em várias aplicações, como testes baseados em fluxo de dados, program slicing e otimizações de compiladores [Rapp 85, Weis 81, Aho 86].

A partir do grafo resultante é possível executar uma técnica de análise de fluxo de dados para inferir quais são os tipos de exceções que podem ser lançadas em dado ponto do modelo. O fluxo de exceções é demonstrado por arestas que ligam os nós que lançam as exceções com os nós que capturam e fazem o seu tratamento. No caso das exceções não capturadas, são gerados nós adicionais para representar as saídas excepcionais do procedimento. Deste modo, é possível validar se o método lança esse tipo de exceção para ser tratada por outro procedimento ou se houve uma falha na modelagem.

A abordagem proposta é capaz de validar o fluxo de exceções completo do modelo, identificando quais tratadores de exceções são utilizados e quais exceções não são capturadas. É possível também identificar os tratadores de exceção genéricos que são capazes de capturar qualquer tipo de exceção e acabam capturando aquelas exceções que necessitam um tratamento especial que não foi modelado apropriadamente.

Além disto, faz parte deste trabalho a apresentação de uma ferramenta de apoio para a validação do fluxo excepcional. Esta ferramenta, denominada ADEX, foi implementada na linguagem de programação Java[1] e aceita como entrada um arquivo XMI (XML Metadata Interchange) [OMG 07b] formato padrão utilizado para exportação de modelos UML. A ferramenta converte o arquivo XMI em instâncias de objetos que representam esse modelo em total conformidade com a especificação da UML 2.0. Esta ferramenta implementa todos algoritmos que serão apresentados neste trabalho para a conversão do diagrama de atividades no grafo de fluxo de controle interprocedimental e também provê recursos para a visualização gráfica da representação do fluxo de controle e fluxo excepcional do modelo.

Por fim, para demonstrar a viabilidade da abordagem proposta, apresentamos dois estudos de casos: o primeiro é um sistema de mineração (Mining System, apresentado em [Slom 87]). Este sistema controla a extração de mineral que é um processo que além de produzir água, também libera gás metano no ambiente. O sistema de controle da mineração é usado para drenar a água acumulada em um reservatório, além de extrair o metano da mina quando seu nível se torna alto. O segundo estudo de caso é um

[1]Java: http://www.java.com

sistema utilizado em máquinas automáticas para venda de café e refrigerantes (Vending Machine, apresentado em [Sinh 99]), este estudo de caso é apresentado para possibilitar uma comparação da abordagem proposta com a abordagem apresentada por Sinha et al.

1.3 Estrutura da Dissertação

Este documento foi dividido em seis capítulos, organizados da seguinte forma:

- **Capítulo 2 - Fundamentação Teórica:** O segundo capítulo apresenta os fundamentos teóricos necessários para embasar este trabalho. Esses conceitos estão classificados em três categorias: (i) tratamento de exceções, (ii) análise de fluxo de dados e (iii) diagrama de atividades. Por fim, são abordados neste capítulo os trabalhos relacionados.

- **Capítulo 3 - Abordagem proposta:** Neste capítulo apresentamos a abordagem proposta para a validação do fluxo excepcional do diagrama de atividades da UML 2.0. O capítulo está subdividido em: (i) descrição da abordagem e (ii) contribuições do trabalho.

- **Capítulo 4 - Grafo de Fluxo de Controle:** Neste capítulo apresentamos os passos necessários para a construção do grafo de fluxo de controle. O capítulo está subdividido em: (i) construção do grafo de atividades; (ii) obtenção do grafo de fluxo de controle a partir do grafo de atividades e (iii) criação do fluxo de controle intraprocedimental.

- **Capítulo 5 - Grafo de Fluxo de Controle Interprocedimental:** Neste capítulo apresentamos os passos necessários para a construção do grafo de fluxo de controle interprocedimental. O capítulo está subdividido em: (i) geração do grafo de fluxo de controle interprocedimental; (ii) construção do grafo resumida da interface abordagem e (ii) construção do grafo de fluxo de controle interprocedimental e (iii) criação do fluxo de controle interprocedimental.

- **Capítulo 6 - Ferramenta de Validação:** Neste capítulo apresentamos a ferramenta desenvolvida para efetuar a validação do fluxo excepcional. São apresentados os detalhes de arquitetura, interface gráfica e funcionalidades.

- **Capítulo 7 - Estudos de Caso:** Neste capítulo apresentamos dois estudos de casos para demonstrar a abordagem proposta, são eles: (i) Sistema de Mineração e (ii) Vending Machine.

- **Capítulo 8 - Conclusões:** Neste capítulo são apresentadas as conclusões deste trabalho. Além disso, são apontadas as principais contribuições e alguns direcionamentos para os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os fundamentos teóricos utilizados para o desenvolvimento do trabalho. Os conceitos estão agrupados em quatro seções. A seção 2.1 apresenta os conceitos sobre tolerância a falhas e tratamento de exceções. A seção 2.3 apresenta as estruturas de dados, conceitos e definições relacionados à análise de fluxo de dados. A seção 2.2 apresenta os conceitos e definições sobre o Diagrama de Atividades da UML 2.0 e sobre a semântica das ações da UML. Por fim a seção 2.4 apresenta os trabalhos relacionados.

2.1 Tratamento de Exceções

O comportamento normal de um sistema é responsável por oferecer os serviços especificados nos seus requisitos. Porém existem circunstâncias que impedem a execução adequada desses serviços. Como se espera que estas circunstâncias ocorram raramente, programadores referem-se a elas como exceções. Apesar de se esperar uma ocorrência rara, na prática não é bem isto que se observa. Devido à grande influência do meio externo nos sistemas computacionais, as situações excepcionais são bastante frequentes [Cris 89b].

A importância do tratamento de exceções é reconhecida por projetistas de sistemas e engenheiros de software. O tratamento de exceções é muitas vezes a parte mais importante do sistema, pois lida com situações anormais. O objetivo dos mecanismos de tratamento de exceções é tornar os programas mais robustos e confiáveis. No entanto, em algumas situações, apesar de mais da metade do código ser dedicado a detecção e tratamento de exceções, muitas falhas são causadas por tratamentos incorretos ou incompletos dessas situações anormais. Análise de acidentes efetuadas em sistemas controlados por computador, mostram que muitas vezes as causas tendem a ser o tratamento inadequado das situações excepcionais (ver, por exemplo, [Lion 96]).

Na década de 1970, pesquisas sobre tratamento de exceções foram realizadas princi-

palmente no contexto do projeto de linguagens. Agora a situação é diferente: as questões sobre o tratamento de exceções estão sendo estudadas e pesquisadas nos mais diversos contextos, tais como sistemas tolerantes a falhas, dependabilidade, engenharia de software, sistemas orientados a objetos, projeto de linguagens de programação, sistemas de tempo real, processos e workflows, etc. Muitas destas disciplinas têm enfrentado requisitos únicos em modelos de desenvolvimento de tratadores de exceções [Perr 00].

2.1.1 Definições: Falhas, Erros e Defeitos

Uma *falha* pode ser ocasionada por algum fenômeno natural de origem interna ou externa ao sistema ou por alguma intervenção humana intencional ou acidental. A ocorrência de uma falha em um componente de software pode levar a manifestação de um *erro*. Caso este erro não seja tratado devidamente pode se transformar em um *defeito* do sistema. Um defeito é considerado um desvio da especificação que deve ser evitado. Além de indicar um estado errôneo irreversível, a ocorrência de um defeito pode gerar novas falhas causando um fenômeno conhecido como propagação da falha. Portanto, um defeito deve ser contido através do seu confinamento [Lapr 95].

As falhas podem ser classificadas conforme a sua frequência, podemos dividi-las em três categorias: (a) *transientes*, quando existe a possibilidade de ocorrer mais de uma vez, mas não obrigatoriamente, muitas vezes se a operação é repetida, a falha desaparece, por exemplo, uma falha numa transmissão usando micro-ondas causada por algum objeto que obstrui temporariamente a comunicação. (b) *intermitentes*, quando a falha se repete dentro de um período de tempo, não necessariamente em períodos definidos, ocorre de maneira aleatória e imprevisível, por exemplo, um conector com mau contato. (c) *permanentes*: são aquelas caracterizadas pela ocorrência constante no sistema, por exemplo, falhas de especificação e implementação.

A ocorrência de uma falha pode resultar em nenhum, um ou vários erros. Em outros casos, várias falhas distintas podem acarretar no mesmo erro. Todos estes fatores tornam muito difícil a tarefa de detecção de falhas, pois a detecção de um erro não garante a detecção da falha que deu origem a este erro. Desta maneira, é necessário uma análise rigorosa do contexto de manifestação dos erros para poder inferir suas causas e efetuar um tratamento adequado.

2.1.2 Mecanismos de Tratamento de Exceções

Com a crescente complexidade dos softwares, os mecanismos de tratamento de erros oferecidos pelas linguagens de programação tornaram-se cada vez mais importantes. Técnicas de tratamento de erros tradicionais, como o uso de variáveis globais para indicar um erro (modelo utilizado na linguagem C), dificultam a leitura e tornam o código ainda mais

propenso a erros. Uma alternativa para estas técnicas foi introduzida pelo C++. O mecanismo de tratamento de exceções desta linguagem oferece uma técnica de tratamento de erros elegante que rapidamente se tornou a principal ferramenta para lidar com erros em linguagens orientadas a objeto. O conceito do tratamento de exceções é baseado na separação do processamento de erros do resto do código, redirecionando o fluxo de controle de execução normal para o bloco de tratamento da exceção. Dada a capacidade de capturar, lançar e propagar uma exceção, os desenvolvedores passaram a contar com uma ferramenta poderosa para criar componentes robustos e encapsulados.

Os Mecanismos de Tratamento de Exceções (MTE) capacitam os desenvolvedores a definirem o comportamento excepcional dos sistemas, que consiste tanto nas atividades de detecção e de lançamento das exceções, quanto nos seus tratadores correspondentes. Quando um erro é detectado, uma exceção é gerada, ou lançada e o MTE da linguagem de programação ativa automaticamente o tratador de exceções mais próximo que se adeque ao tipo da exceção lançada. Dessa forma, se a mesma exceção puder ser lançada em diferentes partes do programa, é possível que diferentes tratadores sejam executados. Por esse motivo, o local de lançamento da exceção também é conhecido como Contexto de Tratamento da Exceção (CTE). Um CTE é uma região de um programa onde exceções de um mesmo tipo são tratadas de maneira uniforme. Cada CTE possui um conjunto de tratadores associados a ele e cada tratador, por sua vez, está associado a um tipo de exceção em particular. Uma exceção lançada dentro de um CTE pode ser capturada por um dos seus tratadores. Quando a exceção é tratada, o sistema pode retomar o seu comportamento normal de execução; caso contrário, a exceção é sinalizada para o CTE de nível imediatamente superior [Good 75].

2.1.3 Tratamento de Exceções em Sistemas Baseados em Componentes

Segundo Szyperski [Szyp 02], um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (interfaces providas), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através de suas interfaces requeridas.

Segundo Lee e Anderson [Lee 90a], um sistema computacional é definido em termos de componentes que interagem entre si para fornecer a funcionalidade desejada. Esta definição é recursiva, um componente pode ser um sistema, formado por diversos componentes. O termo componente é empregado em um sentido genérico, isto é, um modelo abstrato que se refere tanto a componentes de software quanto componentes de hardware. Componentes recebem requisições de serviços e produzem respostas a estas requisições.

Com o intuito de produzir respostas a uma dada requisição, um componente pode solicitar a seus sub-componentes um determinado serviço e assim sucessivamente.

Existem algumas diferenças entre o tratamento de exceções no modelo orientado a objetos e no do Desenvolvimento Baseado em Componentes (DBC), algumas particularidades não são totalmente satisfeitas pelas linguagens de programação atuais. Uma das dificuldades para o desenvolvimento de sistema tolerante a falhas baseados em componentes é a falta da separação entre o tratamento excepcional e a implementação do comportamento normal. Isto prejudica o seu entendimento e principalmente a sua utilização [Luer 00].

Outra necessidade para os MTE no DBC é a conversão de alguns tipos exceções que são propagadas entre os componentes. Durante a propagação do fluxo excepcional, pode ser necessário converter os tipos de algumas exceções entre componentes que adotam modelos de tratamento de exceções distintos [Guer 04, Mesq 01b].

Componentes de software recebem requisições de serviço e produzem respostas, estas respostas podem ser dividas em duas categorias distintas: normais e excepcionais (ou anormais). *Respostas normais* correspondem aquelas situações onde o componente provê o seu serviço satisfatoriamente. *Respostas excepcionais* são aquelas produzidas quando alguma situação excepcional ocorreu durante a execução do serviço que não pôde ser realizado com sucesso. As respostas excepcionais de um componente são denominadas exceções [Good 75].

Exceções podem ser classificadas em duas categorias diferentes: *exceções internas* são aquelas lançadas por um componente com o objetivo de invocar a sua própria atividade de recuperação de erros, e, se a exceção for tratada satisfatoriamente, o componente pode voltar a prestar o seu serviço normalmente; e *exceções externas* são aquelas lançadas quando um componente sinaliza, por alguma razão, que ele não pode prestar o seu serviço especificado. As exceções externas são divididas nas seguintes categorias: *exceções de interface* que são lançadas devido a uma solicitação de serviço inválido, *exceções de defeito* que são lançadas devido a uma falha no processamento de um pedido válido. Dessa maneira, exceções e tratamento de exceções fornecem um mecanismo adequado para a estruturação das atividades de tolerância a falhas de um sistema [Brit 05].

Como consequência do particionamento das respostas em duas categorias, a atividade de um componente também pode ser particionada em atividade normal, que implementa os serviços normais do componente e a atividade excepcional (ou anormal), que implementa as medidas de tolerância as falhas que causaram a manifestação das exceções. A Figura 2.1 mostra um componente ideal tolerante a falhas [Lee 90b], que é um conceito estruturado para construção de sistemas tolerantes a falhas através de técnicas de tratamento de exceções. Um componente ideal promove a separação entre atividade normal e excepcional. Ao receber uma solicitação de serviço, um componente ideal produz três

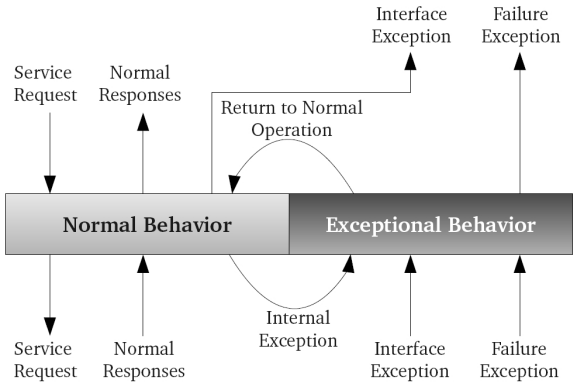


Figura 2.1: Componente ideal Tolerante a Falhas

tipos de respostas: respostas normais, as exceções de interface, e as exceções de defeito.

Componentes ideais podem ser organizados em camadas, de modo que os componentes podem manipular exceções lançadas pelos componentes localizados em outras camadas. Idealmente, a arquitetura de software do sistema é dividida em camadas que compõem os diferentes níveis de abstração. Assim, cada camada é responsável pela manipulação das exceções lançadas pela camada imediatamente abaixo dela.

2.2 Diagrama de Atividades

A UML (Unified Modeling Language) [OMG 07a] é uma linguagem de modelagem e documentação de software que surgiu em 1996 com o objetivo de acelerar a transição para o paradigma de orientação à objetos. Teve sua origem na compilação das melhores práticas de engenharia que provaram ter sucesso na modelagem de sistemas computacionais grandes e complexos. Os conceitos de Booch [Booc 93], Rumbaugh [Rumb 90] e Jacobson [Jaco 94] foram fundidos numa única linguagem de modelagem. Atualmente a linguagem UML é mantida pela OMG (Object Management Group).

A UML é uma notação amplamente utilizada para fins de documentação e especificação durante o processo de desenvolvimento de um software. Rapidamente tornou-se um padrão no desenvolvimento orientado a objetos. A UML é dividida em duas especificações, estrutural e comportamental. A primeira modela os aspectos estáticos de um sistema, já a segunda os aspectos dinâmicos. Neste segundo grupo se enquadram os modelos comportamentais, que especificam como os aspectos estruturais de um sistema mudam através do tempo. O Diagrama de Atividades (DA) é um exemplo desse tipo de modelo, sendo considerado a escolha natural (dentro da UML) para modelar os aspectos comportamentais de um sistema, de uma classe ou mesmo de uma operação. É utilizado para ilustrar uma sequência de atividades que foram modeladas para representar os aspectos

dinâmicos de um processo computacional. Este modelo detalha o fluxo de controle e o fluxo de dados de uma determinada atividade, mostrando as ramificações de controle e as situações onde existem processamento paralelo.

Apresentamos a seguir os elementos do diagrama que serão utilizados nesse trabalho. O leitor interessado poderá consultar a especificação da UML [OMG 07a] para maiores detalhes sobre o diagrama de atividades.

O diagrama de atividades é decomposto em Atividades e Ações. Uma atividade é representada por um retângulo de cantos arredondados, especifica um processamento complexo, que pode ser dividido em várias outras atividades originando um novo diagrama, por esse motivo o próprio Diagrama de Atividades é considerado uma atividade. A Figura 2.2 exibe um exemplo de Diagrama de Atividades com fluxo de controle e dados.

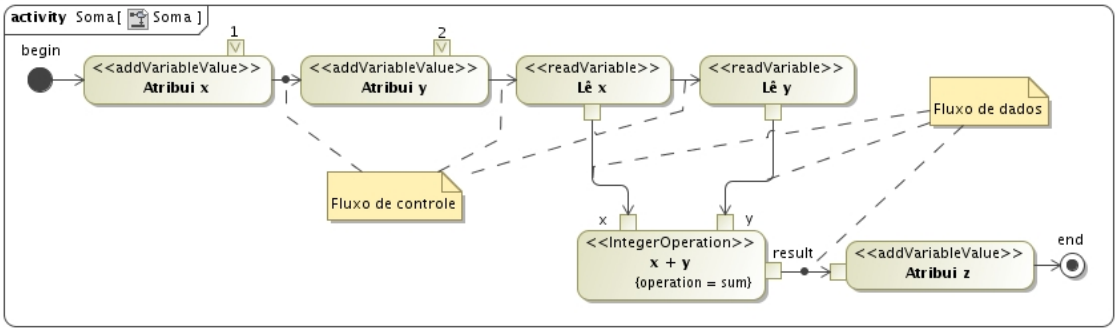


Figura 2.2: Diagrama de Atividades

2.2.1 Atividades e Ações

As atividades são comportamentos que modelam uma execução não-atômica, já uma ação representa um processamento atômico [Bock 03a]. As atividades (*Activity*) coordenam as ações (*Action*). Essa coordenação é capturada como um grafo, onde os nós (*ActivityNode*) representam as ações e são conectados por arestas (*ActivityEdge*). O Fluxo de dados é representado por nós de objetos (*ObjectNode*) e por arestas de fluxo de dados (*ObjectFlow*). O controle e os dados fluem ao longo das arestas e são manipulados pelos nós, são roteados para outros nós ou são armazenados temporariamente. Mais especificamente, os nós de ação manipulam o controle e dados recebidos através das arestas do grafo e proveem controle e dados para outras ações; os nós de controle roteiam o controle e dados através do grafo; os nós de objetos armazenam os dados temporariamente até serem movidos através do grafo. O termo *token* é utilizado para denominar o controle e os valores dos dados que fluem através de uma atividade.

Para ser executada, uma ação precisa saber quando iniciar e quais são suas entradas. Estas condições são determinadas pelo fluxo de controle e fluxo de dados. Uma ação pode ter entradas e saídas, que são chamadas de pinos. Os pinos são conectados por arestas de fluxo de objetos, que identificam o tipo do objeto que está sendo enviado. Na Figura 2 é possível observar a utilização dos pinos `x` e `y` como parâmetros da ação "`x + y`"; a execução ocorrerá somente quando os dois valores estiverem disponíveis [Bock 03b].

O meta-modelo de atividades da UML está organizado em vários níveis. A seguir é apresentada uma breve descrição dos pacote relevantes e das suas atividades:

1. *Atividades Fundamentais (FundamentalActivities):* O nível fundamental define as atividades e as ações. Uma atividade contém nós que podem ser ações ou outras atividades aninhadas.

2. *Atividades Básicas (BasicActivities):* Este nível inclui o fluxo de controle e de dados entre as ações, inclui também os seguintes elementos (Figura 2.3):

   - *Nó de Inicialização (InitialNode):* é o nó inicial do fluxo de uma atividade. Uma atividade pode conter mais de um nó de inicialização, indicando que, quando a atividade é iniciada, vários fluxos são iniciados também.
   - *Nó Final de Atividade (ActivityFinalNode):* a chegada do primeiro token a este nó termina toda a atividade, terminando todos os fluxos concorrentes, se existirem.

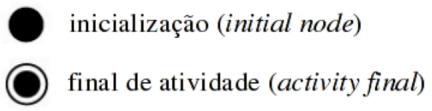


Figura 2.3: Atividades Básicas

3. *Atividades Intermediárias (IntermediateActivities):* Este nível intermediário possibilita a modelagem de diagramas de atividades que incluem fluxos de controle e de dados concorrentes e também os nós de decisão e mesclar (Figura 2.4), descritos a seguir:

   - *Nó de Decisão (DecisionNode):* direciona o fluxo em uma direção ou outra, dependendo da avaliação da expressão associada.
   - *Nó Mesclar (MergeNode):* tem a mesma notação que um nó de ramificação, a diferença é que chegam várias arestas de fluxos alternativos e segue apenas uma aresta.

- *Nó de Bifurcação (ForkNode):* divide o fluxo em múltiplos fluxos concorrentes. Os tokens de dados e controle que chegam a bifurcação são duplicados.
- *Nó de União (JoinNode):* sincroniza os múltiplos fluxos, combinando os tokens recebidos de cada fluxo.
- *Nó Final de Fluxo (FlowFinalNode):* destrói o token de controle ou de dado que alcança o nó. Se existirem outros fluxos, a atividade continua. É utilizado em fluxos concorrentes.

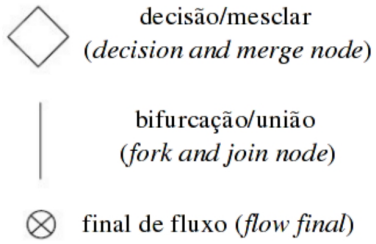


Figura 2.4: Atividades Intermediárias

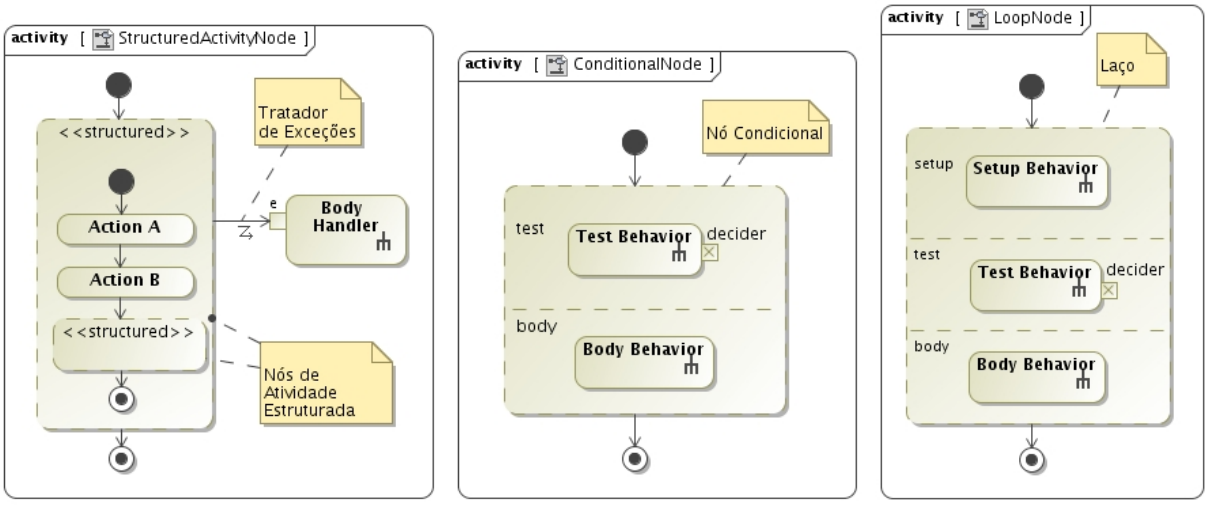
4. *Atividades Estruturadas (StructuredActivities)* e *Atividades Estruturadas Completas (CompleteStructuredActivities):* Estes pacotes permitem a modelagem de construções tradicionais de programação estruturada, como laços e nós condicionais. Adicionam o suporte para o fluxo de dados incluindo os pinos de saídas e entrada, conforme descrito a seguir:

   - *Variável (Variable):* Variáveis são elementos utilizados para compartilhar dados entre as ações. O resultado de uma ação pode ser escrito em uma variável e utilizado como uma entrada de uma ação subsequente, isto caracteriza um caminho indireto de fluxo de dados. Não há nenhuma relação pré-definida entre as ações que leem e escrevem dados nas variáveis, estas ações devem ser sequenciadas pelo fluxo de controle para prevenir condições de corrida entre ações que leem e escrevem na mesma variável. Todos os valores contidos por uma variável devem estar em conformidade com o tipo da variável e ter cardinalidades compatível com multiplicidade definida pela variável. Uma variável possui um escopo, que pode ser uma atividade (propriedade *activityScope*) ou um nó de atividade estruturada (propriedade *scope*), mas não ambos. Os valores armazenados por uma variável podem ser acedidos somente pelo grupo de ações ou atividades pertencentes ao escopo da variável. As variáveis foram introduzidas para simplificar a tradução de linguagens de programação comuns em modelos de atividades.

- *Nó de Atividade Estruturada (StructuredActivityNode):* São nós executáveis que podem conter outras atividades subordinadas. Os nós subordinados devem pertencer a apenas um nó de atividade estruturada, embora possam ser aninhados. Uma atividade estruturada também pode conter tratadores de exceção. Um exemplo pode ser visto na Figura 2.5a.
- *Nó Condicional (ConditionalNode):* É uma atividade estruturada que representa uma escolha exclusiva entre algumas alternativas (Figura 2.5b). Esta atividade pode possuir diversas cláusulas, sendo que uma cláusula é composta de duas seções: teste *(test)* e corpo *(body)*. A seção *test* deve possuir uma ação que retorne um valor booleano. Caso esta ação retorne verdadeiro a ação vinculada à seção *body* é executada. Caso contrário a próxima cláusula da atividade, se existir, é testada. A primeira cláusula satisfeita interrompe a verificação das demais. No exemplo da Figura 2.5b a ação *"Test Behavior"* possui um pino de saída *decider* que é utilizado para fazer a verificação; caso o valor retornado pelo pino *decider* seja verdadeiro, a ação *"Body Behavior"* é executada.
- *Laço (LoopNode):* É uma atividade estruturada utilizada para representar um laço. Esta atividade é dividida em três seções: *setup* onde é feita a inicialização das variáveis, *test* que representa o teste condicional para determinar a condição de saída do laço e *body* que representa o corpo do laço. A seção de teste pode preceder ou suceder a seção do corpo, o que define a ordem é o valor boleano da propriedade *isTestedFirst*, se for verdadeiro o teste é executado antes da execução do corpo, o valor padrão é falso. A seção de inicialização é executada apenas uma vez na entrada do laço, as seções de teste e corpo são executadas repetidamente até que o teste produza um valor falso. Na Figura 2.5c a seção *setup* é representada pela ação *"Setup Behavior"*, a seção *test* é representada pela ação *"Test Behavior"* que possui um pino de saída *decider* o qual é avaliado para determinar a condição de saída do laço. Por fim a ação *"Body Behavior"* representa o corpo do laço.

5. *Atividades Estruturadas Extras (ExtraStructuredActivities):* Este nível permite representar o tratamento de exceções.

   - *Tratador de Exceções (ExceptionHandler):* Representa uma aresta de exceção, indicando que uma atividade estruturada trata as exceções ocorridas no seu interior (Figura 2.5a). No exemplo da Figura 2.5a existe um *ExceptionHandler* ligando a atividade estruturada com a ação *"Body Handler"* que efetivamente fará o tratamento da exceção. Esta ação possui um pino de entrada *e* que



(a) (b) (c)

Figura 2.5: Atividades Estruturadas

representa a referência ao objeto de exceção. O tipo de exceção capturada pela aresta de exceção é especificada pela propriedade *Exception Type*. Uma aresta de exceção é capaz de capturar diversos tipos de exceções, a propriedade *Exception Type* especifica uma coleção de tipos de exceção tratadas por esta aresta. Maiores detalhes sobre o tratamento de exceções no DA serão apresentados adiante neste documento.

Para reduzir a complexidade a abordagem proposta não irá tratar dos fluxos concorrentes e nem das ações com comportamento não-determinístico. O subconjunto das atividades permitidas é mostrado na Tabela 2.1.

2.2.2 Semântica de Ações da UML 2.0

Uma ação é uma unidade fundamental de especificação de comportamento. Uma ação toma um conjunto de entradas e converte em um conjunto de saídas, embora um ou outro, ou mesmo ambos os conjuntos podem ser vazios. Ações estão contidas em comportamentos, os quais definem o seu contexto. Comportamentos também definem restrições entre as ações para determinar quando as ações executam e quais são as suas entradas. De um modo geral, uma ação pode expressar variáveis, operações lógicas e aritméticas, e várias outras funcionalidades básicas de linguagens de programação imperativas.

O modelo de ações da UML [OMG 07a] apresenta um conjunto de ações com uma semântica bem definida. No contexto de fluxo de dados uma ação pode ser classificadas

Tabela 2.1: *Subconjunto de Atividades Permitidas*

Pacote	**Atividades**
Atividades Básicas	Nó Inicial (InitialNode) Nó Final de Atividade (ActivityFinalNode)
Atividades Fundamentais	Atividade (Activity) Ação (Action)
Atividades Intermediárias	Nó de Decisão (DecisionNode) Nó Mesclar(MergeNode) Nó Final de Fluxo (FlowFinalNode)
Atividades Estruturadas	Nó Condicional (ConditionalNode) Laço (LoopNode) Atividade Estruturada (StructuredActivityNode) Variável (Variable)
Atividades Estruturadas Extras	Tratador de Exceções (ExceptionHandler)

como: *ação de escrita*: aquela que faz a definição de dados ou *ação de leitura*: aquela que faz uso de algum dado definido previamente. Podem existir ações que se enquadram em ambas ou em nenhuma das classificações. A seguir é apresentada uma breve descrição das ações relevantes ao contexto deste relatório:

- *Ação de chamada de comportamento (CallBehaviorAction):* É identificada pelo símbolo em forma de tridente e representa a inclusão de outro diagrama de atividades que detalha o seu comportamento. Quando todos os pré-requisitos para a execução da ação forem satisfeitos, esta ação faz uma chamada ao comportamento especificado. Os valores disponíveis nos pinos de entrada são passados como argumentos. Os pinos da ação devem ser equivalentes aos parâmetros do comportamento em número, tipo, direção (entrada/saída) e multiplicidade. O comportamento chamado deve ser capaz de aceitar e retornar o controle. No caso de uma chamada assíncrona, a ação é completada imediatamente sem dependências de execução entre o comportamento chamado e o comportamento onde a ação está contida. Se a chamada for síncrona, a execução da ação é bloqueada até receber uma resposta do comportamento chamado. Quando a execução do comportamento chamado termina, os valores dos resultados são disponibilizados nos pinos de saída da ação. Se a execução do comportamento chamado produzir uma exceção, a exceção é transmitida para a ação *CallBehaviorAction* para iniciar a procura por um tratador de exceção.

- *Ação de chamada de Operação (CallOperationAction):* Representa a chamada de uma operação do objeto alvo disponibilizado no pino de entrada *target*. Esta ação realiza a invocação do comportamento associado à operação, como por exemplo, o método de uma classe que esta modelado em outro diagrama de atividades. Além do

objeto alvo, este tipo de ação pode ter parâmetros de entrada e saída. Os pinos de entrada representam os argumentos, já os pinos de saída representam os parâmetros de retorno da operação. Os parâmetros de entrada e de saída da operação devem ser equivalentes aos pinos incluídos na ação em número, tipo, direção (entrada/saída) e multiplicidade. Quando todos os pré-requisitos para a execução da ação são satisfeitos, as informações sobre a operação e os argumentos de entrada são enviados para o objeto alvo. Caso a chamada seja síncrona, a ação é bloqueada e aguarda o término da execução da operação chamada. Quando a resposta comunicando o fim da execução do comportamento chega até a ação, os valores de retorno são disponibilizados nos pinos de saída da ação e a execução da ação é completada. Se a chamada for assíncrona, o comportamento chamador avança imediatamente e a execução da ação é completada. Nenhum retorno da operação chamada é enviado de volta para o comportamento chamador. Se a execução da operação chamada produzir uma exceção, a exceção é transmitida ao comportamento chamador onde é novamente lançada como uma exceção da ação *CallOperationAction*. Os possíveis tipos de exceção podem ser especificados na operação chamada.

- *Ação opaca (OpaqueAction):* É uma ação com a semântica específica determinada pela sua implementação; sendo utilizada para representar operações aritméticas e booleanas. Esta ação semanticamente não realiza nenhum uso ou definição de dados. Em relação ao fluxo de dados, consideramos que os usos ocorreram nas ações de leitura antecedentes que disponibilizam os valores nos pinos de entrada e as definições ocorrerão nas ações de escrita subsequentes que utilizem os resultados disponibilizados pelos pinos de saída da ação.

- *Definição de característica estrutural (AddStructuralFeatureValueAction):* Esta ação é classificada como uma ação de escrita, ou seja, realiza uma definição de dados. O pino de entrada *value* recebe o valor que será inserido em uma característica estrutural (Ex: atributo de uma classe). Também é possível utilizar o pino *insertAt* para identificar a posição da característica estrutural que receberá o valor, no caso de uma característica multivalorada (Ex: uma coleção). O nome da característica estrutural é especificado estaticamente pela propriedade *StructuralFeature*. A referência do objeto que detém a característica estrutural é informada pelo pino de entrada *object*.

- *Leitura de característica estrutural (ReadStructuralFeatureAction):* Esta ação é classificada como uma ação de leitura. O pino de saída *result* recebe o valor lido de uma característica estrutural (Ex: atributo de uma classe). O nome da característica estrutural é especificado estaticamente pela propriedade *StructuralFeature*. A referência do objeto que detém a característica estrutural é informada pelo pino

de entrada *object*. A multiplicidade da característica estrutural deve ser compatível com o pino de saída. Por exemplo, o modelador pode definir a multiplicidade do pino para suportar múltiplos valores mesmo quando a característica estrutural somente disponibiliza um único valor. Dessa maneira o modelo de ações não será afetado por mudanças na multiplicidade da característica estrutural.

- *Criação de Objeto (CreateObjectAction):* Cria um objeto de acordo com a classe especificada estaticamente na propriedade *Classifier*. O objeto criado é colocado no pino de saída e não possui nenhum valor em suas características estruturais. Uma ação deste tipo é considerada uma ação de escrita, pois o objeto criado pode ser utilizado pela ação subsequente, se esta for uma ação de leitura.

- *Leitura do objeto hospedeiro (ReadSelfAction):* Recupera o objeto hospedeiro da ação, fornecendo acesso ao objeto do contexto quando ele não está disponível como um parâmetro. A ação deve estar contida em um comportamento de uma classe, que represente o corpo de um método; neste caso o método não pode ser estático. O tipo do pino de saída deve ser do mesmo tipo do objeto hospedeiro.

- *Especificação de valor (ValueSpecificationAction):* Esta ação retorna como resultado o valor definido estaticamente na propriedade *value*. O valor é disponibilizado no pino de saída da ação. O tipo do valor especificado deve ser compatível com o tipo do pino de saída. Esta ação semanticamente não realiza nenhum uso ou definição de dados.

- *Definição de variável (AddVariableValueAction):* Ação de escrita, o valor fornecido no pino de entrada *value* é escrito na variável especificada estaticamente na propriedade *Variable*. Uma variável deve ser definida para ser utilizada por este tipo de ação. No caso de variáveis multivaloradas, para especificar a posição do elemento que deve ser atualizado é utilizado o pino de entrada *insertAt*.

- *Leitura de variável (ReadVariableAction):* É uma ação de leitura, que recupera o valor da variável, definida estaticamente pela propriedade *Variable*, disponibilizando esse valor no pino de saída *result*. No caso de variáveis multivaloradas, a ação lê os valores na ordem armazenada pela coleção.

- *Lançamento de exceção (RaiseExceptionAction):* É uma ação que provoca a ocorrência de uma exceção. A referência do objeto de exceção é colocada no pino de entrada *exception* e a execução desta ação lança este objeto como uma exceção. Esta ação é considerada de leitura por fazer uso de uma definição de um objeto de exceção. Quando uma exceção é lançada, a execução do nó estruturado ou atividade que contém esta ação é imediatamente terminada, iniciando em seguida uma busca nos

escopos aninhados por um tratador de exceções correspondente ao tipo do objeto de exceção.

A Tabela 2.2 apresenta um resumo da classificação das ações da UML 2.0 quanto à escrita ou leitura de dados. Na primeira coluna é apresentado o tipo da classificação, na segunda coluna é apresentado o nome da ação, a terceira coluna indica o nome da propriedade da ação que define o nome e o tipo de dado utilizado, a última coluna descreve a ocorrência da definição ou uso gerada pela ação.

Tabela 2.2: *Classificação das Ações UML quanto à Escrita/Leitura de Dados*

Tipo	**Ação**	**Propriedade**	**Descrição**
Escrita	AddStructuralFeatureValue	StructuralFeature	Definição de uma característica estrutural
	AddVariableValue	Variable	Definição de uma variável
	CallOperation	OutputPin [0..*]	Definição dos parâmetros de retorno
	ClearStructuralFeature	StructuralFeature	Definição de uma característica estrutural
	ClearVariable	Variable	Definição de uma variável
	CreateLinkObject	OutputPin	Definição de uma referência de objeto
	CreateObject	Classifier	Definição de uma referência de objeto
	DestroyObject	InputPin	Definição de uma referência de objeto
	Reduce	OutputPin	Definição do resultado da redução
	RemoveStructuralFeatureValue	StructuralFeature	Definição de uma característica estrutural
	RemoveVariableValue	Variable	Definição de uma variável
	Unmarshall	OutpuPin[1..*]	Definição de características estruturais
Leitura	AddVariableValue	InputPin	Leitura do parâmetro de entrada
	CallOperation	InputPin [0..*]	Uso dos parâmetros de entrada
	CreateLink	InputPin[0..*]	Uso dos parâmetros de entrada
	CreateLinkObject	InputPin[0..*]	Uso dos parâmetros de entrada
	Opaque	InputPin[0..*]	Uso dos parâmetros de entrada
	RaiseException	InputPin	Uso de uma referência de exceção
	Reduce	InputPin	Uso da referência de uma coleção
	ReadExtent	Classifier	Uso de uma referência de objeto
	ReadVariable	Variable	Uso de uma variável
	ReadStructuralFeature	StructuralFeature	Uso de uma característica estrutural
	TestIdentify	InputPin[2..2]	Uso dos parâmetros de entrada
	Unmarshall	InputPin[1..1]	Uso de uma referência de objeto
Nenhum	ReadSelf		
	ValueSpecification		

2.2.3 Tratamento de exceções no DA

Na maioria das linguagens orientadas a objetos, uma exceção é um evento que ocorre durante a execução de um programa, interrompendo o fluxo normal das instruções. Quando ocorre um erro dentro de um método, um objeto de exceção é criado. Este objeto contém

informações sobre o erro, incluindo o tipo e o estado do programa no momento que o erro ocorreu. Quando um objeto de exceção é lançado o programa procura na pilha de execução por um método que contenha um bloco de código que possa tratar a exceção. Este bloco de código é chamada de tratador de exceções. A procura começa dentro do método em que o erro ocorreu e continua através da pilha de execução na ordem reversa em que os métodos foram chamados. Quando um tratador de exceção apropriado é encontrado, o programa passa a exceção ao tratador de exceções. Um tratador de exceções é considerado apropriado se o tipo do objeto de exceção lançado é equivalente ao tipo que pode ser capturado pelo tratador de exceções [Gosl 05].

Exceções em Java pode ser classificadas como síncronas e assíncronas. Uma exceção síncrona ocorre em um ponto específico de um programa, podendo ser causada pela avaliação de uma expressão, execução de uma declaração ou um comando *throw* explícito. Por outro lado, uma exceção assíncrona pode ocorrer arbitrariamente em pontos não determinados do programa. Uma exceção síncrona pode ser classificada como verificada (*checked exception*) ou não verificada (*unchecked exception*). No caso de uma exceções verificada, o compilador deve procurar um tratador de exceções dentro do método ou deve existir uma declaração na assinatura do método para indicar que a exceção será lançada. Para as exceções não verificadas, o compilador não tenta encontrar um tratador de exceções ou alguma declaração na assinatura do método. Uma exceção síncrona é explicita se for lançada pelo comando *throw*, já uma exceção síncrona implícita é lançada se for causada na chamada de alguma rotina externa ao programa ou pelo ambiente de execução. Na Figura 2.6 é mostrada a classificação das exceções Java.

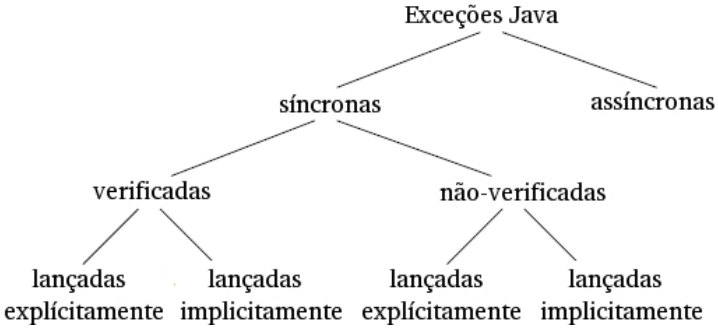


Figura 2.6: Classificação dos Tipos de Exceção do Java

A técnica que estamos apresentando neste documento não abrange as exceções assíncronas e exceções síncronas que são lançadas implicitamente, pois esses tipos de exceções não são modeladas explicitamente. Portanto, serão tratadas somente as exceções síncronas que são modeladas explicitamente.

Fazendo uma analogia com a linguagem de programação Java, a ação *RaiseExcepti-*

onAction é equivalente ao comando *throw*, o nó protegido é equivalente ao bloco *try*, o tratador de exceções (*ExceptionHandler*) é equivalente a uma cláusula *catch* e o corpo do tratador de exceções (*CallBehaviorAction*) é equivalente ao bloco da cláusula acionada. Na Figura 2.7 podemos observar cada uma dessas construções. A ação *"raise E2"* é um exemplo de uso da ação *RaiseExceptionAction*. Esta ação recebe um objeto de exceção no pino de entrada *e2* e lança uma exceção do tipo *E2*. Os nós *"Protected Node 1"* e *"Protected Node 2"* são exemplos de nós protegidos. O primeiro possui dois tratadores de exceção capazes de capturar exceções do tipo *E2* e *E3*, já o segundo possui um tratador de exceção capaz de capturar exceções do tipo *E1*. Cada tratador de exceção liga o nó protegido a uma ação do tipo *CallBehaviorAction* que abstrai o tratamento da exceção. As ações *"Body Handler E1"*, *"Body Handler E2"* e *"Body Handler E3"* são exemplos dessas ações.

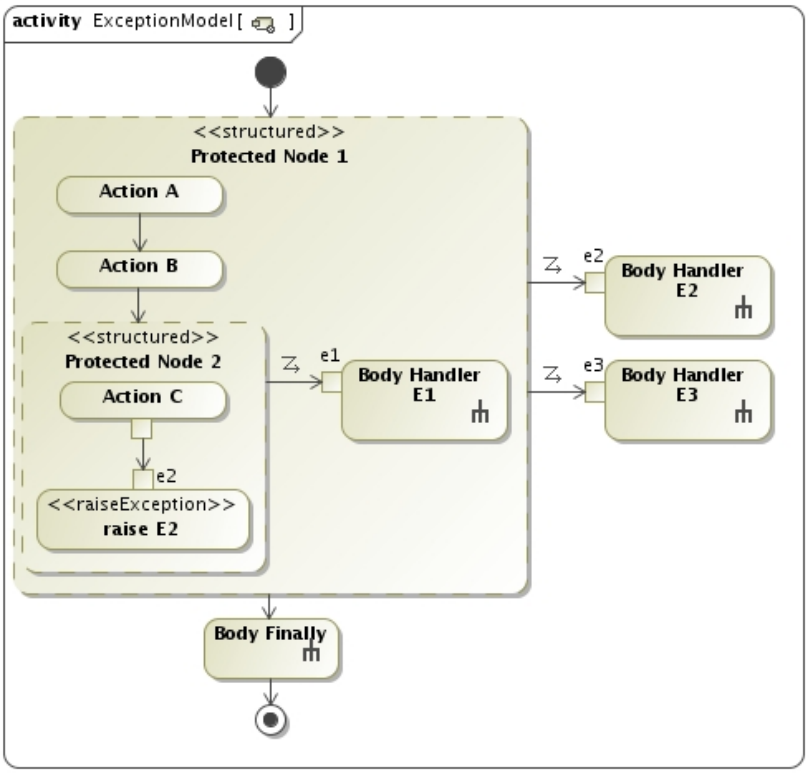


Figura 2.7: Modelo de Exceções do DA

Quando uma ação *RaiseExceptionAction* é executada, todos os tokens do nó protegido onde esta ação estiver contida são finalizados. Em seguida, o conjunto de tratadores de exceção do nó protegido são examinados para verificar se são compatíveis com a exceção lançada. Um tratador de exceções é compatível se o tipo da exceção é o mesmo ou é

uma subclasse do tipo especificado pelo tratador. Se os tipos forem compatíveis, então o tratador captura a exceção. Se existirem múltiplos tratadores compatíveis, apenas um irá capturar a exceção. Após a execução do tratador de exceções o fluxo de controle retorna logo depois do nó protegido vinculado ao tratador de exceções.

Como podem haver nós protegidos aninhados, se uma exceção não é capturada por nenhum dos tratadores de exceção do nó protegido interno, o processo se repete, propagando a procura para o próximo nó protegido, onde o nó interno estiver contido. No exemplo da Figura 2.7 uma exceção do tipo *E2* é lançada dentro do nó protegido *"Protected Node 2"*, os tratadores de exceção do nó protegido são avaliados para verificar se são capazes de tratar a exceção. No exemplo existe apenas um tratador do tipo *E1*, portanto a busca é propagada para o nó protegido *"Protected Node 1"* até encontrar um tratador compatível. Neste caso o tratador compatível é encontrado e o controle é passado para a ação *"Body Handler E2"* que irá invocar o comportamento responsável pelo tratamento da exceção lançada. Após a execução do tratador de exceções o fluxo de controle segue pela aresta de controle que liga o nó protegido *"Protected Node 1"* até a ação *"Body Finally"*.

A ação que representa o corpo do tratador de exceções não possui nenhuma aresta explícita, seja de entrada ou de saída. Contudo, esta ação pertence ao mesmo contexto do nó protegido. Assim, os tokens resultantes desta ação tornam-se tokens resultantes do nó protegido. Qualquer aresta de controle que deixe o nó protegido recebe os tokens de controle resultantes da execução da ação. Quando a execução da ação é completada, é como se o próprio nó protegido tivesse sido completado. Tendo em vista este comportamento, se desejarmos modelar um bloco de finalização (*finally*), podemos inserir uma ação (*CallBehaviorAction*) na sequência do nó protegido, inserindo também uma aresta de controle ligando o nó protegido com o bloco de finalização. Dessa forma, o bloco de finalização é executado na sequência da execução do nó protegido, seja uma execução normal ou excepcional. No exemplo da Figura 2.7 a ação *"Body Finally"* é utilizada para modelar um bloco de finalização.

2.3 Análise de Fluxo de Dados

Nesta seção serão apresentados todas as estruturas de dados, conceitos e definições referentes a representação do fluxo de controle e análise de fluxo de dados.

