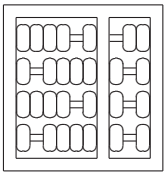


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Fluxo de Exceções Intraprocedimentais a partir do Diagrama de Atividades da UML 2.0

J. Ferreira and E. Martins

Technical Report - IC-10-11 - Relatório Técnico

March - 2010 - Março

Fluxo de Exceções Intraprocedimentais a partir do Diagrama de Atividades da UML 2.0

Jeferson Ferreira* Eliane Martins†

Resumo

Com a evolução do Desenvolvimento Dirigido por Modelos torna-se necessário evoluir as técnicas de extração das informações implícitas no modelo que possam auxiliar no processo de desenvolvimento, validação ou testes. Um desafio para o modelador é a construção de modelos comportamentais complexos que sejam consistentes e confiáveis. Outro desafio é modelar e validar o fluxo de exceções desses modelos. O objetivo deste relatório é apresentar 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.

1 Introdução

A UML (Unified Modeling Language) [1] 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. Até a versão 1.5 da UML, o DA era baseado em máquinas de estados capazes de processar somente um único fluxo de tokens. A partir da UML 2.0 esse diagrama tornou-se independente e capaz de processar múltiplos fluxos de tokens, possibilitando modelar, além do fluxo de controle, o fluxo de dados.

Durante o desenvolvimento de um software, muitos modelos são criados e acabam não sendo reusados nas fases seguintes do processo, como por exemplo, na fase de implementação

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

e também durante os testes. Dessa forma os modelos não são mantidos e tornam-se obsoletos. Uma das abordagens recentes para resolver este problema é o paradigma de Desenvolvimento Dirigido por Modelos (Model-Driven Development - MDD) [2]. Uma das realizações de MDD que têm obtido resultados concretos é a arquitetura MDA (Model-Driven Architecture) [3]. O foco principal deste conceito é preencher a lacuna existente entre as fases de análise e projeto e a fase 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, transformando o modelo independente de plataforma PIM (Platform Independent Model) em um modelo específico de plataforma PSM (Platform Specific Model), que inclui informações sobre tipo de dados, banco de dados, interfaces, etc. Finalmente o modelo PSM é transformado em código 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. [5]. 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 idéia 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 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 [4] 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.

O objetivo da abordagem apresentada neste relatório é fazer com que técnicas de análise, empregadas para detectar anomalias no código, possam ser usadas mais cedo no ciclo de desenvolvimento. Dessa forma, corrige-se o modelo, o qual pode servir de base tanto para os testes quanto para a geração de código, em um paradigma com o MDD. A proposta apresentada extrai as informações de fluxo de controle, fluxo de dados e fluxo de exceções do modelo. O DA é transformado em outro modelo, um Grafo de Fluxo de Dados (GFD). A semântica das ações da UML é utilizada para identificar as definições e usos das variáveis e também identificar os pontos onde as exceções são lançadas. 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.

Embora a natureza do fluxo de exceções seja interprocedimental, este trabalho é limitado ao contexto intraprocedimental, que é o primeiro passo para a análise interprocedimental.

Apesar disso, esta análise é útil para determinar, no nível de operação, algumas anomalias do fluxo de dados, como por exemplo, se existem exceções não capturadas por uma ação que não são definidas em sua assinatura.

Este relatório está estruturado da seguinte forma. A Seção 2 apresenta os conceitos básicos abordados neste trabalho. Na Seção 3 são apresentados os detalhes da transformação dos modelos, a ferramenta implementada para realizar a transformação e os detalhes da análise de fluxo de dados utilizada. Na Seção 4 é feita uma análise dos trabalhos relacionados e por fim na Seção 5 são apresentadas as conclusões e trabalhos futuros.

2 Fundamentação Teórica

Nesta seção são apresentados alguns conceitos básicos sobre fluxo de controle e fluxo de dados. São apresentados o grafo de fluxo de controle e o grafo de fluxo de dados.

2.1 Fluxo de Controle

As técnicas de análise de fluxo de controle 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. Um caminho p é uma seqüência de nós $[n_1, n_2, \ldots, n_m]$, onde cada par de nós é uma aresta. O comprimento de p é igual ao número de arestas do caminho. Um subcaminho de p é uma subseqüência de nós em p [6, 7].

2.2 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 1 apresenta um exemplo de GFD.

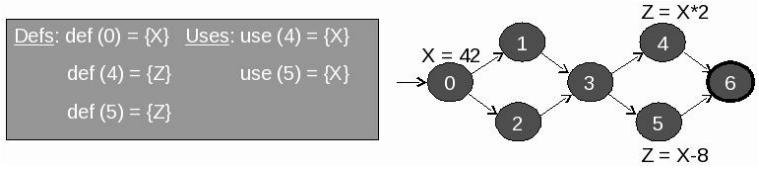


Figura 1: Grafo de Fluxo de Dados

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 $x_1, ..., x_n$ [17].

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 [6].

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 $du(l_n, l_m, v)$ [7].

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 [17].

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

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

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 [8].

Na seção 3.3 é apresentado um exemplo de análise de fluxo de dados utilizada para inferir quais são os tipos de exceção que podem ser lançadas em um dado ponto do modelo.

3 O Diagrama de Atividades e a Semântica de Ações da UML

A UML (Unified Modeling Language)[1] é uma linguagem de modelagem e documentação de software que surgiu em 1996 com o objetivo de acelerar a transição para a Orientação à Objeto. 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[12], Rumbaugh[13] e Jacobson[14] foram fundidos numa única linguagem de modelagem. Atualmente a linguagem UML é mantida pela OMG (Object Management Group).

O diagrama de atividades é um dos diversos diagramas definidos pela UML. 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.