2.3.1 Grafo de Fluxo de Controle

As técnicas de análise de fluxo de dados utilizam uma representação conhecida como *Grafo de Fluxo de Controle (GFC)*. Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos. A execução do primeiro comando de um bloco acarreta a

execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, com exceção do primeiro e do último comando, têm um único predecessor e exatamente um único sucessor.

Usualmente, a representação de um programa P como um GFC ($G = (N, E, s)$) consiste em estabelecer uma correspondência entre os vértices (nós) e blocos, e em indicar possíveis fluxos de controle entre blocos por meio das arestas (arcos). Assume-se que um GFC é um grafo orientado, com um único nó de entrada $s \in N$, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco de comandos tem as seguintes características: uma vez que o primeiro comando do bloco é executado, todos os demais são executados sequencialmente, não existe desvio de execução para nenhum comando dentro do bloco [Dela 07].

2.3.2 Grafo de Fluxo de Dados

O grafo *Def-Uso*, ou *Grafo de Fluxo de Dados (GFD)*, é uma extensão do *GFC* onde são adicionadas as informações referentes ao fluxo de dados, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável (definição da variável) e pontos nos quais esse valor é utilizado (referência ou uso da variável). Este grafo é obtido a partir do *GFC* associando-se a cada nó n: o conjunto de variáveis com uso no vértice n, o conjunto de variáveis com definições no vértice n, e para cada aresta *(n,m)* o conjunto de variáveis com usos na aresta *(n,m)*. A Figura 2.8 apresenta um exemplo de GFD.

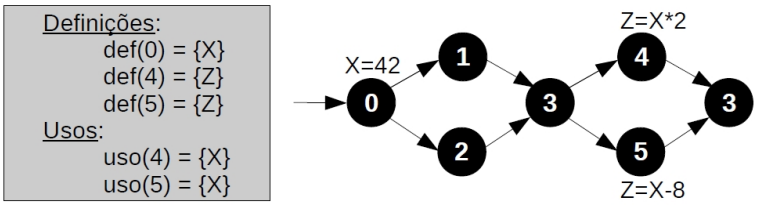


Figura 2.8: Grafo de Fluxo de Dados

Cada ocorrência de uma variável pode ser classificada como sendo uma definição *def*, um uso computacional *c-uso* ou um uso predicativo *p-uso*. Por exemplo, a declaração $y = f(x_i, ..., x_n)$ contém *c-usos* de $x_1, .., x_n$ seguido de uma definição de y. Já a declaração de transferência condicional *if* $p(x_1, .., x_n)$ *then* $\langle ... \rangle$ contém *p-usos* de $x_1, ..., x_n$ [Rapp 85].

Um par *definição-uso* é um par de localizações $(l_n,\ l_m)$ no qual uma variável v é definida em l_n e usada em l_m. Um caminho de l_n até l_m é um caminho livre de definição com respeito a variável v se v não receber outro valor em nenhum dos nós do caminho.

Se existe um caminho livre de definição de l_n até l_m com respeito a variável v, a definição de v em l_n alcança o uso em l_m [Dela 07].

Um *du-caminho* é um subcaminho simples que é livre de definição com respeito a v da definição de v até o uso de v. O conjunto de *du-caminhos* de l_n até l_m com respeito a v é definido como *du(l_n, l_m, v)* [Amma 08].

Uma vez que estamos interessados em capturar o fluxo de dados entre os nós, qualquer definição que é usada dentro do mesmo nó em que ocorreu a definição, tem pouca importância. Assim fazemos a seguinte distinção: *c-uso* de uma variável x é um *c-uso global*, se não existir nenhuma definição de x precedendo o *c-uso* dentro do mesmo bloco. Isto é, o valor de x deve ter sido definido em algum outro bloco diferente daquele em que está sendo usado. De outra forma isto seria um *c-uso local*.Como um *p-uso* está associado somente com arestas, nenhuma distinção precisar ser feita entre *p-usos locais* e *globais*.

Uma definição de uma variável x em um nó i é uma *definição global* se esta é a última ocorrência de x dentro do bloco associado com o nó i e existe um caminho livre de definição do nó i até o nó que contenha o *c-uso global* de x ou até uma aresta que contenha um *p-uso* de x [Rapp 85].

Equações de Fluxo de Dados: As informações de fluxo de dados podem ser coletadas estabelecendo e solucionando-se sistemas de equações que relacionam informações em vários pontos do programa. Estas equações são chamadas de equações de fluxo de dados. Uma equação típica possui a seguinte forma

$$saida[B] = geradas[B] \bigcup (entrada[B] - mortas[B])$$

e pode ser lida como *"as informações ao final de um bloco B ou são geradas dentro do bloco ou entram ao início e não são mortas à medida que o controle flui através do bloco"*. As noções de *"gerar"* e *"matar"* dependem da informação desejada, isto é, do problema de análise de fluxo de dados a ser resolvido [Aho 86].

Definições Incidentes: Uma definição d incide num ponto p se existir um percurso do ponto que se segue imediatamente a d até p, tal que d não seja "morta" ao longo do percurso. Intuitivamente, se uma definição d de alguma variável a incide em um ponto p, d poderia ser o local no qual o valor de a usado em p teria sido definido. Matamos uma definição de uma variável a se, entre dois pontos ao longo do percurso, existir uma nova definição de a [Aho 86].

Neste tipo de análise somente os conjuntos de definições são utilizados. As definições geradas em um nó "matam" as definições correspondentes à mesma variável nos demais nós. Um algoritmo clássico utilizado para o cômputo das definições incidentes foi proposto por Aho et al. [Aho 86]. Este algoritmo utiliza a seguinte equação de fluxo de dados:

$$
\begin{aligned}
geradas[n] &= \text{definições geradas pelo nó } n \\
mortas[n] &= \text{definições mortas pelo nó } n \\
entrada[n] &= \bigcup_{P \in predecessores[n]} saida[P] \\
saida[n] &= geradas[n] \bigcup (entrada[n] - mortas[n])
\end{aligned}
$$

Nos capítulos 4 e 5 são apresentados exemplos desta análise de fluxo de dados que é utilizada para inferir quais são os tipos de exceção que podem ser lançadas por um determinado nó do GFC.

2.3.3 Grafo de Fluxo de Controle Interprocedimental

Um *Grafo de Fluxo de Controle Interprocedimental (GFCI)* para um programa P consiste em um conjunto de GFCs para cada operação em P; cada *nó de chamada* é conectado ao *nó de entrada* da operação chamada por uma *aresta de chamada*, e o *nó de saída* da operação chamada é conectado com o *nó de retorno* correspondente por uma *aresta de retorno* [Sinh 99].

Quando são utilizadas estruturas para o tratamento de exceções, um GFCI deve possuir arestas adicionais para representar o fluxo de controle interprocedimental, são as *arestas de retorno de excepcional*. Uma aresta de retorno excepcional é uma aresta que conecta um *nó de saída excepcional* da operação chamada com o *nó de captura de exceção* ou com outro nó de saída excepcional da operação chamadora. Se uma operação propaga uma exceção que é capturada pelo chamador daquela exceção, o nó de saída excepcional para aquele tipo de exceção é conectado com o nó apropriado na operação chamadora por uma *aresta de exceção* Uma operação pode propagar uma exceção que não é tratada na operação chamadora imediata, mas é tratada em uma operação que fica mais acima na cadeia de chamadas [Sinh 99].

A Figura 2.9 apresenta o fluxo de controle interprocedimental no nível de bloco. Esta figura adapta o fluxo de controle normal e excepcional de um programa Java para o contexto do DA da UML [Sinh 99]. No topo é mostrada a operação chamada, B, e logo abaixo é mostrada a operação chamadora, A. A chamada de B dentro da região protegida de A é mostrada pelo nó de chamada. Se B completar sua execução normalmente, o controle retorna para A através da saída normal de B. Por outro lado, se B propagar uma exceção, o retorno retorna de B através de uma saída excepcional. Seguindo a saída excepcional de B, o fluxo de controle pode seguir para dois lugares: (i) se o bloco protegido de A tiver um tratador de exceção associado, e o tratador de exceções for compatível com a exceção lançada em B, o fluxo segue para este tratador; (ii) operação A também sai pela saída excepcional, e a procura por um tratador continua na operação que chamou A.

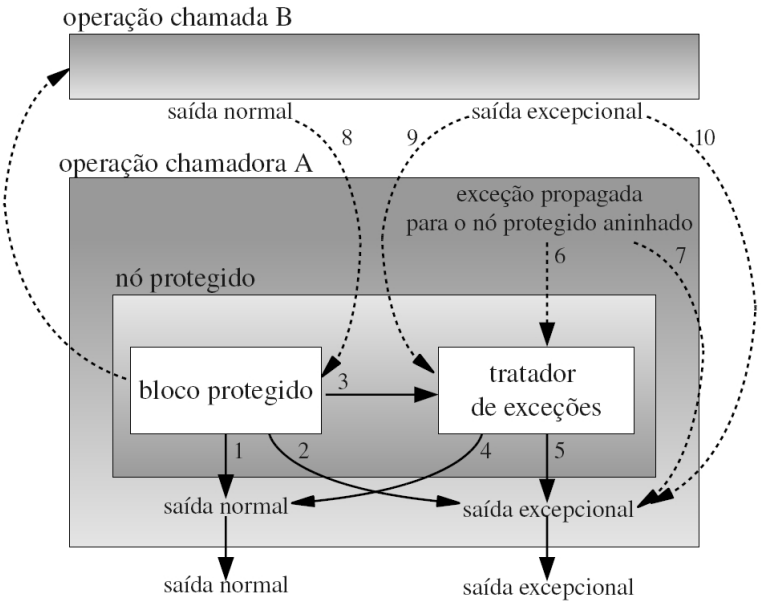


Figura 2.9: Fluxo de controle excepcional do Diagrama de Atividades da UML

2.3.4 Grafo Resumido da Interface

Um *Grafo Resumido da Interface (GRI)* contém um subgrafo para cada operação. Cada subgrafo representa as informações locais sobre a operação que é abstraída para a análise interprocedimental, incluindo as informações sobre os parâmetros formais da operação e dos parâmetros atuais utilizados pelos nós de chamada da operação. O grafo é construído considerando as informações que vinculam os parâmetros formais e atuais para conectar os subgrafos [Harr 94b].

A abordagem proposta usa quatro tipos de nós, originalmente definidos no trabalho prévio [Harr 94b], são eles: (en) *nó de entrada*, (ex) *nó de saída*, (cn) *nó de chamada* e (rn) *nó de retorno*. Além destes nós, para adaptar este grafo às necessidades da abordagem apresentada neste trabalho foram definidos alguns nós complementares para representar as operações com parâmetros de retorno e também nós utilizados para representar a semântica do modelo de exceções da UML, são eles: (rsn) *nó do resultado*, (rvn) *nó de retorno de valor*, (ee) *nó de saída excepcional*, (er) *nó de retorno de exceção*, (thn) *nó de lançamento* e (ctn) *nó de captura de exceções* [Ferr 11].

Em um GRI, os pontos de entrada e saída de uma operação são denotados, respectivamente, pelo nós en_f^m e ex_f^m, ambos criados para cada parâmetro formal f de cada operação m. A chamada e retorno de uma operação são representados pelos nós $cn_x^{m \to n}$ e $rn_x^{m \to n}$, respectivamente, ambos os nós são criados para cada parâmetro x de cada chamada da operação m para a operação n. Os parâmetros de retorno de uma operação são denotados

pelos nós rsn_r^n, criados para cada parâmetro de retorno r da chamada de uma operação n e os nós $rvn_s^{m \to n}$ são criados para cada parâmetro de resultado s da operação m. Uma vez que, cada nó representa uma simples variável, os conjuntos de definições de um nó correspondem a um simples parâmetro formal, atual, de retorno ou de resultado. A propagação de exceções entre operações são denotadas pelos nós ee^n e $er^{m \to n}$, onde estes nós representam o conjunto de todas as definições de exceções que são propagadas entre as operações. Finalmente, a propagação de definições de exceções do nó de lançamento para os nós de captura são denotadas pelos nós $thn_e^{r \to h}$ e ctn_e^h, e ambos são criados para cada parâmetro de exceção e que é lançado a partir do nó de lançamento r no bloco protegido h.

No trabalho prévio foram definidos três tipos de arestas, são eles: i) (*reaching*) *aresta de alcance:* abstrai as informações de fluxo de controle da operação. Por exemplo, a aresta de alcance de um nó de entrada para um nó de chamada indica que uma definição de um parâmetro formal que alcança o início da operação também alcança o nó de chamada onde é utilizado como um parâmetro atual. Este tipo de aresta é estritamente intraprocedimental uma vez que é computada sem incorporar a estrutura de controle da operação chamada pelo nó de chamada; ii) (*binding*) *aresta de ligação:* corresponde às ligação entre um parâmetro formal e atual, ou então, corresponde à ligação de um parâmetro de exceção com o seu respectivo tratador de exceção; iii) (*interreaching*) *aresta de alcance interprocedimental:* liga o nó de chamada ao nó de retorno dentro de uma mesma operação chamadora, abstraindo as informações de controle da operação chamada. Esta aresta indica que uma definição que alcança a chamada da operação pode ser preservada após o retorno da operação. Este tipo de aresta é utilizado para preservar o contexto das operações chamadas durante a análise de fluxo interprocedimental [Harr 94b].

Além das aresta originais, para adaptar este grafo às necessidades da abordagem apresentada neste trabalho foram definidos algumas arestas complementares, são elas: i) (*return*) *aresta de retorno de valor:* liga o nó de retorno de valor de um procedimento ao nó de resultado do procedimento chamador é uma extensão da aresta de ligação criada caso o procedimento possua um parâmetro de retorno; ii) (*propagate*) *aresta de propagação de exceção:* liga o nó de saída excepcional de um procedimento ao nó de retorno excepcional correspondente do procedimento chamador, esta aresta é utilizada na propagação das exceções na pilha de chamada; iii) (*uncaught*) *aresta de exceções não capturadas:* liga um nó de retorno excepcional ao nó de saída excepcional do procedimento, é utilizada para transportar as exceções não capturadas ou relançadas no procedimento até o nó de saída excepcional [Ferr 11].

A Figura 2.10 apresenta os nós e arestas do GRI.

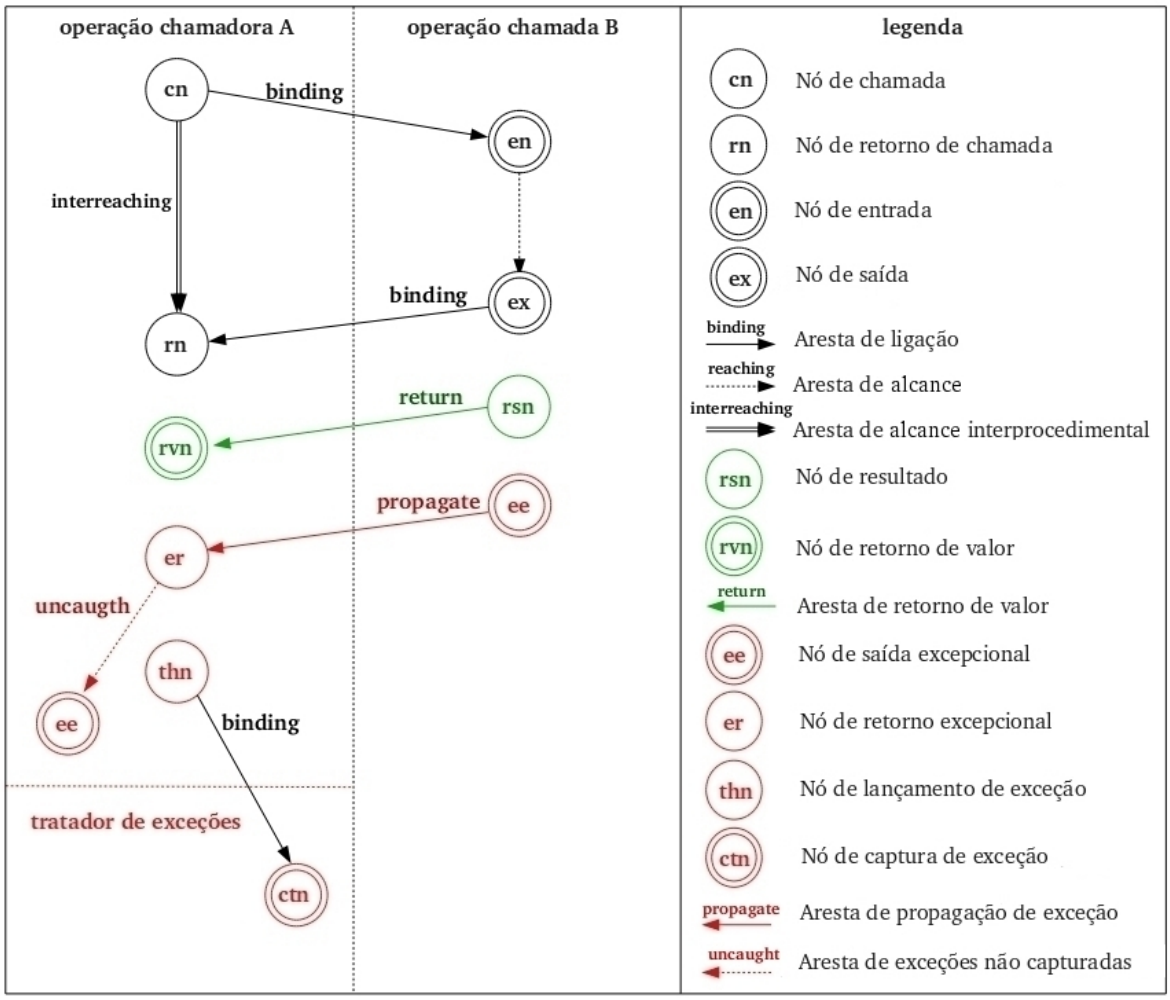


Figura 2.10: Grafo Resumido da Interface

2.4 Trabalhos Relacionados

A obtenção de um GFC a partir do DA já foi proposta por outros autores mesmo na versão 1.5 da UML [Bind 99]. A abordagem proposta neste trabalho estende a técnica proposta por Perez e Martins [Pere 07], que apresenta um método para gerar casos de teste a partir de DAs que representam o comportamento de um componente. O processo utilizado para a geração do GFC é semelhante ao apresentado neste trabalho, porém abordando somente o fluxo de controle. O diferencial é a extensão ao trabalho anterior acrescentando o fluxo de dados. Uma outra diferença entre essas abordagens é o tratamento da ação *CallBehaviorAction*. No trabalho anterior, como a semântica das ações não era considerada, esta ação era tratada como uma chamada interprocedimental. Neste trabalho, de acordo com a semântica das ações da UML, uma ação do tipo *CallBehaviorAction* semanticamente não

representa uma chamada interprocedimental e sim um recurso de modelagem que permite abstrair um comportamento detalhado em outra atividade. A utilização da semântica da ações para determinar o fluxo de dados de um DA também foi apresentado por Ferreira e Martins em outro trabalho [Ferr 09].

A Análise de Fluxo de Dados (AFD) é um processo que analisa os programas com base nos dados e computa as associações e relacionamentos entre esses dados [Aho 86, Dela 07, Amma 08, Rapp 85]. Técnicas de AFD podem ser aplicadas tanto em abordagens baseadas em modelos quanto em abordagens baseadas no código-fonte [Harr 94a, Tabi 08, Ferr 09]. Uma abordagem que utiliza modelos foi proposta por Waheed et al. [Tabi 08], essa abordagem baseia-se no diagrama de estados e na semântica das ações da UML 2.1. Uma AFD é executada para encontrar as associações definição-uso entre as ações definidas no modelo. Esta técnica é aplicada em modelos executáveis que utilizam uma linguagem de ações que permita sua compilação e/ou execução antes da sua implementação. O objetivo final é obter os pares definição-uso entre todas as variáveis e características estruturais envolvidas no fluxo de dados dentro de um estado ou entre os múltiplos estados que podem ser alcançados por um determinado estado. Além de utilizar o diagrama de estados, outra diferença em relação a abordagem apresentada é o fato de não abordar o fluxo de exceções. Este trabalho também utiliza a semântica das ações da UML 2.1 e apresenta um mapeamento das ações UML para a definição e uso das variáveis e características estruturais muito semelhante ao apresentado neste relatório. A diferença é que baseia-se em linguagens de ações que possuem gramáticas para definir as operações entre as ações UML. Essa técnica utiliza a sintaxe abstrata das linguagens baseadas na semântica das ações da UML para gerar o fluxo de dados existente entre as ações.

O fluxo de dados do DA também foi abordado por Störrle [Stor 05, Stor 04], este trabalho define formalmente o fluxo de dados do DA mapeando os elementos do diagrama para os elementos correspondentes numa Rede de Petri. Este trabalho aborda o fluxo de dados existente no DA, porém não utiliza a semântica das ações UML. Desta maneira não fica clara a utilização de uma variável ou característica estrutural definida no modelo, também não é possível identificar uma definição ou uso dessa variável ou característica estrutural.

Similarmente ao tratamento de exceções nas linguagens de programação, um dos principais problemas das linguagens de modelagem é não permitir a especificação explícita dos fluxos excepcionais globais. Estas linguagens exigem que os desenvolvedores entendam a origem de uma exceção e o lugar onde ele é tratada. Cacho et al. [Cach 08] apresentam um modelo de tratamento de exceções que proporciona abstrações para descrever explicitamente uma visão global do fluxo de controle. Este trabalho define um modelo de tratamento de exceções chamado *EFlow*. O principal objetivo é tornar o fluxo de exceções explícito através de canais de fluxo de exceções denominados *Explicit exception channels*

(Eec). Também são definidos tratadores de exceção modularizados denominados *Pluggable Handlers*. Os canais ligam os locais onde as exceções são lançadas até o local onde são tratadas. A implementação desta proposta estende as construções orientadas a aspectos e a análise de fluxo de controle da linguagem de modelagem Motorola WEAVR [Cott 06] com o objetivo de promover uma melhora na robustez e modularização do programa. A linguagem de modelagem Motorola WEAVR é uma abordagem orientada a aspectos para diagrama de estados da UML 2.0 que inclui uma semântica de ações. Os modelos executáveis são refinados até o nível de granularidade que permite a geração completa do código para uma plataforma específica. Neste tipo de modelo o comportamento é definido imperativamente pela incorporação de ações nas transições de um diagrama de estado. As ações são executadas durante a transição de um estado para outro. Uma vantagem desta abordagem é que o modelo executável pode ser testado logo no início do processo de desenvolvimento. Além de modelar o fluxo de controle excepcional, a abordagem proposta verifica se todas as exceções definidas por meio dos lançadores de exceção, são tratadas por um *pluggable handler*. A semântica do *Eec* permite a validação do modelo através de uma análise de fluxo de controle.

Uma similaridade com a abordagem proposta neste trabalho é a possibilidade da visualização dos fluxos excepcionais globais. Através do GRI é possível identificar o caminho do fluxo excepcional através das operações do modelo, de maneira similar ao fluxo excepcional representado por um *Eec*.

A representação de programas que lançam exceções explicitamente é abordada por diversos trabalhos [Sinh 99, Fu 04, Chan 01, Choi 99].

Sinha e Harold [Sinh 99] apresentam uma abordagem que utiliza técnicas de análise como fluxo de controle, fluxo de dados e dependência de controle para programas Java e C++, onde os efeitos da ocorrência de exceções e tratamento de exceções devem ser considerados. Uma vez que estas análises são utilizadas em várias tarefas de engenharia de software existe a necessidade de encontrar representações para programas onde exceções são lançadas e capturadas explicitamente e então para definir algoritmos que usam estas representações para executar estes tipos de análise. Esta abordagem define um grafo de fluxo de controle interprocedimental que representa o fluxo de exceções. Os autores apresentam técnicas para a representação de programas com ocorrência de exceções que são lançadas explicitamente por instruções *throw*. Esta técnica aborda o tratamento de exceções intraprocedimental e interprocedimental. Quando uma operação lança uma exceção e não provê o seu tratamento, uma saída excepcional é criada para representar o fluxo excepcional interprocedimental. Esta abordagem é baseada no código-fonte e utiliza uma inferência de tipos para determinar quais são as exceções que podem ser lançadas por uma determinada instrução do programa. Os autores incluíram o resultado de três estudos empíricos para ilustrar a frequência do uso de exceções em um conjunto de programas

Java, estes estudos indicam uma baixa frequência para os casos onde a inferência de tipos é necessária para identificar as exceções. Porém, o conjunto de programas selecionados não inclui nenhum programa tolerante a falhas, neste contexto a necessidade de inferência é bem maior, pois as exceções são capturadas e relançadas com muita frequência.

Fu et al. [Fu 04] apresenta uma análise do fluxo de exceções, baseada nas definições e usos de variáveis de exceção, para testar um programa Java. Com o objetivo de de testar o programa, esta análise identifica o fluxo entre as definições e usos das exceções que devem ser testados, determinando o tipo de falha e o ponto apropriado do código para instrumentação. Esta técnica incorpora uma análise de fluxo de dados interprocedimental e sensível ao contexto *(points-to analysis)* para identificar a ausência de alcançabilidade do fluxo de dados das referências dos objetos, confirmando desta forma a inviabilidade de alguns fluxos excepcionais. A similaridade com a nossa abordagem é a utilização de uma análise de fluxo de dados que calcula para cada bloco tratador de exceção, todos os comandos que lançam exceções que potencialmente podem ser capturadas pelo tratador. Durante a execução, qualquer operação de lançamento ou captura de exceções são, respectivamente, definições e usos de uma variável. Assim, esta técnica utiliza uma variação da tradicional análise de definições incidentes *(reaching definitions)* para analisar este problema.

Chang et al. [Chan 01] apresenta uma análise estática interprocedimental em programas Java para estimar o fluxo excepcional independente da especificação do programador. A justificativa é que o compilador depende muito da especificação do programador para poder validar as exceções não tratadas. O objetivo é identificar os tratadores de exceções desnecessários e/ou sugerir outros tratadores de exceções mais especializados. O trabalho também apresenta resultados da análise proposta, demostrando que é capaz de detectar eficientemente exceções não tratadas para programas reais.

Choi et. al [Choi 99] sugerem uma nova representação do fluxo de controle interprocedimental chamado de *Factored Control Flow Graph* (FCFG). Esta representação leva em consideração todas as instruções que podem lançar exceções, verificadas ou não verificadas. Esta instruções são denominadas PEIs (Potentially Excepting Instructions); O FCFG é base para uma análise de fluxo de dados utilizada para otimizar o compilador de uma máquina virtual Java. A ideia de representação reduzida do GFD poderia ser utilizada pela nossa abordagem, caso seja necessário reduzir o tamanho do grafo.

Também apresentamos alguns exemplos de ferramentas que capturam e validam o fluxo de exceções propostas por [Robi 03, Filh 05].

Robillard e Murphy [Robi 03, Robi 99] apresentam *Jex*, uma ferramenta de análise estática desenvolvida para capturar as informações do fluxo excepcional de sistemas Java. A ferramenta extrai informações sobre a estrutura de exceções em programas Java, fornecendo uma visão dos tipos reais de exceção que podem surgir em pontos diferentes do

programa e também fornece informações sobre os tratadores de exceção que estão presentes. Além disso, verifica se as exceções são capturadas por tratadores muito genéricos. Uma diferença em relação a abordagem proposta é o fato da ferramenta não realizar uma inferência de tipos para determinar os possíveis tipos de exceções podem ser lançados por uma instrução *throw*. Esta informação pode ser incompleta já que qualquer subtipo do tipo estático também pode ser lançado.

Castor et al. [Filh 05] apresentam um framework de análise do fluxo de controle excepcional no nível arquitetural. O framework *Aereal* é baseado nas linguagens e ferramentas de apoio à descrição e análise do fluxo excepcional em arquiteturas de software. Este framework utiliza *Alloy* [Jack 02], uma linguagem relacional de primeira ordem e *ACME* [Garl 00], uma linguagem de intercâmbio para a descrição de arquiteturas e define como as exceções fluem entre os elementos arquiteturais (componentes e conectores) para cada estilo arquitetural utilizado na descrição da arquitetura. Esse trabalho é semelhante ao proposto em relação ao objetivo de verificação (por exemplo, a existência de exceções não capturadas e tratadores de exceção não utilizados), mas difere na forma como os fluxos de exceção de controle são representados. Outra diferença importante é o modelo utilizado, enquanto na abordagem proposta é utilizado o DA, o Aereal requer a definição exaustiva de complicados formalismos para representar somente os fluxos entre os dois elementos arquiteturais interligados.

Tabela 2.3: *Quadro Comparativos dos Trabalhos*

Abordagem	**Fluxo de Controle**	**Fluxo de Dados**	**Fluxo de Exceções**	**Semântica de Ações**	**Modelo/Fonte**
Método proposto	Sim	Sim	Sim	Sim	Diagrama de Atividades
Perez e Martins	Sim	Não	Sim	Não	Diagrama de Atividades
Waheed et al.	Sim	Sim	Não	Sim	Diagrama de Estados
Cacho et al.	Sim	Não	Sim	Sim	Diagrama de Estados
Sinha e Harold	Sim	Sim	Sim	Não	Código Fonte
Fu et al.	Sim	Sim	Sim	Não	Código Fonte
Störrle	Sim	Sim	Não	Não	Diagrama de Atividades
Choi et al.	Sim	Sim	Não	Não	Código Fonte
Robillard e Murphy	Sim	Sim	Sim	Não	Código Fonte
Castor et al.	Não	Não	Sim	Não	Descrição da Arquitetura

Capítulo 3

A Abordagem Proposta

Este capítulo apresenta a descrição da abordagem utilizada para validar o fluxo excepcional a partir dos mecanismos de tratamento de exceções modelados no diagrama de atividades. O ponto de partida é um modelo UML composto por um conjunto de diagramas de classes e de atividades. A abordagem proposta realiza uma série de transformações com o objetivo de gerar um grafo de fluxo de controle interprocedimental, que representa o fluxo de controle normal e excepcional de um componente de software. Neste processo também utilizamos a semântica das ações da UML, utilizada para a identificar as ações responsáveis pela criação, definição, lançamento e captura dos objetos de exceção.

A seguir apresentamos a sequência de passos da abordagem proposta:

1. Modelagem dos Diagramas de Atividades

2. Transformação dos Modelos

   (i) Geração do Grafo de Atividades (GA)

   (ii) Geração do Grafo de Fluxo de Controle (GFC)

   (iii) Obtenção do Fluxo de Exceções Intraprocedimentais

   (iv) Geração do Grafo de Fluxo de Controle Interprocedimental (GFCI)

   (v) Geração do Grafo Resumido da Interface (GRI)

   (vi) Obtenção do Fluxo de Exceções Interprocedimentais

O capítulo está organizado da seguinte maneira, na seção 3.1 apresentamos os detalhes sobre a modelagem dos diagramas, na seção 3.2 apresentamos os detalhes sobre a transformação dos modelos.

3.1 Modelagem do Diagrama de Atividades

O principal diagrama do processo é o Diagrama de Atividades (DA). Este modelo deve ser elaborado por uma ferramenta de modelagem que atenda os seguintes requisitos:

1. Conformidade com a especificação da UML 2.0 ou versão superior;
2. Suporte a semântica das ações UML;
3. Funcionalidade de exportação do modelo para o formato XMI (XML Metadata Interchange) [OMG 07b]

Os modelos apresentados no Capítulo 7 foram modelados pela ferramenta MagicDraw[1].

A modelagem do DA fica restrita ao subconjunto de atividades e ações apresentados na seção 2.2, pois a abordagem proposta não abrange todas as possibilidades de modelagem oferecidas na especificação da UML. Por exemplo, a modelagem de fluxos concorrentes não foi considerada com o intuito de reduzir a complexidade da abordagem proposta.

Outro fator importante na modelagem do DA é a utilização da semântica das ações da UML, fundamental para a identificação das ações de escrita e leitura das informações de fluxo de dados. A Tabela 2.2 apresenta um resumo da classificação das ações da UML quanto à escrita ou leitura de dados. Além deste papel, a semântica das ações da UML também é utilizada para identificar as ações responsáveis pelo lançamento dos objetos de exceção.

A modelagem do comportamento normal e excepcional de um componente de software não se resume apenas na construção de um conjunto de diagramas de atividades. É necessário especificar as classes utilizadas na modelagem do comportamento normal e também a hierarquia das exceções que serão lançadas e capturadas pelos tratadores de exceção. A especificação das classes e da hierarquia de exceções são representadas por um conjunto de diagramas de classes.

Para dar início ao processo de validação é necessário efetuar a exportação completa do modelo UML. Todos os diagramas contidos no modelo são exportados para um arquivo no formato XMI (XML Metadata Interchange) [OMG 07b]. O arquivo XMI do modelo é o artefato de entrada para a ferramenta de validação.

3.2 Transformação dos Modelos

A abordagem proposta é uma composição de várias abordagens anteriores que foram adaptadas ou estendidas para a utilização do modelo UML ao invés do código fonte.

[1]MagicDraw: http://www.magicdraw.com

A representação do fluxo de controle intraprocedimental através do GFC e do fluxo de controle interprocedimental através do GFCI, foi baseada na abordagem proposta por Sinha e Harrold [Sinh 99]. A utilização do GRI para efetuar a análise de fluxo de dados interprocedimental foi baseada no trabalho apresentado por Harrold e Soffa [Harr 94b]. A abordagem proposta estende o GRI, inserindo novos nós e arestas para proporcionar a representação do fluxo excepcional interprocedimental e permitir uma análise de fluxo de dados para propagação das exceções através da pilha de chamadas.

O primeiro passo efetuado pela ferramenta é a leitura do arquivo XMI e a instanciação do modelo UML. Na sequência o modelo é examinado para a geração do Grafo de Atividades (GA). Este é um grafo direcionado que representa a sequência de execução das atividades e ações de um procedimento modelado no DA. O GA é uma representação intermediária utilizada apenas durante o processo de construção do GFC. Durante a geração do GA, as informações de fluxo de dados do modelo são examinadas. As informações são obtidas através das definições e usos das referências dos objetos de exceção. A semântica das ações UML é utilizada para identificar as definições e os usos das variáveis, a semântica das ações também possibilita a identificação dos pontos onde as exceções são lançadas e capturadas. As definições e usos identificadas são associadas aos nós do GA. Maiores informações sobre a construção do GA são apresentadas na seção 4.1.

No segundo passo, o GA gerado no passo anterior é transformado em um Grafo de Fluxo de Controle (GFC). Cada nó do GFC representa um bloco básico de atividades do GA, o qual não possui nenhum desvio de controle no seu interior, exceto ao final do bloco. Assim, um conjunto de nós do GA podem ser agrupados em um único nó do GFC. Depois de gerado o GFC a próxima etapa é a criação do fluxo de exceções intraprocedimental. Para isso é necessária a identificação dos tipos de exceção que podem ser lançados por um determinado nó. No caso de uma exceção que é tratada dentro do próprio procedimento, uma aresta de exceção irá ligar este nó com o nó correspondente ao tratador de exceção compatível. No caso da exceção não ser tratada, uma aresta de exceção irá ligar este nó com o nó correspondente à saída excepcional do procedimento. Para determinar quais são os tipos de exceção que podem ser lançados por um nó, utilizamos uma análise de fluxo de dados sobre o GFC. Esta análise consiste no cômputo das definições incidentes *(reaching definitions)*. O conjunto das definições incidentes é utilizado para determinar quais são os possíveis tipos de exceção que um determinado nó é capaz de lançar. Na seção 4.2 são mostrados os detalhes da geração do GFC e na seção 4.3 são mostrados os detalhes para a obtenção do fluxo excepcional intraprocedimental.

O terceiro passo é a criação do Grafo de Fluxo de Controle Interprocedimental (GFCI). Cada operação existente no modelo possui o seu GFC correspondente. O GFCI consiste no grafo resultante da interligação de todos os GFCs. Cada nó de chamada de uma operação chamadora é conectado ao nó de entrada da operação chamada por uma aresta de chamada

e cada nó de saída da operação chamada é conectado ao nó de retorno correspondente por uma aresta de retorno. Para representar o fluxo de controle interprocedimental, o GFCI contém arestas de retorno de exceções. Uma aresta de retorno de exceção é uma aresta interprocedimental que conecta um nó de saída excepcional da operação chamada com o nó de captura de exceção ou com outro nó de saída excepcional da operação chamadora. Se uma operação propaga uma exceção que é capturada pelo chamador daquela exceção, o nó de saída excepcional para aquele tipo de exceção é conectado com o nó apropriado na operação chamadora por uma aresta de exceção. Uma operação pode propagar uma exceção que não é tratada na operação chamadora imediata, mas é tratada em uma operação que fica mais acima na cadeia de chamadas. A descrição completa do processo de obtenção do GFCI é apresentada na seção 5.1.

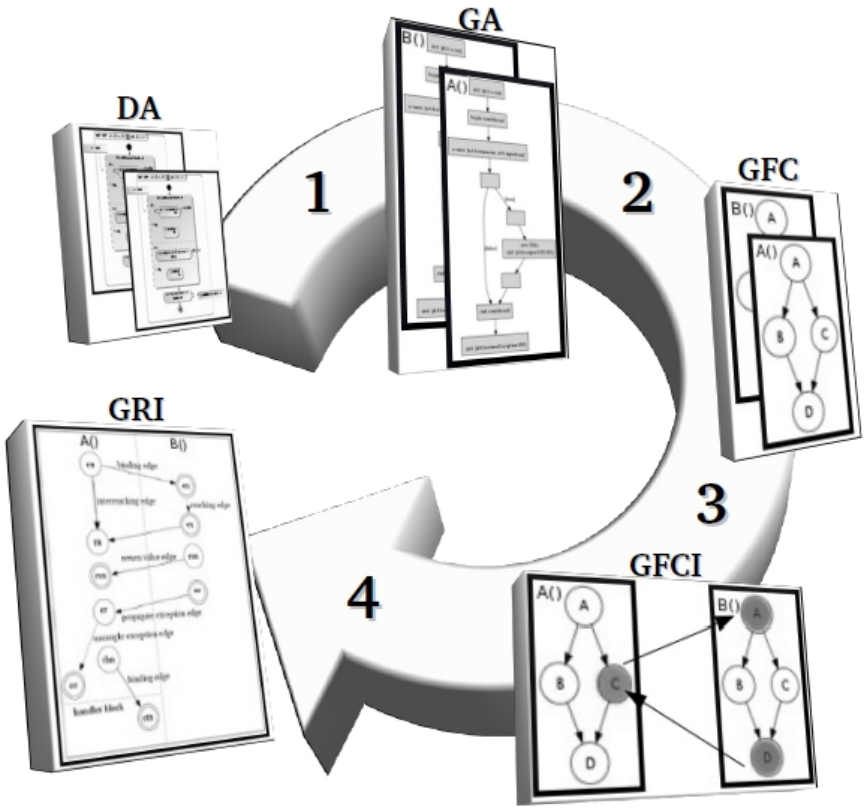


Figura 3.1: Processo de Transformação dos Modelos

O quarto passo é a obtenção do fluxo excepcional interprocedimental, para isso é necessário executar uma análise de fluxo de dados interprocedimental. A análise utilizada é o cômputo das definições incidentes interprocedimentais *(interprocedural reaching definitions)*. Com o objetivo de simplificar a execução da análise de fluxo de dados e também

preservar o contexto das operações chamadas, um novo grafo é gerado a partir da interface das operações do modelo, denominado de Grafo Resumido da Interface (GRI). Um GRI contém um subgrafo para cada operação. Cada subgrafo representa as informações locais sobre a operação que é abstraída para a análise interprocedimental, incluindo as informações sobre os parâmetros formais da operação e dos parâmetros atuais utilizados pelos nós de chamada da operação. O grafo é construído considerando as informações que vinculam os parâmetros formais e atuais para conectar os subgrafos. Após a execução da análise de fluxo de dados no GRI, os conjuntos de definições incidentes computados são utilizados para a criação das arestas de exceções ligando os nós que lançam a exceção com os nós responsáveis pelo seu tratamento. Caso a exceção não seja tratada pela operação a aresta de exceção liga o nó que lança a exceção com o nó de saída excepcional da operação. Os detalhes da geração do GRI são mostrados na seção 5.2 e os detalhes sobre a obtenção do fluxo excepcional interprocedimental são mostrados na seção 5.3.

A figura 3.1 resume o processo de transformação dos modelos:

1. Transformação do modelo UML (no formato XMI), convertendo o conjunto de DAs em um conjunto de GAs, um GA para cada procedimento modelado;

2. Transformação do GA em um GFC, através do agrupamento dos nós do GA em um único nó do GFC;

3. Geração do GFCI a partir da interligação dos GFCs gerados no passo anterior;

4. Geração do GRI a partir da abstração dos nós da interface do GFCI.

Para apoiar o processo de transformação dos modelos e efetuar a validação do fluxo excepcional foi implementada uma ferramenta, denominada *ADEX* (Activity Diagram EXceptional flow analyzer). Além de implementar os algoritmos para as transformações dos modelos, a ferramenta também provê recursos para a visualização dos grafos resultantes. O capítulo 6 apresenta maiores detalhes sobre a ferramenta.

Capítulo 4

Grafo de Fluxo de Controle

Neste capítulo apresentamos o processo de obtenção do Grafo de Fluxo de Controle de uma operação modelada no DA. Este processo pode ser dividido em três passos: i) na seção 4.1 apresentamos a obtenção do Grafo de atividades, um grafo intermediário que mapeia diretamente cada atividade ou ação do diagrama de atividades; ii) na seção 4.2 apresentamos a obtenção do Grafo de Fluxo de Controle a partir do grafo de atividades, neste passo os nós do grafo intermediário são agrupados em blocos básicos de atividades quando não existe desvio de controle entre eles; iii) na seção 4.3 apresentamos a obtenção do fluxo de exceções intraprocedimental, demonstrando quais os tipos de exceções são capturados ou propagados pela operação representada pelo Grafo de Fluxo de Controle.

4.1 Construção do Grafo de Atividades

O Grafo de Atividades (GA) é um grafo direcionado que representa a sequência de execução das atividades e ações de um procedimento modelado no DA. Este grafo é uma representação intermediária utilizada apenas durante o processo de construção do GFC. Cada nó do GA pode representar uma atividade básica (Ex: Nó de Inicialização, Nó Final de Atividade), um nó de controle (Ex: Nó de Decisão, Nó Mesclar) ou nós auxiliares que são criados durante a transformação das atividades estruturadas. As arestas do GA representam o fluxo de controle entre os nós, e são geradas pelo mapeamento direto de arestas de controle do DA ou pela conversão de uma ou mais arestas de fluxo de dados do DA em uma aresta de controle do GA. As principais justificativas sobre a utilização do GA são as seguintes: (i) o DA admite decomposição, e cada atividade pode ser decomposta em outro DA; (ii) os DAs não são interconectados, e possuem atividades desconexas, como por exemplo, o nó condicional, que tem seções que não estão interligadas no DA.

Formalmente um nó do GA pode ser definido como:

$$ActivityNode \in \{Action, BasicActivity, ControlNode, AuxiliaryNode\}$$

onde:

$Action$	ação UML
$BasicActivity$	atividade básica (*InitialNode*, *ActivityFinalNode*)
$ControlNode$	nó de controle (*DecisionNode*, *MergeNode*)
$AuxiliaryNode$	nó auxiliar gerado na transformação de uma atividade estruturada

Já uma aresta do GA pode ser definida como:

$$ActivityEdge \in \{NormalEdge, AuxiliaryEdge\}$$

onde:

$NormalEdge$	aresta de fluxo de controle entre nós $ActivityNode$
$AuxiliaryEdge$	aresta de fluxo de controle entre nós $AuxiliaryNode$, gerada na transformação de uma atividade estruturada

O Grafo de Atividades é definido formalmente pela seguinte tupla:

$$ActivityGraph = \langle Nodes, iN, fN, Edges \rangle$$

onde:

$Nodes$	$\subseteq ActivityNode$, conjunto de todos os nós internos do GA
iN, fN	nó inicial e nó final do GA
$Edges$	$\subseteq ActivityEdge$, conjunto de todos as arestas do GA

O processo de geração do GA é dividido em três etapas; cada DA contido no modelo passa por essas três etapas até dar origem aos grafos resultantes. No final deste processo, cada procedimento especificado no modelo terá gerado o seu respectivo GA. Lembrando que para modelar um procedimento podem ser necessários vários diagramas de atividade.

A primeira etapa consiste em determinar a ordem de execução das atividades e ações contidas no DA. O Algoritmo 1 é utilizado para realizar esta ordenação.

Algoritmo 1 OrderingActivityNodes(n, e, p)

Entrada: Nó do DA n, Aresta do DA e, Nó predecessor p.
Saída: Grafo ordenado g.

```
1: if n.visited = false then {Nó ainda não visitado}
2:     outgoings ⇐ n.outgoingsEdges {Conjunto de arestas de saída}
3:     incomings ⇐ n.incomingsEdges {Conjunto de arestas de entrada}
4:     if n is an Action then {Nó é uma Ação}
5:         if incomings.size > 1 then {Existe mais de uma aresta de entrada}
6:             incomings ⇐ incomings \ e
7:             if incomings.size = 0 then {Conjunto de arestas de entrada vazio}
8:                 n.visited ⇐ true
9:                 g.nodes ⇐ g.nodes ⋃ n
10:                for all i in n.incomingsEdges do {Processa arestas de entrada}
11:                    source ⇐ i.source {Nó origem}
12:                    source.edges ⇐ source.edges ⋃ Edge(source, n)
13:                for all o in outgoings do {Processa arestas de saída}
14:                    OrderingActivityNodes(o.target, o, n)
15:        else if outgoings.size > 1 then {Existe mais de uma aresta de saída}
16:            g.nodes ⇐ g.nodes ⋃ n
17:            p.edges ⇐ p.edges ⋃ Edge(p, n)
18:            for all o in outgoings do {Processa arestas de saída}
19:                OrderingActivityNodes(o.target, o, n)
20:        else
21:            n.visited ⇐ true{Marca nó como visitado}
22:            p.edges ⇐ p.edges ⋃ Edge(p, n)
23:            for all o in outgoings do {Processa arestas de saída}
24:                OrderingActivityNodes(o.target, o, n)
25:            if n is an Action then {Nó é uma Ação}
26:                for all output in n.outputPins do {Processa pinos de saída}
27:                    OrderingActivityNodes(output, null, n)
28:    else if n is an InputPin then {Nó é um pino de entrada}
29:        if n.owner is an Action then {Pino pertence a uma ação}
30:            if n.owner.visited = false then {Nó da ação ainda não visitada}
31:                n.owner.visited ⇐ true
32:                inputs ⇐ n.inputPins \ n {Conjunto de pinos de entrada}
33:            else
34:                inputs ⇐ inputs \ n
35:            if inputs.size = 0 then {Conjunto de pinos de entrada vazio}
36:                g.nodes ⇐ g.nodes ⋃ n.owner
37:                p.edges ⇐ p.edges ⋃ Edge(p, n.owner)
38:                OrderingActivityNodes(n.owner, null, n)
```

O grafo de atividades resultante é um grafo direcionado, onde cada nó representa uma atividade ou ação mapeada do DA e cada aresta representa o fluxo de controle entre os nós. Nesta etapa, as arestas de fluxo de controle e de fluxo de dados do DA são utilizadas para identificar as dependências de controle e as dependências de dados entre as ações. Para começar a executar, uma ação deve saber quando começar e quais são as suas entradas. Estas condições representam as dependências de controle e dados, respectivamente. Uma ação começa executar quando todas as suas entradas de controle e dados estiverem disponíveis. A finalidade do grafo é representar a possível ordem de execução entre as atividades e ações do DA, de acordo com as dependências de controle explícitas (arestas de fluxo de controle) e as dependências de controle implícitas pela dependência de dados (arestas de fluxo de dados).

Na segunda etapa, cada DA existente no modelo é visitado novamente; os nós internos são visitados de acordo com a ordenação gerada na primeira etapa. Essa segunda etapa permite criar a representação das atividades estruturadas do DA, quais sejam os nós condicionais (*ConditionalNode*) e os laços (*LoopNode*). Para cada DA existente no modelo será gerado um subgrafo de atividades correspondente. Para cada atividade estruturada contida em um DA, ou aninhada dentro de outra atividade estruturada, também será um gerado subgrafo de atividades. O Algoritmo 2 é responsável pela geração dos subgrafos de cada DA e também de suas atividades estruturadas aninhadas.

No caso das atividades estruturadas Nó Condicional (*ConditionalNode*) e Laço (*LoopNode*), nós e arestas auxiliares são inseridos para ligar as seções da atividade estruturada, mostrando explicitamente a representação do fluxo de controle que antes era implícita. A Figura 4.1 mostra os nós e arestas auxiliares utilizados para representar o fluxo de controle dessas atividades.

A Figura 4.1a apresenta um Nó Condicional *ConditionalNode* e o seu respectivo fluxo de controle gerado durante a transformação dos modelos. Neste subgrafo, os nós *begin* e *end* representam o nó inicial e o nó final e os demais nós representam uma cláusula condicional.Uma cláusula é formada por um par de seções: *test* e *body*. Os nós *test* e *body* representam respectivamente as seções *test* e *body* de uma cláusula condicional. Quando a condição da ação *test* é verdadeira o fluxo de controle segue pela aresta (*test*→*body*). A aresta (*test*→*end*) representa o fluxo de controle quando a condição testada é falsa, e por fim a aresta (*body*→*end*) representa o fim da execução de um nó condicional, quando o fluxo de controle sai da ação *body* e segue para o nó *end*. Um nó condicional pode conter diversas cláusulas, as seções *test* são examinadas até que uma condição seja satisfeita, quando isto ocorre o fluxo segue para a seção *body* correspondente. A execução de uma cláusula implica na execução do nó condicional, mesmo se houverem mais cláusulas neste nó condicional, as demais não serão examinadas.

Algoritmo 2 ProcessAD(*activity*)

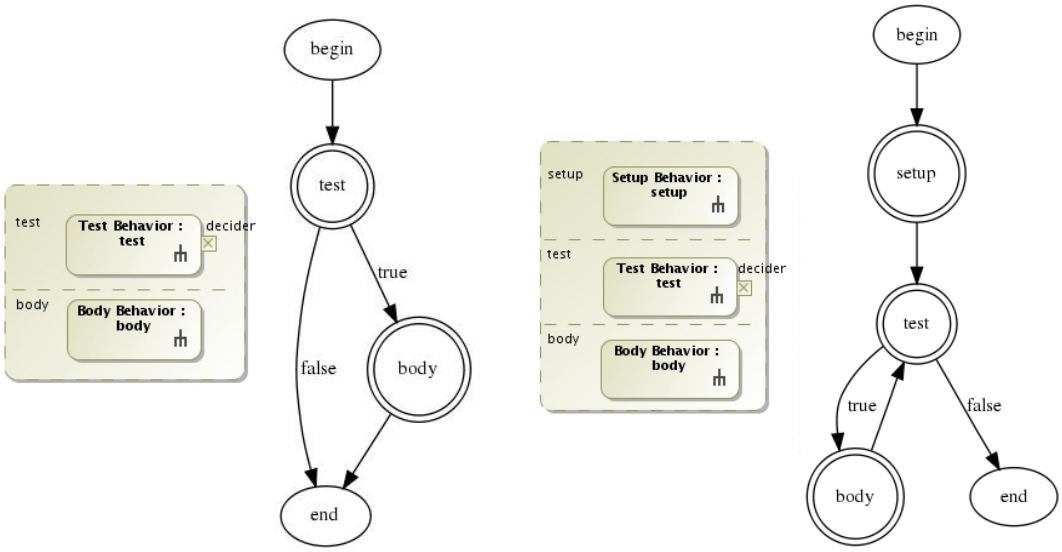
Entrada: Diagrama de Atividades *activity*.
Saída: Grafo de Atividades *graph*.

```
1: orderedGraph ⇐ OrderingActivityNodes(activity.initialNode, null, null)
2: graph ⇐ ActivityGraph(activity)
3: nodes ⇐ activity.nodes
4: structuredNodes ⇐ activity.structuredNodes
5: for all n in nodes do {Processa cada nó da atividade}
6:     if n is an Action then {Nó é uma Ação}
7:         graph.nodes ⇐ graph.nodes ⋃ ActivityGraphNode(n)
8:         dataFlowMapping(n) {Efetua o mapeamento do fluxo de dados}
9: for all s in structuredNodes do {Processa cada atividade estruturada aninhada}
10:    graph.nodes ⇐ graph.nodes ⋃ ActivityGraphNode(s)
11:    if s is a uml2.ConditionalNode then {Nó condicional}
12:        {Cria os nós auxiliares para cada seção do nó condicional}
13:    else if s is an uml2.LoopNode then {Laço}
14:        {Cria os nós auxiliares para cada seção do laço}
15:    else
16:        ProcessAD(s)
17: for all n in orderedGraph.nodes do {Processa cada nó do grafo ordenado}
18:    for all e in n.edges do {Processa cada aresta do nó}
19:        graph.nodes ⇐ graph.nodes ⋃ ActivityGraphNode(n)
```

A representação gráfica de um Laço *LoopNode* e o seu respectivo fluxo de controle são exibidos na Figura 4.1b. Da mesma forma que na atividade anterior os nós *begin* e *end* representam o nó inicial e o nó final. Nesta atividade existe um nó para cada seção *setup*, *test* e *body*. O laço é representado pelo par de arestas (*test*→*body*) e (*body*→*test*). A aresta (*test*→*end*) representa a condição de saída do laço quando a ação *test* retornar um valor falso. O subgrafo apresentado na Figura 4.1b, representa um laço com a propriedade *isTestedFirst* setada com valor true. Esta propriedade determina se o teste da condição de saída do laço será executado antes da execução do corpo ou não, o valor padrão é falso.

Cada seção de uma atividade estruturada é representada no GA por uma ação do tipo *CallBehaviorAction* o que implica a inclusão de outro diagrama de atividades. Na Figura 4.1 as seções são representadas por um círculo duplo, para indicar que estes nós serão substituídos posteriormente pelo subgrafo da atividade vinculada.

A obtenção das informações de fluxo de dados está implícita no processo de geração do GA (Algoritmo 2, procedimento: $dataFlowMapping(n)$). Durante este processo cada ação do DA é avaliada e todas as informações referentes ao fluxo de dados são capturadas.



(a) Nó Condicional (ConditionalNode) (b) Laço (LoopNode)

Figura 4.1: Fluxo de Controle das Atividades Estruturadas

As definições e usos das variáveis capturadas são adicionadas ao nó de atividade correspondente. No passo seguinte, quando o GA é transformado em um GFC, cada nó do GFC representa um agrupamento de nós do GA. Todas as definições e usos desse conjunto de nós são transferidas para o nó do GFC.

Para representarmos uma *definição* formalmente utilizamos uma quádrupla:

$$Def = \langle d_i : reference : type : object \rangle$$

onde: d_i identificador da i-ésima ocorrência de uma definição
$reference$ nome da variável ou pino de saída que sofreu a definição
$type$ tipo de dado da referência
$object$ tipo de dado do objeto referenciado

A representação utilizada para um *c-uso* ou *p-uso* é mais simples, ambos podem ser representada pela tripla:

$$Use = \langle u_i : reference : type \rangle$$

onde: u_i identificador da i-ésima ocorrência de uso
$reference$ nome da variável ou pino de entrada que foi utilizado
$type$ tipo de dado da referência

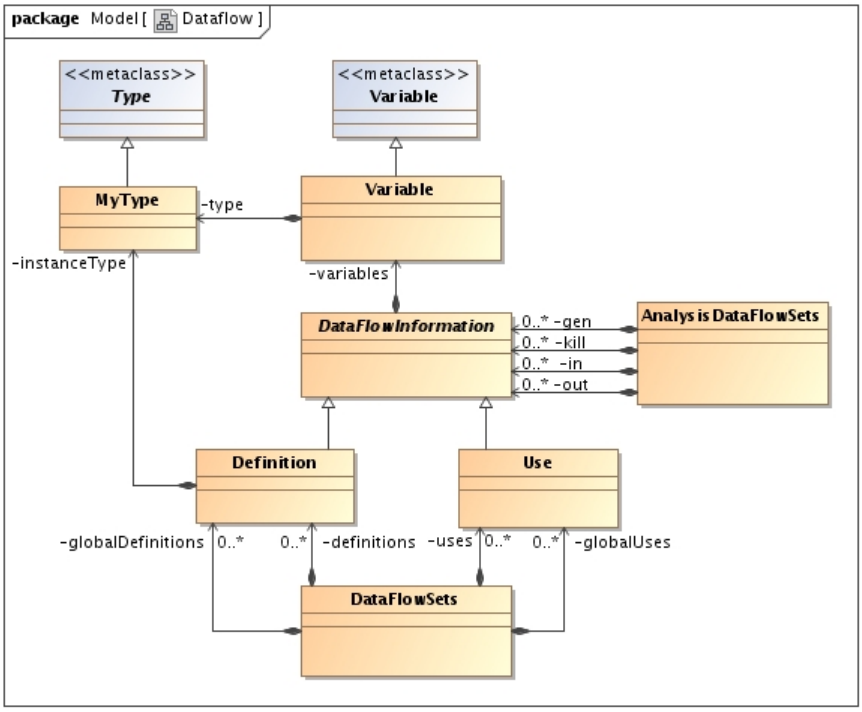


Figura 4.2: Modelagem das Informações de Fluxo de Dados

Na Figura 4.2 são apresentadas as classes responsáveis pela modelagem das informações de fluxo de dados: *MyType* estende a classe *Type* do metamodelo da UML para definir um tipo de dado de uma variável ou característica estrutural; *Variable* estende a classe *Variable* do metamodelo da UML para definir uma variável; *DataFlowInformation* é uma classe abstrata que define uma informação de fluxo de dados; *Definition* e *Use* especializam a classe *DataflowInformation* para representar, respectivamente, um definição e uso de uma variável; A classe *DataFlowInformation* possui um relacionamento com a classe *Variable* para identificar a variável associada com a definição ou uso. A classe *Definition* possui um relacionamento com a classe *MyType*, este relacionamento define o atributo *instanceType*, que determina o tipo de dado da instância do objeto referenciado pela variável. A classe *AnalysisDataFlowSets* possui quatro relacionamentos com a classe *DataflowInformation* para definir quatro conjuntos de informações de fluxo de dados (definições ou usos) que são utilizados pelos nós durante uma análise de fluxo de dados: *(gen)* conjunto de informações geradas, *(kill)* conjunto de informações mortas, *(in)* conjunto de informações de entrada e *(out)* conjunto de informações de saída.

Na Figura 4.3 são apresentadas as classes responsáveis pela modelagem do GA. As classes abstratas *Graph*, *GraphNode* e *GraphEdge*, modelam respectivamente, um grafo, seus nós e suas arestas. A classe *Graph* possui um relacionamento bidirecional com a

classe *GraphNode* definindo o atributo *nodes* que representa o conjunto de nós do grafo. Do lado da classe *GraphNode* este relacionamento define o atributo *owner* que identifica o grafo que o nó está contido. A classe *GraphNode* possui um relacionamento bidirecional com a classe *GraphEdge* definindo o atributo *edges* que representa o conjunto de arestas que partem do nó. Do lado da classe *GraphEdge* este relacionamento define o atributo *owner* que identifica o nó que a aresta está contida. A classe *GraphEdge* ainda possui outras duas referências para a classe *GraphNode* que definem os atributos *source* e *target*, que identificam respectivamente, os nós de origem e destino da aresta. Neste pacote também é definida a classe *Model*, que serve como um contêiner para o modelo. Esta classe possui um relacionamento bidirecional com a classe *Graph* definindo o atributo *graphs* que representa o conjunto de grafos contidos no modelo. Do lado da classe *Graph*, este relacionamento define o atributo *owner* que indica o modelo que o grafo está contido.

A classe *ActivityGraph* especializa a classe abstrata *Graph* definindo o próprio GA. A classe *ActivityGraphNode* especializa a classe abstrata *GraphNode* definindo um nó do GA. Esta classe também possui uma referência para a classe *ActivityNode* do metamodelo da UML, que representa um nó do diagrama de atividades. Para cada nó do DA existe um nó correspondente no GA, mas o inverso não é verdadeiro, pois alguns nós estruturados do DA acabam gerando mais de um nó no GA. Por exemplo, no mapeamento das atividades estruturadas *LoopNode* e *ConditionalNode* para o GA, o nós auxiliares inseridos para representar o início e fim do fluxo de controle dos grafos resultantes não possuem atividade correspondente no DA (Figura 4.1). A classe *AuxiliarNode* especializa a classe *ActivityNode* definindo um nó auxiliar. As classes *InitialAuxiliarNode* e *FinalAuxiliarNode* são subclasses de *AuxiliarNode*. A classe *DataFlowSets* possui um relacionamento com a classe *ActivityGraphNode* para especificar os conjuntos de usos e definições das variáveis pelos nós.

Também são apresentadas as classes *Scope* e *Handler*, a primeira é utilizada para representar uma região protegida do DA (Por exemplo: Atividade Estruturada, Laço ou Nó Condicional) e a segunda é utilizada para representar um tratador de exceções. Como uma região protegida pode estar aninhada em outras, um determinado nó do GA pode estar contido em várias regiões protegidas. O relacionamento entre a classe *ActivityGraphNode* e a classe *Scope* define o atributo *scopes*, que representa uma pilha de escopos (regiões protegidas) na qual o nó do GA está contido. A classe *Scope* por sua vez possui um relacionamento com a classe *Handler* definindo o atributo *handlers* que identifica o conjunto de tratadores de exceção que estão associados com a região protegida. As arestas do GA são representadas pela classe *ActivityGraphEdge* que especializa a classe abstrata *GraphEdge*.

A terceira etapa do processo consiste na substituição dos nós do GA correspondentes às atividades estruturadas e ações *CallBehaviorAction* pelos seus respectivos subgrafos. Cada grafo gerado na etapa anterior é percorrido. Quando um nó correspondente a uma

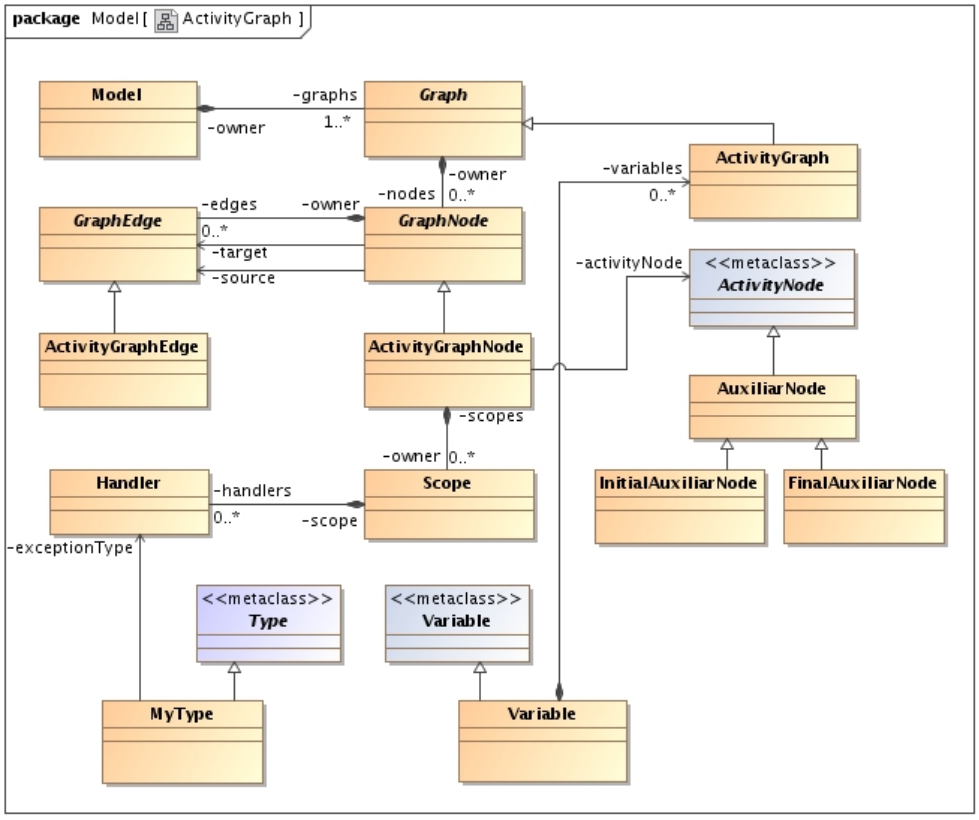


Figura 4.3: Modelagem do Grafo de Atividades (GA)

ação *CallBehaviorAction* é encontrado, este nó é substituído pelo subgrafo do comportamento vinculado à ação. O nó é retirado do grafo original, as arestas de entrada do nó substituído são direcionadas para o nó inicial do subgrafo, e as arestas de saída do nó substituído são transferidas para o nó final do subgrafo. Na Figura 4.1 é possível observar que cada seção de uma atividade estruturada, representada por um círculo duplo, pode conter uma ação *CallBehaviorAction*. Cada um destes nós é removido e substituído pelo subgrafo correspondente. Para os nós correspondentes às ações estruturadas o procedimento é similar, ou seja, os nós também são substituídos pelos respectivos subgrafos. Este processo é feito recursivamente até que não existam mais nós para serem substituídos. No final deste processo cada operação (procedimento) existente no modelo terá o seu GA completo.

Para demonstrar o processo de construção do GA vamos apresentar a especificação da operação *service()*, composta pelo conjunto de atividades mostrados na Figura 4.4. Este exemplo modela o comportamento de um componente de software abstrato que possui um único serviço. A Figura 4.4(a) apresenta a atividade principal desse serviço, que

recebe como entrada dois inteiros, representados pelos parâmetros *a* e *b*. Os parâmetros são repassados para os nós internos através de arestas de fluxo de dados que ligam o parâmetro até o pino de entrada do nó destino. A atividade *service()* é constituída por uma atividade estruturada (*s1*). Esta atividade é um nó protegido que permite a utilização de tratadores de exceção. Neste caso, existem dois tratadores de exceção que ligam o nó protegido com as ações responsáveis pelo tratamento: *h1* e *h2*. As ações que representam o corpo do tratador de exceções são do tipo *CallBehaviorAction*, ou seja, abstraem um comportamento especificado em outra atividade. As atividades que especificam o tratamento das exceções são mostradas nas Figuras 4.4(e) e 4.4(f).

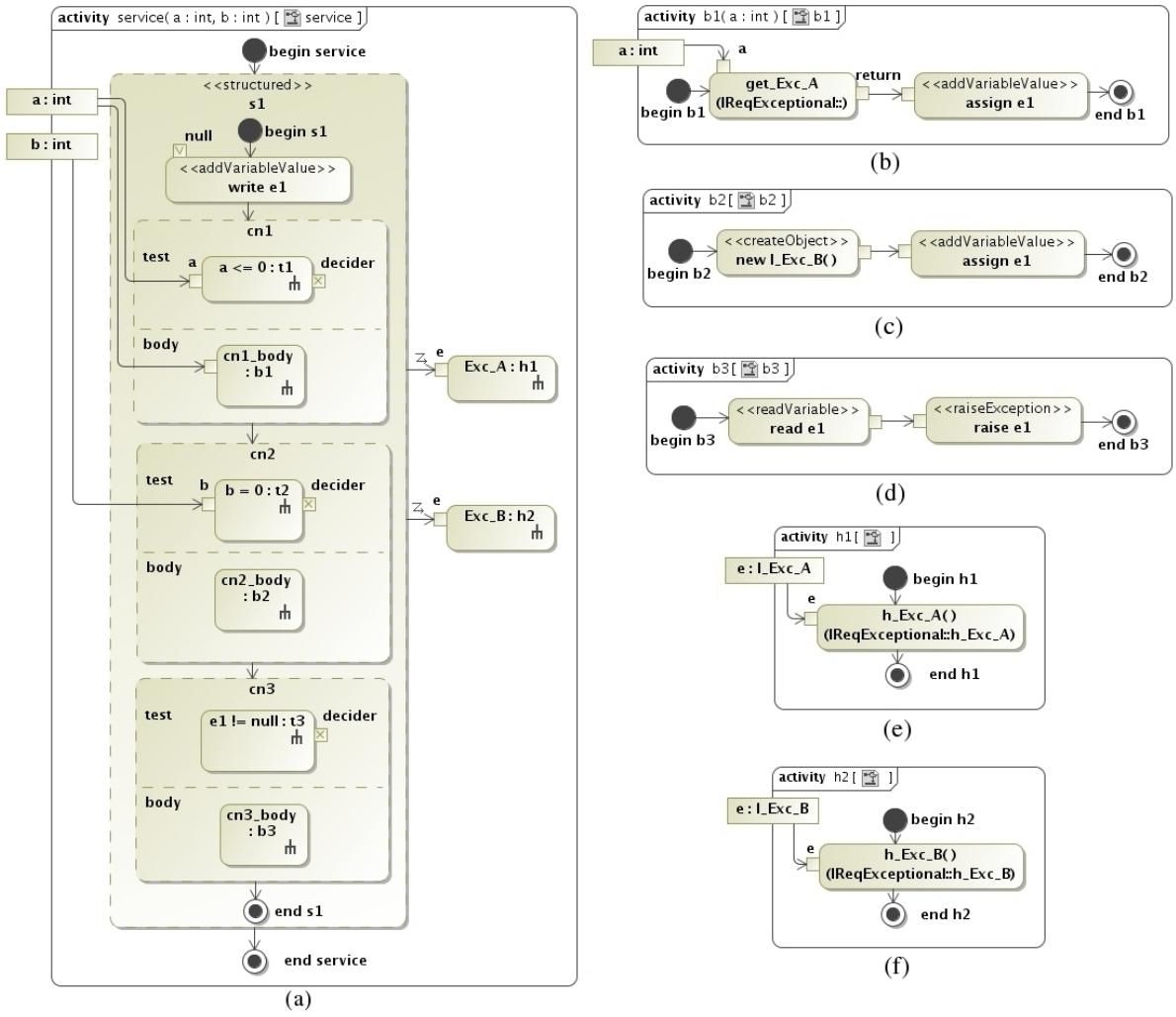


Figura 4.4: Especificação da operação *service()*

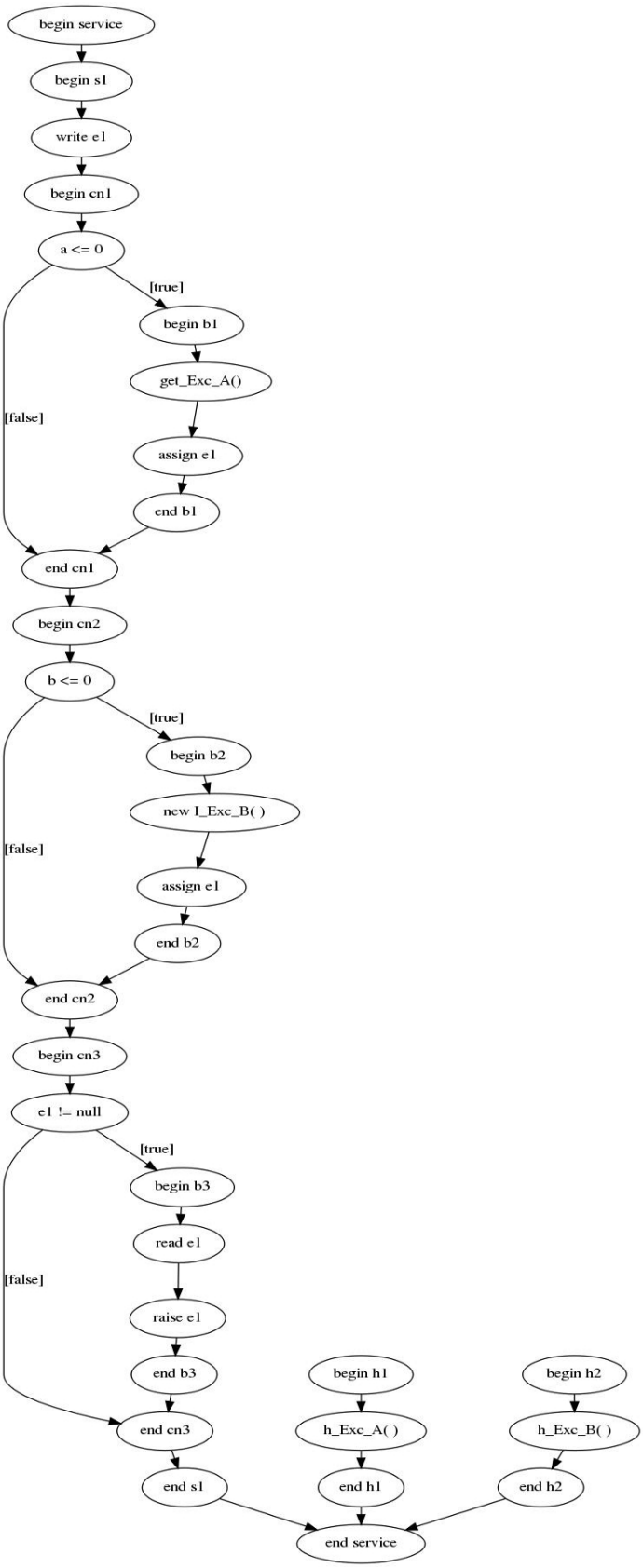


Figura 4.5: Grafo de Atividades da operação *service()*

A atividade estruturada *s1* possui três nós condicionais internos: *cn1*, *cn2* e *cn3*. Cada um deles possui apenas uma cláusula, ou seja, um par de seções. A primeira seção correspondente ao teste condicional e a segunda seção corresponde ao corpo do nó condicional. Cada corpo é constituído por uma ação do tipo *CallBehaviorAction*. As Figuras 4.4(b), 4.4(c) e 4.4(d) são as atividades correspondentes aos corpos dos nós condicionais *cn1*, *cn2* e *cn3* respectivamente. As atividades que correspondentes às seções de teste *t1*, *t2* e *t3*, não foram apresentadas por questões de espaço. Para facilitar o entendimento as condições de teste foram inseridas como rótulo das ações.

Submetendo o conjunto de atividades da Figura 4.4 como entrada para os Algoritmos 1 e 2, obtemos o grafo de atividades mostrado na Figura 4.5. O grafo resultante é a combinação dos subgrafos individuais de cada atividade e representa o fluxo de controle da operação *service()*.

4.2 Obtenção do Grafo de Fluxo de Controle a partir do GA

No segundo passo, o GA gerado no passo anterior é transformado em um Grafo de Fluxo de Controle (GFC). Cada nó do GFC representa um bloco básico de atividades do GA, o qual não possui nenhum desvio de controle no seu interior, exceto ao final do bloco. Assim, um conjunto de nós do GA podem ser agrupados em um único nó do GFC.

Formalmente um GFC pode ser representado pela tupla:

$$ControlFlowGraph = \langle CFGNodes, CFGEdges \rangle$$

onde $CFGNodes$ e $CFGEdges$ são novamente particionados em respectivas metaclasses que são apresentadas a seguir:

$$CFGNodes = \langle N, iN, fN, cN, rN, rsN, ctN, eN \rangle$$

onde: N conjunto de nós do GFC
iN, fN nó inicial e nó final
cN, rN conjunto de nós de chamada e conjunto de nós de retorno de chamada
rsN, ctN conjunto de nós de lançamento e conjunto de nós de captura de exceção
eN conjunto de nós de saída excepcional

$$CFGEdges = \langle PE, EE \rangle$$

onde: PE conjunto de arestas normais entre os nós do GFC
EE conjunto de arestas de exceção entre N e EN

Algoritmo 3 PROCESSCFG(m, n, cfg)

Entrada: Nó do grafo de atividades m
Entrada: Nó do grafo de fluxo de controle n
Saída: Grafo de fluxo de controle cfg.

1: **if** $m.visited =$ **false then** {Nó ainda não visitado}
2: $n.visited \Leftarrow$ **true**{Marca nó como visitado}
3: $a \Leftarrow m.activity$ {Atividade UML associada ao nó do GA}
4: $predecessor \Leftarrow n$
5: **if** s is a $uml2.CallOperationAction$ **then** {Ação de chamada de procedimento}
6: $n \Leftarrow CFGCallNode()$
7: $cfg.nodes \Leftarrow cfg.nodes \bigcup n$
8: $predecessor.edges \Leftarrow predecessor.edges \bigcup CFGEdge(predecessor, n)$
9: **else if** a is a $uml2.DecisionNode$ or $uml2.MergeNode$ **then** {Decisão/Mesclar}
10: **if** $a.edges.size > 1$ **then** {Nó com mais de uma aresta}
11: $n \Leftarrow CFGNode()$
12: $cfg.nodes \Leftarrow cfg.nodes \bigcup n$
13: $predecessor.edges \Leftarrow predecessor.edges \bigcup CFGEdge(predecessor, n)$
14: **else**
15: $N \Leftarrow predecessor$
16: **else if** a is a $AuxiliarNode$ **then** {Nó Auxiliar}
17: $n \Leftarrow CFGNode()$
18: $cfg.nodes \Leftarrow cfg.nodes \bigcup n$
19: $predecessor.edges \Leftarrow predecessor.edges \bigcup CFGEdge(predecessor, n)$
20: **else if** a is a $uml2.RaiseExceptionAction$ **then** {Ação de exceção}
21: $s \Leftarrow a.sourceException$
22: **if** s is a $uml2.ReadVariableAction$ **then** {Variável de exceção}
23: $n \Leftarrow CFGRaiseVariableNode()$
24: **else**
25: $n \Leftarrow CFGRaiseNode()$
26: $cfg.nodes \Leftarrow cfg.nodes \bigcup n$
27: $predecessor.edges \Leftarrow predecessor.edges \bigcup CFGEdge(predecessor, n)$
28: $n.activities \Leftarrow n.activities \bigcup m$ {Adiciona o nó do GA ao nó do GFC}
29: $n.uses \Leftarrow n.uses \bigcup m.uses$ {Transfere os usos}
30: $n.definitions \Leftarrow n.definitions \bigcup m.definitions$ {Transfere as definições}
31: **for all** e in $m.edges$ **do** {Percorre cada aresta do nó do GA}
32: $target \Leftarrow e.target$
33: PROCESSCFG$(target, n, cfg)$

Durante a construção do GFC alguns nós do GA são mapeados para nós exclusivos no GFC, são eles: i) *CallOperationAction*: esta ação representa a chamada de um procedimento, no GFC este nó representa um nó de chamada a outro procedimento; ii) *DecisionNode* e *MergeNode*: estes dois nós representam respectivamente a ramificação e a junção do fluxo de controle; iii) *AuxiliarNode*: representa o nó auxiliar inserido no GA para representar o desvio do fluxo de controle de uma atividade estruturada; iv) *RaiseExceptionAction*: esta ação representa o lançamento de uma exceção, e o nó correspondente indica o início do fluxo de exceção; v) *Exceptional Exit*: nó de saída excepcional criado quando uma exceção não é capturada por nenhum dos tratadores de exceção definidos dentro da operação. Os nós de chamada de procedimento *CallOperationAction* e o nó de saída excepcional *Exceptional Exit* são criados para permitir a representação do fluxo de controle e de exceções interprocedimental. No final deste processo cada operação (procedimento) existente no modelo terá o seu GFC gerado. O Algoritmo 3 é utilizado na construção do GFC.

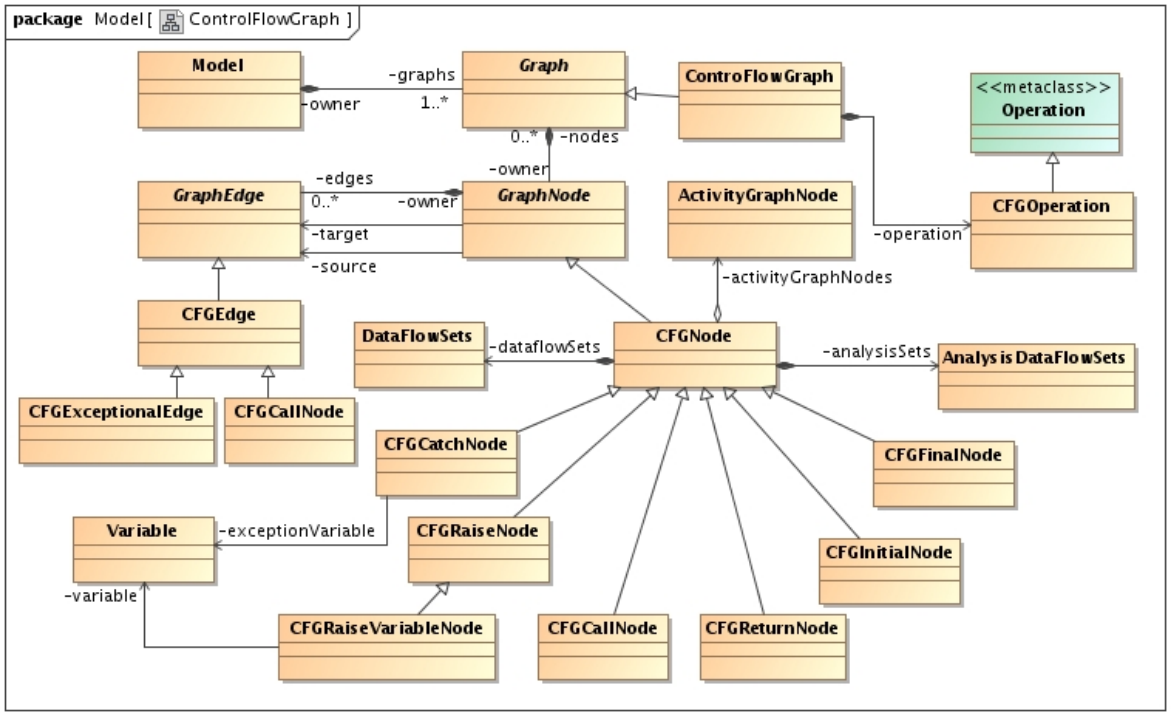


Figura 4.6: Modelagem do Grafo de Fluxo de Controle (GFC)

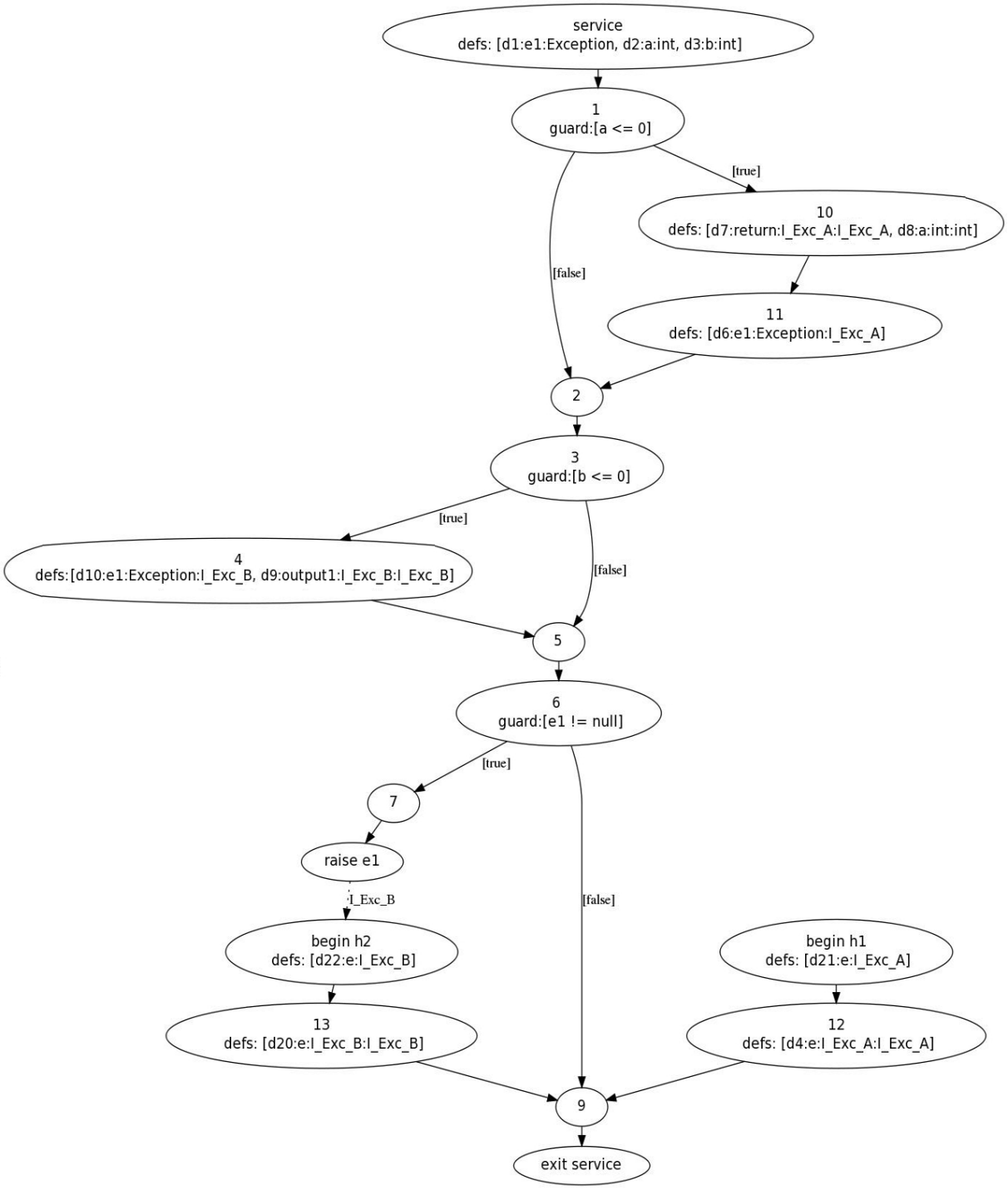


Figura 4.7: Grafo de Fluxo de Controle da operação *service()*

A Figura 4.6 mostra o diagrama de classes utilizado para modelar o Grafo de Fluxo de Controle. A classe *ControlFlowGraph* especializa a classe abstrata *Graph* definindo

o próprio GFC. A classe *CFGOperation* representa a operação que está associada ao GFC, esta classe possui uma referência a classe *Operation* do metamodelo da UML. A classe *CFGNode* especializa a classe abstrata *GraphNode* para modelar um nó do GFC. A classes *CFGInitialNode* e *CFGFinalNode* especializam a classe *GraphNode* para modelar, respectivamente, o nó inicial e o nó final do GFC. As classes *CFGCallNode* e *CFGReturnNode* também são subclasses de *GraphNode* e representam o nó de chamada e nó de retorno de uma chamada de operação. As classes *CFGRaiseNode*, *CFGRaiseVariableNode* e *CFGCatchNode* também especializam a classe *CFGNode* e são os nós responsáveis pelo fluxo excepcional do GFC. Os dois primeiros responsáveis pelo lançamento e o último responsável pelo tratamento de uma exceção.

A classe *CFGNode*também possui um relacionamento entre as classes *CFGNode* e *ActivityGraphNode* definindo o atributo *activityGraphNodes* que identifica o conjunto de nós do GA que estão contidos em um nó do GFC. A classe *CFGEdge* especializa a classe abstrata *GraphEdge* para modelar uma aresta do GFC. A classe *CFGExceptionalEdge* é uma subclasse de *CFGEdge* e representa uma aresta de fluxo excepcional ligando o nó que lança a exceção até o nó correspondente ao tratador de exceções ou saída excepcional. A classe *CFGCallEdge* também é uma subclasse de *CFGEdge* e representa uma aresta de chamada interprocedimental que liga o nó de chamada ao nó inicial do grafo correspondente a operação chamada.

A classe *AnalysisDataFlowSets* possui um relacionamento com a classe *CFGNode* e define quatro conjuntos de variáveis que são utilizados pelos nós durante uma análise de fluxo de dados. A classe *DataFlowSets* possui um relacionamento com a classe *CFGNode* para especificar os conjuntos de usos e definições das variáveis pelos nós.

Na Figura 4.7 é apresentado o GFC resultante da execução do Algoritmo 3. As definições geradas pelos nós são exibidas no conjunto de definições *def*. Estas definições serão utilizadas nas análises de fluxos de dados para gerar tanto o fluxo de exceções intraprocedimental quanto o fluxo de interprocedimental.

4.3 Fluxo de Exceções Intraprocedimentais

O primeiro passo para a criação do fluxo de exceções intraprocedimental no GFC é a identificação dos tipos de exceção que podem ser lançados por um determinado nó. O que determina se um nó do GFC lança uma exceção é a ocorrência de uma ação do tipo *RaiseExceptionAction*. Sempre que uma ação deste tipo é encontrada no GA, durante a geração do GFC, um nó exclusivo para esta ação é criado, conforme descrito na seção 4.2. A partir deste nó terá início o fluxo de exceções. No caso de uma exceção que é tratada dentro do próprio procedimento, uma aresta de exceção irá ligar este nó com o nó correspondente ao tratador de exceção compatível. No caso da exceção não ser tratada,

uma aresta de exceção irá ligar este nó com o nó correspondente à saída excepcional do procedimento.

De acordo com o modelo definido (Figura 4.6), no GFC existem dois tipos de nós que podem lançar uma exceção, o primeiro é o *CFGRaiseNode* e o segundo é o *CFGRaiseVariableNode*. O que determina se um nó do GA, que contém uma ação do tipo *RaiseExceptionAction*, será mapeado para um nó do GFC do tipo *CFGRaiseNode* ou *CFGRaiseVariableNode*, é o tipo de ação que antecede a ação *RaiseExceptionAction* no DA. Por exemplo, na Figura 4.4(d) a ação que fornece o objeto de exceção é do tipo *ReadVariableAction*. Neste caso, o nó criado no GFC é do tipo *CFGRaiseVariableNode*, pois a ação *ReadVariableAction* lê o conteúdo de uma variável é passa este valor como entrada para a ação que lança a exceção, o que implica na dependência da definição da variável para a identificação do tipo de exceção que pode ser lançada. Caso a ação que fornece o objeto de exceção seja do tipo *CreateObjectAction*, não existe a dependência de uma variável. Assim, o nó criado no GFC será do tipo *CFGRaiseNode*, pois o tipo de exceção lançada pode ser determinado diretamente pelo tipo do objeto criado pela ação *CreateObjectAction*.

Para determinar quais são os tipos de exceção que podem ser lançados por um nó do tipo *CFGRaiseVariableNode*, utilizamos uma análise de fluxo de dados sobre o GFC. Esta análise consiste no cômputo das definições incidentes *(reaching definitions)* definida na seção 2.3.2.

Após o cômputo das definições incidentes, precisamos inferir quais são os tipos de exceções que podem ser lançados e quais são os possíveis tratadores de exceção que podem capturar essas exceções. Para isso, utilizamos também as informações do contexto das regiões interruptíveis, das atividades estruturadas e das informações dos tratadores de exceção extraídas do modelo.

No caso da ocorrência de uma variável do tipo *Exception*, a dificuldade para inferir o tipo de exceção que pode ser lançada é o fato desta variável poder armazenar uma referência de qualquer umas das exceções que são subclasses concretas de *Exception*. O problema a ser resolvido é identificar, em um dado nó do GFC que lança uma exceção a partir do uso de uma variável e do tipo *Exception*, quais são os tipos de exceções que podem ser lançados. Para isso, precisamos computar quais são as definições da variável e que incidem neste nó. A solução adotada foi a seguinte: como cada definição possui a informação do tipo de dado da instância do objeto referenciado, a partir do conjunto de definições incidentes identificamos o conjunto dos tipos de exceções que podem estar associados a variável e e consequentemente obtemos o conjunto dos tipos de exceções que podem ser lançados pelo nó.

Além de determinar quais tipos de exceção podem ser lançados a partir de um determinado nó, precisamos saber quais tratadores de exceção podem capturar a exceção lançada.

Como um tratador de exceção (*ExceptionHandler*) pode estar associado a qualquer atividade estruturada, é necessário empilhar os contextos (atividades estruturadas) em que cada ação está incluída. Dessa forma, quando uma ação do tipo *RaiseExceptionAction* é identificada, cada contexto empilhado deve ser analisado para verificar a existência de tratadores de exceção. Se existir algum, é preciso verificar se é capaz de capturar alguma, dentre as possíveis exceções lançadas.

Caso exista um tratador de exceção equivalente, ou seja, que trate um tipo de exceção igual ou mais genérico daquele que foi lançado, uma aresta de exceção é criada ligando o nó que lança a exceção com o nó que faz o tratamento da mesma. Caso não exista um tratador de exceção equivalente, um nó de saída excepcional é criado.

Para demonstrarmos a construção do fluxo de exceções intraprocedimental voltamos ao exemplo do componente de software abstrato mostrado na Figura 4.4. Apresentamos agora a hierarquia das exceções contidas no modelo (Figura 4.8). O diagrama de classes contém onze exceções, as exceções internas ao componente são identificadas pelo prefixo *I*, já as exceções externas, ou seja, aquelas que são propagadas pelo componente, são identificadas pelo prefixo *E*.

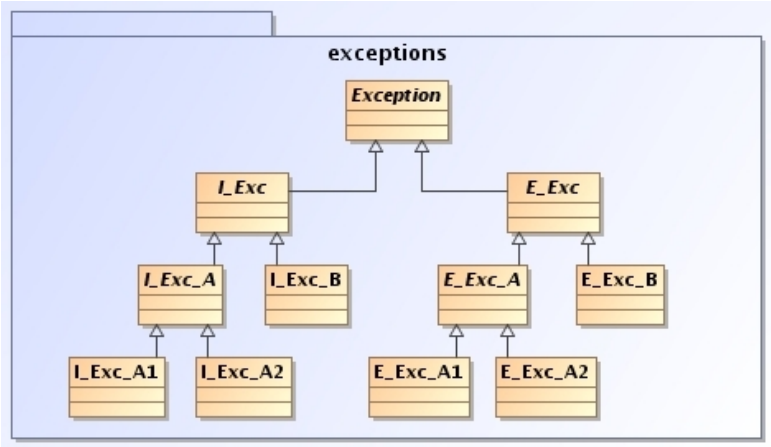


Figura 4.8: Hierarquia de Exceções

No caso do GFC da operação *service*, mostrado na Figura 4.7, o nó *"raise e1"* lança uma exceção baseado na variável *e1*. Considerando que esta variável é do tipo *Exception*, podemos afirmar que pode ser definida por instâncias de qualquer subclasse de *Exception*. Como as classes *Exception*,*I_Exc*, *I_Exc_A*, *E_Exc* e *E_Exc_A* são abstratas e não podem ser instanciadas, restam as classes concretas *I_Exc_A1*, *I_Exc_A2*, *I_Exc_B*, *E_Exc_A1*, *E_Exc_A2* e *E_Exc_B*. Somente estas informações não são suficientes para identificarmos quais destas exceções de fato podem ser lançadas pelo nó *"raise e1"*, para isso temos que executar uma análise de fluxo de dados.

Apresentamos a seguir o resultado do cômputo das definições incidentes (*reaching definitions*), realizado sobre o GFC da Figura 4.7. Na Tabela 4.1 podemos ver os conjuntos de definições globais geradas em cada nó do GFC e na Tabela 4.2 apresentamos o resultado da análise de fluxo de dados.

Tabela 4.1: *Conjunto de Definições do GFC*

n	**definições[n]**	**geradas[n]**	**mortas[n]**
service	$\langle d1 : e1 : Exception \rangle$, $\langle d2 : a : int \rangle$, $\langle d3 : b : int \rangle$	$d1$, $d2$, $d3$	$\varnothing$
4	$\langle d9 : output1 : I\_Exc\_B : I\_Exc\_B \rangle$, $\langle d10 : e1 : Exception : I\_Exc\_B \rangle$	$d9$, $d10$	$d1$
10	$\langle d7 : return : I\_Exc\_A : I\_Exc\_A \rangle$, $\langle d8 : a : int \rangle$	$d7$, $d8$	$d2$
11	$\langle d6 : e1 : Exception : I\_Exc\_A \rangle$	$d6$	$d1$, $d10$
begin h1	$\langle d21 : e : I\_Exc\_A \rangle$	$d21$	$d4$
12	$\langle d4 : e : I\_Exc\_A : I\_Exc\_A \rangle$	$d4$	$d21$
begin h2	$\langle d22 : e : I\_Exc\_B \rangle$	$d22$	$d20$
13	$\langle d20 : e : I\_Exc\_B : I\_Exc\_B \rangle$	$d20$	$d22$

Tabela 4.2: *Resultado da Análise das definições incidentes*

n	$in[n]$	$out[n]$
service	$\varnothing$	$d1, d2, d3$
1	$d1, d2, d3$	$d1, d2, d3$
10	$d1, d2, d3$	$d1, d3, d7, d8$
11	$d1, d3, d7, d8$	$d3, d6, d7, d8$
2	$d1, d2, d3, d6, d7, d8$	$d1, d2, d3, d6, d7, d8$
3	$d1, d2, d3, d6, d7, d8$	$d1, d2, d3, d6, d7, d8$
4	$d1, d2, d3, d6, d7, d8$	$d1, d2, d3, d6, d7, d8$
5	$d1, d2, d3, d6, d7, d8, d9, d10$	$d1, d2, d3, d6, d7, d8, d9, d10$
6	$d1, d2, d3, d6, d7, d8, d9, d10$	$d1, d2, d3, d6, d7, d8, d9, d10$
7	$d1, d2, d3, d6, d7, d8, d9, d10$	$d1, d2, d3, d6, d7, d8, d9, d10$
raise e1	$d1, d2, d3, d6, d7, d8, d9, d10$	$d1, d2, d3, d6, d7, d8, d9, d10$
9	$d1, d20, d21, d22, d2, d3, d4, d6, d7, d8, d9, d10$	$d1, d20, d21, d22, d2, d3, d4, d6, d7, d8, d9, d10$
begin h1	$\varnothing$	$d21$
12	$d21$	$d4, d21$
begin h2	$\varnothing$	$d22$
13	$d22$	$d20, d22$
exit service	$d1, d2, d3, d4, d6, d7, d8, d9, d10, d20, d21, d22$	$d1, d2, d3, d4, d6, d7, d8, d9, d10, d20, d21, d22$

O conjunto de definições incidentes que alcança o nó *"raise e1"* é o seguinte:

$$entrada[raise\ e1] = d1, d2, d3, d6, d7, d8, d9, d10$$

Desse conjunto apenas as definições $d1$, $d6$ e $d10$, que são definições da variável $e1$, são examinadas para determinar os possíveis tipos de exceções lançadas pelo nó. A definição $d1$ foi gerada pela inicialização da variável $e1$, representada pela ação *write e1* (Figura 4.4(a)). Neste caso foi atribuída uma referência nula a variável, por este motivo a definição $d1$ não possui tipo de dado da referência. Já a definição $d6$ foi gerada pela atribuição do retorno do método *get_Exc_A()*. Como o tipo de retorno desta operação é uma classe abstrata (*I_Exc_A*) que possui outras duas subclasses (I_Exc_A1 e I_Exc_A2), não podemos

determinar neste momento o tipo de dado da referência. Esta definição somente ocorrerá após a execução da análise de fluxo de dados interprocedimental. Dessa forma, a única definição válida é $\langle d10 : e1 : Exception : I\_Exc\_B \rangle$, portanto apenas a exceção *I_Exc_B* foi identificada pela análise intraprocedimental.

O próximo passo para a construção do fluxo excepcional é a procura por um tratador de exceção (*ExceptionHandler*) compatível com o tipo *I_Exc_B*. Esta procura começa pela ação do tipo *RaiseExceptionAction* responsável pelo lançamento da exceção no DA. A Figura 4.4(d) mostra a atividade *b3* onde a ação *raise e1* está contida. Esta atividade é chamada pela ação *cn3_body*, que representa a seção do corpo do nó condicional *cn3* da atividade *service* mostrada na Figura 4.4(a). O nó condicional *cn3* é a primeira região protegida que o nó *raise e1* está contido, como este nó não possui tratadores de exceção a busca pelo tratador de exceções compatível continua. A próxima região protegida que encapsula o nó condicional e a atividade estruturada *s1*. Esta atividade possui dois tratadores de exceção o primeiro *Exc_A* está associado ao tipo de exceção *I_Exc_A* que não é compatível com o tipo procurado, já o segundo *Exc_B* tratador está associado ao tipo de exceção *I_Exc_B* que é exatamente o tipo de exceção identificado pela análise de fluxo de dados.

O passo final é a criação da aresta de exceção que liga o nó origem, onde a exceção foi lançada, com o nó destino, responsável pelo tratamento da exceção. No nosso exemplo o nó *raise e1* é a origem e o nó destino é identificado a partir do tratador de exceção *Exc_B*. Como o tratador de exceção invoca a atividade *h2*, o nó destino é o nó *begin h2*, que corresponde ao nó inicial da atividade *h2*.

Capítulo 5

Grafo de Fluxo de Controle Interprocedimental

Quando uma exceção é lançada dentro de uma região protegida, o controle é transferido para um tratador de exceções capaz de manipular a exceção lançada. Mas, se o tratador de exceções não estiver presente na operação chamada, o objeto de exceção é propagado para a operação chamadora. A propagação das exceções pela pilha de chamada cria o fluxo de controle interprocedimental.

Para representar o fluxo de controle interprocedimental vamos utilizar o Grafo de Fluxo de Controle Interprocedimental (GFCI), apresentado anteriormente na seção 2.3.3. A representação utilizada nesta abordagem foi proposta por Sinha e Harrold [Sinh 99]. Para realizar a análise de fluxo de dados interprocedimental recorremos a outra estrutura, o Grafo Resumido da Interface (GRI) apresentado na seção 2.3.4. O GRI original foi proposto por Soffa e Harrold [Harr 94b]. Este grafo é uma abstração dos nós de interface do GFCI e foi escolhido por simplificar a análise de fluxo de controle interprocedimental. Neste trabalho também apresentamos uma extensão do GRI, foram incluídos novos nós e arestas para representar a propagação das exceções pela pilha de chamada.

A seguir apresentamos o processo de obtenção do fluxo de controle interprocedimental. Este processo esta dividido em três etapas: i) na seção 5.1 iremos apresentar os passos necessários para a criação do GFCI; ii) na seção 5.2 apresentamos a criação do GRI a partir da abstração dos nós de interface do GFCI; iii) na seção 5.3 apresentamos as análises de fluxo de dados necessárias para a criação das arestas de exceção interprocedimentais.

5.1 Geração do Grafo de Fluxo de Controle Interprocedimental

O fluxo de controle interprocedimental é representado pelo GFCI. Um GFCI para um modelo M consiste em um conjunto de GFCs interligados pelas arestas interprocedimentais, cada GFC corresponde a um procedimento contido em M. Portanto, o processo de geração do GFCI consiste apenas na criação das arestas interprocedimentais que interligam os GFCs. Nesta etapa são criadas apenas as arestas de fluxo de controle normal, o fluxo de controle excepcional é criado mais tarde, após uma análise de fluxo de dados interprocedimental.

O Algoritmo 4 apresenta os passos utilizados na construção do GFCI.

Algoritmo 4 ProcessICFG($cfg, icfg$)

Entrada: Grafo de Fluxo de Controle (GFC) cfg
Saída: Grafo de Fluxo de Controle Interprocedimental (GFCI) $icfg$.

1: **if** $cfg.visited =$ **false then** {GFC ainda não visitado}
2: $icfg.nodes \Leftarrow icfg.nodes \bigcup cfg.nodes$ {Incorpora os nós do GFC ao GFCI}
3: $cfg.visited \Leftarrow$ **true**{Marca o GFC como visitado}
4: **for all** n in $cfg.nodes$ **do** {Percorre cada nó do GFC}
5: **if** n is a $CFGCallNode$ **then** {Nó de chamada}
6: $sucessors \Leftarrow n.sucessors$ {Recupera todos os nós sucessores de n}
7: $n.edges \Leftarrow \varnothing$ {Remove todas as aresta do nó de chamada}
8: $operation \Leftarrow n.operation$ {Recupera a operação chamada}
9: $callCFG \Leftarrow operation.cfg$ {Recupera o GFC da operação}
10: $n.label \Leftarrow$ "$call$" + $operation.label$
11: ProcessICFG($callCFG, icfg$)
12: $i \Leftarrow callCFG.initialNode$ {Recupera o nó inicial do GFC}
13: $f \Leftarrow callCFG.finalNode$ {Recupera o nó final do GFC}
14: $n.edges \Leftarrow n.edges \bigcup CFGCallEdge(n, i)$ {Aresta de chamada}
15: $r \Leftarrow CFGReturnNode(operation)$ {Nó de retorno}
16: $r.label \Leftarrow$ "$return$" + $operation.label$
17: $icfg.nodes \Leftarrow icfg.nodes \bigcup r$ {Adiciona o nó ao GFCI}
18: $f.edges \Leftarrow f.edges \bigcup CFGCallEdge(f, r)$ {Aresta de retorno}
19: **for all** s in $sucessors$ **do** {Para cada sucessor do nó de chamada}
20: $r.edges \Leftarrow r.edges \bigcup CFGEdge(r, s)$
21: **return** $icfg$

O processo de geração das arestas interprocedimentais é simples. Cada GFC é percorrido para encontrar os *nós de chamada*. Para cada *nó de chamada* identificado, uma *aresta de chamada* é criada conectando o *nó de chamada* com o *nó de entrada* do pro-

cedimento correspondente. Na sequência, uma *aresta de retorno* é criada para conectar o *nó de saída* do procedimento com um *nó de retorno*. As arestas de fluxo de controle que ligavam o *nó de chamada* aos seus sucessores, são transferidas para o *nó de retorno* recém criado.

5.2 Construção do Grafo Resumido da Interface

Para a obtenção do fluxo excepcional interprocedimental é necessário executar uma análise de fluxo de dados. A análise utilizada é o cômputo das definições incidentes interprocedimentais *(interprocedural reaching definitions)*. Com o objetivo de simplificar a execução da análise de fluxo de dados, um novo grafo é gerado a partir da interface das operações do modelo, denominado de Grafo Resumido da Interface (GRI). Uma característica importante do GRI são as *arestas de alcance interprocedimental* que são utilizadas para preservar o contexto das chamadas durante a análise de fluxo de dados.

Após a execução da análise de fluxo de dados no GRI, os conjuntos de definições incidentes computados são utilizados para a criação das arestas de exceções ligando os nós que lançam a exceção com os nós responsáveis pelo seu tratamento. Caso a exceção não seja tratada pela operação a aresta de exceção liga o nó que lança a exceção com o nó de saída excepcional da operação.

O GRI é uma extensão do grafo apresentado por Harrold e Soffa [Harr 94b]. As extensões propostas incluem o fluxo de controle e dados causado pelos tratadores de exceção e fluxo de dados gerados a partir dos parâmetros de retorno dos procedimentos.

Na abordagem proposta o GRI é gerado a partir do GFCI. Todos os nós de interface das chamadas procedimentais são convertidos em nós do GRI de acordo com a especificação da seção 2.3.4. As informações intraprocedimentais computadas sobre as definições dos parâmetros formais e atuais são anexadas aos nós do grafo. Nos pontos de entrada e de saída de uma operação, as informações sobre as definições locais são abstraídas dos parâmetros formais, enquanto que nos pontos de chamada e de retorno, as informações são extraídas dos parâmetros atuais envolvidos na chamada da operação. Os algoritmos 5 e 6 apresentam os passos utilizados na obtenção do GRI.

Algoritmo 5 GenerateISGNodes($icfg$)

Entrada: Grafo de Fluxo de Controle Interprocedimental (GFCI) $icfg$.
Saída: Grafo Resumido da Interface (GRI) isg.

```
1: for all n in icfg.nodes do {Cada nó do GFCI}
2:     if n is a CFGCallNode then {Nó de chamada}
3:         o ⇐ n.operation {Recupera a operação chamada}
4:         for all a in o.actualParameters do {Cada parâmetro atual}
5:             cn ⇐ CallNode(a)
6:             isg.nodes ⇐ isg.nodes ⋃ cn
7:     else if n is a CFGReturnNode then {Nó de retorno}
8:         o ⇐ n.operation {Recupera a operação chamada}
9:         for all a in o.actualParameters do {Cada parâmetro atual}
10:            rn ⇐ ReturnCallNode(a)
11:            isg.nodes ⇐ isg.nodes ⋃ rn
12:        for all s in n.scopes do {Cada bloco protegido do nó}
13:            for all h in s.handlers do {Cada tratador de exceção}
14:                types ⇐ types ⋃ h.exceptionType {Tipos de exceção com tratador}
15:        ern ⇐ ExceptionalReturnNode(types)
16:        isg.nodes ⇐ isg.nodes ⋃ ern
17:    else if n is a CFGInitialNode then {Nó de entrada}
18:        o ⇐ n.operation {Recupera a operação}
19:        for all f in o.formalParameters do {Cada parâmetro formal}
20:            en ⇐ EntryNode(f)
21:            isg.nodes ⇐ isg.nodes ⋃ en
22:    else if n is a CFGFinalNode then {Nó de saída}
23:        o ⇐ n.operation {Recupera a operação}
24:        for all f in o.formalParameters do {Cada parâmetro formal}
25:            ex ⇐ ExitNode(f)
26:            isg.nodes ⇐ isg.nodes ⋃ ex
27:        for all r in o.resultParameters do {Cada parâmetro de saída}
28:            rsn ⇐ ResultNode(r)
29:            isg.nodes ⇐ isg.nodes ⋃ rsn
30:        een ⇐ ExceptionalExitNode(exceptionsDef)
31:        isg.nodes ⇐ isg.nodes ⋃ een
32:    else if n is a CFGCatchNode then {Nó de captura de exceção}
33:        ctn ⇐ CatchNode(n.exceptionVariable)
34:        isg.nodes ⇐ isg.nodes ⋃ ctn
35: return isg
```

Algoritmo 6 GenerateISGEdges(*isg*)

Entrada: Grafo Resumido da Interface (GRI) *isg*.

```
1: for all n in isg.nodes do {Cada nó do GRI}
2:     if n is a ISGEntryNode or ISGReturnCallNode then {Nó entrada/retorno}
3:         for all m in n.operationNodes do {Cada nó da operação}
4:             if m is a ISGExitNode or ISGCallNode then
5:                 {Cria aresta de alcance}
6:     else if n is a ISGCallNode then {Nó de chamada}
7:         for all m in isg.nodes do {Cada nó do GRI}
8:             if m is a ISGEntryNode then
9:                 {Cria aresta de ligação}
10:            else if m is a ReturnCallNode then
11:                {Cria aresta de alcance interprocedimental}
12:    else if n is a ISGCatchNode then {Nó de captura de exceção}
13:        for all m in n.operationNodes do {Cada nó da operação}
14:            if m is a ISGCallNode then
15:                {Cria aresta de alcance}
16:    else if n is a ISGExceptionalExitNode then {Nó de saída excepcional}
17:        for all m in isg.nodes do {Cada nó do GRI}
18:            if m is a ISGExceptionalReturnNode then
19:                {Cria aresta de propagação de exceção}
20:    else if n is a ISGExceptionalReturnNode then {Nó de retorno excepcional}
21:        for all m in n.operationNodes do {Cada nó da operação}
22:            if m is a ISGExceptionalExitNode then
23:                {Cria aresta de exceções não capturadas}
24:    else if n is a ISGExitNode then {Nó de saída}
25:        for all m in isg.nodes do {Cada nó do GRI}
26:            if m is a ISGReturnCallNode then
27:                {Cria aresta de ligação}
28:    else if n is a ISGResultNode then {Nó de resultado}
29:        for all m in isg.nodes do {Cada nó do GRI}
30:            if m is a ReturnValueNode then
31:                {Cria aresta de retorno de valor}
32:    else if n is a ISGReturnValueNode then {Nó de retorno de valor}
33:        for all m in n.operationNodes do {Cada nó da operação}
34:            if m is a ISGResultNode or ISGCallNode then
35:                {Cria aresta de alcance}
36:    else if n is a ISGThrowNode then {Nó de lançamento de exceção}
37:        for all m in isg.nodes do {Cada nó do GRI}
38:            if m is a ISGCatchNode then
39:                {Cria aresta de ligação}
```

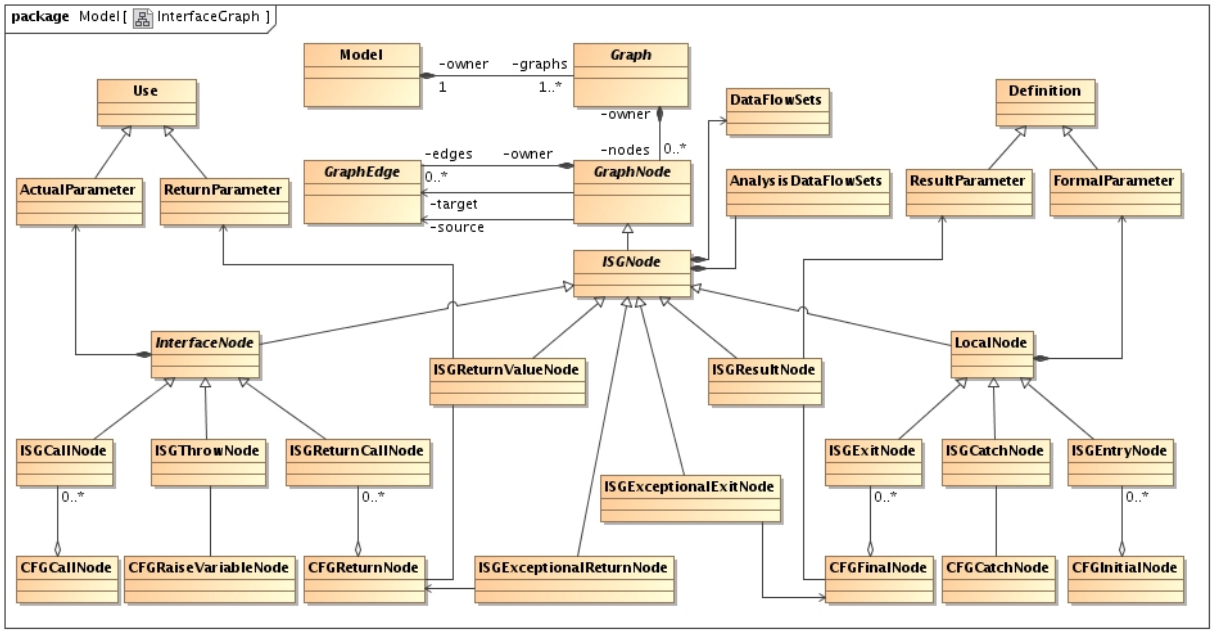


Figura 5.1: Modelagem do Grafo Resumido da Interface (GRI)

Para demonstrar o processo de construção do GRI voltamos a utilizar o exemplo do componente abstrato *service()* apresentado no capítulo 4. Na Figura 7.17 apresentamos o diagrama de classes do modelo, onde são especificadas outras operações. A especificação do componente possui duas interfaces, a interface provida *INormalBehavior* e a interface requerida *IReqExceptional*. A interface *INormalBehavior* especifica a operação *service()* apresentada em detalhes no capítulo 4 e a interface *IReqExceptional* define as operações requeridas pelo componente para executar o tratamento do comportamento excepcional. Esta última interface define três operações: i) *get_Exc_A()* esta operação recebe como parâmetro um inteiro, e retorna uma exceção interna do componente do tipo *I_Exc_A*; ii) *h_Exc_A()* esta operação recebe como parâmetro de entrada uma exceção interna do tipo *h_Exc_A()* e define o comportamento do tratador de exceções para este tipo de exceção e seus subtipos *I_Exc_A1* e *I_Exc_A2*; iii) *h_Exc_B()* esta operação define o comportamento de um tratador de exceções que recebe como parâmetro de entrada uma exceção interna do tipo *I_Exc_B*.

A Figura 5.3 apresenta a especificação do comportamento da operação *get_Exc_A()*, que recebe como entrada um inteiro, representado pelo parâmetro *a*. O comportamento da operação é modelado por três atividades: *get_Exc_A*, *b4* e *b5*. A atividade *get_Exc_A* é constituída por um nó condicional (cn4) que possui duas cláusulas condicionais utilizadas para modelar o comportamento de uma estrutura de controle do tipo *if-else*. O corpo da primeira cláusula é modelado pela atividade *b4*. Esta atividade, por sua vez, é composta

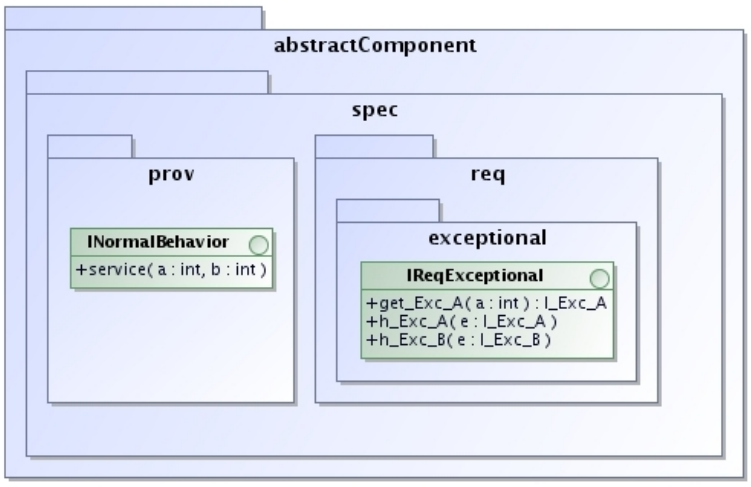


Figura 5.2: Diagrama de Classes do Componente Abstrato

por duas ações, a primeira é uma ação *CreateObjectAction* e a segunda é uma ação *AddVariableValueAction*. A primeira ação cria um objeto de exceção do tipo *I_Exc_A1* e a segunda define a variável *e2* com a referência do objeto de exceção criado pela primeira. O corpo da segunda cláusula condicional é modelado pela atividade *b5*. Esta atividade, por sua vez, é composta por duas ações, a primeira cria um objeto de exceção do tipo *I_Exc_A2* e a segunda ação também define a variável *e2* com a referência do objeto de exceção criado pela ação anterior.

A Figura 5.4 apresenta a especificação do comportamento da operação *h_Exc_A()*, que define um tratador para exceções do tipo *I_Exc_A1* e *I_Exc_A2*, conforme a Figura 4.8 as duas exceções são subtipos concretos da classe de exceção abstrata *I_Exc_A*. A operação é modelado por três atividades: *h_Exc_A*, *h3* e *h4*. A atividade *h_Exc_A* é a principal, esta atividade é composta por um parâmetro de entrada *e* do tipo *I_Exc_A* e uma atividade estruturada *s2* que possui dois tratadores de exceção *Exc_A1* e *Exc_A2*, modelados respectivamente pelas atividades *h3* e *h4*. A atividade estruturada *s2* também possui uma ação interna *thrown* do tipo *RaiseExceptionAction*. Esta ação é responsável pelo relançamento da exceção interna recebida pelo parâmetro *e*. A exceção relançada é imediatamente capturada por um dos tratadores de exceção vinculados a atividade estruturada. A atividade *h3* possui duas ações, a primeira cria um objeto de exceção tipo *E_Exc_A1* que representa uma exceção externa não tratada pelo componente, a segunda ação lança o objeto de exceção criado pela primeira. A atividade *h4* possui um comportamento semelhante à atividade anterior, a diferença é o tipo de exceção criada, neste caso a exceção externa é do tipo *E_Exc_A2*. Em resumo, o comportamento da operação *h_Exc_A()* é capturar subtipos da exceção interna *I_Exc_A* e transformá-los em exceções externas ao componente, criando e lançando novos objetos de exceção correspondentes às exceções internas

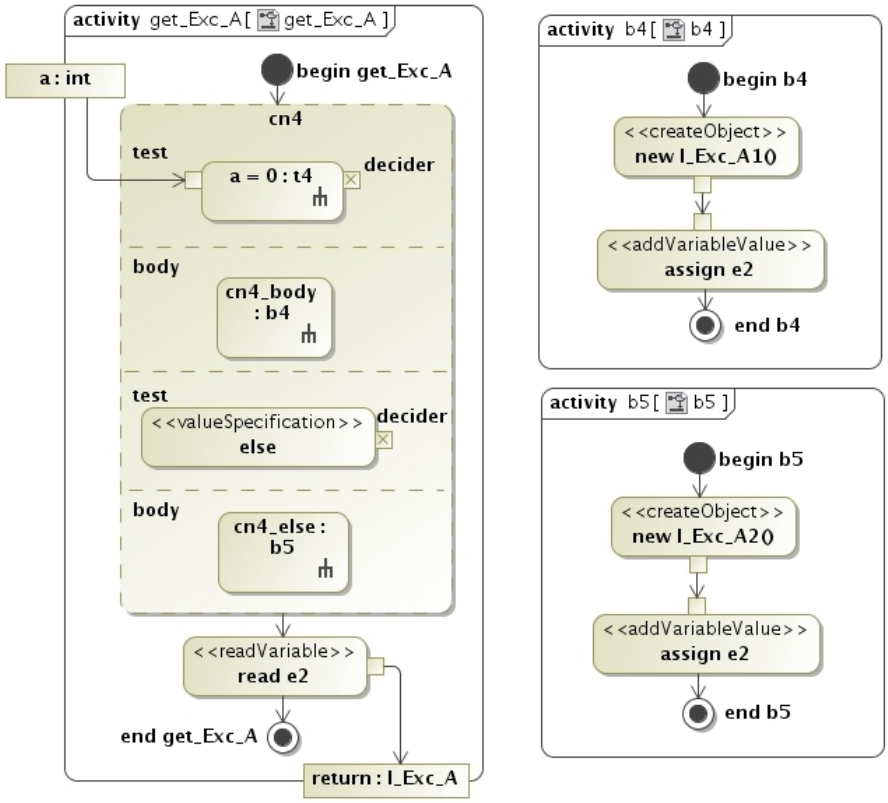


Figura 5.3: Especificação da operação *get_Exc_A()*

capturadas.

A Figura 5.5 apresenta a especificacão do comportamento da operação *h_Exc_B()*. Esta operação define um tratador para exceções do tipo *I_Exc_B*. A operação é composta por duas atividades: *h_Exc_B* e *h5*. A atividade *h_Exc_B* é a principal, composta por um parâmetro de entrada *e* do tipo *I_Exc_B* e pela atividade estruturada *s3* que possui um tratador de exceção *Exc_B*. A atividade *s3* também possui uma ação interna *thrown* do tipo *RaiseExceptionAction*, responsável pelo relançamento da exceção interna recebida pelo parâmetro *e*. A exceção relançada é imediatamente capturada pelo tratador de exceção vinculado à atividade estruturada. Este tratador de exceção é modelado pela atividade *h5*, esta atividade possui duas ações, a primeira cria um objeto de exceção tipo *E_Exc_B* que representa uma exceção externa não tratada pelo componente, a segunda ação lança o objeto de exceção criado pela primeira. Em resumo, o comportamento da operação *h_Exc_B()* é capturar a exceção interna *I_Exc_B* e transformá-la em uma exceção externa ao componente, criando e lançando um novo objeto de exceção.

O GRI resultante é apresentado na Figura 5.6, o grafo está dividido em quatro raias principais, cada uma representando um operação do modelo, são elas: *service()*,

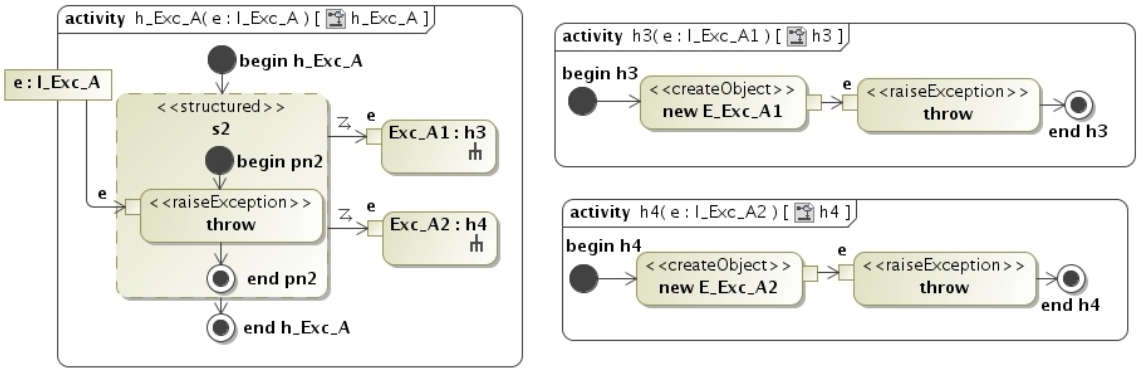


Figura 5.4: Especificação da operação *h_Exc_A()*

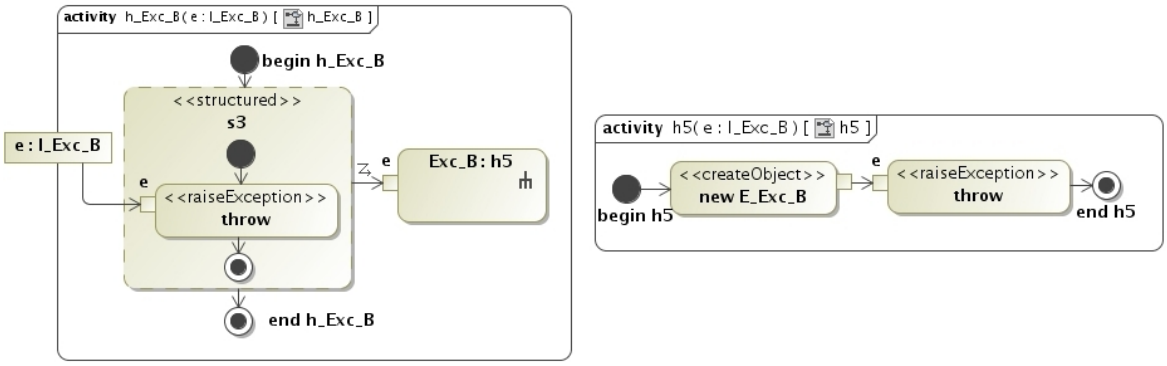


Figura 5.5: Especificação da operação *h_Exc_B()*

get_Exc_A(), *h_Exc_A()* e *h_Exc_B()*. Cada raia do grafo correspondente a uma operação, uma raia também pode ser subdividida em raias secundárias, sendo uma para comportamento normal e outras para a representação do comportamento excepcional. Para cada tratador de exceção da operação é criada uma raia secundária. Por exemplo, a operação *service()* possui três raias secundárias, a primeira para os nós que fazem parte do comportamento normal, a segunda para o tratador de exceção *Exc_A* e a terceira para o tratador de exceção *Exc_B*.

A raia referente ao comportamento normal da operação *service()* possui os seguintes nós:

(1) nó de retorno de valor da chamada da operação *get_Exc_A()*, este nó é criado pelo fato da operação chamada retornar uma referência a um objeto de exceção. Apesar da operação possuir um parâmetro de entrada *a* do tipo inteiro, não existe nenhum nó de chamada para esta operação, pois somente os parâmetros que são referências à objetos dão origem aos nós de chamada; os parâmetros com tipos primitivos são desconsiderados durante a construção do GRI;

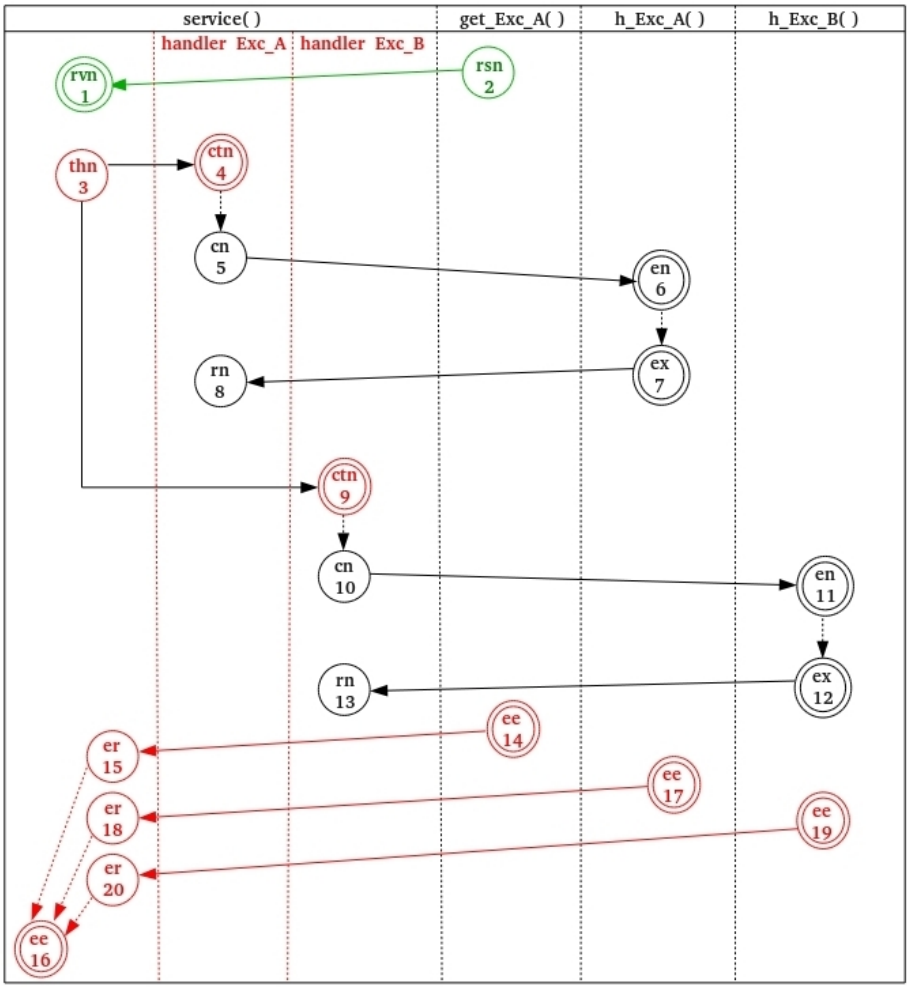


Figura 5.6: GRI do Componente Abstrato

(3) nó de lançamento de exceção gerado pela presença da ação *raise e1* do tipo *RaiseExceptionAction* 4.4(d). Este tipo de nó é criado somente quando uma ação *RaiseExceptionAction* está diretamente vinculada a uma variável, pois as definições da variável de exceção que alcançam a ação devem ser propagadas para os tratadores de exceção compatíveis com o tipo da variável, para o caso da exceção ser relançada pelo tratador de exceção. De outra maneira não seria possível determinar se existe algum outro tratador de exceção compatível com a exceção relançada. Durante a análise de fluxo de dados interprocedimental as definições da variável de exceção que alcançam a ação *RaiseExceptionAction* fluem pela aresta de ligação até os nós de captura de exceção que sejam compatíveis para o tratamento das exceções lançadas. No caso do nó (3) existem dois nós de captura compatíveis com os tipos de exceção lançadas, os nós (4) e (9). O nó (3) possui uma aresta de ligação com cada um

desses nós;

(15, 18, 20) nós de retorno excepcional, para cada operação chamada pela operação *service()* é criado um nó de retorno excepcional. Durante a análise de fluxo de dados interprocedimental, esse tipo de nó é responsável por receber as definições das exceções propagadas pelo método chamado e caso não sejam tratadas pelo método chamador, essas definições fluem pela aresta de exceções não capturadas até o nó de saída excepcional do método chamador;

(16) nó de saída excepcional, únicao para cada operação. O seu papel na análise de fluxo de dados interprocedimental é propagar as definições das exceções não capturadas pelo método. Essas definições fluem pela aresta de propagação de exceção até o nó de retorno excepcional do método chamador, se existir. No caso da operação *service()*, como não existe um método chamador, todas as definições que alcançarem este nó devem ser exceções externas, pois serão propagadas para fora dos limites do componente modelado;

A raia referente ao tratador de exceções *Exc_A* da operação *service()* possui os seguintes nós:

(4) nó de captura de exceção que representa um tratador de exceção. Este nó é correspondente ao nó inicial da atividade *h1* 4.4(e). Durante a análise de fluxo de dados interprocedimental, são consideradas somente as definições da variável de exceção que chegam pela aresta de ligação e que sejam compatíveis com o tipo de exceção tratada pelo nó, neste caso do tipo *I_Exc_A*. Como a definição da variável de exceção tratada pelo nó (4) alcança o nó de chamada (5) existe um aresta de alcance unindo estes dois nós.

(5) nó de chamada do procedimento *h_Exc_A()*, esta procedimento contém parte do comportamento excepcional do componente abstrato. Este nó foi gerado pela ação *h_Exc_A()* do tipo *CallOperationAction* 4.4(e).

(8) nó de retorno de chamada do procedimento *h_Exc_A()*.

A raia referente ao tratador de exceções *Exc_B* da operação *service()* possui os seguintes nós:

(9) nó de captura de exceção que representa um tratador de exceção. Este nó é correspondente ao nó inicial da atividade *h2* 4.4(f). Durante a análise de fluxo de dados interprocedimental, são consideradas somente as definições da variável de exceção que chegam pela aresta de ligação e que sejam compatíveis com o tipo de exceção tratada pelo nó, neste caso do tipo *I_Exc_B*.

(10) nó de chamada do procedimento *h_Exc_B()*, esta procedimento contém parte do comportamento excepcional do componente abstrato. Este nó foi gerado pela ação *h_Exc_B()* do tipo *CallOperationAction* 4.4(f).

(13) nó de retorno de chamada do procedimento *h_Exc_B()*.

A raia referente ao comportamento da operação *get_Exc_A()* possui os seguintes nós:

(2) nó de resultado gerado pelo fato da operação *get_Exc_A()* retornar uma referência de um objeto de exceção. Este nó possui uma aresta de retorno de valor para o nó de retorno de valor (1).

(14) nó de saída excepcional, as definições das exceções não capturadas pelo procedimento *get_Exc_A()* fluem pela aresta de propagação de exceção até o nó de retorno excepcional do método chamador. Portanto, existe uma aresta de propagação de exceção ligando o nó (14) ao nó de retorno excepcional (15).

A raia referente ao comportamento da operação *h_Exc_A()* possui os seguintes nós:

(6) nó de entrada do procedimento *h_Exc_A()*. Este nó corresponde ao nó inicial da atividade *h_Exc_A*(Figura 5.4. Como a definição do parâmetro formal vinculado ao nó (6) alcança o nó final do procedimento *h_Exc_A()*, existe uma aresta de alcance para o nó de saída (7).

(7) nó de saída do procedimento *h_Exc_A()*. Este nó possui uma aresta de ligação para o nó de retorno de chamada (8).

(17) nó de saída excepcional, as definições das exceções não capturadas pelo procedimento *h_Exc_A()* fluem pela aresta de propagação de exceção até o nó de retorno excepcional do método chamador. Portanto, existe uma aresta de propagação de exceção ligando o nó (17) ao nó de retorno excepcional (18).

A raia referente ao comportamento da operação *h_Exc_B()* possui os seguintes nós:

(11) nó de entrada do procedimento *h_Exc_B()*. Este nó corresponde ao nó inicial da atividade *h_Exc_B*(Figura 5.5. Como a definição do parâmetro formal vinculado ao nó (11) alcança o nó final do procedimento *h_Exc_B()*, existe uma aresta de alcance para o nó de saída (12).

(12) nó de saída do procedimento *h_Exc_B()*. Este nó possui uma aresta de ligação para o nó de retorno de chamada (13).

(19) nó de saída excepcional, as definições das exceções não capturadas pelo procedimento *h_Exc_B()* fluem pela aresta de propagação de exceção até o nó de retorno excepcional do método chamador. Portanto, existe uma aresta de propagação de exceção ligando o nó (19) ao nó de retorno excepcional (20).

5.3 Fluxo de Exceções Interprocedimentais

Nossa abordagem é constituída pela combinação de duas análises de fluxo de dados, uma intraprocedimental e outra interprocedimental. Primeiro as informações intraprocedimentais são computadas, na sequência estas informações são propagadas através do GRI por uma análise de fluxo de dados interprocedimental. Por fim, adicionamos o fluxo excepcional interprocedimental ao GFCI.

A análise de fluxo de dados utilizada é o cômputo das definições incidentes interprocedimentais *(interprocedural reaching definitions)*. Esta análise consiste na propagação das informações sobre as definições locais. As definições são propagadas pelo GRI, levando em consideração o contexto das operações chamadas. O conjunto $Entrada_{def}[n]$ se refere ao conjunto das definições incidentes associada ao nó n do GRI. Para preservar o contexto das operações chamadas e garantir que somente os caminhos válidos sejam considerados pela análise de fluxo de dados, os conjuntos de fluxos de dados são computados em duas fases.

Na primeira fase da propagação, são consideradas todas as arestas do GRI e os nós: cn, rn, ex, thn, er, rsn e rvn. Na segunda fase, são considerados todos os nós do GRI, porém as arestas são restritas ao seguinte conjunto: arestas de ligação das chamadas, arestas de alcance e arestas de alcance interprocedimental. A propagação deve ser restrita a este conjunto de arestas do GRI, para evitar que sejam tomados caminhos que não representem o fluxo de execução do programa. A equação de fluxo de dados para as definições incidentes interprocedimentais é a mesma utilizada pela análise intraprocedimental. Além da análise de definições incidentes interprocedimentais, também é utilizada uma equação de fluxo de dados para a propagação das exceções. Esta análise é aplicada aos nós de retorno er de cada operação o, a equação utilizada é a seguinte:

$$
\begin{aligned}
gen[er] &= \text{conjunto de tipos de exceção que podem ser lançadas por } o \\
kill[er] &= \text{conjunto de tipos de exceção capturados dentro da operação } o \\
in[er] &= \bigcup_{P \in predecessors[er]} out[P] \\
out[er] &= gen[er] \bigcup (in[er] - kill[er])
\end{aligned}
$$

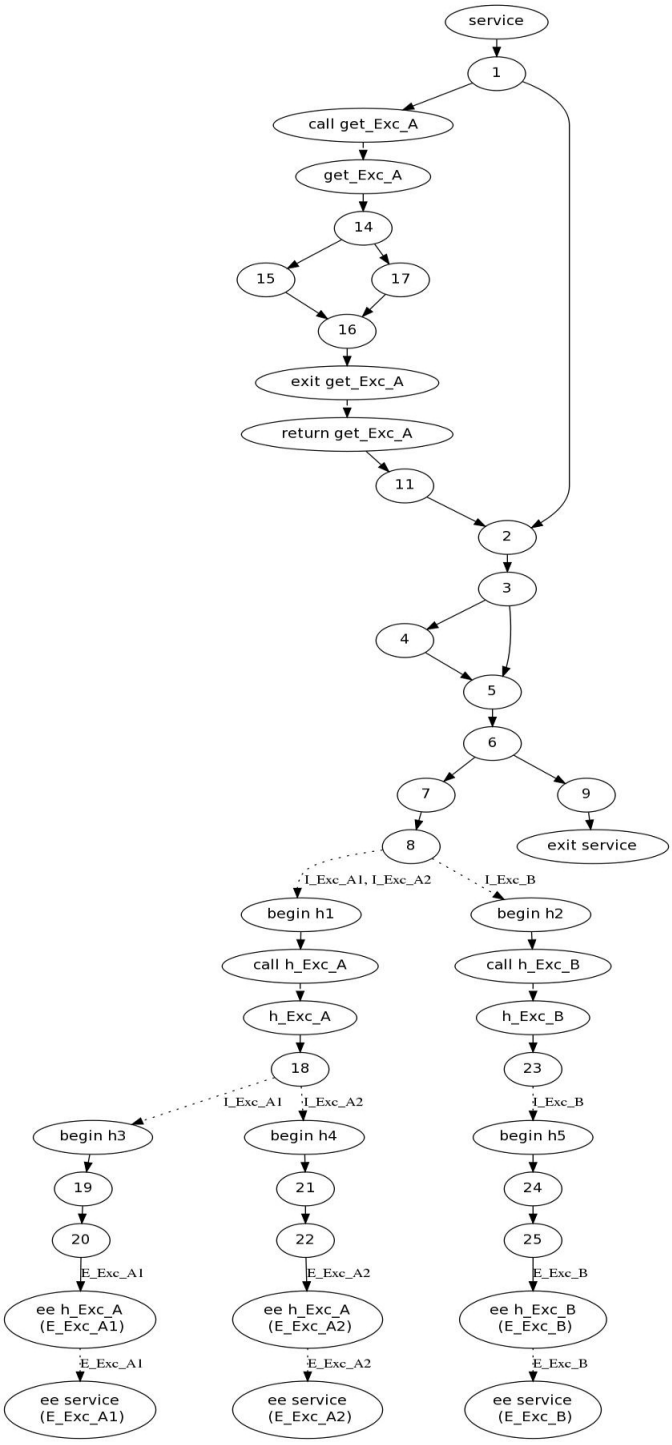


Figura 5.7: Grafo de Fluxo de Controle Interprocedimental

Capítulo 6

Ferramenta de Validação

Este capítulo apresenta as funcionalidades e os detalhes de implementação de uma ferramenta que foi desenvolvida para apoiar o processo de validação do fluxo excepcional proposto neste trabalho.

O capítulo está organizado da seguinte maneira, na seção 6.1 apresentamos uma breve introdução ao capítulo, na seção 6.2 apresentamos os conceitos básicos sobre alguns frameworks e padrões utilizadas pela ferramenta, na seção 6.3 apresentamos as funcionalidades implementadas e a interface gráfica da ferramenta, na seção 6.4 apresentamos os aspectos de implementação e arquitetura da ferramenta.

6.1 Introdução

A ferramenta *ADEX*, um acrônimo para o termo em inglês *"Activity Diagram EXception flow analyzer"*, foi desenvolvida com o objetivo de apoiar o processo de validação do fluxo excepcional, oferecendo ao usuário um conjunto mínimo de funcionalidades para executar todos os passos da abordagem proposta neste trabalho. A ferramenta recebe como entrada um modelo UML composto por um conjunto de diagramas de classes e de atividades. Em seguida, são realizadas as transformações de modelos com o objetivo de gerar um grafo de fluxo de controle interprocedimental, contendo fluxo de controle normal e excepcional do componente de software modelado.

A ferramenta proposta oferece recursos para visualização dos grafos gerados e inspeção das informações armazenadas pelos nós, como por exemplo, as definições e usos das variáveis, os conjuntos gerados durante as análises de fluxo de dados, entre outras informações. A ferramenta também oferece o resultado de algumas análise realizadas sobre os grafos, como por exemplos um resumo das exceções capturadas pelos tratadores de exceção, esta análise também pode indicar quais exceções não estão sendo tratadas.

Na seção 2.4 apresentamos dois exemplos de ferramentas relacionadas que capturam

e validam o fluxo de exceções. A primeira ferramenta é a *Jex*, proposta por Robillard e Murphy [Robi 03, Robi 99]. Esta ferramenta executa uma analise estática desenvolvida para capturar as informações do fluxo excepcional de sistemas Java. A segunda é o framework *Aereal* proposta por Castor et al. [Filh 05], que efetua uma análise do fluxo de controle excepcional no nível arquitetural, baseado em linguagens de descrição de arquiteturas de software.

6.2 Conceitos Básicos

Nesta seção apresentamos alguns conceitos básicos que são utilizados para descrever as funcionalidades e aspectos de implementação da ferramenta.

XML Metadata Interchange (XMI): é um formato padrão da Object Management Group (OMG) para troca de informações baseadas em XML [OMG 07b]. O principal objetivo do XMI é permitir um fácil intercâmbio de metadados entre ferramentas de modelagem baseadas na Unified Modeling Language (UML). O padrão XMI integra três principais padrões da indústria:

- XML - eXtensible Markup Language;
- UML - Unified Modeling Language;
- MOF - Meta Object Facility, um padrão da OMG para escrever metamodelos.

Eclipse Modeling Framework (EMF): é um framework de modelagem que possui recursos para geração de código [Stei 09]. Este framework é utilizado na construção de ferramentas e aplicações baseadas em um modelo estruturado de dados. A partir de uma especificação de um modelo descrito em XMI, o framework EMF fornece ferramentas para transformar o seu modelo em código Java eficiente e facilmente personalizável. O EMF permite a geração de código a partir de alguns modelos de entrada, são eles:

- XML Schema
- Diagrama UML
- Modelo Ecore
- Interfaces Java anotadas

Uma vez que você especifique um modelo EMF, o gerador de código cria um conjunto de classes de implementação Java correspondentes ao modelo. E possível editar as classes geradas para adicionar métodos e variáveis de instância, e caso seja necessário regerar o código é possível preservar o código adicionado. Outro fator importante é a qualidade do código gerado, pois o EMF utiliza diversos padrões de projeto, o resultado é um código bem organizado.

Outro recurso importante do framework EMF e a interoperabilidade com outras ferramentas e aplicativos baseados em EMF. O framework oferece recursos para persistir as classes geradas, como todas as classes são serializáveis é possível persistir o resultado gerando outro modelo XMI. O framework também oferece recursos para fazer a operação inversa, instanciar as classes persistidas a partir do modelo XMI.

Graphviz: oferece um conjunto de ferramentas de código aberto utilizadas para desenhar e visualizar grafos. A ferramenta *Graphviz*[1] é capaz de interpretar uma descrição textual de um grafo e gerar uma imagem gráfica correspondente. A descrição textual do grafo obedece a uma linguagem de construção chamada *DOT* [Gans 00], utilizada para construir grafos hierárquicos direcionados. O algoritmo de layout do Graphviz tenta otimizar a disposição dos nós e arestas para para facilitar a sua visualização, tentando reduzir o tamanho dos arcos e evitar o cruzamento dos mesmos. Embora o *Graphviz* funcione como um sistema autônomo, seus programas e bibliotecas foram concebidos para serem incorporados por outras aplicações.

6.3 Funcionalidades

Nesta seção iremos apresentar a interface gráfica e as funcionalidades implementadas na ferramenta *ADEX*. A ferramenta continua em desenvolvimento, para este trabalho foram implementadas as funcionalidades mínimas da interface gráfica, somente o essencial para apoiar o processo de validação do fluxo excepcional.

A Figura 6.1 mostra a barra de ferramentas que fica na parte superior da tela, onde estão os botões de ação . A seguir apresentamos cada um desses botões descrevendo a ação associada:

- Open XMI: exibe uma caixa de diálogo para selecionar o arquivo XMI que será aberto. Após a seleção do arquivo o conteúdo do mesmo é mostrado em um editor localizado no canto superior direito da aba XMI (Figura 6.1).

- Generate Graphs: está é a principal ação da ferramenta, o conteúdo do arquivo XMI mostrado no editor e passado como entrada para o algoritmo que utiliza os recursos

[1] Graphviz:http://www.graphviz.org/

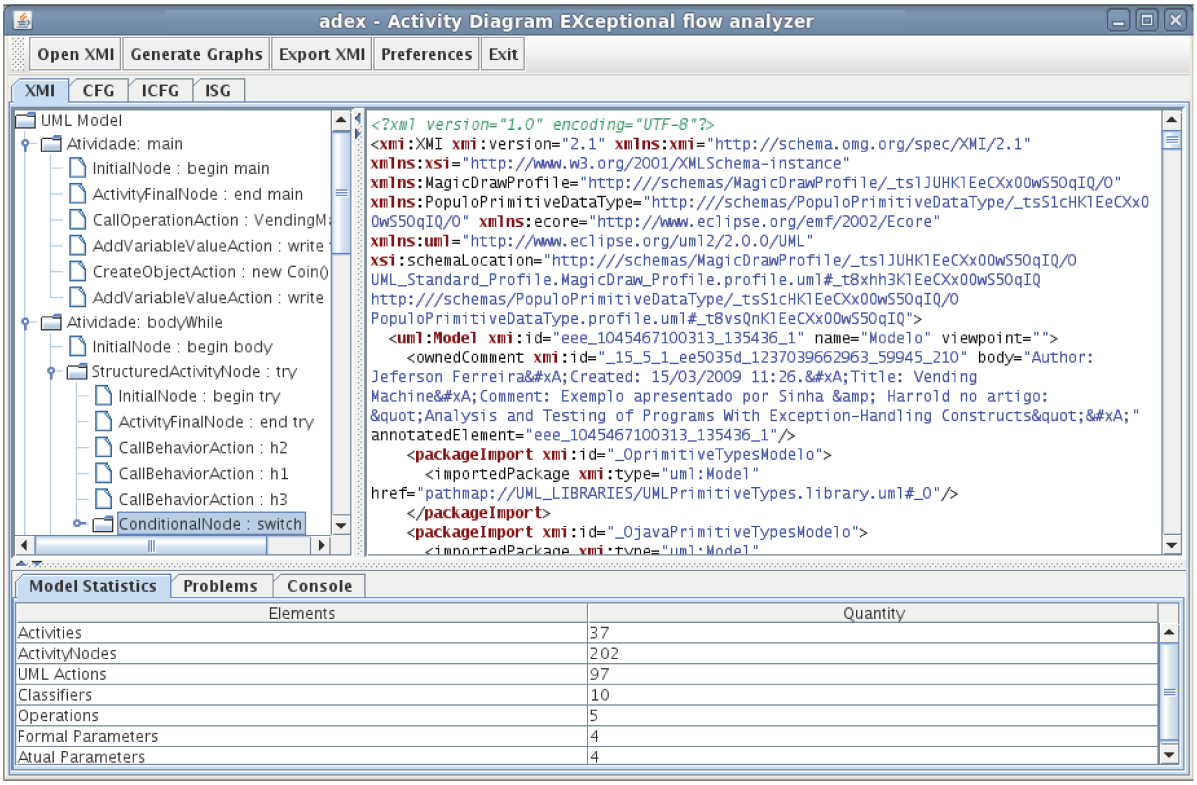


Figura 6.1: Ferramenta ADEX - Aba XMI - Model Statistics

do framework EMF para instanciar os objetos persistidos no arquivo XMI. O resultado é uma instância de um objeto Java que contém o modelo UML completo. Este modelo é utilizado como entrada nos algoritmos de transformação, dando origem a todos os grafos apresentados neste trabalho (GA, GFC, GFCI e GRI). A partir desse momento é possível navegar pelas abas do aplicativo para visualizar os grafos e consultar as informações sobre nós, exceções e tratadores de exceção.

- Export XMI: exibe uma caixa de diálogo para o usuário indicar o diretório e o nome do arquivo XMI que será gerado através da persistência do resultado da análise. Como todos os grafos utilizados pela ferramenta também foram modelados através do framework EMF, é possível exportá-los para o formato XMI. Assim, podemos obter como saída da ferramenta um modelo, que serve como um contêiner para todos os grafos gerados durante as transformações, contendo todas as informações de fluxo de dados e de propagação de exceções.

- Preferences: exibe todas as propriedades que o usuário poderá customizar para a geração das imagens dos grafos. Como por exemplo, o formato do nó (círculo, elipse ou retângulo), a cor, a fonte, as informações que serão exibidas no interior

do nó (definições, usos, condições de guarda). Esta funcionalidade ainda está em desenvolvimento.

- Exit: saída do aplicativo.

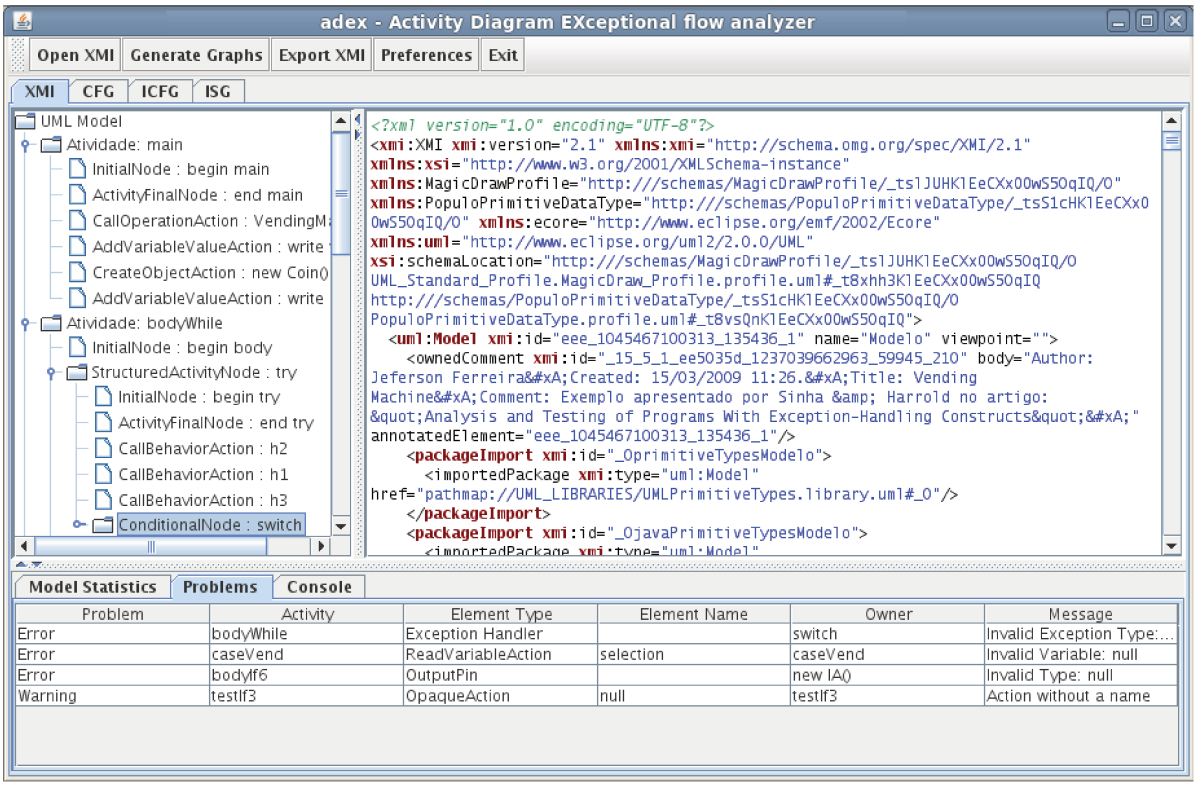


Figura 6.2: Ferramenta ADEX - Aba XMI - Problems

A Figura 6.1 também mostra a aba XMI, a qual está divida em três regiões. A primeira região é mostrada no canto superior esquerdo da tela, onde é exibida uma árvore que mostra a hierarquia do modelo UML. O nó raiz representa o próprio modelo UML, os diagramas de atividade e as atividades estruturadas são apresentados como nós internos da árvore. As ações e nós de atividades contidas nos diagramas são apresentadas como folhas. A segunda região é a área do editor, onde o usuário pode consultar o conteúdo do arquivo XMI. A terceira região fica na parte inferior da tela onde são apresentadas mais três abas:

- Model Statistics: apresenta uma estatística sobre o modelo UML, com as seguintes informações: i) Activities: número de diagramas de atividade contidos no modelo; ii) ActivityNodes: número total de nós de atividades ou ações contidas no modelo; iii) UML Actions: número de ações UML contidas no modelo; iv) Classifiers: número

de classes definidas pelos diagramas de classe; v) Operations: número de operações definidas nas classes; vi) Formal Parameters: número de parâmetros formais contidos nas definições das operações; vii) Actual Parameters: número de parâmetros atuais contidos nas chamadas das operações.

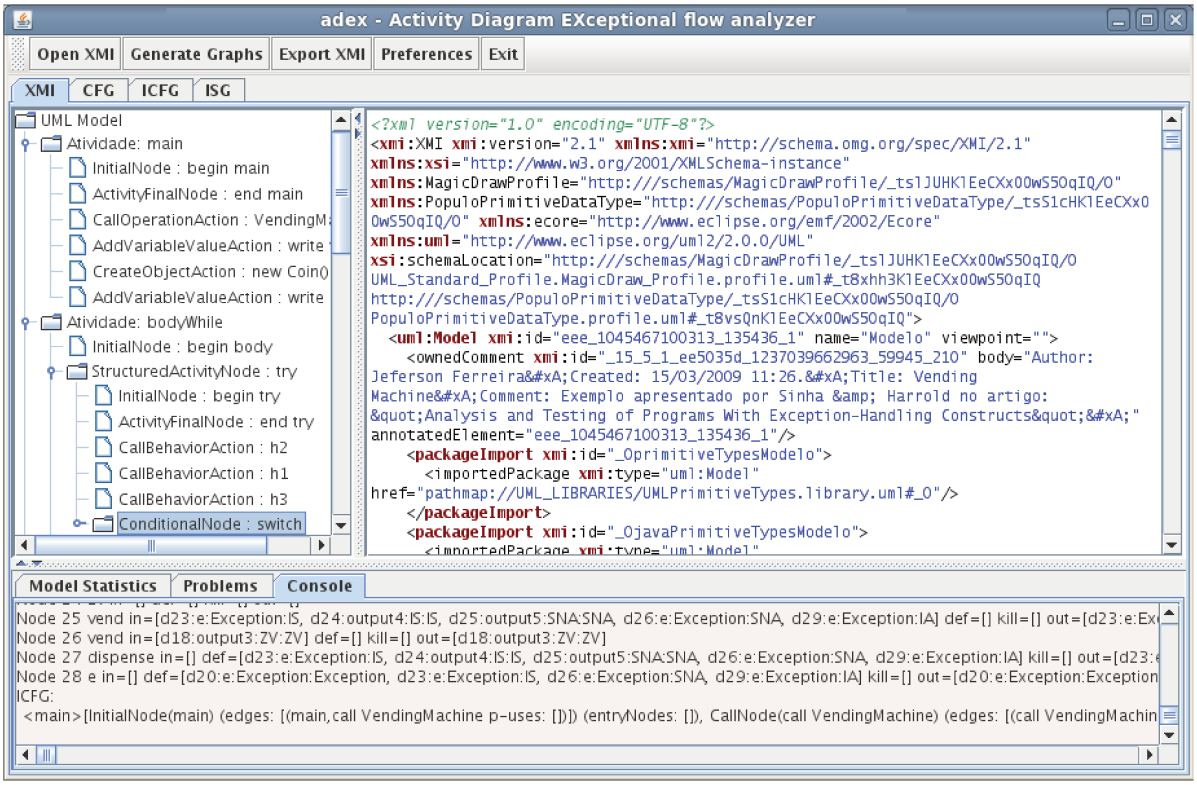


Figura 6.3: Ferramenta ADEX - Aba XMI - Console

- Problems: apresenta uma tabela contendo os problemas encontrados durante a geração dos grafos (Figura 6.2). O objetivo desta tabela é guiar o usuário na solução de erros ou inconsistências inseridas no modelo UML. As informações apresentadas são as seguintes: i) Problem: identifica o tipo do problema, que pode ser um erro ou uma advertência. Somente no caso da ocorrência de um erro a geração dos grafos é abortada; ii) Activity: identifica o nome do diagrama de atividades onde ocorreu o problema; iii) Element Type: identifica o tipo do elemento do diagrama onde o problema ocorreu, por exemplo: pino de entrada, pino de saída, tipo de ação; iv) Element Name: identifica o nome do elemento; v) Owner: identifica o nome do contêiner onde o elemento está inserido. Esta informação é útil para identificar uma ação aninhada dentro de um atividade estruturada, neste caso o nome da atividade estruturada é exibido neste campo; vi) Message: informa a mensagem de erro ou

advertência.

- Console: apresenta um log das operações realizadas após o botão Generate Graphs ser pressionado(Figura 6.3). Toda a saída de tela é direcionada para esta aba. Caso ocorra algum erro durante a execução dos algoritmos ele pode ser consultado aqui.

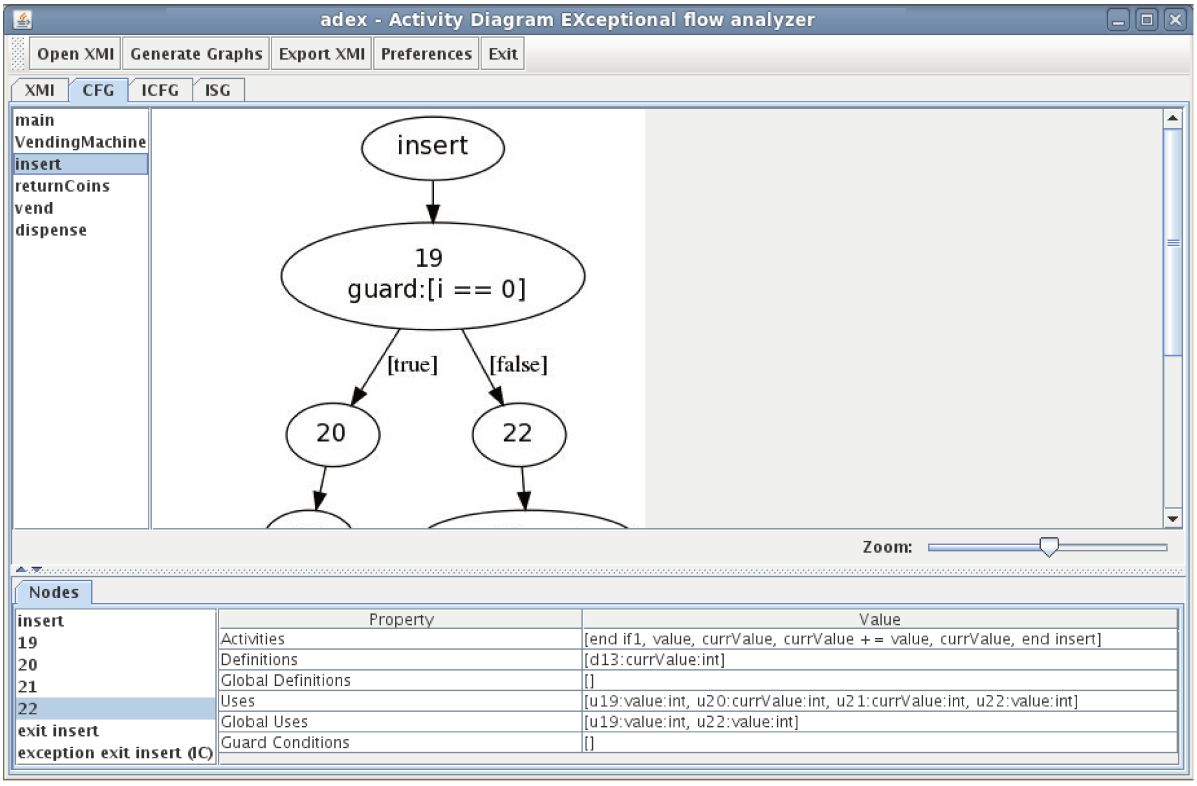
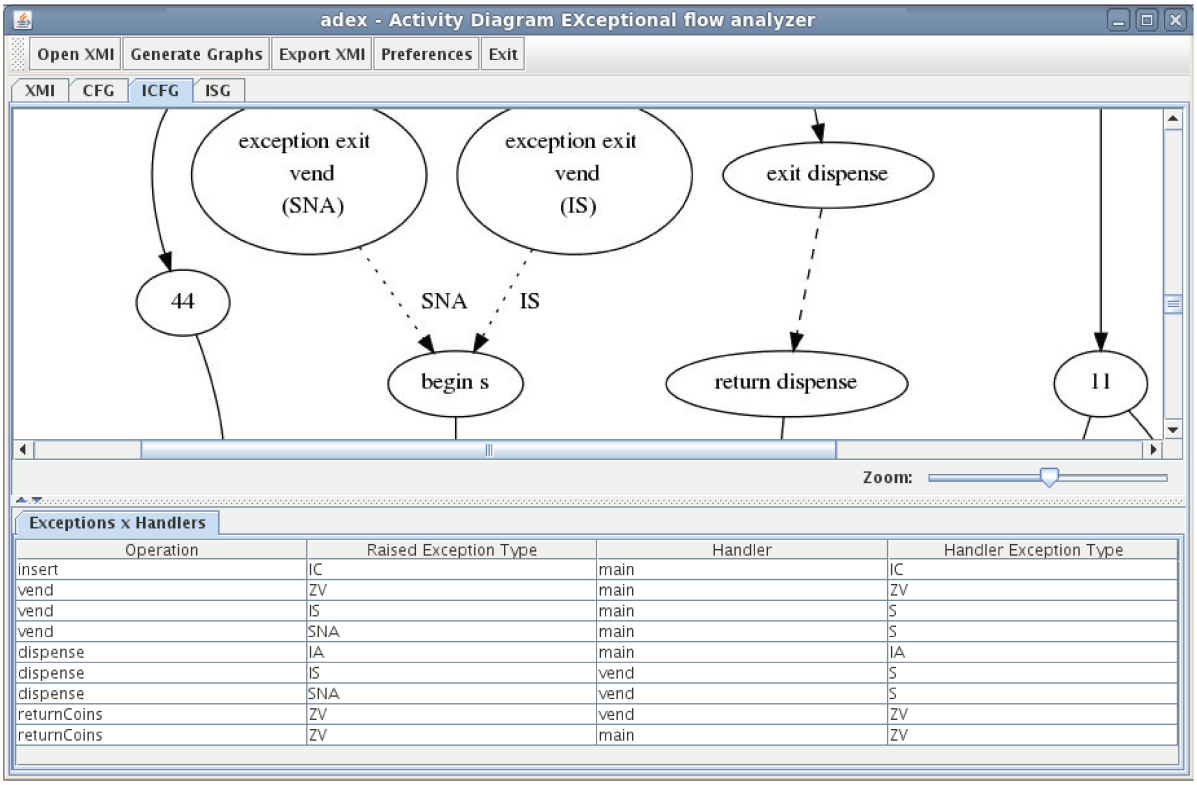


Figura 6.4: Ferramenta ADEX - Aba CFG

A Figura 6.4 mostra a aba CFG, esta aba está divida em quatro regiões. A primeira região é mostrada no canto superior esquerdo da tela, onde é exibida uma lista com todas as operações contidas no modelo. A medida que o usuário, seleciona uma das operações a imagem do GFC é exibida. A segunda região é a área de visualização do GFC, o usuário pode aumentar ou diminuir a imagem através do botão Zoom. A terceira região da tela fica na parte inferior esquerda, sempre que um GFC é selecionado, uma lista contendo os nós do GFC é apresentada. Quando o usuário seleciona um dos nós as informações sobre o nó são exibidas. A quarta região da tela apresenta uma tabela contendo as informações sobre o nó selecionado. Nesta tabela são apresentadas as seguintes informações: i) Activities: apresenta o conjunto de nós do GA contidos no GFC; ii) Definitions: apresenta o conjunto de definições do nó; iii) Global Definitions: apresenta o conjunto de definições globais do

nó; iv) Uses: apresenta o conjunto de usos do nó; v) Global Uses: apresenta o conjunto de usos globais do nó; vi) Guard Conditions: apresenta as condições de guarda do nó, caso ele seja condicional.



Operation	Raised Exception Type	Handler	Handler Exception Type
insert	IC	main	IC
vend	ZV	main	ZV
vend	IS	main	S
vend	SNA	main	S
dispense	IA	main	IA
dispense	IS	vend	S
dispense	SNA	vend	S
returnCoins	ZV	vend	ZV
returnCoins	ZV	main	ZV

Figura 6.5: Ferramenta ADEX - Aba ICFG

A Figura 6.5 mostra a aba ICFG, esta aba está divida em duas regiões. A primeira região é mostrada na parter superior da tela, onde é exibido o GFCI do modelo. O usuário pode aumentar ou diminuir a imagem do GFCI através do botão Zoom. A segunda região fica na parte inferior da tela, onde é apresentada uma tabela contendo as informações sobre quais exceções são lançadas e qual é o tratador de exceção responsável por capturar cada exceção. Nesta tabela são apresentadas as seguintes informações: i) Operation: mostra o nome da operação onde ocorre uma exceção; ii) Raised Exception Type: apresenta o tipo de exceção lançada na operação; iii) Handler: mostra o nome da operação que possui o tratador de exceção; iv) Handler Exception Type: mostra o tipo de exceção definida no tratador de exceções. Esta análise é útil para identificar exceções que não possuem um tratador de exceção capaz de tratá-las, ou então identificar um tratador de exceções que não é compatível com nenhuma exceção lançada. Pode também identificar aqueles tratadores de exceção muito genéricos que acabam capturando muitos tipos de exceção.

A Figura 6.6 mostra a aba ISG, esta aba está divida em três regiões. A primeira

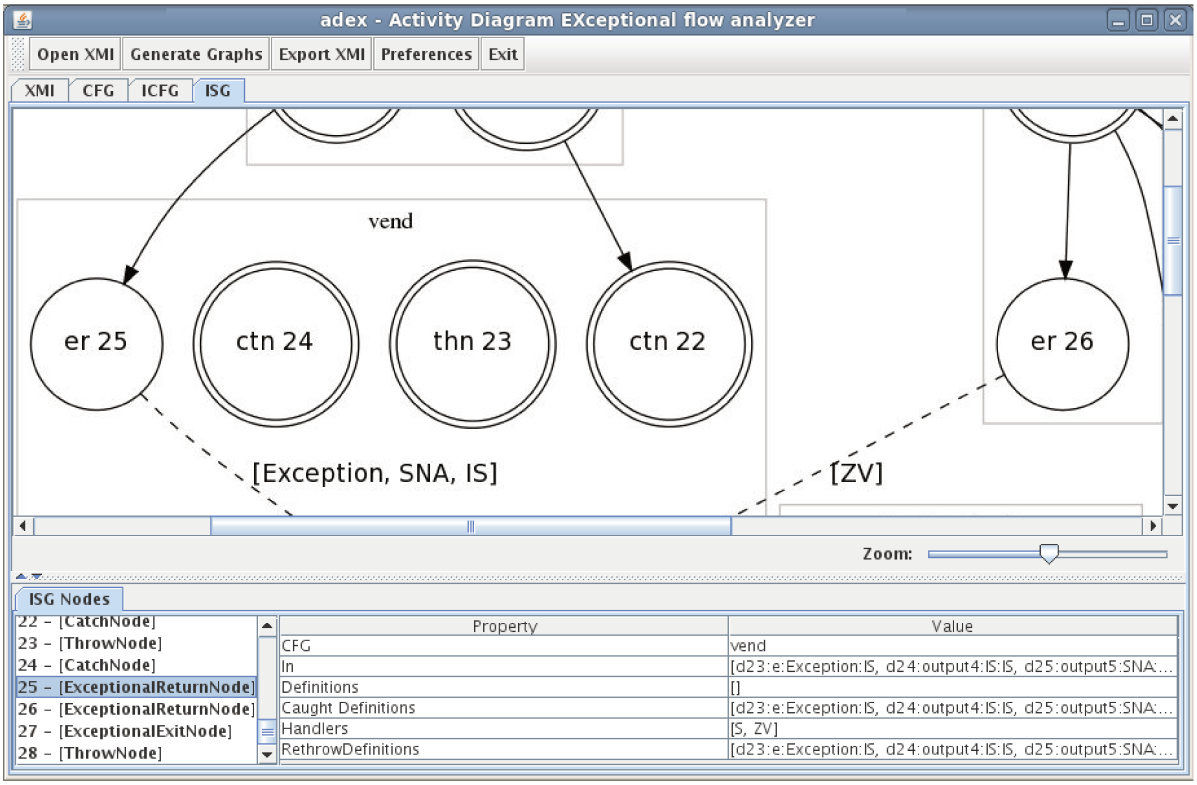


Figura 6.6: Ferramenta ADEX - Aba ISG

região é mostrada na parte superior da tela, onde é exibido o GRI do modelo. O usuário pode aumentar ou diminuir a imagem do GRI através do botão Zoom. A segunda região da tela fica na parte inferior esquerda da tela onde é mostrada uma lista contendo os nós pertencentes ao GRI. Quando o usuário seleciona um desses nós, as suas informações são exibidas em uma tabela. A terceira região da tela apresenta uma tabela contendo as informações sobre o nó selecionado. Nesta tabela são apresentadas as informações de acordo com o tipo do nós selecionado, no exemplo da Figura 6.6, o nó selecionado é do tipo *ExceptionalReturnNode*. Para este tipo de nó são apresentadas as seguintes informações: i) CFG: mostra o nome da operação que o nó pertence; ii) In: apresenta o conjunto de definições que alcançam o nó de retorno excepcional, estas definições foram propagadas pela operação chamada; iii) Definitions: apresenta o conjunto de definições do nó; iv) Caught Definitions: mostra quais as definições de objetos de exceção são capturadas dentro da operação; v) Handlers: apresenta o conjunto de tipos de exceções tratadas pela operação; vi) RethrowDefinitions: apresenta o conjunto de definições que foram relançadas pela operação.

6.4 Aspectos de Implementação e Arquitetura

A ferramenta *ADEX* foi implementada na linguagem de programação Java e será distribuída como software livre sobre a licença *Eclipse Public License* (EPL). A escolha dessa licença deve-se ao fato da ferramenta incorporar muitos recursos do *Eclise Modeling Framework* (EMF) que é distribuído sobre a mesma licença. Um exemplo é a manipulação de um arquivo XMI, o framework EMF permite instanciar os objetos persistidos no arquivo XMI de acordo com a implementação do metamodelo da UML 2.

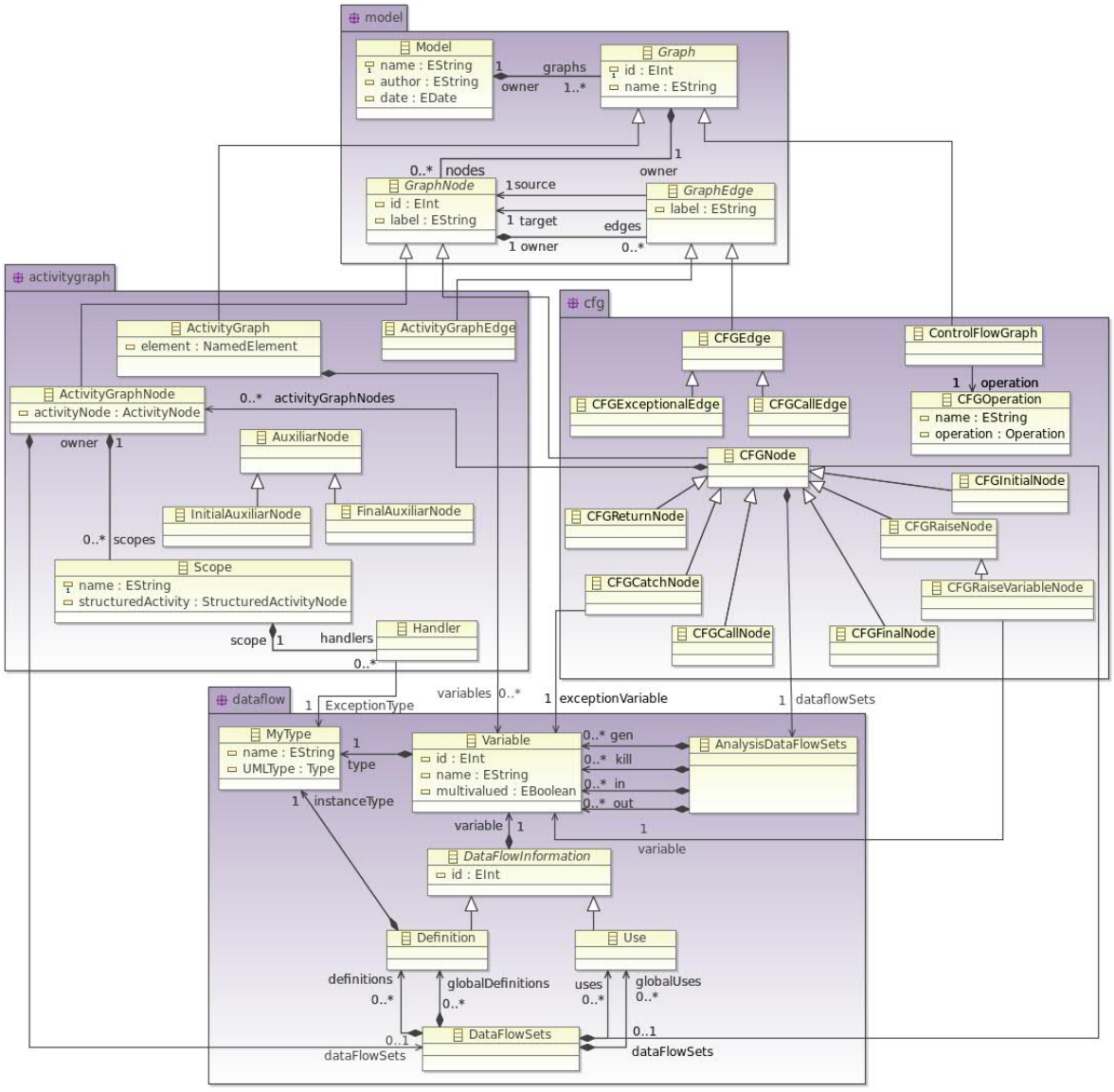


Figura 6.7: Modelo Ecore utilizado na geração do código-fonte

O metamodelo utilizado pela ferramenta é baseado no framework EMF e faz parte do projeto *Model Development Tools* (MDT/UML2) [Ecli 10]. A implementação completa do metamodelo da UML 2.x foi o fator decisivo para optar pelo uso do framework EMF na inspeção e transformação dos modelos, ao invés de utilizar as ferramentas convencionais de transformação.

A Figura 6.7 mostra o modelo Ecore utilizado pelo framework EMF para a geração do código-fonte dos grafos. Nesta Figura são mostrados quatro pacotes: i) model: define o modelo que servirá de contêiner para os grafos juntamente com as classes abstratas que modelam um grafo genérico. ii) activitygraph: define as classes utilizadas na modelagem do GA, a descrição completa destas classes foi apresentada na seção 4.1; iii) cfg: define as classes utilizadas na modelagem do GFC, a descrição completa destas classes foi apresentada na seção 4.2; iv) dataflow: define as classes utilizadas para modelar as informações de fluxo de dados, a descrição completa destas classes foi apresentada na seção 4.1.

Para a visualização dos grafos são utilizados os recursos da ferramenta *Graphviz*, portanto, a instalação do *Graphviz* é um pré-requisito para o funcionamento da ferramenta. A *Graphviz* é capaz de interpretar uma descrição textual de um grafo e gerar uma imagem gráfica correspondente. A ferramenta ADEX gera as descrições textuais dos grafos de acordo com a linguagem DOT. Os grafos textuais são lidos pela *Graphviz* e convertidos em imagens PNG, que são exibidas pela ferramenta *ADEX*.

Capítulo 7

Estudos de Caso

Neste capítulo apresentamos alguns estudos de caso para validação da abordagem proposta. O capítulo está dividido em duas seções. A primeira seção mostra o estudo de caso de uma versão simplificada de um Sistema de Mineração apresentado por Sloman e Kramer [Slom 87]. Este exemplo é utilizado para ilustrar a validação do fluxo de exceções propagadas entre os componentes de um sistema tolerante à falhas. A segunda seção mostra o estudo de caso de uma Máquina de Vendas apresentada por Sinha e Harrold [Sinh 99]. Este exemplo é utilizado para a validação do fluxo excepcional gerado a partir do modelo de tratamento de exceções do diagrama de atividades. Além disso, este exemplo justifica a utilização de uma análise de fluxo de dados interprocedimental para apoiar a construção do fluxo excepcional completo, pois são usadas variáveis que são definidas com referências de objetos de exceção. Assim, uma operação lança um exceção a partir de uma destas variáveis para ser capturada e relançada em outra operação, exigindo que as definições ultrapassem as fronteiras da operação.

7.1 Sistema de Mineração

O processo de extração de minerais da mina além de produzir água, também libera gás metano no ambiente. Por isso, além da extração de minerais, o sistema de mineração deve ainda manter o nível de água acumulada em um reservatório e também controlar o nível de metano no interior da mina. O sistema é composto por três subsistemas: a)*MineralExtractorController* (MEC), responsável pela extração de minerais; b) *PumpController* (PC), responsável por drenar a mina e manter o nível de água do reservatório; c) *AirExtractorController* (AEC), responsável pelo controle do nível de metano. Quando a água atinge um nível alto, a bomba é ligada para drenar o reservatório até a água atingir um nível apropriado. Um sensor de fluxo de água é capaz de detectar o fluxo de água na tubulação. Porém, a bomba está situada no subterrâneo e por razões de segurança, não

deve iniciar ou continuar a executar quando a quantidade de metano na mina exceder o limite de segurança. Para controlar o nível de metano, existe um sistema de controle do exaustor de ar que monitora o nível de metano. Quando o nível de metano está elevado, o sistema liga o exaustor de ar para remover o ar carregado de metano do interior da mina. Todo o sistema é controlado a partir da superfície através de um console que deve atender a qualquer emergência identificada pelo sistema.

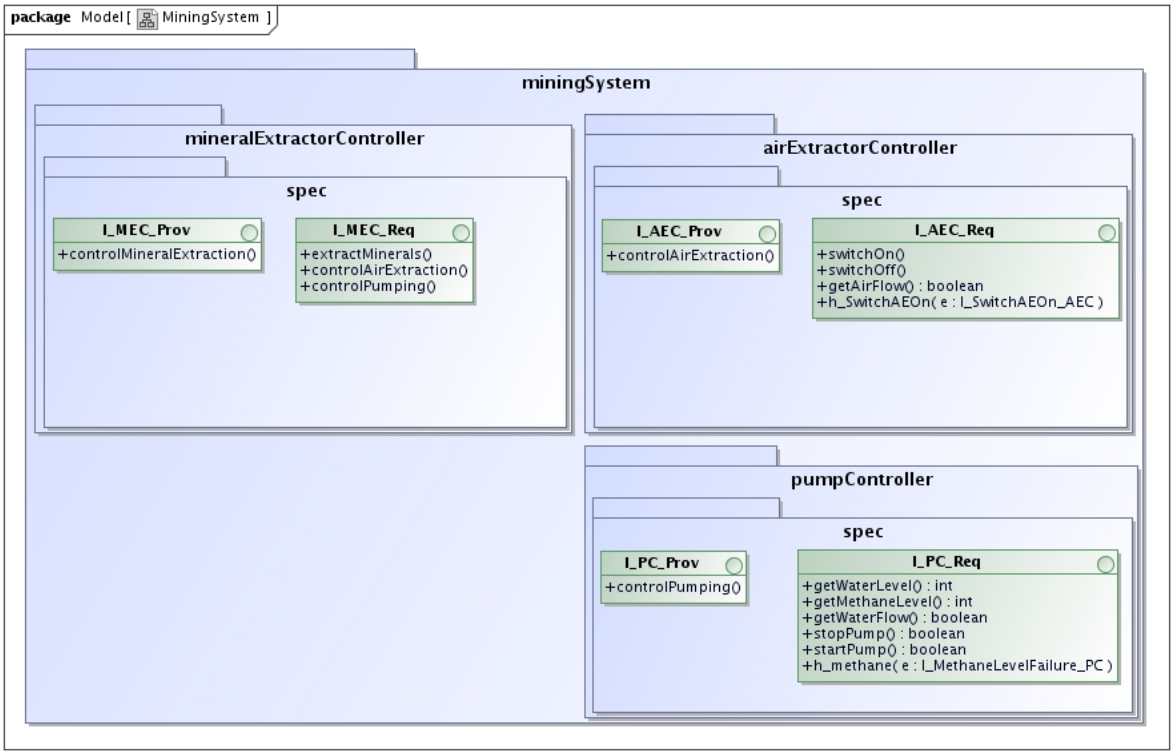


Figura 7.1: Diagrama de Classes do Sistema de Mineração

A Figura 7.1 mostra um diagrama de classes que contém o pacote principal *MiningSystem* e os pacotes dos três componentes que representam os subsistemas: *MineralExtractorController* (MEC), *PumpController* (PC) e *AirExtractorController* (AEC). Cada componente possui um pacote de especificação *(spec)* onde estão modeladas as interfaces providas e requeridas pelos componentes.

O modelo completo do Sistema de Mineração pode ser consultado no seguinte endereço: *http://www.students.ic.unicamp.br/~ra066147/MiningSystem/MiningSystem.html*

Este estudo de caso está dividido em duas seções, na Seção 7.1.1 iremos validar o fluxo excepcional interno de cada componente de software, verificando se as exceções internas são tratadas pelos próprios componentes. Na Seção 7.1.2 iremos validar o fluxo

excepcional externo aos componentes, demonstrando quais são as exceções propagadas e capturadas pelos componentes.

A Figura 7.2 mostra a hierarquia das exceções de cada um dos componentes. Neste exemplo, as exceções com prefixo *I* são consideradas internas e as exceções com prefixo *E* são consideradas externas.

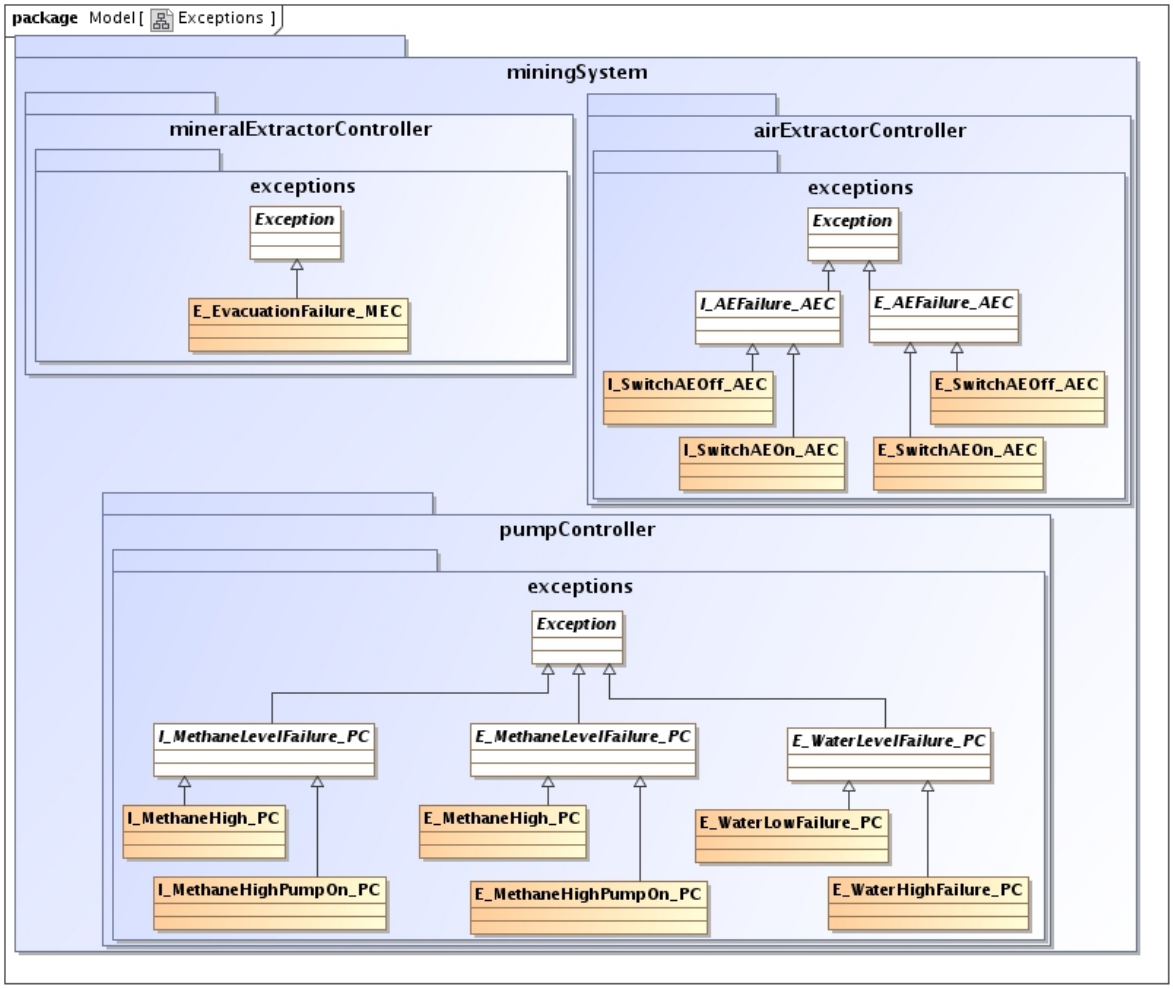
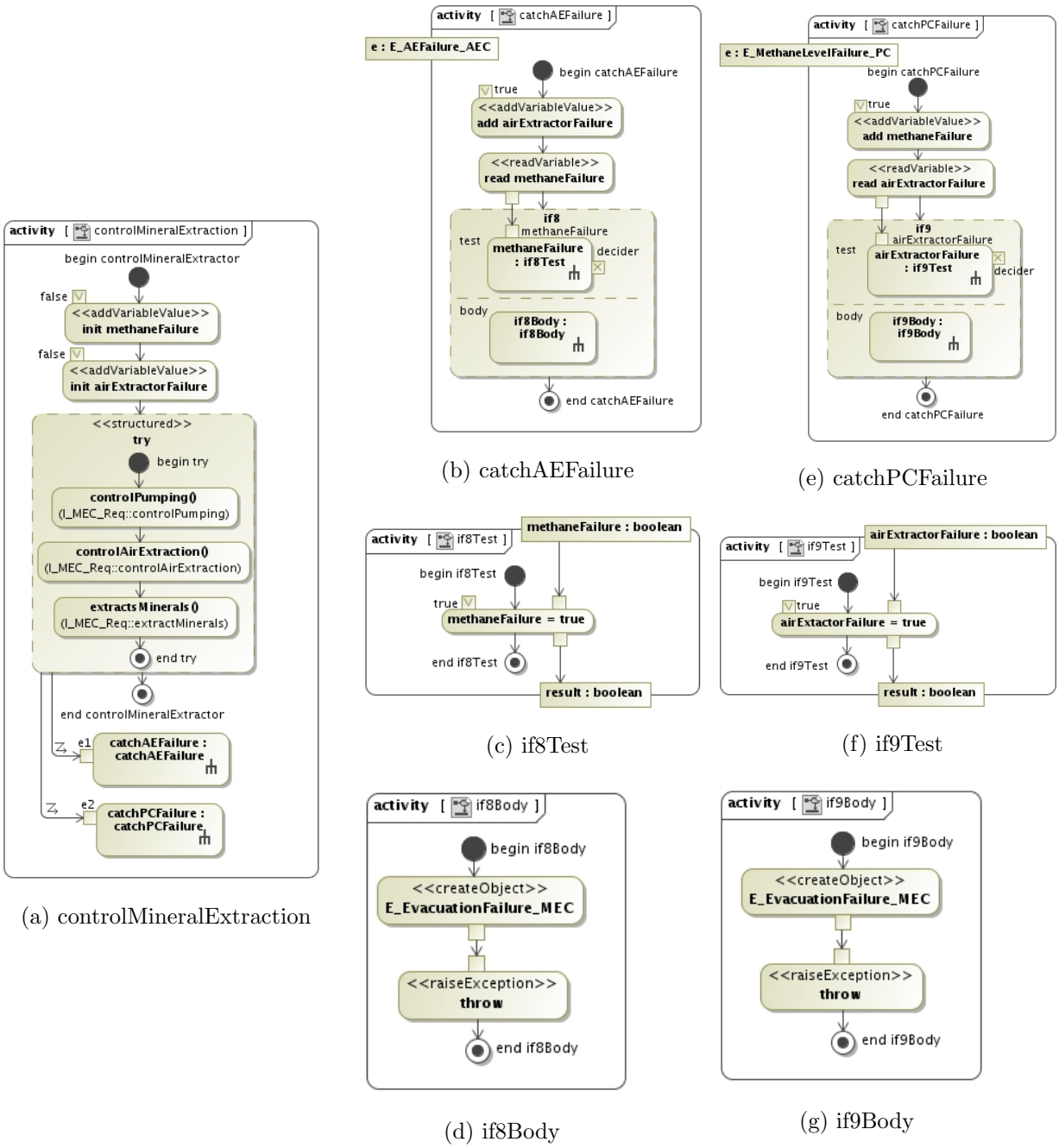


Figura 7.2: Hierarquia das Exceções

7.1.1 Fluxo de Controle Intraprocedimental

Neste estudo de caso o componente *MEC* desempenha o papel principal, pois utiliza os serviços dos outros dois componentes. O serviço *controlMineralExtraction()* provido pelo componente *MEC* é o ponto de partida do nosso exemplo. A Figura 7.3 apresenta um conjunto de atividades utilizadas para modelar o comportamento da operação *controlMi-*

neralExtraction().



(a) controlMineralExtraction

(b) catchAEFailure

(e) catchPCFailure

(c) if8Test

(f) if9Test

(d) if8Body

(g) if9Body

Figura 7.3: Operação controlMineralExtraction()

A Figura 7.3a exibe a atividade principal da operação. Esta atividade possui um nó protegido *try* que contém três ações do tipo *CallOperationAction*, responsáveis pela chamada aos serviços da interface requerida. O nó protegido *try* possui dois tratadores de

exceção: *catchAEFailure* e *catchPCFailure* modelados respectivamente pelas atividades das Figuras 7.3b e 7.3e. O primeiro tratador de exceções é do tipo *E_AEFailure_AEC*, capaz de tratar todas as exceções externas do componente *AEC* (*E_SwitchAEOff_AEC* e *E_SwitchAEOn_AEC*). O segundo tratador de exceções é do tipo *E_MethaneLevelFailure_PC*, capaz de tratar parte das exceções externas do componente *PC* (*E_MethaneHigh_PC* e *E_MethaneHighPumpOn_PC*). Quando o nível da água e o nível de metano estiverem elevados e as exceções do tipo *E_AEFailure_AEC* e *E_MethaneLevelFailure_PC* ocorrerem simultaneamente uma exceção do tipo *E_EvacuationFailure_MEC* é lançada, como mostram as atividades *if8Body* (Figura 7.3d) e *if9Body* (Figura 7.3g).

Figura 7.4: GA da operação controlMineralExtraction()

Para efetuar a validação do fluxo excepcional da operação, primeiro temos que converter o conjunto de atividades utilizado para modelar o comportamento da operação em um grafo de atividades (GA). Durante essa conversão cada atividade é processada mapeando seus nós e arestas para o grafo. Nesta primeira transformação os nós inicial e final do

diagrama são mapeados respectivamente para os nós inicial e final do GA, cada ação do diagrama de atividades é mapeada para um nó do GA. No caso das atividades estruturadas, nós e arestas auxiliares são inseridos para ligar as seções da atividade estruturada, tornando possível a representação do fluxo de controle. Para realizar as transformações são utilizados os Algoritmos 1 e 2, apresentados na Seção 4.2.

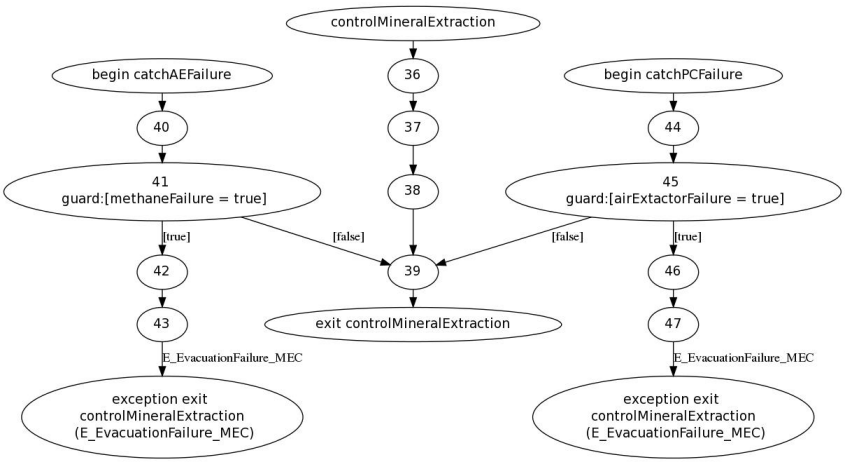


Figura 7.5: GFC da operação controlMineralExtraction()

A obtenção das informações de fluxo de dados do modelo ocorrem durante o processo de geração do GA. Cada ação do DA é avaliada e todas as informações referentes ao fluxo de dados são capturadas de acordo com o mapeamento apresentado na Tabela 2.2 apresentada na Seção 2.2.2. As definições e usos das variáveis capturadas são adicionadas ao nó de atividade correspondente.

A Figura 7.4 exibe o grafo de atividades da operação *controlMineralExtraction()* gerado a partir do conjunto de atividades da Figura 7.3.

Na sequência o GA gerado no passo anterior é convertido em um Grafo de Fluxo de Controle (GFC). Cada nó do GFC corresponde a um bloco básico de atividades do GA, no qual não existe desvio do fluxo de controle, exceto no final do bloco. Com exceção de alguns nós do GA que acabam gerando nós exclusivos no GFC, com por exemplo os nó 36, 37 e 38, correspondentes aos nós de chamada das operações: *controlPumping()*, *controlAirExtraction()* e *extractsMinerals()*. Os conjuntos de definição e uso das variáveis de cada nó do GA, são transportados para os nós do GFC. Para gerar o GFC utilizamos o Algoritmo 3 apresentado na Seção 4.2.

Após a geração do GFC é preciso completar o grafo com o fluxo de controle excepcional, inserindo arestas que representem o fluxo da exceção do ponto que foi lançada até o ponto que será tratada dentro do GFC. Caso a exceção não seja tratada dentro da própria operação é necessário incluir um nó de saída excepcional no GFC. O primeiro passo para

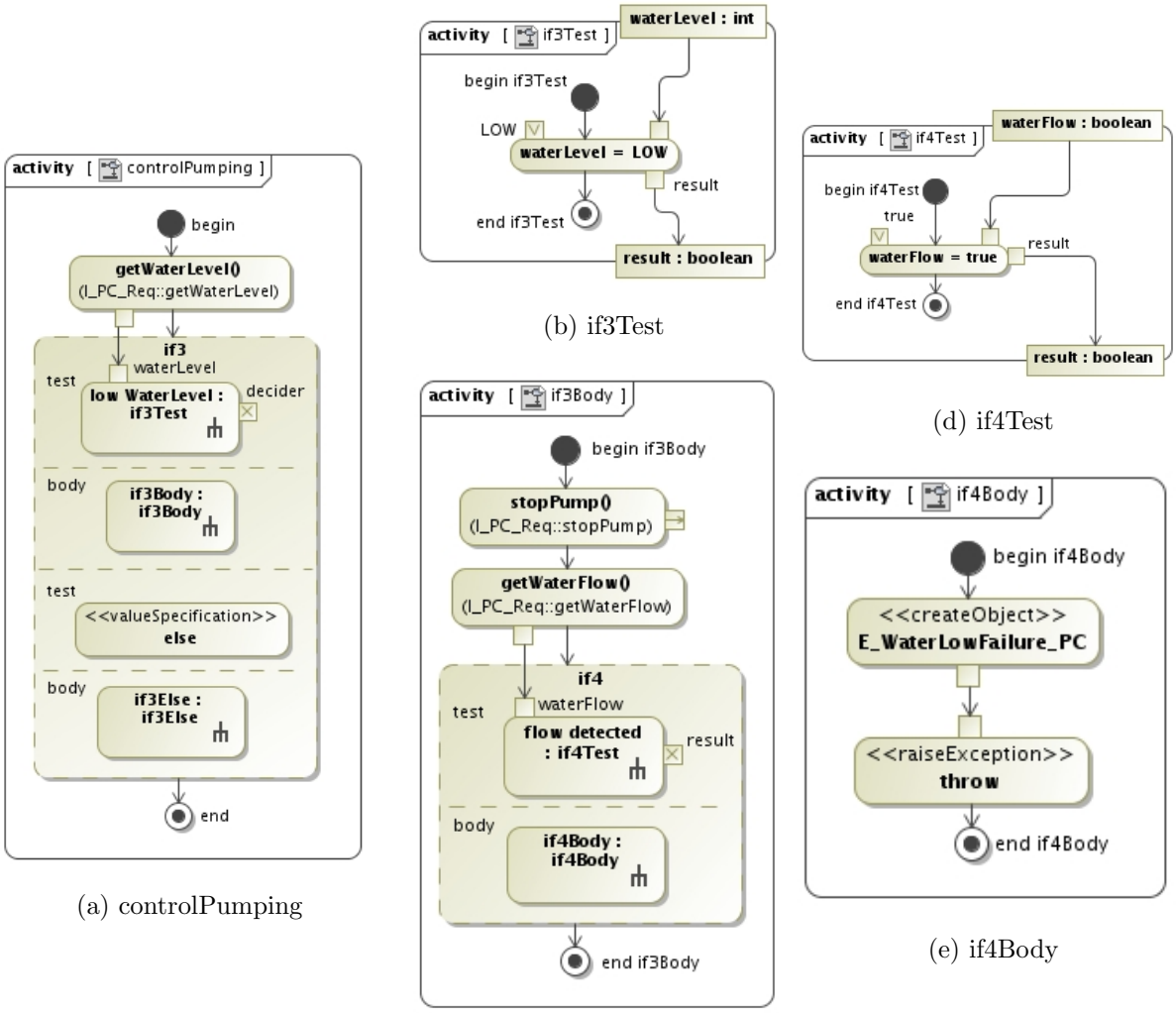
a criação do fluxo de exceções intraprocedimental é a identificação dos tipos de exceção que podem ser lançados por um determinado nó.

No caso da operação *controlMineralExtraction()*, existem duas atividades do tipo *RaiseExceptionAction*, que são responsáveis pelo lançamento das exceções (Figuras 7.3d e 7.3g). Sempre que uma ação deste tipo é encontrada no GA, durante a geração do GFC, um nó exclusivo para esta ação é criado no GFC. Conforme a regra apresentada na Seção 4.3 este nó pode ser um *CFGRaiseNode* ou um *CFGRaiseVariableNode*. O que determina se um nó do GA, que contém uma ação do tipo *RaiseExceptionAction*, será mapeado para um nó do GFC do tipo *CFGRaiseNode* ou *CFGRaiseVariableNode*, é o tipo de ação que antecede a ação *RaiseExceptionAction* no DA. Se a ação for do tipo *ReadVariableAction* é criado um nó do tipo *CFGRaiseVariableNode*, pois a ação *ReadVariableAction* lê o conteúdo de uma variável é passa este valor como entrada para a ação que lança a exceção, o que implica na dependência da definição da variável para a identificação do tipo de exceção que pode ser lançada. Caso a ação que fornece o objeto de exceção seja do tipo *CreateObjectAction*, não existe a dependência de uma variável. Assim, o nó criado no GFC será do tipo *CFGRaiseNode*, pois o tipo de exceção lançada pode ser determinado diretamente pelo tipo do objeto criado pela ação *CreateObjectAction*.

No caso das atividades *if8Body* e *if9Body* foram criados dois nós do tipo *CFGRaiseNode* (nós 43 e 47 do GFC da Figura 7.5) que dispensam a análise de fluxo de dados para identificar o tipo de exceção lançada. A partir desses nós é iniciado o fluxo excepcional da operação, no caso de uma exceção que é tratada dentro da própria operação, uma aresta de exceção irá ligar este nó com o nó correspondente ao tratador de exceção compatível. No caso da exceção não ser tratada, uma aresta de exceção irá ligar este nó com o nó correspondente à saída excepcional da operação.

Após identificação dos tipos de exceção precisamos inferir quais são os possíveis tratadores de exceção que podem capturar essas exceções. Para isso, utilizamos as informações do contexto das regiões interruptíveis, das atividades estruturadas e das informações dos tratadores de exceção extraídas do modelo. Como as atividades *if8Body* e *if9Body* lançam exceções do tipo *E_EvacuationFailure_MEC* e não existe nenhum tratador de exceções compatível com este tipo de exceção, foram criados dois nós de saída excepcional. A Figura 7.5 exibe o grafo de fluxo de controle completo da operação *controlMineralExtraction()*.

A seguir iremos apresentar as atividades das demais operações, apresentando sempre o grafo de atividades e o grafo de fluxo de controle resultante das transformações.



(a) controlPumping

(b) if3Test

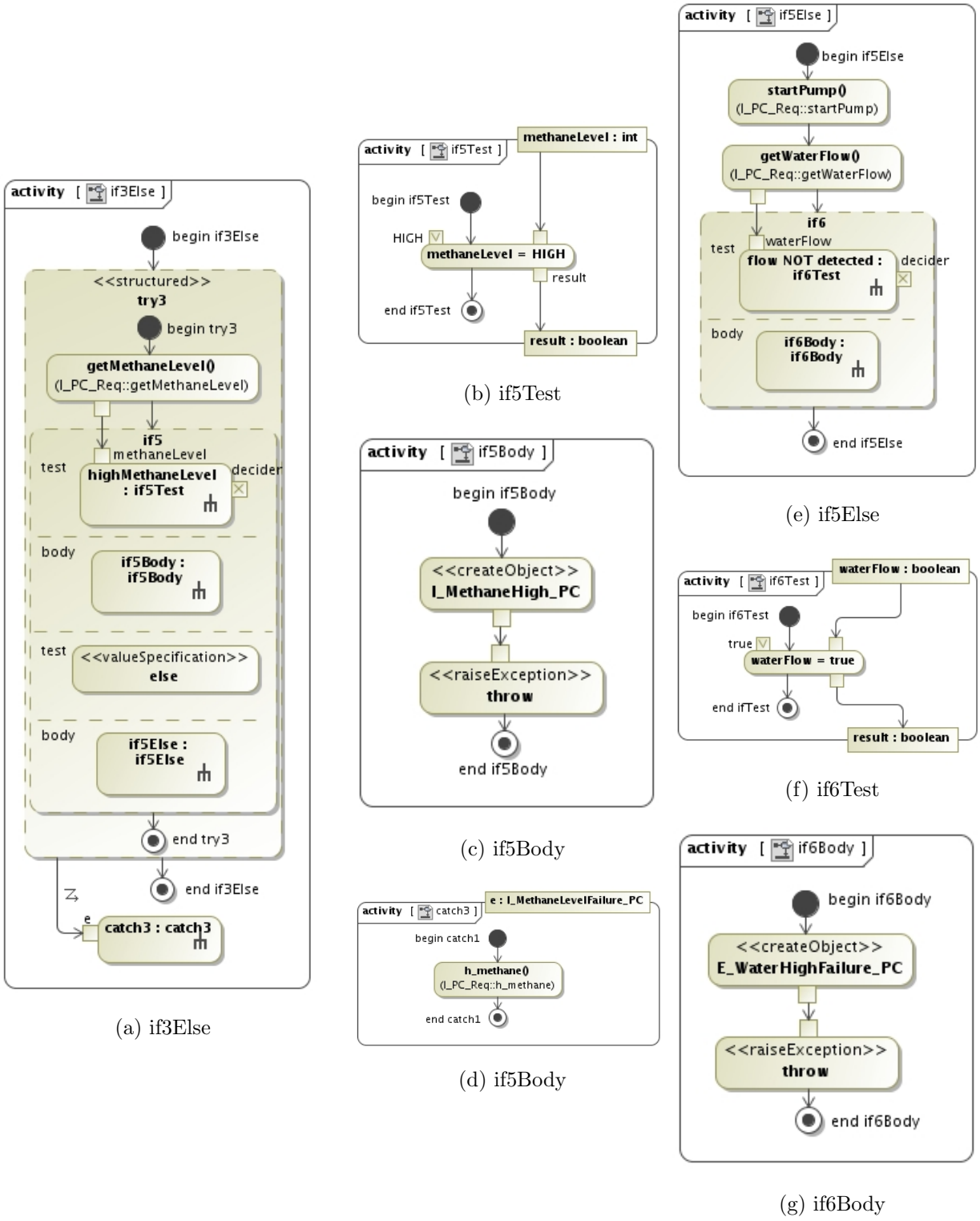
(c) if3Body

(d) if4Test

(e) if4Body

Figura 7.6: Operação controlPumping() - Parte 1

As Figuras 7.6 e 7.7 exibem um conjunto de atividades utilizadas para modelar o comportamento da operação *controlPumping()*. Esta operação representa o comportamento normal do componente *PC*. A Figura 7.6a exibe a atividade principal da operação. Esta atividade possui um nó condicional *if3* com duas cláusulas condicionais (if-else). A primeira cláusula testa se o nível da água está baixo, neste caso a atividade *if3Body* (Figura 7.6c) é executada. Caso contrário a segunda cláusula é satisfeita e a atividade *if3Else* (Figura 7.7a) é executada. A atividade *if3Body* desliga a bomba de água e verifica o sensor de fluxo de água para certificar-se que a bomba foi desligada. Caso o desligamento da bomba tenha falhado, o componente lança uma exceção externa do tipo *E_WaterLowFailure_PC* (Figura 7.6e).



(a) if3Else

(b) if5Test

(c) if5Body

(d) if5Body

(e) if5Else

(f) if6Test

(g) if6Body

Figura 7.7: Operação controlPumping() - Parte 2

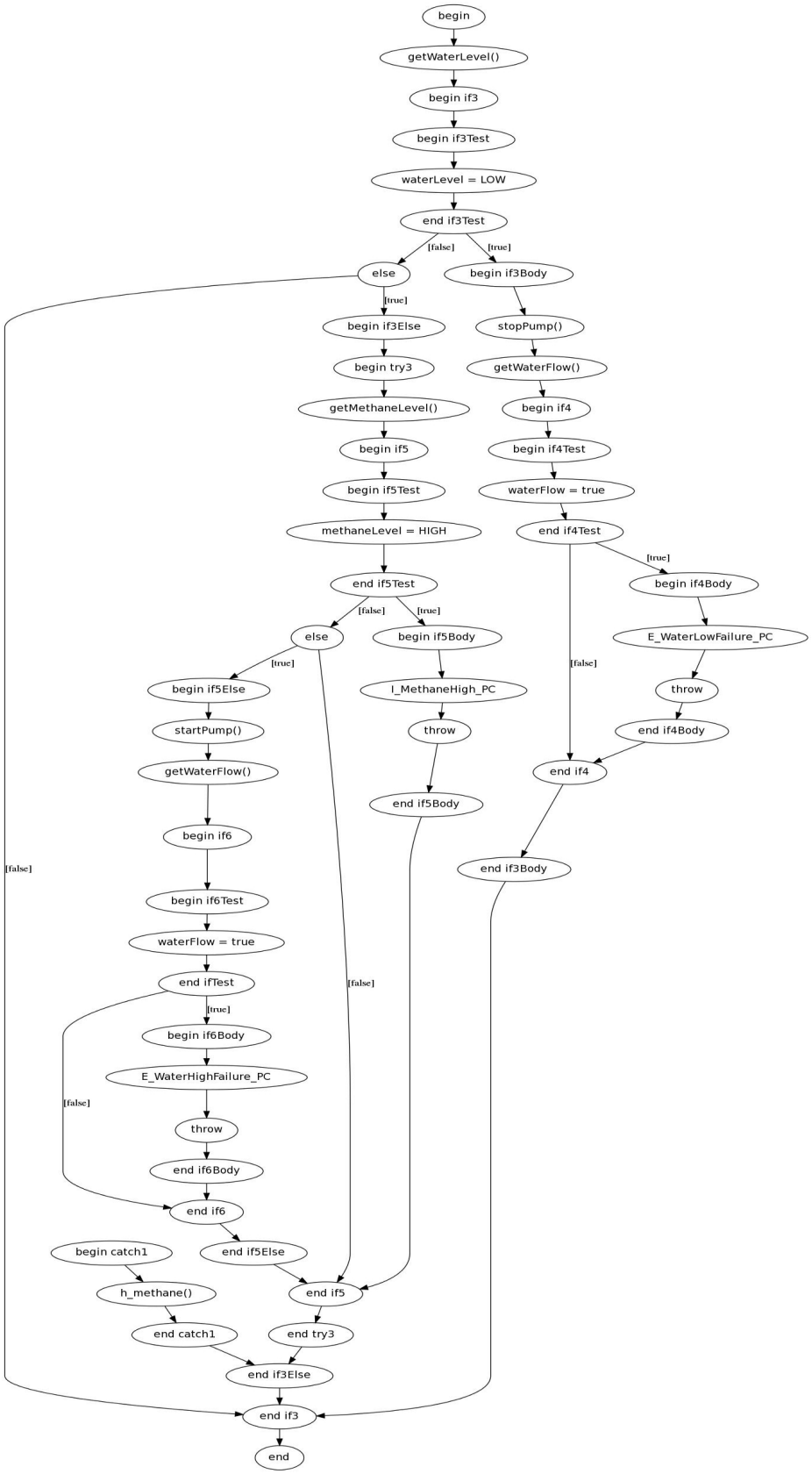


Figura 7.8: GA da operação controlPumping()

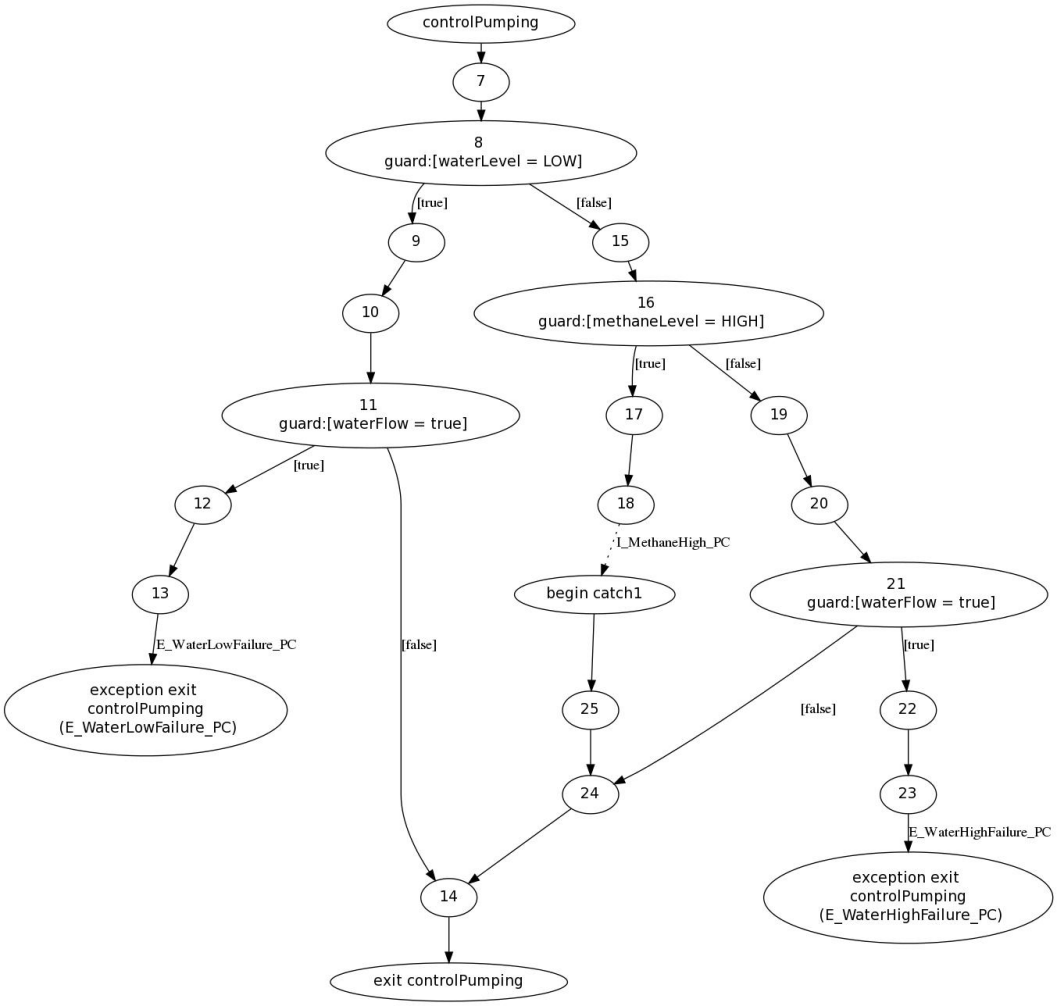
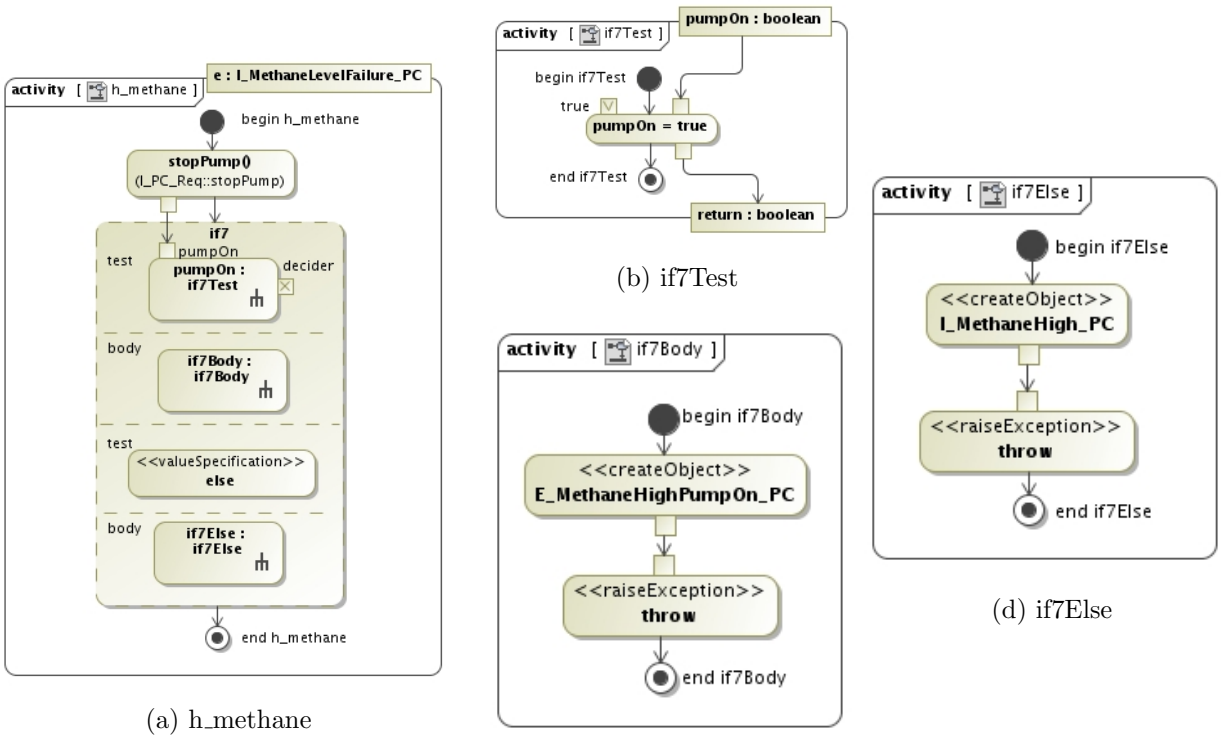


Figura 7.9: GFC da operação controlPumping()

A Figura 7.7a exibe a atividade *if3Else*. Esta atividade possui um nó condicional *if3* com duas cláusulas condicionais (if-else). A primeira cláusula testa se o nível da água está baixo, neste caso a atividade *if3Body* (Figura 7.6c) é executada. Caso contrário a segunda cláusula é satisfeita e a atividade *if3Else* (Figura 7.7a) é executada. A atividade *if3Body* desliga a bomba de água e verifica o sensor de fluxo de água para certificar-se que a bomba foi desligada. Caso o desligamento da bomba tenha falhado, o componente lança uma exceção externa do tipo *E_WaterLowFailure_PC* (Figura 7.6e). A atividade *if3Else* possui também um nó protegido *try3* que possui um tratador para exceções internas do tipo *I_MethaneLevelFailure_PC*. A atividade *catch3* (Figura 7.7d) representa o corpo do tratador de exceções, que contém uma chamada para a operação *h_methane()*.



(a) h_methane

(b) if7Test

(c) if7Body

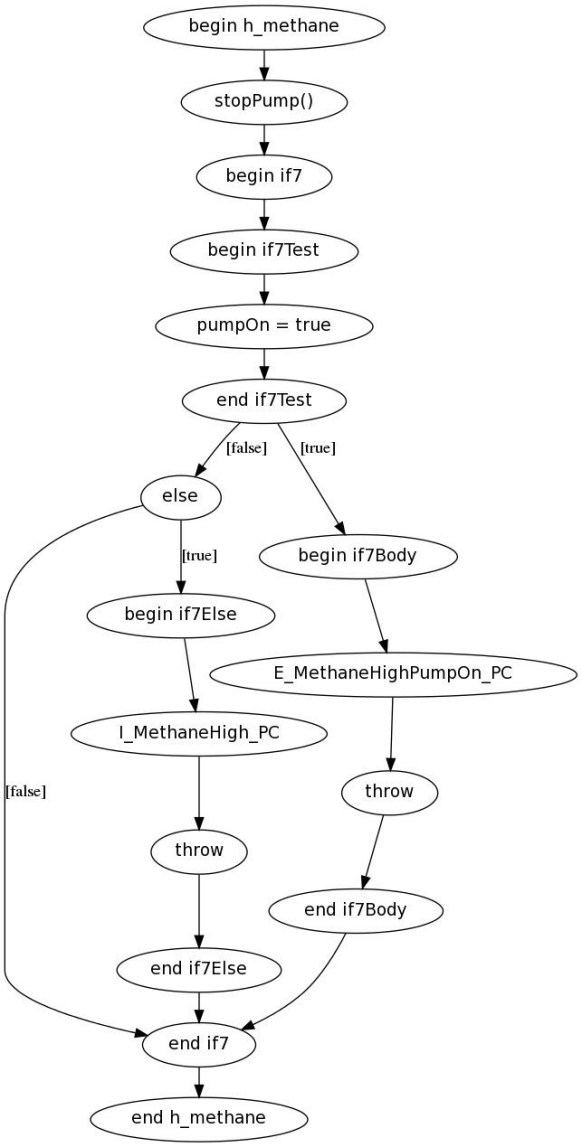
(d) if7Else

Figura 7.10: Operação h_methane()

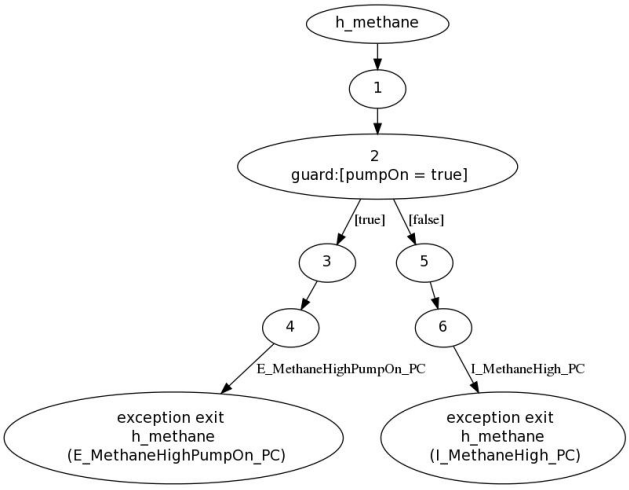
A Figura 7.10 exibe um conjunto de atividades utilizadas para modelar o comportamento da operação *h_methane()*. Esta operação representa o comportamento excepcional do componente *PC* na presença de uma exceção causada pelo nível elevado de metano. A Figura 7.10a exibe a atividade principal da operação, quando o nível de metano esta elevado a primeira providência e desligar a bomba de água. Na sequência esta atividade utiliza o nó condicional *if7* para verificar se a bomba continua ligada. Caso a bomba continue ligada, a primeira cláusula do nó condicional é satisfeita e a atividade *if7Body* é executada. Esta atividade é responsável pelo lançamento de uma exceção externa do tipo *E_MethaneHighPumpOn_PC*. Caso a bomba esteja desligada, a segunda cláusula do nó condicional é satisfeita e a atividade *if7Else* é executada lançando uma exceção interna do tipo *I_MethaneHigh_PC*.

A Figura 7.8 exibe o grafo de atividades da operação *controlPumping()* gerado a partir do conjunto de atividades utilizados para modelar o comportamento da operação.

A Figura 7.9 exibe o grafo de fluxo de controle da operação *controlPumping()* gerado a partir do grafo de atividades da Figura 7.8. Os nós de saída excepcional mostram os tipos de exceções propagadas pela operação *controlPumping()*, neste caso a operação propaga as exceções *E_WaterLowFailure_PC* e *E_WaterHighFailure_PC*.



(a) GA

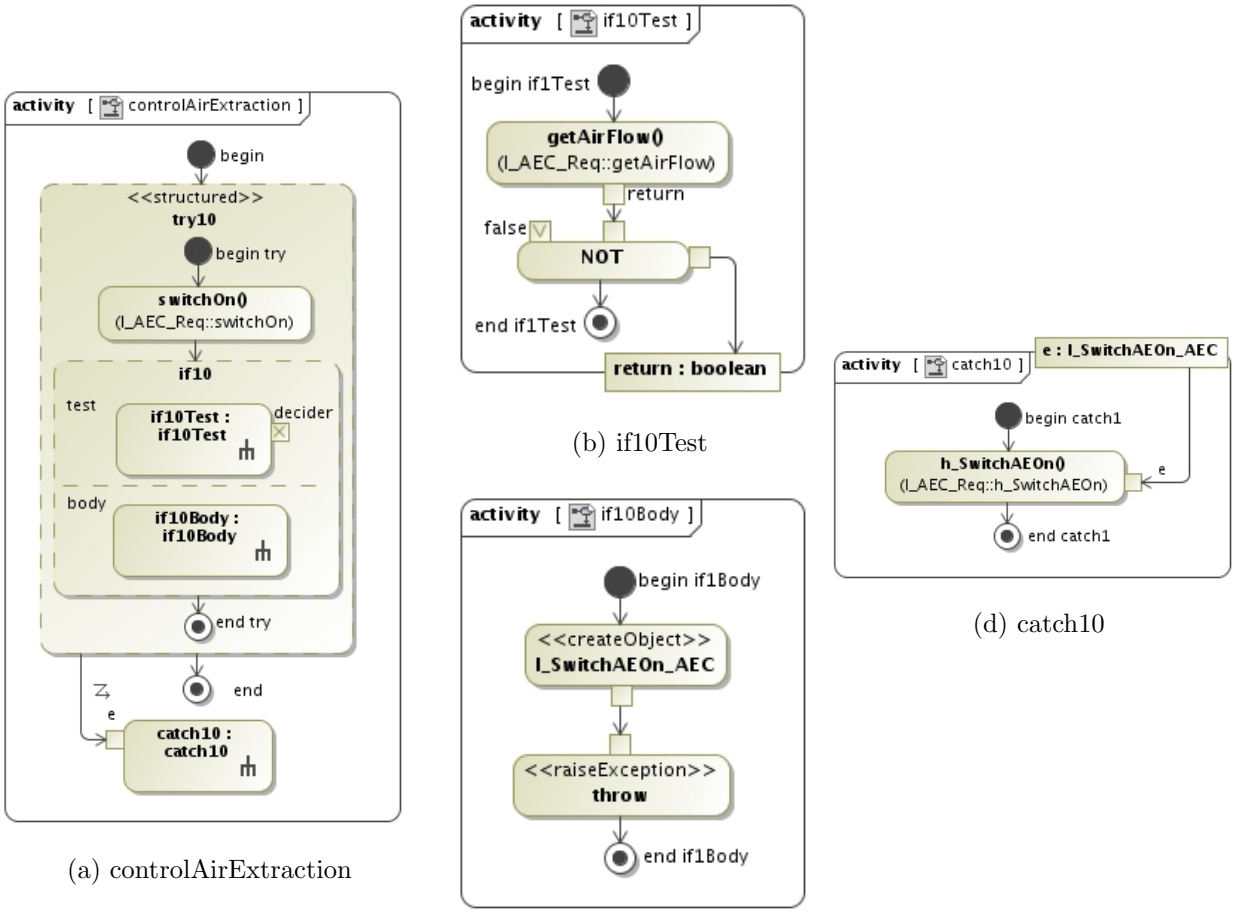


(b) GFC

Figura 7.11: Grafos da operação h_methane()

A Figura 7.11a exibe o grafo de atividades da operação *h_methane()* gerado a partir do conjunto de atividades utilizados para modelar o comportamento da operação, mostrado na Figura 7.10.

A Figura 7.11b exibe o grafo de fluxo de controle da operação *h_methane()* gerado a partir do agrupamento dos nós do grafo de atividades da Figura 7.11a. Os nós de saída excepcional mostram os tipos de exceções propagadas pela operação *h_methane()*, neste caso a operação propaga as exceções *E_MethaneHighPumpOn_PC* e *I_MethaneHigh_PC*.



(a) controlAirExtraction

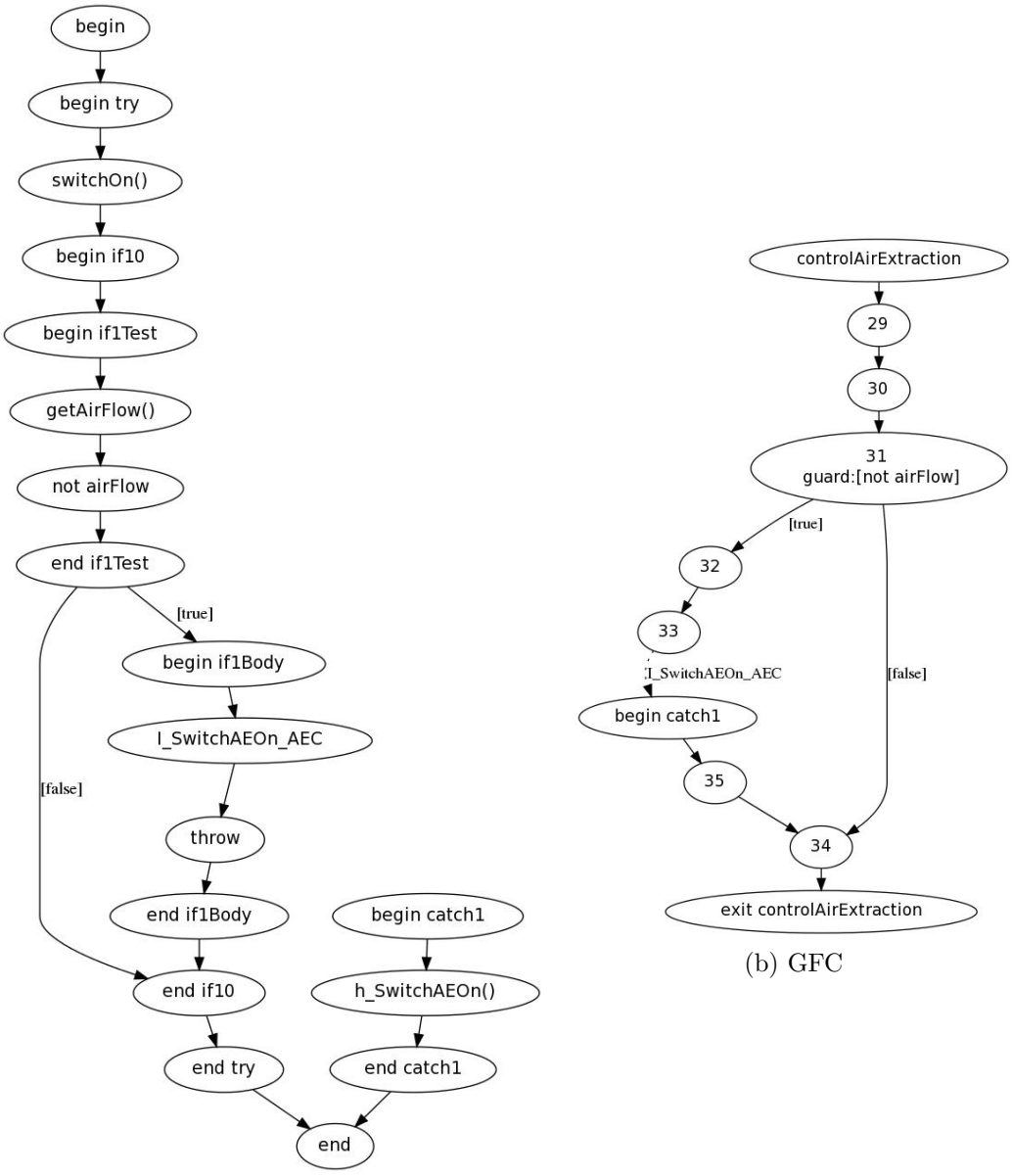
(b) if10Test

(c) if10Body

(d) catch10

Figura 7.12: Operação controlAirExtraction()

A Figura 7.12 exibe um conjunto de atividades utilizadas para modelar o comportamento da operação *controlAirExtraction()*. Esta operação representa o comportamento normal do componente *AEC*. A Figura 7.12a exibe a atividade principal da operação. A primeira ação desta atividade é ligar o exaustor de ar (*switchOn()*), na sequência o nó condicional *if10* testa o sensor do fluxo de ar para verificar se o exaustor está funcionando. Caso não exista fluxo de ar, a cláusula condicional é satisfeita e a atividade *if10Body* (Figura 7.12c) é executada. Esta atividade lança uma exceção interna do tipo *I_SwitchAEOn_AEC*. A atividade *controlAirExtraction* também possui um nó protegido *try10* que contém um tratador para exceções internas do tipo *I_SwitchAEOn_AEC*. A atividade *catch10* (Figura 7.12d) representa o corpo do tratador de exceções que realiza uma chamada para a operação *h_SwitchAEOn()*. Esta atividade modela o comportamento excepcional da operação *controlAirExtraction()*.



(a) GA

(b) GFC

Figura 7.13: Grafos da operação controlAirExtraction()

A Figura 7.13a exibe o grafo de atividades da operação *controlAirExtraction()* gerado a partir do conjunto de atividades da Figura 7.12.

A Figura 7.13b exibe o grafo de fluxo de controle da operação *controlAirExtraction()* gerado a partir do grafo de atividades. Podemos observar que a exceção interna lançada pela operação (I_SwitchAEOn_AEC) e capturada por um tratador de exceções da própria operação, representado no grafo pelo nó *begin catch1*.

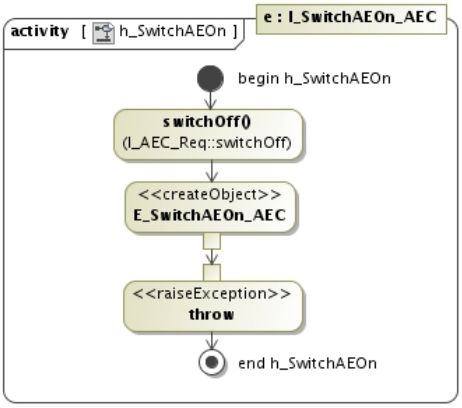
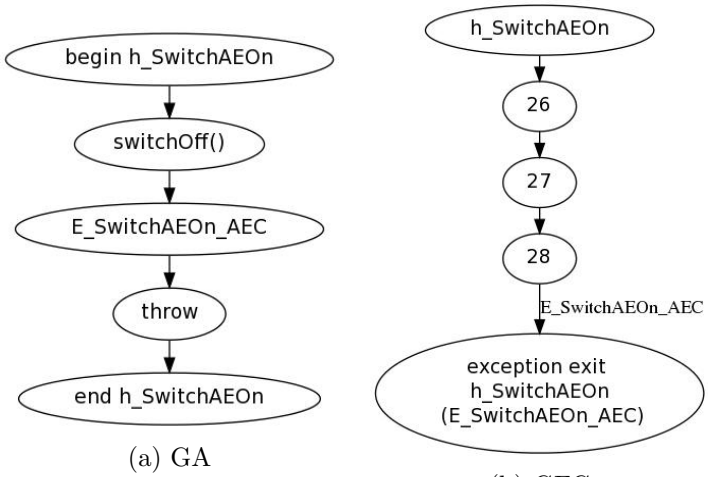


Figura 7.14: Operação h_SwitchAEOn()

A Figura 7.14 exibe a atividade utilizada para modelar o comportamento da operação *h_SwitchAEOn()*. Esta operação representa o comportamento excepcional do componente *AEC* na presença de uma exceção causada pelo falha do exaustor de ar. Esta atividade lança uma exceção externa do tipo *E_SwitchAEOn_AEC*.



(a) GA

(b) GFC

Figura 7.15: Grafos da operação h_SwitchAEOn()

A Figura 7.15a exibe o grafo de atividades da operação *h_SwitchAEOn()* gerado a partir do diagrama de atividades da Figura 7.14. A Figura 7.15b exibe o grafo de fluxo de controle da operação *h_SwitchAEOn()* gerado a partir do grafo de atividades da Figura 7.15b. Como a operação não possui nenhum tratador de exceções apropriado para capturar a exceção do tipo *E_SwitchAEOn_AEC*, um nó de saída excepcional e gerado no GFC.

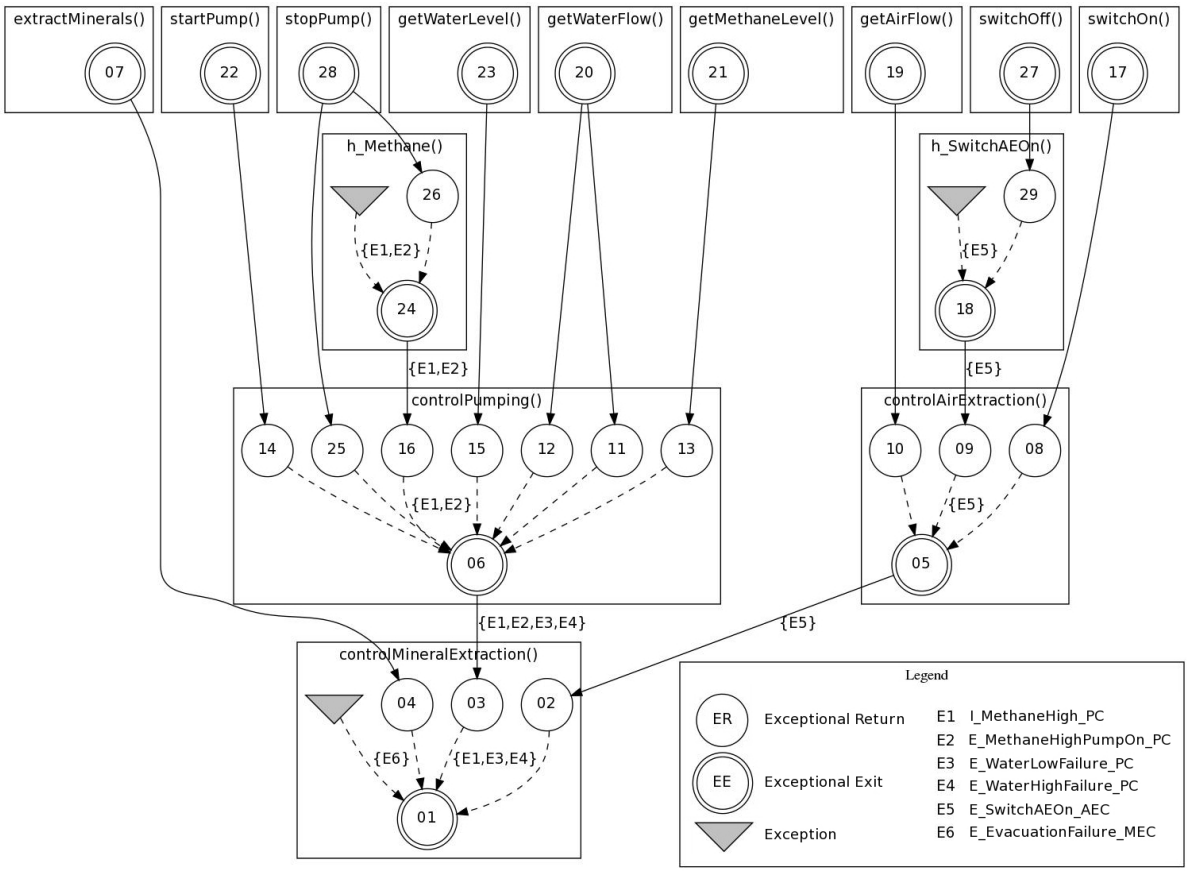


Figura 7.16: GRI do Sistema de Mineração

7.1.2 Fluxo de Controle Interprocedimental

Após a construção do GFC de cada operação contida no modelo o próximo passo é a geração do Grafo de Fluxo de Controle Interprocedimental (GFCI). Neste processo cada GFC é percorrido para encontrar os *nós de chamada* de operação. Para cada *nó de chamada* identificado, uma *aresta de chamada* é criada conectando o *nó de chamada* com o *nó de entrada* da operação correspondente. Na sequência, uma *aresta de retorno* é criada para conectar o *nó de saída* da operação com um *nó de retorno*. Para construção do GFCI utilizamos o Algoritmo 4, apresentado na Seção 5.1.

O grafo resultante do Algoritmo 4 ainda é incompleto, pois falta incluir o fluxo excepcional interprocedimental, que é demonstrado pelas arestas de exceção interprocedimentais. Estas arestas ligam um nó de saída excepcional de uma operação com o nó do respectivo tratador de exceções contido em outra operação. Para obter o fluxo completo é necessário propagar as definições das variáveis de exceção entre os procedimentos através de uma análise de fluxo de dados interprocedimental, para facilitar esta tarefa um novo grafo é

gerado a partir do GFCI. O último grafo gerado é o Grafo Resumido da Interface (GRI), o processo de geração é mostrado na Seção 5.2.

A Figura 7.16 mostra o grafo gerado a partir dos nós de interface do GFCI. O grafo mostrado é parcial, para facilitar a análise do fluxo excepcional foram considerados somente os nós de saída excepcional *(ee)* e os nós de retorno excepcional *(er)*. Por conveniência, foram incluídos triângulos invertidos para representar a ocorrência de exceções não tratadas. Cada triângulo tem uma aresta de exceções não capturadas que faz a ligação com o nó de saída excepcional da operação. Estas arestas são rotuladas com o tipo de exceção que está sendo propagada. Após a execução da análise de fluxo de dados no GRI, é possível identificar quais são os tipos de exceção que são propagados ao longo da pilha de chamadas.

Um exemplo de falha de modelagem mostrado neste grafo é a exceção *E1*. Esta exceção foi gerada pela operação *h_Methane()* e passou pelo nó de saída excepcional (24), o que indica que não foi tratada nesta operação. A exceção passou também pelo nó de retorno excepcional (16) da operação *controlPumping()* e alcançou o nó de saída excepcional (06). Neste ponto já identificamos uma inconsistência, pois a exceção *E1* é uma exceção interna (*I_MethaneHigh_PC*) e não deveria ser propagada para fora do componente *PC*. Porém a exceção foi propagada até o nó de saída excepcional (01) da operação controlMineralExtraction(). Como se trata de uma exceção interna, com certeza não será esperada pelo software que for utilizar o serviço *controlMineralExtraction()*.

A falha da modelagem está na atividade *if?Else* (Figura 7.10d); o tipo da exceção deveria ser *E_MethaneHigh_PC*, uma exceção externa, que poderia ser propagada para fora do componente *PC* e seria capturada pelo tratador de exceções *catchPCFailure* da operação *controlMineralExtraction()*.

Tabela 7.1: *Exceções lançadas X Tratadores de Exceções - Sistema de Mineração*

Operação	**Exceção Lançada**	**Tratador de Exceções**	**Exceção do Tratador**
h_methane()	E_MethaneHigh_PC **(E1)**	controlMineralExtraction()	E_MethaneLevelFailure_PC
h_methane()	E_MethaneHighPumpOn_PC **(E2)**	controlMineralExtraction()	E_MethaneLevelFailure_PC
controlPumping()	I_MethaneHigh_PC	controlPumping()	I_MethaneLevelFailure_PC
controlPumping()	E_WaterLowFailure_PC **(E3)**		
controlPumping()	E_WaterHighFailure_PC **(E4)**		
controlAirExtraction()	I_SwitchAEOn_AEC	controlAirExtraction()	I_SwitchAEOn_AEC
h_SwitchAEOn()	E_SwitchAEOn_AEC **(E5)**	controlMineralExtraction()	E_AEFailure_AEC
controlMineralExtraction()	E_EvacuationFailure_MEC **(E6)**		

Após a correção da falha de modelagem, a Tabela 7.1 mostra os tipos de exceções capturados pelos tratadores de exceção do modelo. A tabela também mostra que todos os tratadores modelados capturam pelo menos uma exceção.

A imagem do GFCI gerado para o Sistema de Mineração é muito grande para ser inserida neste documento e está disponível para consulta no seguinte endereço: *http://www.students.ic.unicamp.br/~ra066147/grafos/ICFG1.jpg*

7.2 Máquina de Vendas

O segundo caso de estudo simula as ações de uma máquina de vendas automática. O aparelho permite que o usuário insira moedas, solicite o reembolso, ou selecione um item utilizando um teclado numérico. Se o usuário selecionar um item válido e inserir moedas com valor suficiente para cobrir o custo do item, o aparelho dispensa o item selecionado. Se o usuário fizer uma seleção incorreta, o aparelho solicita ao usuário para digitar novamente a seleção. O usuário pode digitar novamente ou solicitar um reembolso das moedas inseridas. A máquina mantém o controle do número de seleções errôneas inseridas por um usuário. Uma vez que o número de seleções errôneas excede um valor predeterminado, a máquina aborta a operação e devolve as moedas do usuário. As possíveis condições de erro que podem surgir durante a operação são mostradas na Tabela 7.2.

O modelo completo da Máquina de Vendas pode ser consultado no seguinte endereço: *http://www.students.ic.unicamp.br/~ra066147/VendingMachine/VendingMachine.html*

Tabela 7.2: *Exceções da Máquina de Vendas*

Sigla	**Exceção**	**Descrição**
IA	*IllegalAmountException*	Usuário seleciona um item que custa mais que o valor das moedas inseridas.
IC	*IllegalCoinException*	Usuário insere uma moeda não permitida.
ZV	*ZeroValueException*	Usuário seleciona um item, ou solicita reembolso sem inserir nenhuma moeda.
IS	*IllegalSelectionException*	Usuário entra com um código que não corresponde a nenhuma seleção válida.
SNA	*SelectionNotAvailableException*	Usuário seleciona um item que está temporariamente indisponível.

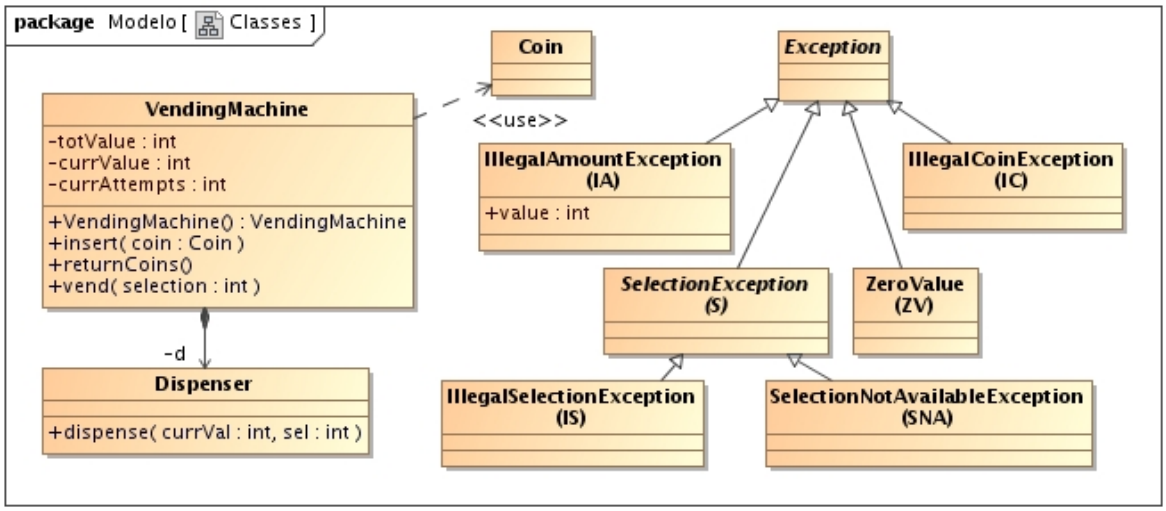


Figura 7.17: Diagrama de Classes da Máquina de Vendas

A Figura 7.17 apresenta o diagrama de classes da Máquina de Vendas. O modelo é composto por três classes *VendingMachine*, *Dispenser* e *Coin*. A classe *VendingMachine* é a classe principal, possui três atributos: *totValue* que acumula o valor vendido; *currValue* valor inserido pelo usuário e *currAttempts* que acumula o número de tentativas de seleção incorretas. A classe *VendingMachine* possui ainda um construtor e mais três operações: *insert()*, *returnCoins()* e *vend()*, cada uma dessas operações representa uma opção no menu do usuário. A operação *insert()* habilita a máquina para recepcionar as moedas, a operação *returnCoins()* efetua a devolução das moedas inseridas e a operação *vend()* permite que o usuário informe através do teclado numérico o código do item desejado. A classe *Coin* representa uma moeda e a classe *Dispenser* representa o mecanismo de entrega do item. Esta última classe possui uma operação *dispense()* responsável pela entrega do item selecionado.

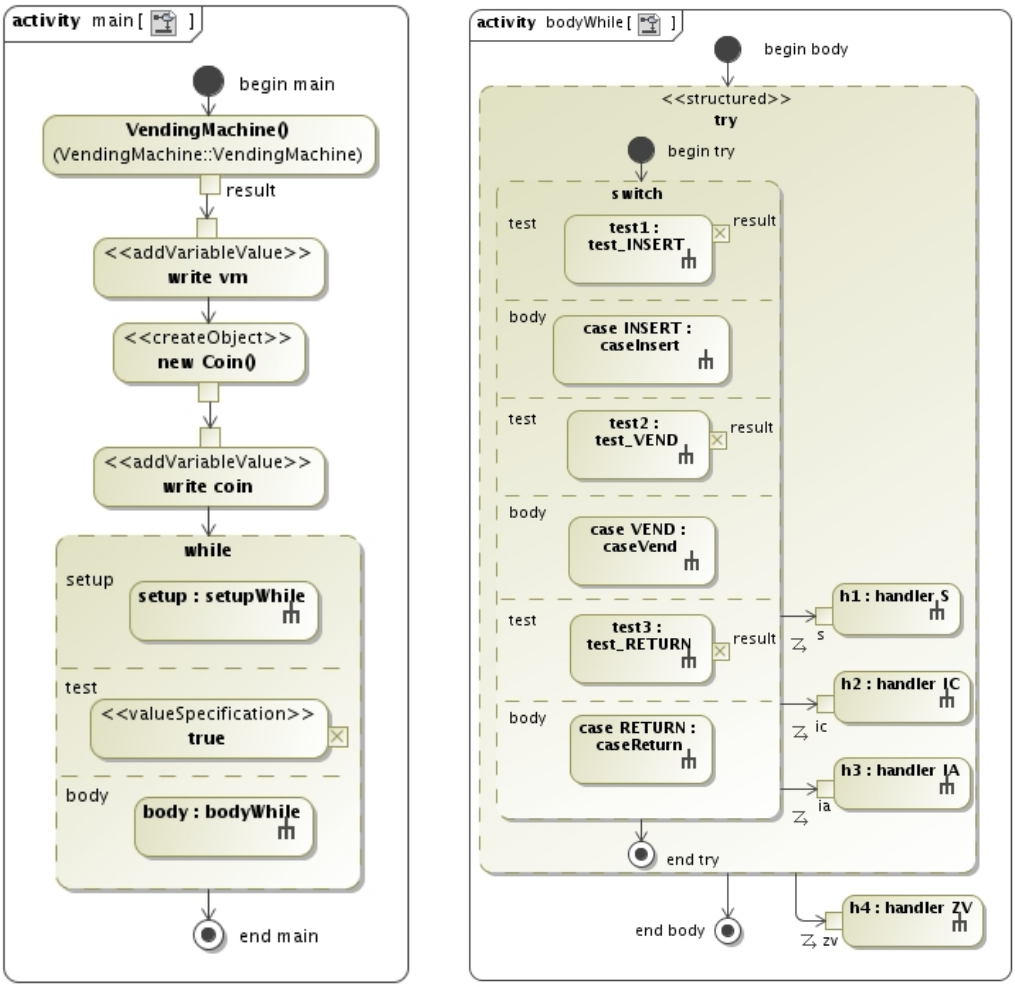
A Figura 7.1 também mostra a hierarquia das classes de exceções. Para reduzir as descrições nas figuras dos diagramas de atividades e dos grafos, a partir daqui as exceções serão identificadas pelas siglas apresentadas na Tabela 7.2.

7.2.1 Fluxo de Controle Intraprocedimental

Neste estudo de caso a operação *main* é a operação principal, a partir dela as demais operações são chamadas. Esta operação apresenta ao usuário três opções, e com base na ação do usuário, uma das três operações definidas na classe *VendingMachine* é chamada para processar a ação.

Assim como descrito no primeiro estudo de casos, para efetuar a validação do fluxo

excepcional da operação, primeiro temos que converter o conjunto de atividades utilizado para modelar o comportamento da operação em um grafo de atividades (GA). Para realizar as transformações são utilizados os Algoritmos 1 e 2, apresentados na Seção 4.2.



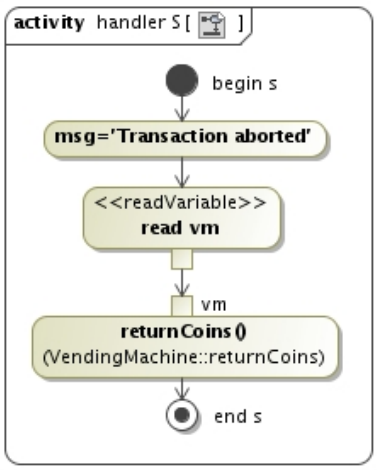
(a) main

(b) bodyWhile

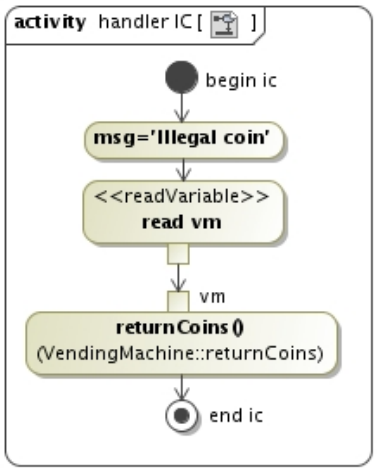
Figura 7.18: Operação main() - Parte 1

As Figuras 7.18, 7.19 e 7.20 apresentam o conjunto de atividades utilizadas para modelar o comportamento da operação *main()*. A Figura 7.18a apresenta o diagrama principal da operação *main()*. A primeira ação (do tipo *CallOperationAction*) invoca o construtor da classe VendingMachine. A referência do objeto criado é atribuído na variável *vm* pela ação *write vm*. A atividade *main* possui uma atividade estruturada (*LoopNode while*) que é utilizada para modelar um laço infinito. O corpo do laço é representado pela atividade bodyWhile, mostrada na Figura 7.18b. Esta atividade possui

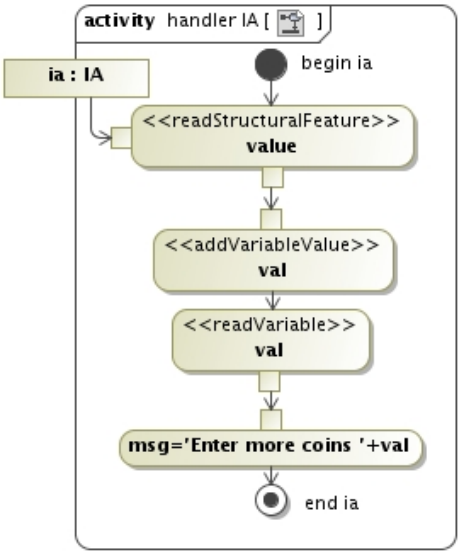
duas atividades estruturadas aninhadas, a primeira denominada *try* é um nó protegido que contém um tratador de exceções do tipo *ZV*. A segunda atividade estruturada é um nó condicional (*switch*) utilizado para modelar um estrutura de controle de múltipla escolha. Este nó condicional possui três cláusulas, uma cláusula para cada ação disponível para o usuário. A primeira clausula é satisfeita caso o usuário tenha selecionado a opção inserir moedas, a segunda cláusula caso o usuário queira selecionar um item para compra e a terceira cláusula caso o usuário tenha solicitado a devolução das moedas inseridas. O nó condicional *switch* também possui três tratadores de exceções: *h1* do tipo *S*, *h2* do tipo *IC* e *h3* do tipo *IA*.



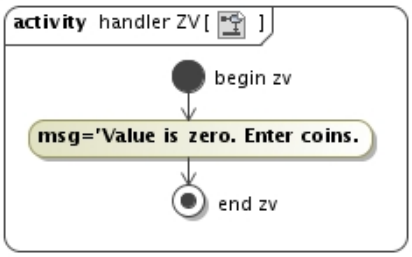
(a) handler_S



(b) handler_IC

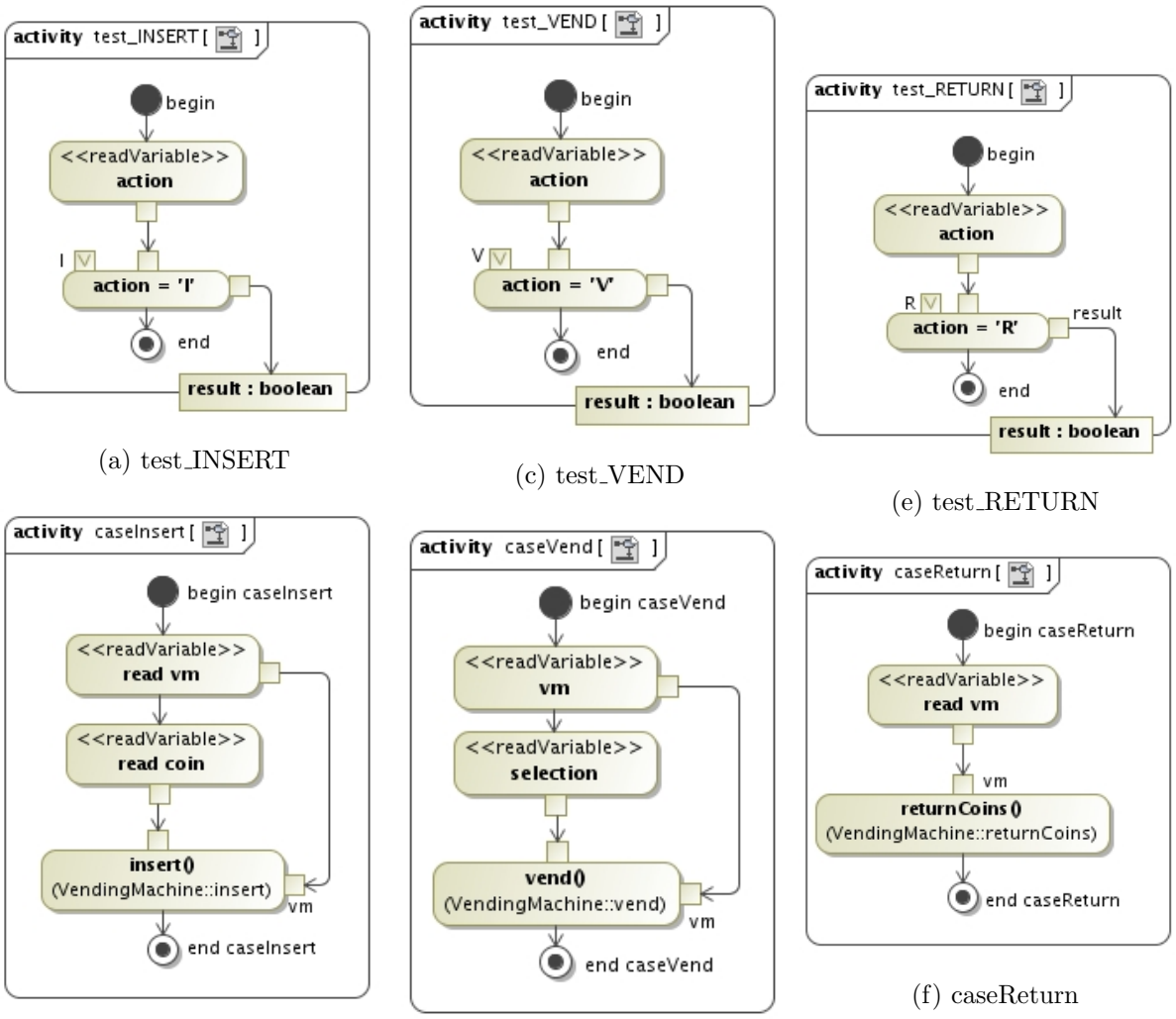


(c) handler_IA



(d) handler_ZV

Figura 7.19: Operação main() - Parte 2



(a) test_INSERT

(c) test_VEND

(e) test_RETURN

(b) caseInsert

(d) caseVend

(f) caseReturn

Figura 7.20: Operação main() - Parte 3

As atividades que modelam o comportamento dos tratadores de exceção da operação *main()* são mostrados na Figura 7.19. A Figura 7.19a mostra o comportamento do tratador de exceções *h1*, acionado na ocorrência de uma exceção do tipo *IS* ou *SNA*, uma mensagem informando que a transação foi abortada é exibida ao usuário e em seguida a operação *returnCoins()* é chamada para devolver as moedas inseridas. A Figura 7.19b mostra o comportamento do tratador de exceções *h2*, acionado na ocorrência de uma exceção do tipo *IC*, uma mensagem informando que foi inserida uma moeda inválida é exibida ao usuário e em seguida a operação *returnCoins()* é chamada para devolver as moedas inseridas.

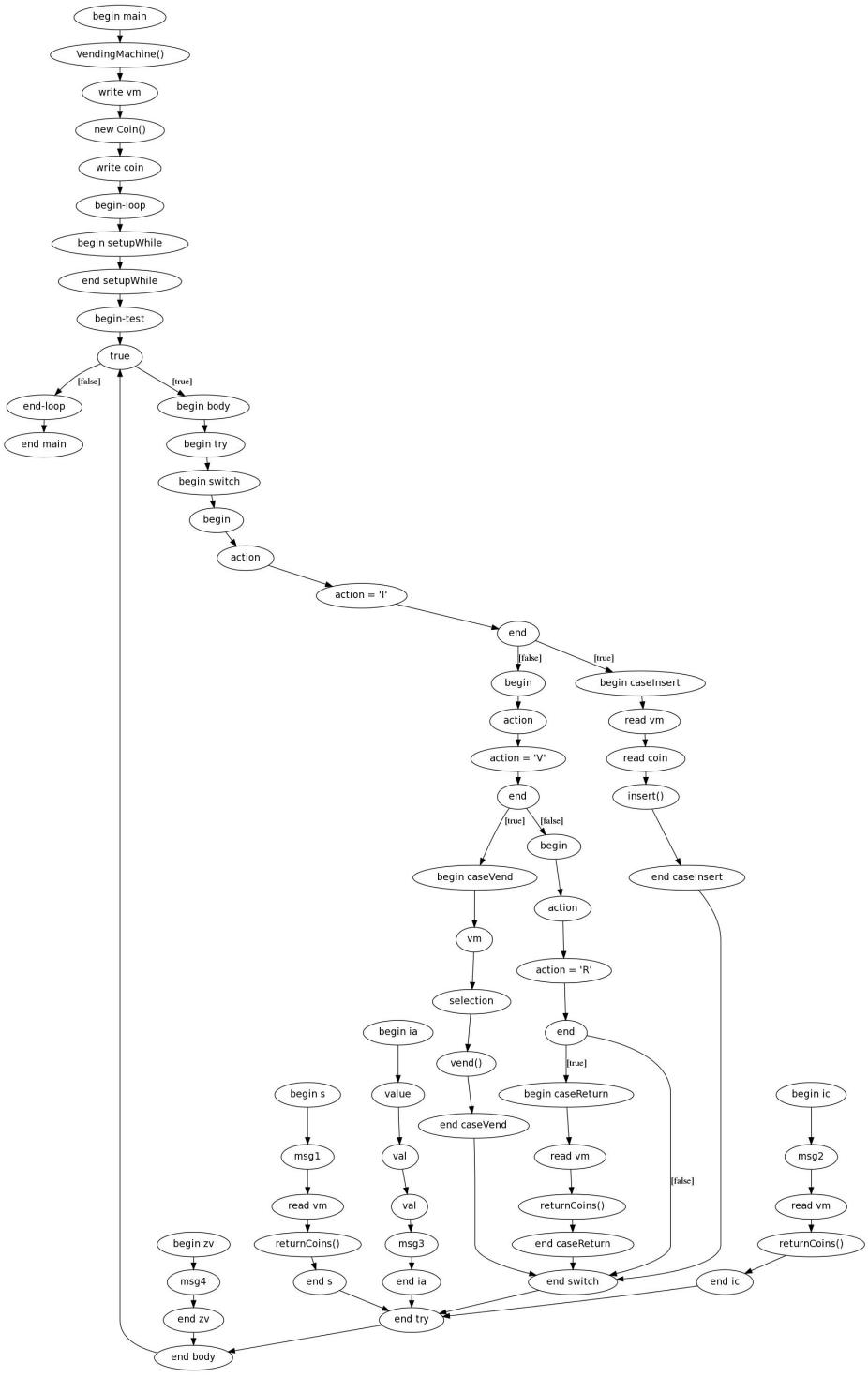


Figura 7.21: GA da operação main()

A Figura 7.19c mostra o comportamento do tratador de exceções *h3*, acionado na

ocorrência de uma exceção do tipo *IA*, uma mensagem é exibida informando o valor que falta ser inserido para a máquina dispensar o item selecionado. O valor é obtido do próprio objeto de exceção que possui um atributo para armazená-lo. A ação *value* do tipo *ReadStructuralFeatureAction* é responsável por recuperar o valor do atributo e disponibilizá-lo no pino de saída da ação. A Figura 7.19d mostra o comportamento do tratador de exceções *h4*, acionado na ocorrência de uma exceção do tipo *ZV*, neste caso uma mensagem é exibida informando ao usuário que nenhuma moeda foi inserida.

As Figuras 7.20a, 7.20c e 7.20e mostram os testes das cláusulas do nó condicional *switch*. A variável *action* é utilizada para determinar a cláusula escolhida. A Figura 7.20b mostra o comportamento da primeira cláusula condicional, neste caso a operação *insert()* é chamada para habilitar a recepção das moedas. A Figura 7.20c mostra o comportamento da segunda cláusula condicional, neste caso a operação *vend()* é chamada para habilitar a seleção do item desejado. A Figura 7.20f mostra o comportamento da terceira cláusula condicional, neste caso a operação *returnCoins()* é chamada para reembolsar o usuário.

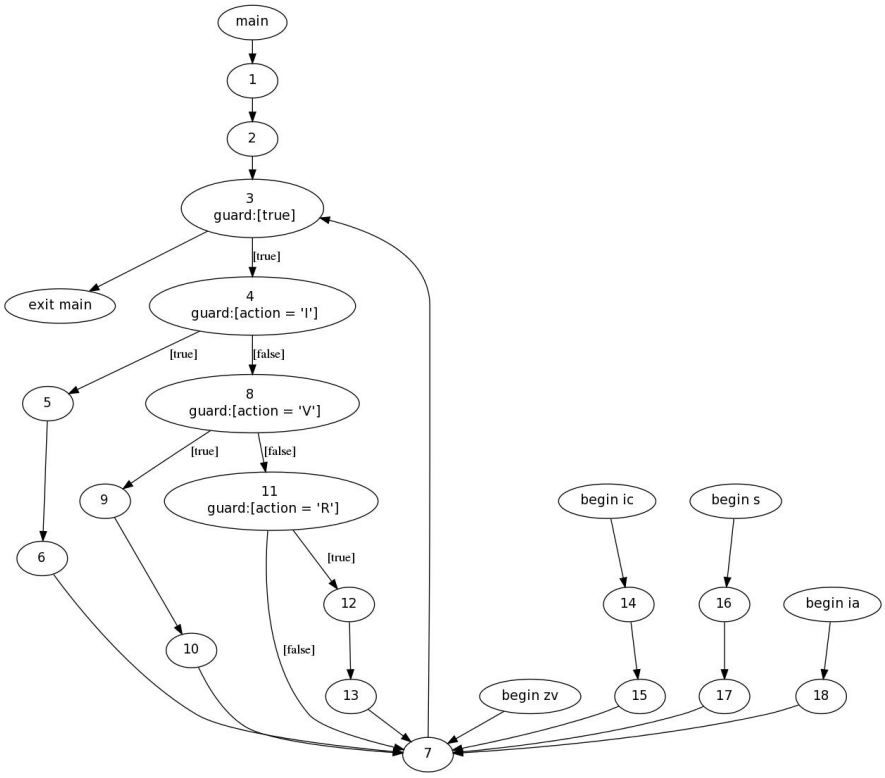


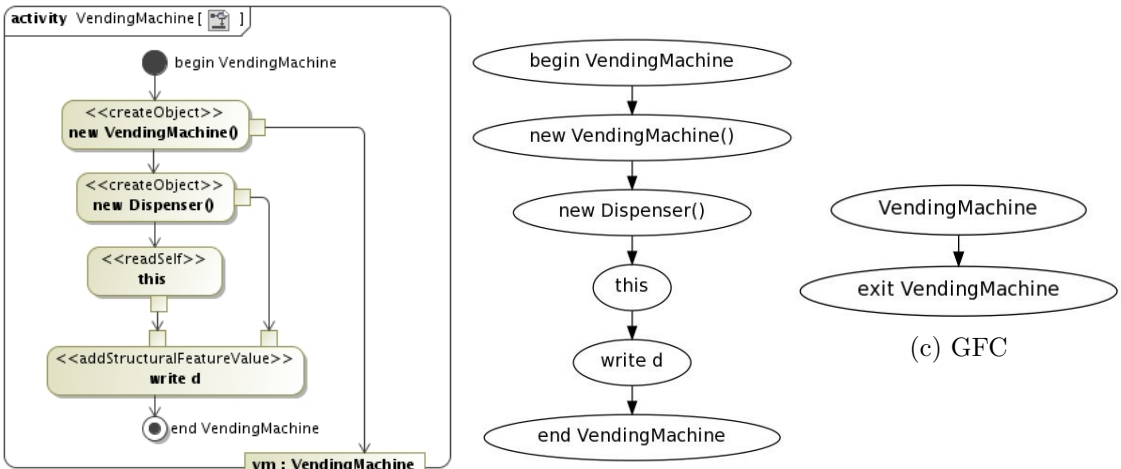
Figura 7.22: GFG da operação main()

A Figura 7.21 mostra o grafo de atividades resultante da transformação do conjunto de diagramas de atividades mostrados nas Figuras 7.18, 7.19 e 7.20. O próximo passo e gerar

o Grafo de Fluxo de Controle (GFC). Cada nó do GFC corresponde a um bloco básico de atividades do GA, no qual não existe desvio do fluxo de controle, exceto no final do bloco. Os conjuntos de definição e uso das variáveis de cada nó do GA, são transportados para os nós do GFC. Para gerar o GFC utilizamos o Algoritmo 3 apresentado na Seção 4.2. A Figura 7.22 mostra o grafo de fluxo de controle da operação *main()*.

Após a geração do GFC o próximo passo é a criação das arestas de exceção. No caso da na operação *main()*, nenhuma exceção é lançada pela operação, por outro lado, ela especifica vários tratadores de exceção que irão capturar as exceções lançadas pelas demais operações. Dessa maneira, o fluxo excepcional somente poderá ser completado a partir da criação do GFCI.

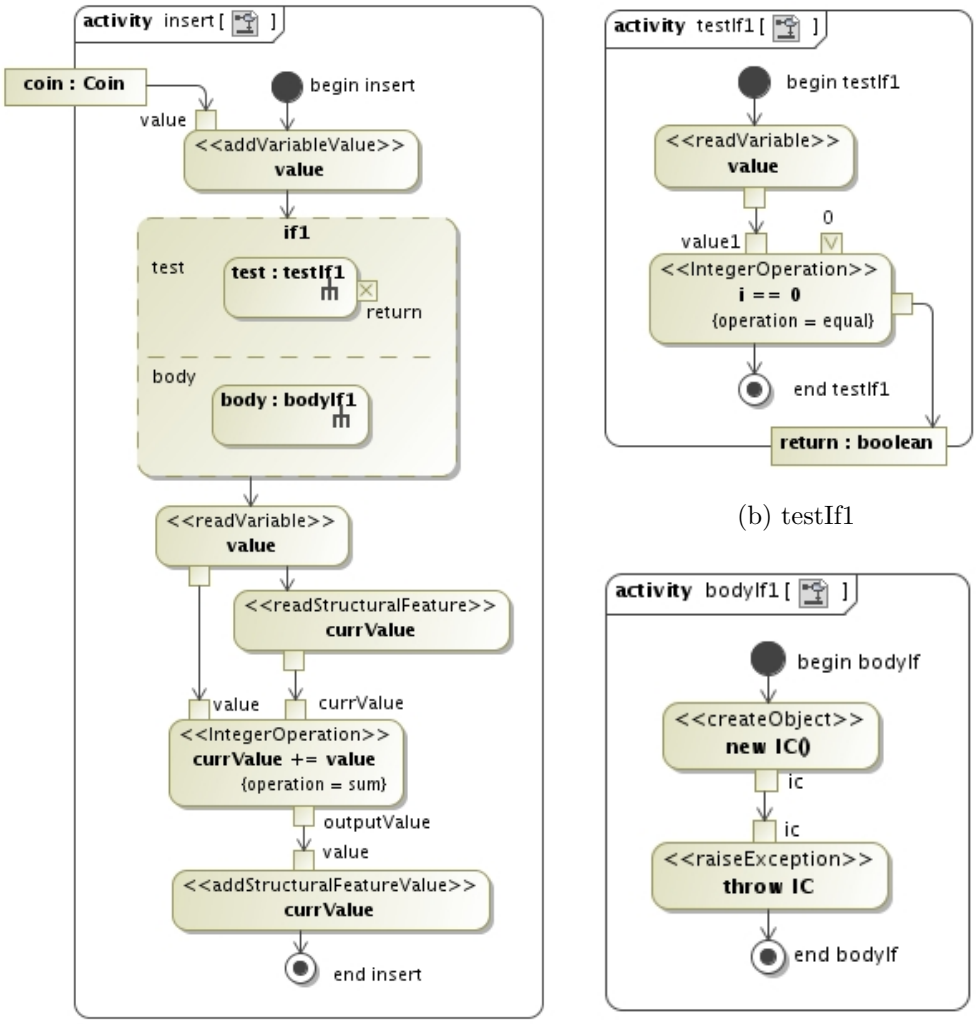
A seguir iremos apresentar as atividades das demais operações, apresentando sempre o grafo de atividades e o grafo de fluxo de controle resultante das transformações.



(a) DA (b) GA (c) GFC

Figura 7.23: Construtor VendingMachine()

A Figura 7.23a mostra o diagrama de atividades que modela o comportamento do construtor da classe *VendingMachine*. Nesta atividade são instanciados dois objetos, o primeiro é a instância da própria classe *VendingMachine* e o segundo é uma instância do módulo dispensador (*Dispenser*). A referência deste último objeto é inserida no atributo *d* da classe *VendingMachine* pela ação *write d* do tipo *AddStructuralFeatureAction*. A Figura 7.23b mostra o grafo de atividades do construtor e a Figura 7.23c mostra o grafo de fluxo de controle do construtor gerado a partir do agrupamento dos nós do GA.



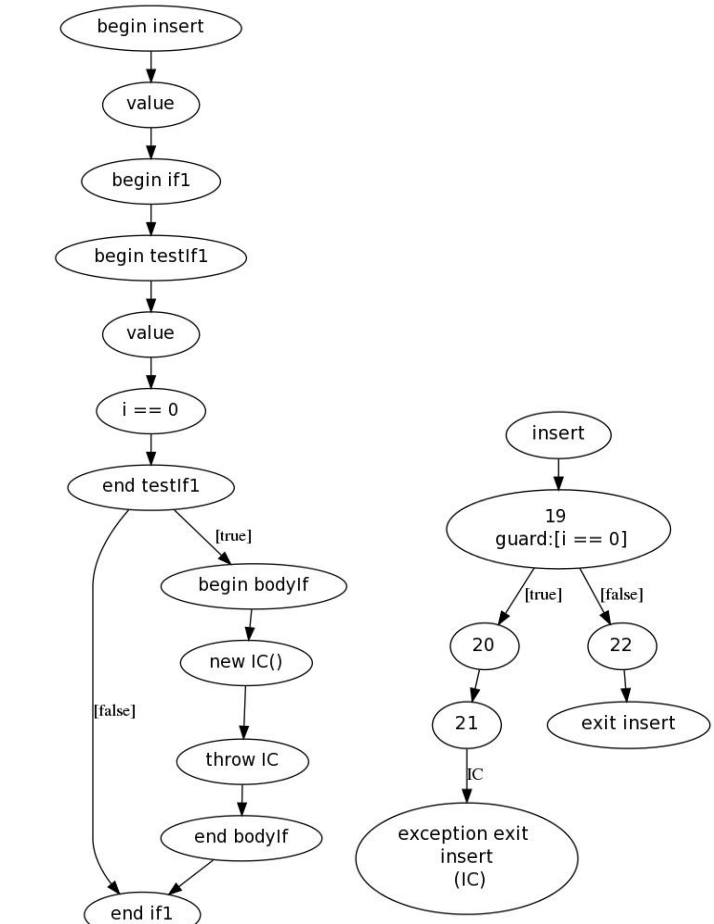
(a) insert

(b) testIf1

(c) bodyIf1

Figura 7.24: Modelagem da Operação insert()

A operação *insert()* primeiro verifica se o usuário entrou com uma moeda válida, na sequência incrementa o valor atual com o valor da moeda; *insert()* gera uma exceção se o usuário inserir uma moeda inválida. A Figura 7.24 mostra o conjunto de atividades que modelam a operação *insert()*. A Figura 7.24a mostra a atividade principal da operação, esta atividade possui um nó condicional (*if1*). O teste condicional é realizado pela atividade *testIf1* (Figura 7.24b), que verifica o valor da moeda inserida, caso o valor seja zero, a cláusula condicional é satisfeita é a atividade *bodyIf1* é executada. A Figura 7.24c mostra o comportamento da atividade *bodyIf1*, que cria um objeto de exceção do tipo *IC* e em seguida lança essa exceção.



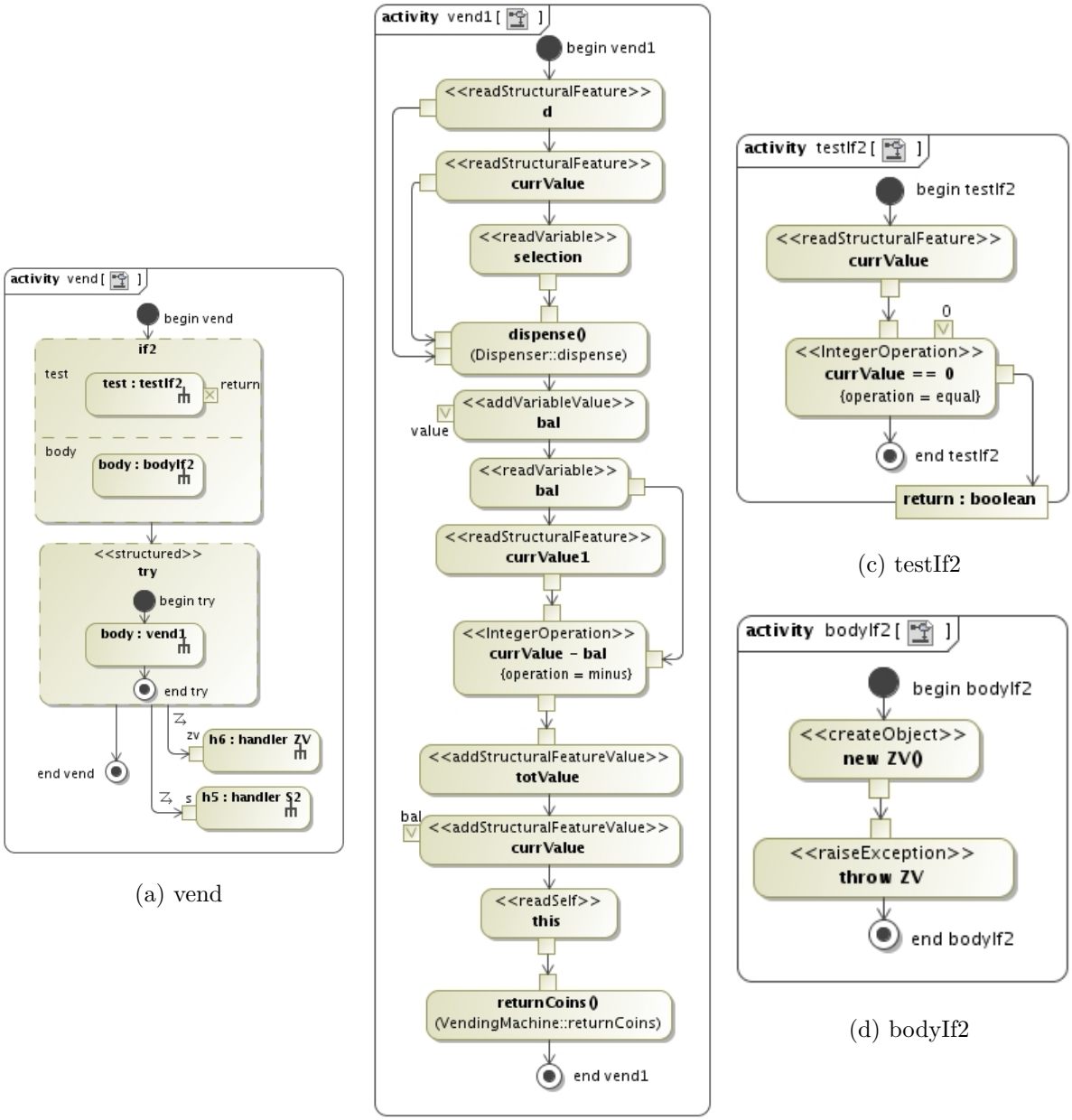
(a) GA

(b) GFC

Figura 7.25: Grafos da Operação insert()

A Figura 7.25 mostra os grafos resultantes das transformações, a Figura 7.25a mostra o grafo de atividades e a Figura 7.25b mostra o grafo de fluxo de controle da operação *insert()*.

No caso da operação *insert()* uma exceção é lançada e não existe nenhum tratador de exceções modelado na operação. Dessa maneira, um nó de saída excepcional é gerado no GFC. A Figura 7.25b mostra o nó de saída excepcional do tipo *IC*. Assim como ocorreu com a operação *main()*, o fluxo excepcional somente será criado a partir da geração GFCI.



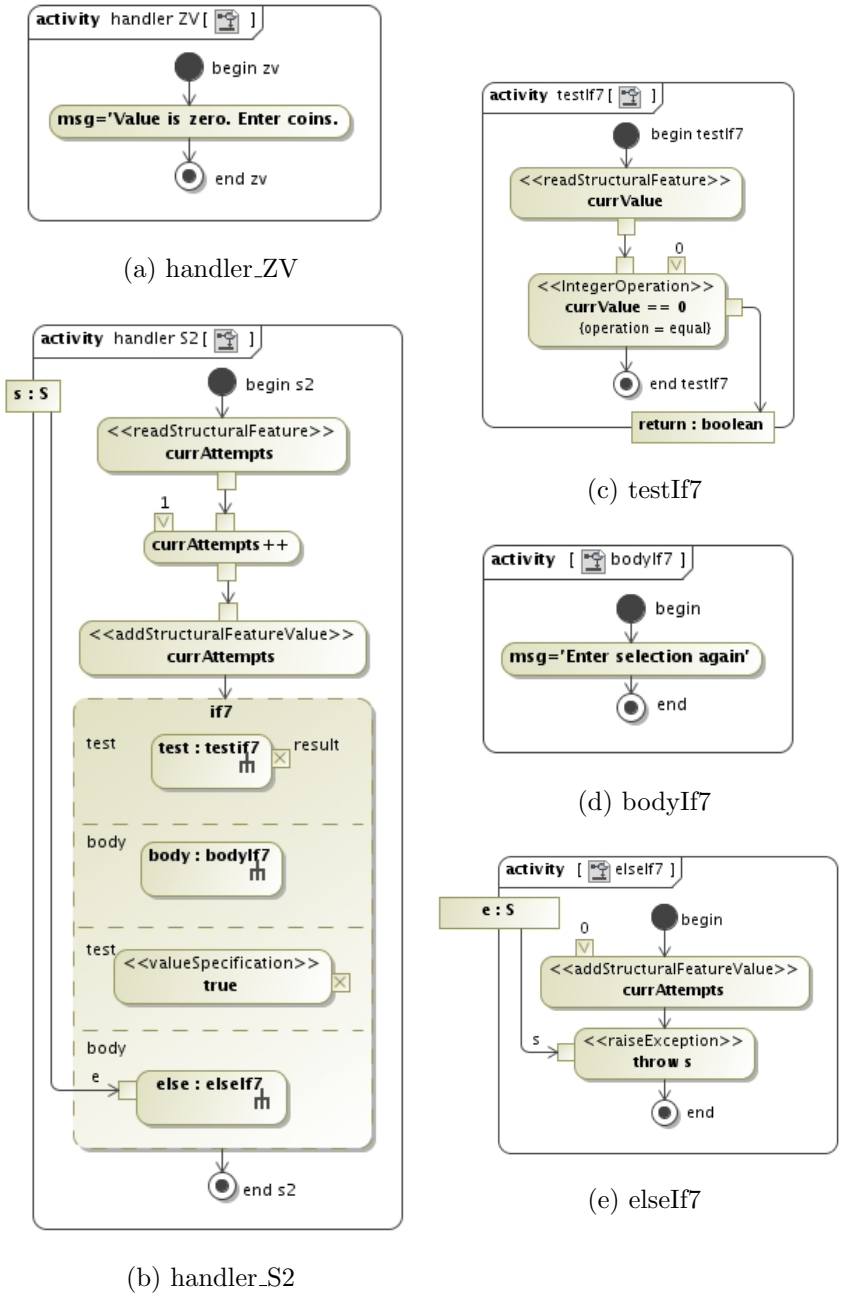
(a) vend

(b) vend1

(c) testIf2

(d) bodyIf2

Figura 7.26: Operação vend() - Parte 1



(a) handler_ZV

(b) handler_S2

(c) testIf7

(d) bodyIf7

(e) elseIf7

Figura 7.27: Operação vend() - Parte 2

A operação *vend()* recebe a seleção do usuário, verifica se o valor atual é diferente de zero. Se o valor atual for zero a operação lança uma exceção, caso contrário chama a operação *dispense()*, definida na classe *Dispenser*, para entregar o item ao usuário. As Figuras 7.26 e 7.27 mostram o conjunto de atividades que modelam a operação.

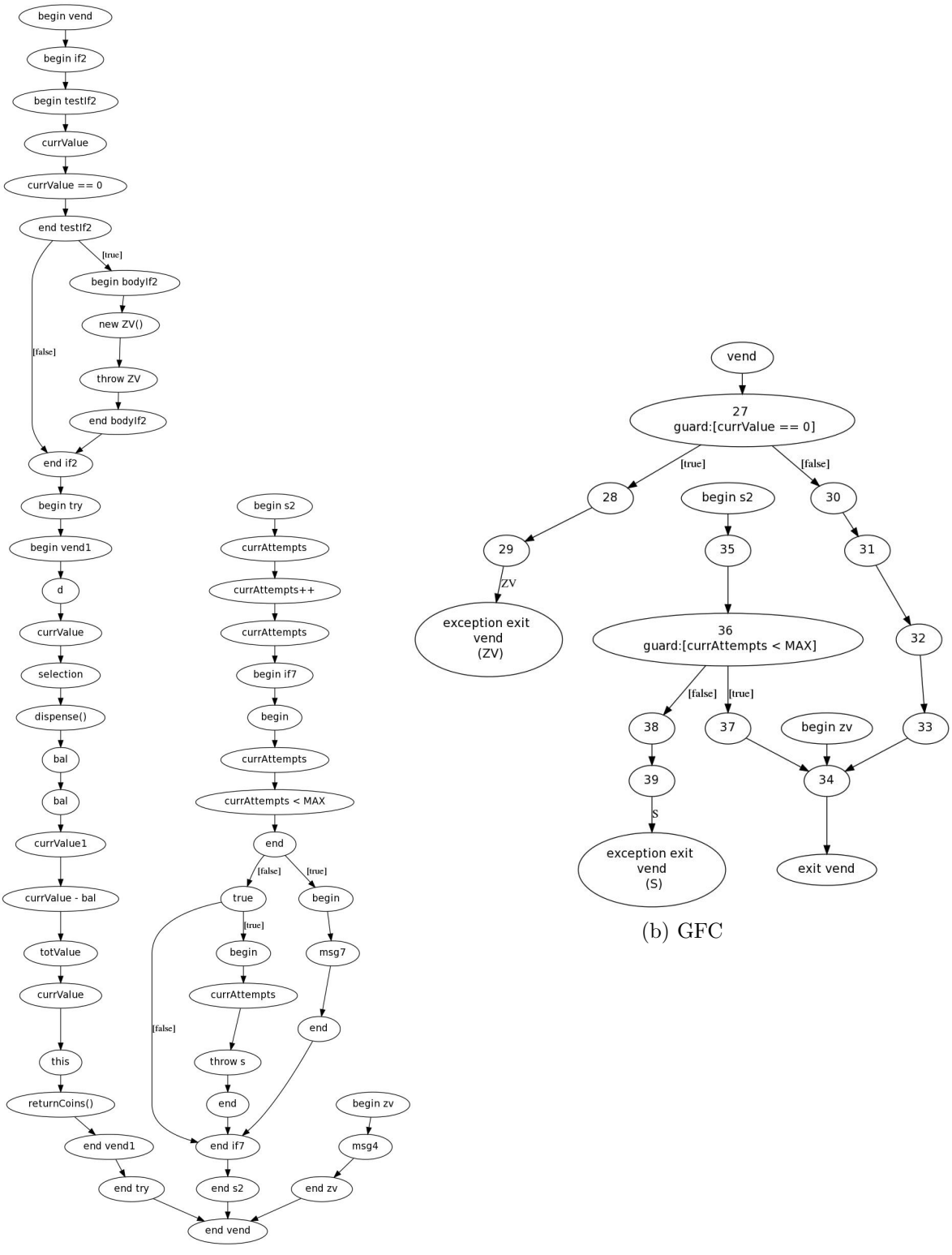
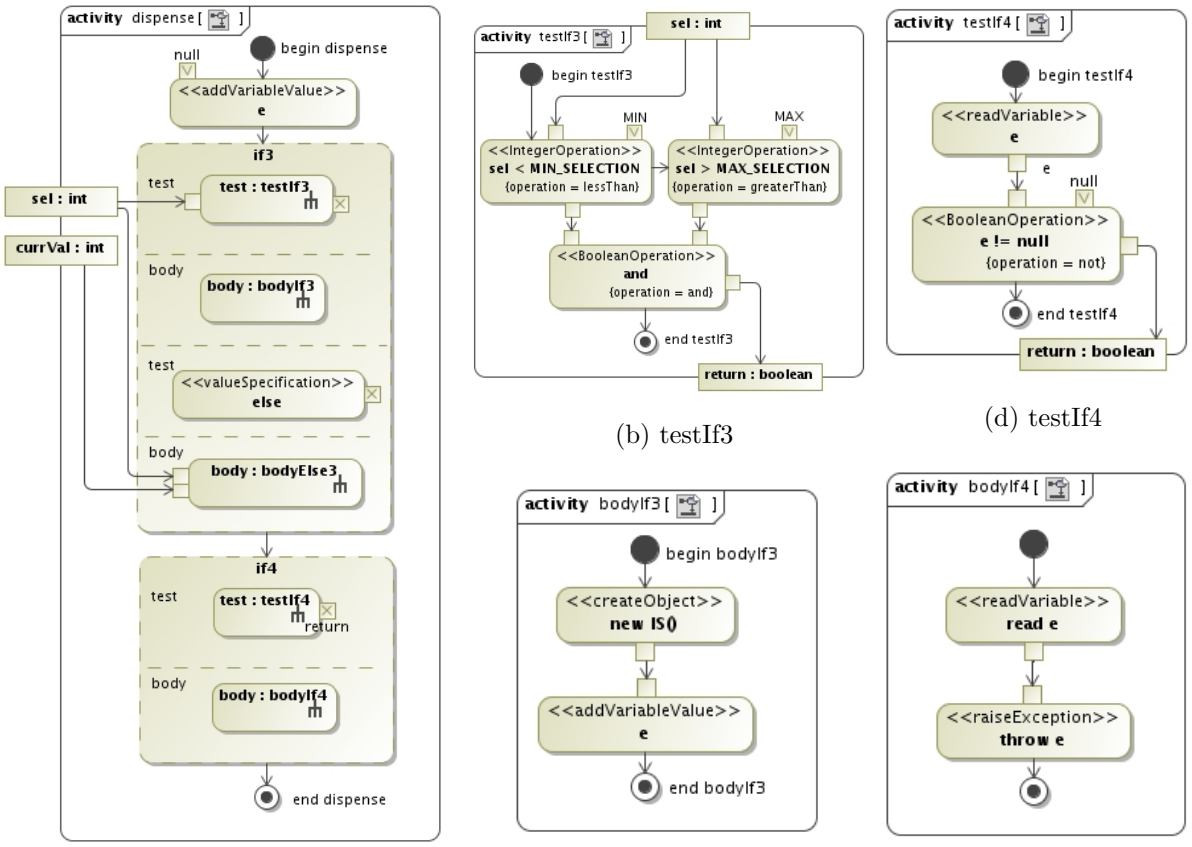


Figura 7.28: Grafos da Operação vend()

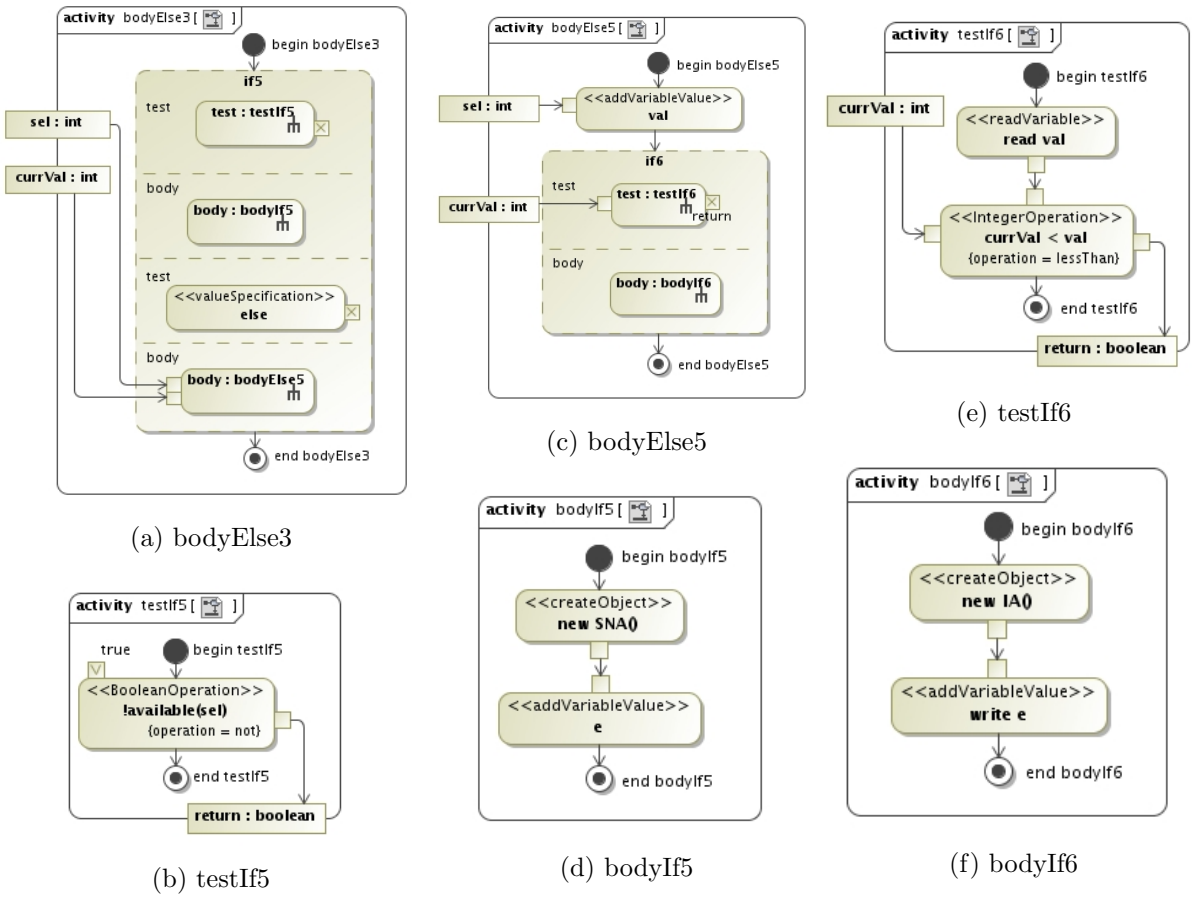
A Figura 7.26a mostra a atividade principal que possui dois nós estruturados. O primeiro é um nó condicional (*if2*) que possui apenas uma cláusula condicional. O segundo é um nó protegido (*try*) que possui dois tratadores de exceções (*h5* e *h6*). Este nó também possui no seu interior uma ação do tipo *CallBehaviorAction*, que chama a atividade *vend1* (Figura 7.26b). As Figuras 7.26c e 7.26d, mostram as atividades *testIf2* e *bodyIf2*, responsáveis respectivamente pelo teste condicional e pelo lançamento de uma exceção do tipo *ZV*. As Figuras 7.27a e 7.27b mostram o comportamento dos tratadores de exceçao (*h5* e *h6*). O primeiro captura um exceção do tipo *ZV* e exibe uma mensagem para o usuário. O segundo captura uma exceção do tipo *S* e dependendo de um teste condicional lança novamente a exceção para ser capturada em outra operação.

A Figura 7.28 mostra os grafos resultantes das transformações, a Figura 7.28a mostra o grafo de atividades e a Figura 7.28b mostra o grafo de fluxo de controle da operação *vend()*.



(a) dispense (b) testIf3 (c) bodyIf3 (d) testIf4 (e) bodyIf4

Figura 7.29: Operação dispense() - Parte 1



(a) bodyElse3

(b) testIf5

(c) bodyElse5

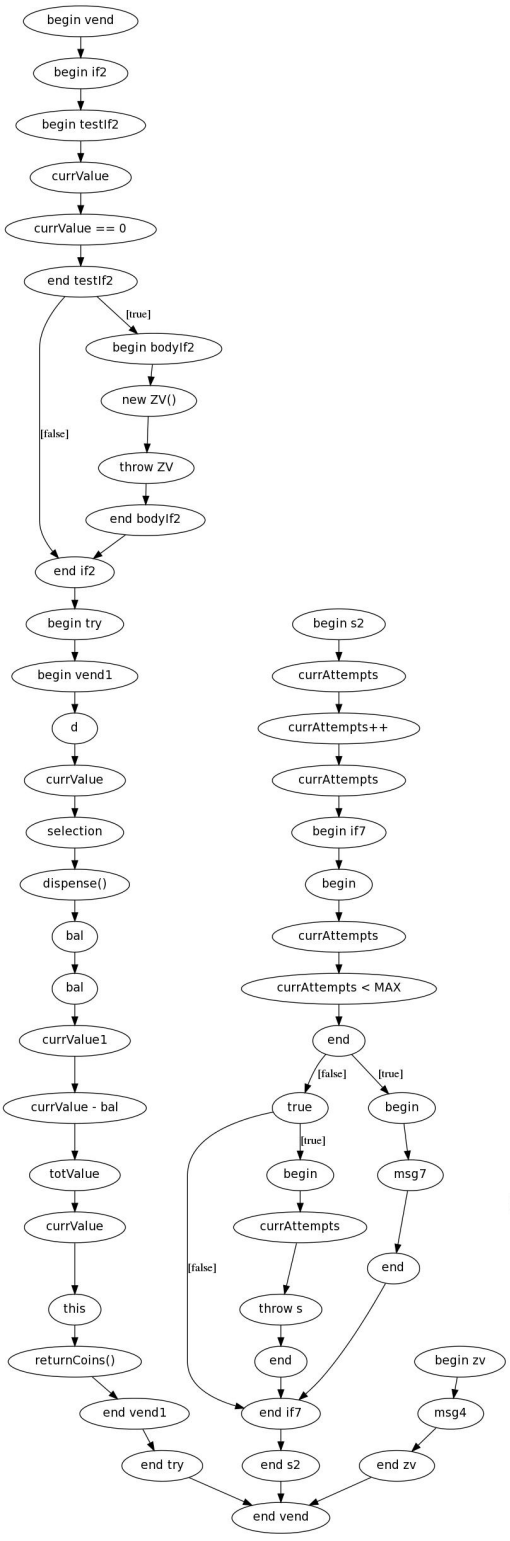
(d) bodyIf5

(e) testIf6

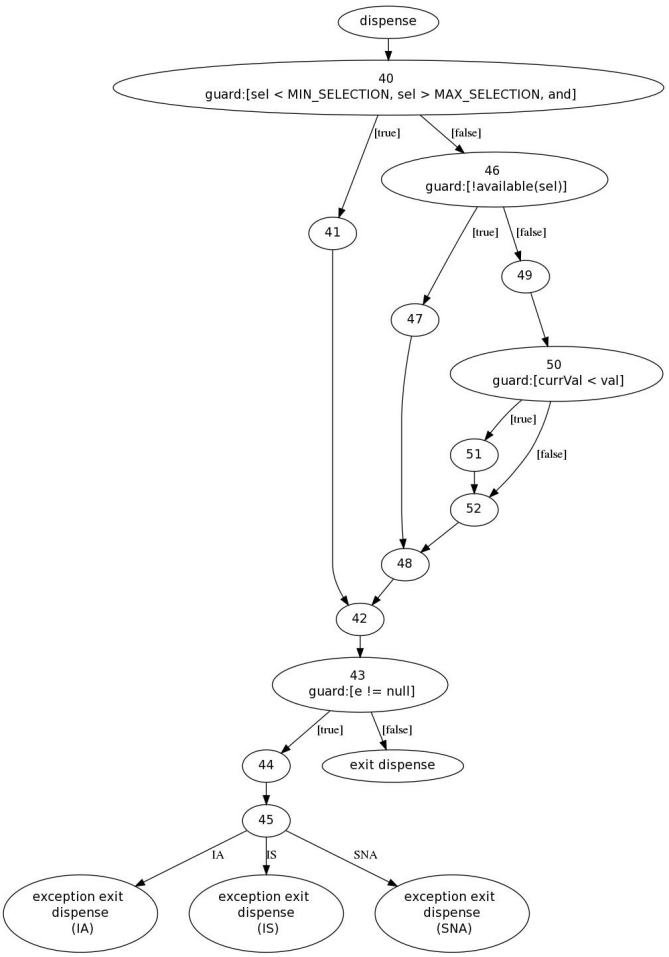
(f) bodyIf6

Figura 7.30: Operação dispense() - Parte 2

A operação *dispense()* executa várias verificações de erros para garantir a seleção de um item válido, garantir que o item esteja disponível para distribuição, e garantir que o valor atual seja capaz de cobrir o custo da seleção. Se qualquer uma dessas verificações falhar, a operação lança uma exceção. Se passar por todas as verificações, a operação *dispense()* simula a distribuição do item, exibindo uma mensagem ao usuário. Após a conclusão bem sucedida da operação *dispense()*, a operação *vend()* atualiza as variáveis e chama a operação *returnCoins()* para devolver o troco para o usuário. Se a operação *dispense()* gerar uma exceção por causa de uma seleção errada, a operação *vend()* trata a exceção e incrementa a variável que controla o número de tentativas incorretas (*currAttempts*). Quando a variável *currAttempts* exceder a constante *MAX_ATTEMPTS*, a operação *vend()* relança a exceção capturada, que faz com que a operação *main()* aborte a transação. As Figuras 7.29 e 7.30 mostram o conjunto de atividades que modelam a operação.



(a) GA



(b) GFC

Figura 7.31: Grafos da Operação dispense()

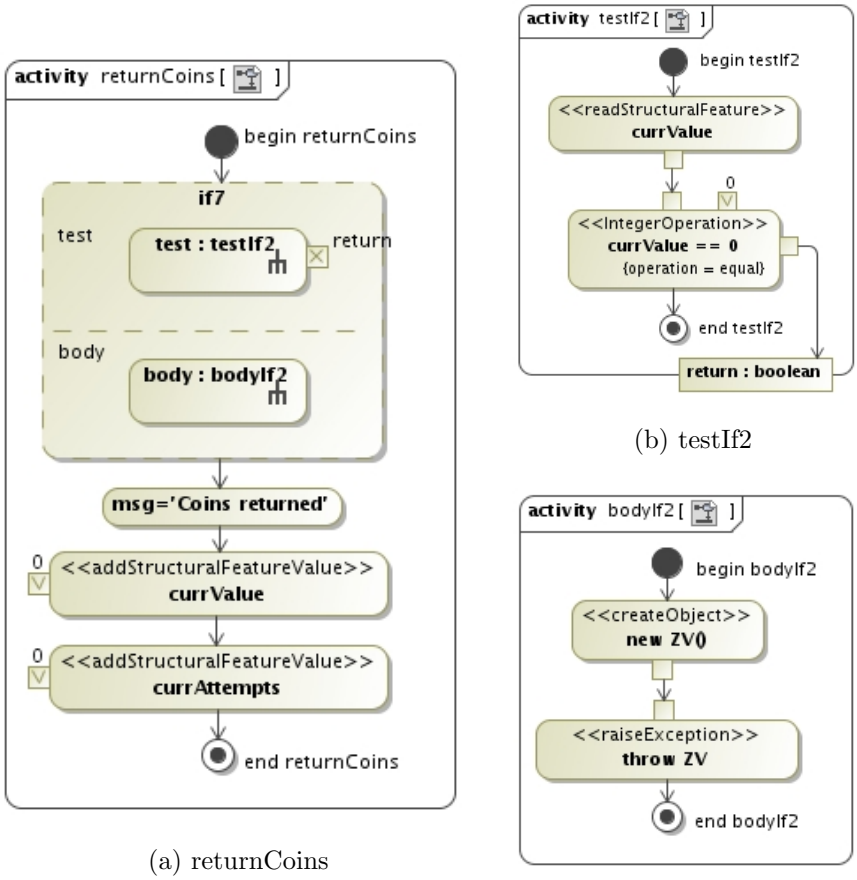
Para efetuar a validação do fluxo excepcional da operação *dispense()*, primeiro temos que converter o conjunto de atividades utilizado para modelar o comportamento da operação em um grafo de atividades (GA). Para realizar as transformações são utilizados os Algoritmos 1 e 2, apresentados na Seção 4.2.

Na sequência o GA gerado no passo anterior é convertido em um Grafo de Fluxo de Controle (GFC). Para gerar o GFC utilizamos o Algoritmo 3 apresentado na Seção 4.2. Após a geração do GFC é preciso completar o grafo com o fluxo de controle excepcional, inserindo arestas que representem o fluxo da exceção ou caso a exceção não seja tratada dentro da própria operação é necessário incluir um nó de saída excepcional. O primeiro passo para a criação do fluxo de exceções intraprocedimental é a identificação dos tipos de exceção que podem ser lançados por um determinado nó.

No caso da operação *dispense()*, existe apenas uma atividade do tipo *RaiseExceptionAction*, que é responsável pelo lançamento das exceções (Figuras 7.29e. Sempre que uma ação deste tipo é encontrada no GA, durante a geração do GFC, um nó exclusivo para esta ação é criado no GFC. Conforme a regra apresentada na Seção 4.3 este nó pode ser um *CFGRaiseNode* ou um *CFGRaiseVariableNode*. O que determina se um nó do GA, que contém uma ação do tipo *RaiseExceptionAction*, será mapeado para um nó do GFC do tipo *CFGRaiseNode* ou *CFGRaiseVariableNode*, é o tipo de ação que antecede a ação *RaiseExceptionAction* no DA. Se a ação for do tipo *ReadVariableAction* é criado um nó do tipo *CFGRaiseVariableNode*, pois a ação *ReadVariableAction* lê o conteúdo de uma variável e passa este valor como entrada para a ação que lança a exceção, o que implica na dependência da definição da variável para a identificação do tipo de exceção que pode ser lançada. Caso a ação que fornece o objeto de exceção seja do tipo *CreateObjectAction*, não existe a dependência de uma variável. Nessa caso, o nó criado no GFC será do tipo *CFGRaiseVariableNode*, pois o tipo de exceção lançada depende das definições da variável *e* que alcançam o nó responsável pelo lançamento das exceções.

No caso da atividade *bodyIf4* foi gerado um nó do tipo *CFGRaiseVariableNode* (nó 45 do GFC da Figura 7.31b). Para determinar quais os tipos de exceções que podem ser lançadas por este nó é necessário executar uma análise de fluxo de dados intraprocedimental, conforme mostrado na seção 4.3. Neste caso, as definições da variável *e* que alcançam o nó 45 foram geradas pelas atividades *bodyIf3*, *bodyIf5* e *bodyIf6*, mostradas nas Figuras 7.29 e 7.30. Por esse motivo, foram gerados três nós de saída excepcional no GFC da Figura 7.31b.

A Figura 7.31 mostra os grafos resultantes das transformações, a Figura 7.31a mostra o grafo de atividades e a Figura 7.31b mostra o grafo de fluxo de controle da operação *dispense()*.



(a) returnCoins

(b) testIf2

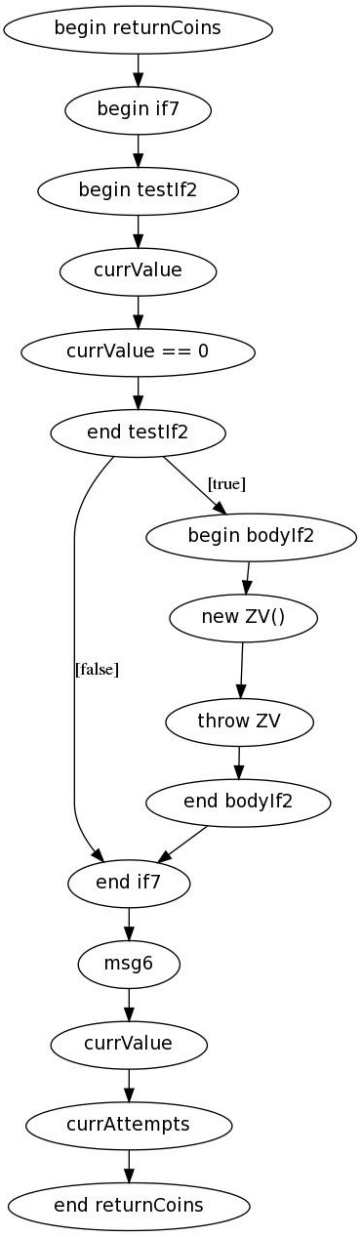
(c) bodyIf2

Figura 7.32: Operação returnCoins()

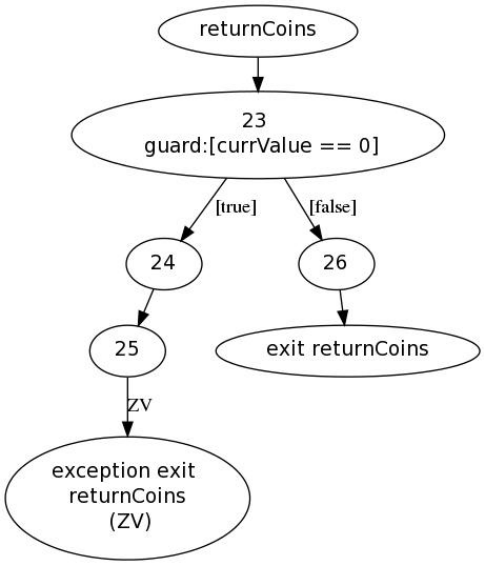
A operação *returnCoins()* devolve as moedas inseridas de acordo com o valor atual, na sequência zera o valor atual; *returnCoins()* gera uma exceção se o valor atual for zero. A Figura 7.32 exibe o conjunto de atividades utilizadas para modelar a operação.

A Figura 7.33 mostra os grafos resultantes das transformações, a Figura 7.33a exibe o grafo de atividades e a Figura 7.33b o grafo de fluxo de controle da operação *returnCoins()*.

Detalhes de algumas operações foram omitidos, como as operações *value()* e *available()* da classe *Dispenser*, que não são relevantes para nossa apresentação por não lançarem nenhuma exceção. Para resumir, também foram omitidos detalhes dos construtores de algumas classes, e das inicializações de constantes, tais como: *MAX_ATTEMPTS* e *MIN_SELECTION*.



(a) GA



(b) GFC

Figura 7.33: Grafos da Operação returnCoins()

7.2.2 Fluxo de Controle Interprocedimental

Após a construção do GFC de cada operação contida no modelo o próximo passo é a geração do Grafo de Fluxo de Controle Interprocedimental (GFCI). Neste processo cada GFC é percorrido para encontrar os *nós de chamada* de operação. Para cada *nó de*

chamada identificado, uma *aresta de chamada* é criada conectando o *nó de chamada* com o *nó de entrada* da operação correspondente. Na sequência, uma *aresta de retorno* é criada para conectar o *nó de saída* da operação com um *nó de retorno*. Para construção do GFCI utilizamos o Algoritmo 4, apresentado na Seção 5.1.

O grafo resultante do Algoritmo 4 ainda é incompleto, pois falta incluir o fluxo excepcional interprocedimental, que é demonstrado pelas arestas de exceção interprocedimentais. Estas arestas ligam um nó de saída excepcional de uma operação com o nó do respectivo tratador de exceções contido em outra operação. Mas para obter o fluxo completo é necessário propagar as definições das variáveis de exceção entre os procedimentos através de uma análise de fluxo de dados interprocedimental, para facilitar esta tarefa um novo grafo é gerado a partir do GFCI. O último grafo gerado é o Grafo Resumido da Interface (GRI), o processo de geração é mostrado na Seção 5.2.

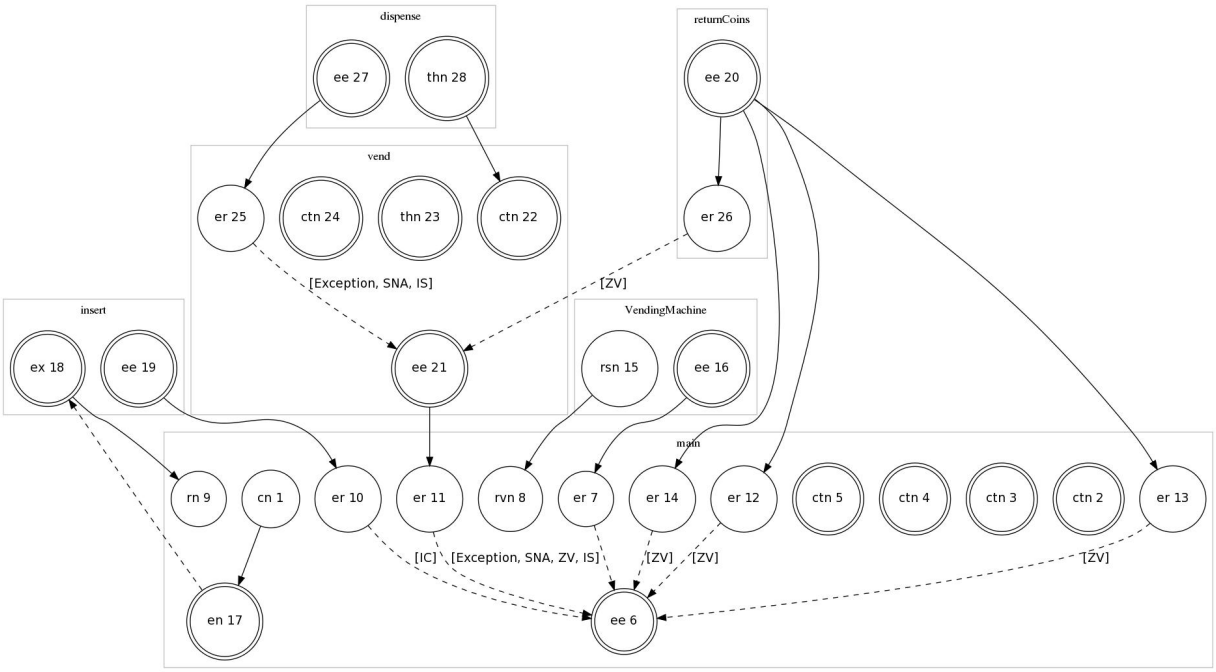


Figura 7.34: GRI da Máquina de Vendas

A Figura 7.34 mostra o grafo gerado a partir dos nós de interface do GFCI. Após a execução da análise de fluxo de dados no GRI, é possível identificar quais são os tipos de exceção que são propagados ao longo da pilha de chamadas.

A imagem do GFCI gerado para a Máquina de Vendas é muito grande para ser inserida neste documento e está disponível para consulta no seguinte endereço: *http://www.students.ic.unicamp.br/~ra066147/grafos/ICFG2.jpg*

Tabela 7.3: *Exceções lançadas X Tratadores de Exceções - Máquina de Vendas*

Operação	**Exceção Lançada**	**Tratador de Exceções**	**Exceção do Tratador**
insert()	IC	main()	IC
vend()	ZV	main()	ZV
vend()	IS	main()	S
vend()	SNA	main()	S
dispense()	IA	main()	IA
dispense()	IS	vend()	S
dispense()	SNA	vend()	S
returnCoins()	ZV	vend()	ZV
returnCoins()	ZV	main()	ZV

A Tabela 7.3 mostra o resultado da análise realizada sobre o GFCI, são mostrados os tratadores de exceções presentes no modelo e quais são as exceções que cada um é capaz de tratar. No caso da Máquina de vendas não existe nenhuma exceção sem um tratador apropriado.

Capítulo 8

Conclusões e Trabalhos Futuros

A abordagem proposta neste trabalho apresenta uma técnica para a validação do fluxo excepcional interprocedimental a partir de um modelo comportamental, o diagrama de atividades da UML 2.0. O objetivo desta validação é identificar as falhas na modelagem do comportamento excepcional, que podem ocorrer no momento da definição e lançamento de uma exceção, ou então, no momento da captura e tratamento da exceção. O trabalho enfoca a modelagem de componentes de software tolerantes a falhas, pois neste contexto o comportamento excepcional é modelado em operações separadas do comportamento normal. Também é muito comum que as exceções sejam propagadas entre os componentes de um sistema, ou que sejam passadas como parâmetros de uma operação, ou ainda, que sejam definidas a partir de um parâmetro de retorno de uma operação. Dessa forma, justifica-se a necessidade de uma análise de fluxo de dados interprocedimental que inclua todas as operações do sistema, para possibilitar a inferência dos tipos de exceções que podem ser lançadas por uma operação.

O método proposto reúne várias técnicas presentes na literatura, algumas foram adaptadas e outras estendidas. Adaptamos o GFC e GFCI apresentados por Sinha e Harrold [Sinh 99] para serem criados a partir do GA, estendemos o GRI proposto por Harrold e Soffa [Harr 94b], incluindo novos nós e arestas responsáveis pela propagação das exceções ao longo da pilha de chamadas. O resultado é uma nova abordagem, que executa análises de fluxo de dados intraprocedimentais e interprocedimentais para poder representar o fluxo excepcional completo do modelo comportamental. Outro resultado do trabalho é a implementação de uma ferramenta de apoio, a ferramenta *ADEX*. A ferramenta oferece uma interface gráfica para o usuário executar as transformações de modelos, visualizar os grafos resultantes e mostra informações úteis para auxiliar o processo de validação.

Uma dificuldade para a aplicação da abordagem proposta no contexto da MDD é o fato de que para atender diferentes plataformas, com uma semântica de tratamento de exceções diferente da semântica do DA, é necessário adaptar o modelo para deixá-lo em

conformidade com a semântica da linguagem alvo. Porém, esta adaptação deve ocorrer somente durante o refinamento do modelo para uma de plataforma específica. O método de validação proposto segue sempre a semântica de tratamento de exceções do DA de acordo com a especificação da UML 2.0. Nesse aspecto a abordagem não é tão flexível como o desejado para uma abordagem MDD. Outra questão é sobre o percentual do código gerado, caso a abordagem MDD utilizada seja capaz de gerar o código completo a validação do fluxo excepcional realizada antes da geração do código é valida, caso contrário, se o usuário tiver que implementar parte do código ou modificar o código gerado, a validação do modelo pode não ser suficiente para atender os requisitos de confiabilidade do componente de software.

A técnica apresentada possibilita a realizar outras análises de fluxo de dados, como por exemplo, uma análise para computar as cadeias de definição-uso contidas no modelo. Dessa forma é possível utilizar as informações de fluxo de dados para a derivação de casos de teste. Outra possibilidade seria utilizar o resultado da análise de fluxo de dados para resolver o problema criado pela herança, polimorfismo e ligação dinâmica na chamada dos métodos. Da mesma forma como as definições incidentes servem para identificar os tipos de exceções lançadas por uma ação *RaiseException*, é factível determinar quais são os métodos que podem ser chamados por uma ação do tipo *CallOperationAction*, a partir das definições do objeto associado ao pino de entrada *target*.

Outro trabalho futuro poderia ser uma extensão da abordagem proposta para tratar fluxos concorrentes, permitindo a utilização de outras atividades disponíveis no DA, como por exemplo, o nó de região expandida, o nó bifurcar e o nó mesclar.

Também será necessário realizar futuramente uma análise qualitativa da metodologia e dos algoritmos para avaliar a ocorrência de falsos positivos e falsos negativos. Um falso negativo ocorre quando o programa contém falhas não encontradas pela ferramenta, dando a falsa sensação de que não existem falhas, na verdade significa que a ferramenta não foi capaz de encontrá-las. Já um falso positivo ocorre quando a ferramenta aponta falhas não existentes.

8.1 Contribuições

Como contribuições deste trabalho ressaltamos:

1. **Classificações das Ações da UML 2.0 quanto à escrita ou leitura de dados**: Neste trabalho apresentamos o resultado de estudo detalhado da semântica das Ações da UML 2.0 para determinar quais propriedades de uma Ação devem ser consideradas para caracterizar um escrita (definição) ou leitura de dados (uso). A Tabela 2.2 apresentada na seção 2.2.2 mostra a classificação das ações.

2. **O Método de validação do fluxo excepcional**: A principal contribuição deste trabalho é o método proposto para a validação do fluxo excepcional a partir do modelo comportamental. O diferencial deste método é a possibilidade da validação do fluxo excepcional antes da geração do código-fonte.

3. **Ferramenta de Validação**: A criação da ferramenta *ADEX* para apoiar a tarefa de análise do fluxo excepcional, possibilitou a automação necessária para verificar a existência de falhas de projeto durante a modelagem do comportamento do sistema.

4. **Integração com o Método MDCE+**: A abordagem proposta neste trabalho foi integrada com sucesso ao método MDCE+ [Brit 05]. Esse método foi proposto para auxiliar na modelagem do comportamento excepcional de sistemas baseados em componentes. Melhorando a confiança no funcionamento do sistema e oferecendo uma maneira sistemática para modelar as exceções e os seus respectivos tratadores. Porém o método prevê a validação do fluxo excepcional apenas no nível arquitetural. O método MDCE+ utiliza diagramas de atividades para representar os cenários arquiteturais utilizados para validações do fluxo excepcional. A integração proposta [Ferr 11] consiste no refinamento dos diagramas até alcançar o nível de detalhamento necessário para realizar a validação com a ferramenta *ADEX*. A maior contribuição deste trabalho é o execução da análise do fluxo excepcional durante as fases de especificação e projeto, o que permite a detecção precoce de falhas nas definições dos mecanismos de tratamento de exceções, diferentemente das abordagens convencionais que buscam por falhas somente depois de implementar o código-fonte.

8.2 Publicações

Durante a condução desse trabalho, foram produzidas três publicações, sendo uma em congresso internacional, uma em congresso nacional e um relatório técnico. A seguir, é mostrado um breve resumo de cada uma delas.

1. **Validation of Exception Handling in the Development of Dependable Component-Based Software Systems, LADC'11 - Fifth Latin-American Symposium on Dependable Computing, 2011 [Ferr 11]**. *Jeferson Ferreira, Patrick H. da Silva Brito, Eliane Martins e Cecilia M. F. Rubira*: Este trabalho apresenta uma abordagem que permite sistematizar a validação do comportamento excepcional do sistema, tanto no nível da arquitetura do software quanto no nível detalhado do projeto. No nível de arquitetura de software, a solução proposta é baseada na especificação e verificação de cenários arquiteturais. No nível detalhado do projeto, a solução proposta consiste na utilização de uma ferramenta de análise

estática para coletar as informações relativas ao fluxo excepcional em um dado modelo comportamental, para auxiliar na tarefa de validação do fluxo excepcional. Esta análise pode antecipar a detecção e consequentemente a correção de falhas durante a fase de especificação.

2. **Fluxo de Exceções Intraprocedimentais a partir do Diagrama de Atividades da UML 2.0, Relatório Técnico IC-10-11, Instituto de Computação (UNICAMP), 2010** [**Ferr 10**]. *Jeferson Ferreira e Eliane Martins*: Este relatório apresenta uma abordagem para a obtenção do fluxo de exceções intraprocedimentais de um modelo comportamental. A abordagem proposta transforma o Diagrama de Atividades da UML em um Grafo de Fluxo de Dados. A semântica das ações da UML é utilizada para determinar as definições e usos das variáveis e também identificar os pontos onde as exceções são lançadas. Uma análise de fluxo de dados é executada no grafo resultante para determinar quais são os tipos de exceções que podem ser lançadas. O fluxo de exceções é demonstrado por arestas que ligam os nós que lançam as exceções com os nós que capturam e fazem o seu tratamento.

3. **Análise de Fluxo de Controle e Dados a partir do Diagrama de Atividades da UML 2.0, SAST'09 - Workshop Brasileiro de Teste de Software Sistemático e Automatizado, 2009** [**Ferr 09**]. *Jeferson Ferreira e Eliane Martins*: Este trabalho apresenta uma abordagem para a transformação do Diagrama de Atividades da UML em um Grafo de Fluxo de Dados. A abordagem proposta captura o fluxo de dados e utiliza a semântica das ações da UML para identificar as definições e usos das variáveis contidas no modelo. A vantagem oferecida é a possibilidade de realizar análises de fluxo de dados no grafo resultante, oferecendo entre outros recursos, subsídios para a geração de requisitos de testes baseados no fluxo de dados.

Referências Bibliográficas

[Aho 86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers-Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 1986.

[Amma 08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[Bass 97] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[Bind 99] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Bock 03a] C. Bock. "UML 2 Activity and Action Models". *Journal of Object Technology*, Vol. 2, No. 4, July-August 2003.

[Bock 03b] C. Bock. "UML 2 Activity and Action Models Part 2: Actions". *Journal of Object Technology*, Vol. 2, No. 5, September-October 2003.

[Booc 93] G. Booch. *Object-oriented analysis and design with applications, 2nd ed.* Benjamin-Cummings Publishing Co., 1993.

[Brit 05] P. H. S. Brito, C. R. Rocha, F. C. Filho, E. Martins, and C. M. F. Rubira. "A Method for Modeling and Testing Exceptions in Component-Based Software Development". *Lectures Notes in Computer Science*, Vol. 3747, pp. 61–79, 2005.

[Cach 08] N. Cacho, T. Cottenier, and A. Garcia. "Improving robustness of evolving exceptional behaviour in executable models". In: *Proceedings of the 4th international workshop on Exception handling*, pp. 39–46, ACM, New York, NY, USA, 2008.

[Chan 01] B. mo Chang, J. wu Jo, K. Yi, and K. moo Choe. "Interprocedural Exception Analysis for Java". 2001.

[Choi 99] J. deok Choi, D. Grove, M. Hind, and V. Sarkarrka. "Efficient and precise modeling of exceptions for the analysis of Java programs". *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 21–31, 1999.

[Cott 06] T. Cottenier, A. V. D. Berg, and T. Elrad. "The Motorola WEAVR: Model Weaving in a Large Industrial Context". In: *in Proceedings of the International Conference on AspectOriented Software Development, Industry Track*, 2006.

[Cris 89a] F. Cristian. "Exception Handling". In: T. Anderson, Ed., *Dependability of Resilient Computers*, pp. 68–97, Blackwell Scientific Publications, 1989.

[Cris 89b] F. Cristian. "Exception Handling". In: T. Anderson, Ed., *Dependability of Resilient Computers*, pp. 68–97, Blackwell Scientific Publications, 1989.

[Dela 07] M. E. Delamaro, J. C. Maldonado, and M. Jino. *Introdução ao Teste de Software*. Editora Campus, 2007.

[Ecli 10] Eclipse. *Model Development Tools (MDT) - UML2*. http://wiki.eclipse.org/MDT-UML2, August 2010.

[Ferr 09] J. Ferreira and E. Martins. "Análise de Fluxo de Controle e Dados a partir do Diagrama de Atividades da UML 2.0". *SAST'2009 - Brazilian Workshop on Systematic and Automated Software Testing*, August 2009.

[Ferr 10] J. Ferreira and E. Martins. "Fluxo de Exceções Intraprocedimentais a partir do Diagrama de Atividades da UML 2.0". Tech. Rep., 2010.

[Ferr 11] J. Ferreira, E. Martins, C. Rubira, and P. Brito. "Validation of Exception Handling in the Development of Dependable Component-Based Software Systems". *Fifth Latin-American Symposium on Dependable Computing (LADC 2011)*, 2011.

[Filh 05] F. C. Filho, P. H. S. Brito, and C. M. F. Rubira. "A framework for analyzing exception flow in software architectures". *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 4, pp. 1–7, 2005.

[Fu 04] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. "Testing of Java Web Services for Robustness". In: *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 23–34, ACM Press, 2004.

[Fuen 08] L. Fuentes, J. Manrique, and P. Sánchez. "Execution and simulation of (profiled) UML models using pópulo". In: *MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering*, pp. 75–81, ACM, New York, NY, USA, 2008.

[Gans 00] E. R. Gansner and S. C. North. "An open graph visualization system and its applications to software engineering". *SOFTWARE - PRACTICE AND EXPERIENCE*, Vol. 30, No. 11, pp. 1203–1233, 2000.

[Garl 00] D. Garlan, R. T. Monroe, and D. Wile. "Foundations of component-based systems". Chap. Acme: architectural description of component-based systems, pp. 47–67, Cambridge University Press, New York, NY, USA, 2000.

[Good 75] J. B. Goodenough. "Exceptional handling: Issues and a proposed notation". *Commun. ACM*, Vol. 18, No. 12, pp. 683–696, 1975.

[Gosl 05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[Guer 04] P. A. d. C. Guerra, F. C. Filho, V. A. Pagano, and C. M. F. Rubira. "Structuring Exception Handling for Dependable Component-Based Software Systems". In: *Proceedings of the 30th EUROMICRO Conference*, pp. 575–582, IEEE Computer Society, Washington, DC, USA, 2004.

[Harr 94a] M. J. Harrold and G. Rothermel. "Performing Data Flow Testing on Classes". *ACM SIGSOFT Symp. on the Foundations of Softw. Eng.*, pp. 154–163, 1994.

[Harr 94b] M. J. Harrold and M. L. Soffa. "Efficient Computation of Interprocedural Definition-Use Chains". *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 2, pp. 175–204, 1994.

[Jack 02] D. Jackson. "Alloy: a lightweight object modelling notation". *ACM Trans. Softw. Eng. Methodol.*, Vol. 11, pp. 256–290, April 2002.

[Jaco 94] I. Jacobson, M. Christerson, and L. L. Constantine. *The OOSE method: a use—case-driven approach*. SIGS Publications Inc., 1994.

[Klep 03] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Knie 08] C. Knieke, M. Huhn, and M. Lochau. "Modeling and Validation of Executable Requirements Using Live Activity Diagrams". In: *Software Engineering Research, Management and Applications, 2008. SERA '08. Sixth International Conference on*, pp. 51 –58, August 2008.

[Lapr 95] J.-C. Laprie. "Dependable computing: concepts, limits, challenges". In: *In Proceedings of the Twenty-Fifth international conference on Fault-tolerant computing*, pp. 42–54, IEEE Computer Society, Washington, DC, USA, 1995.

[Lee 90a] Lee and Anderson. *Fault-Tolerance: Principles and Practice - 2ª Edição*. Springer-Verlag, January 1990.

[Lee 90b] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice, 2nd edition*. Springer-Verlag New York, Secaucus, NJ, USA, 1990.

[Lion 96] J.-L. Lions. "Ariane 5 Flight 501 Failure: Report by the Inquiry Board". *European Space Agency*, 1996.

[Luer 00] C. Lüer and D. S. Rosenblum. "WREN - An Environment for Component-Based Development". *Software Engineering Notes*, Vol. 26, pp. 207–217, 2000.

[Mesq 01a] G. R. de Mesquita Ferreira. *Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes*. Master's thesis, Universidade Estadual de Campinas, 2001.

[Mesq 01b] G. R. de Mesquita Ferreira. *Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes*. Master's thesis, Universidade Estadual de Campinas, 2001.

[OMG 07a] OMG. *UML 2.1.2 Superstructure Normative Specification*. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, November 2007.

[OMG 07b] OMG. *XML Metadata Interchange (XMI) Normative Specification*. http://schema.omg.org/spec/XMI/2.1.1/PDF, December 2007.

[Pere 07] I. R. D. C. Perez, E. Martins, and J. E. Viégas. "Uso de Modelos da UML em Testes de Componentes". *VIII Workshop de Teste e Tolerância a Falhas*, p. 128, 2007.

[Perr 00] D. E. Perry, A. Romanovsky, and A. Tripathi. "Current Trends in Exception Handling". *IEEE Trans. Softw. Eng.*, Vol. 26, pp. 921–922, October 2000.

[Rapp 85] S. Rapps and E. J. Weyuker. "Selecting Software Test Data Using Data Flow Information". *IEEE - Transactions on Software Engineering*, Vol. 11, No. 4, 1985.

[Reim 03] D. Reimer and H. Srinivasan. "Analyzing Exception Usage in Large Java Applications". In: *Proc. of Workshop on Exception Handling in Object-Oriented Systems*, July 2003.

[Robi 03] M. P. Robillard and G. C. Murphy. "Static analysis to support the evolution of exception structure in object-oriented systems". *ACM Trans. Softw. Eng. Methodol.*, Vol. 12, No. 2, pp. 191–221, 2003.

[Robi 99] M. P. Robillard and G. C. Murphy. "Analyzing exception flow in Java programs". In: *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 322–337, Springer-Verlag, London, UK, 1999.

[Rubi 05] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. "Exception handling in the development of dependable component-based systems". *Software—Practice and Experience*, 2005.

[Rumb 90] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1990.

[Sars 05] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. "A New Approach to Combine Models and Code in Model Driven Development". *Conference on Software Engineering Research and Practice*, 2005.

[Sinh 99] S. Sinha and M. J. Harrold. "Criteria for testing exception-handling constructs in java programs". In: *In Proceedings of the International Conference on Software Maintenance*, pp. 265–274, IEEE, 1999.

[Slom 87] M. Sloman and J. Kramer. *Distributed systems and computer networks*. Prentice Hall International (UK) Ltd., Hertfordshire, 1987.

[Stah 06] T. Stahl and M. Volter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons, Chichester, 2006.

[Stei 09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd Ed., 2009.

[Stor 04] H. Storrle. "Structured nodes in UML 2.0 activities". *Nordic J. of Computing*, Vol. 11, pp. 279–302, September 2004.

[Stor 05] H. Storrle. "Semantics and Verification of Data Flow in UML 2.0 Activities". *Eletronic Notes in Theorical Computer Science*, pp. 35–52, 2005.

[Szyp 02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2002.

[Tabi 08] M. Z. Z. I. Tabinda Waheed and Z. I. Malik. *Model Driven Architecture – Foundations and Applications*, Chap. Data Flow Analysis of UML Action Semantics for Executable Models, pp. 79–93. Vol. Volume 5095/2008 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, June 2008.

[Weis 81] M. Weiser. "Program slicing". In: *In Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, 1981.