Até a versão 1.5 da UML o fluxo de dados e controle eram representados pelos diagramas de estado e de interação. Na versão 1.5 surgiu a primeira versão do diagrama de atividades, que herdou a forma do diagrama de estados alterando a notação para parecer uma diagrama de fluxo. Na versão 2.0 da UML esse diagrama se tornou independente, deixando inclusive de se basear em máquinas de estados e passando a se basear em Redes de Petri, utilizadas para modelar comportamento concorrente em sistemas distribuídos. Nesta última versão foram incluídos vários recursos, como: pinos para modelar o fluxo de dados das atividades, parâmetros em atividades, conjunto de parâmetros, regiões interruptíveis e a modelagem explícita do fluxo de objetos.

Apresentamos a seguir os elementos do diagrama que serão utilizados nesse trabalho. O leitor interessado poderá consultar a especificação da UML[1] 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, representa 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 exibe um exemplo de Diagrama de Atividades com fluxo de controle e dados.

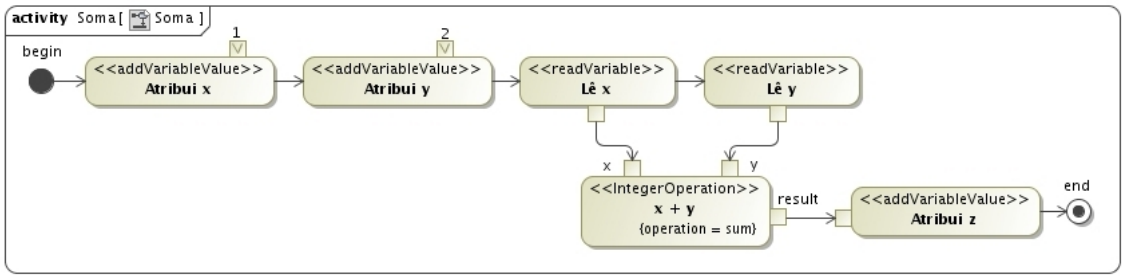


Figura 2: Diagrama de Atividades

3.1 Atividades e Ações

As atividades são comportamentos que modelam uma execução não-atômica, já uma ação representa um processamento atômico[9]. 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 provêem 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[10].

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:

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

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

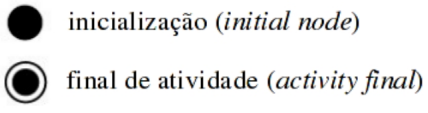


Figura 3: Atividades Básicas

- *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 4), descritos a seguir:
  - *Nó de Decisão (DecisionNode):* direciona o fluxo em uma direção ou outra, dependendo da avaliação da expressão associada.
  - *Nó de Intercalação (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.

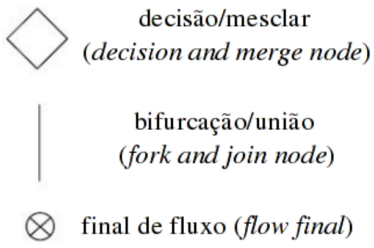


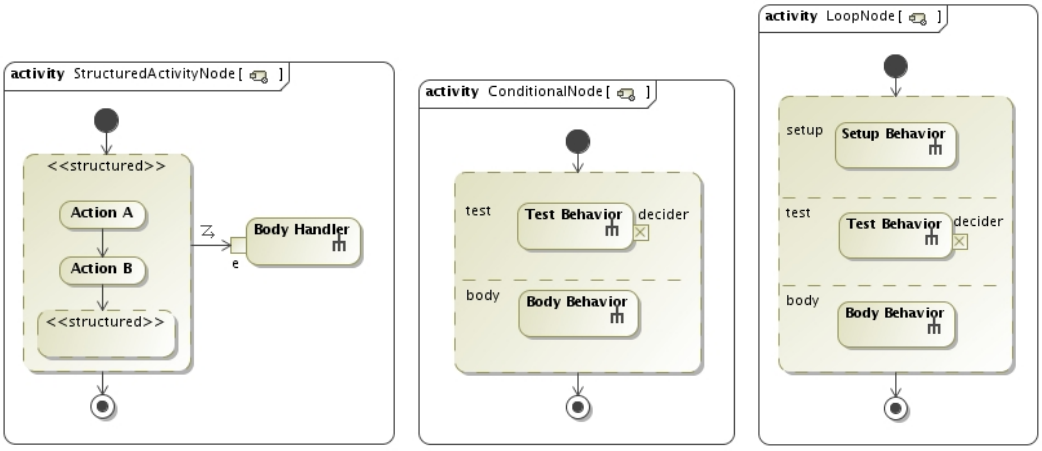
Figura 4: Atividades Intermediárias

- *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 lêem e escrevem dados nas variáveis, estas ações devem ser sequenciadas pelo fluxo de controle para prevenir condições de corrida entre ações que leêm 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 acessados 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, inclusive outros nós de atividade estruturada aninhados. Os nós subordinados devem pertencer a apenas um nó de atividade estruturada, embora possam ser aninhados. Uma atividade estruturada também pode conter manipuladores de exceção. Um exemplo pode ser visto na Figura 5a.
- *Nó Condicional (ConditionalNode):* É uma atividade estruturada que representa uma escolha exclusiva entre algumas alternativas (Figura 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 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 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.

- *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 5a). No exemplo da Figura 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 representa a referência ao objeto de exceção. O tipo de exceção capturada pela aresta de



(a) (b) (c)

Figura 5: Atividades Estruturadas

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 1. A Figura 16 mostra um exemplo mais completo de diagramas de atividades.

Tabela 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 de Intercalação (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)

3.2 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 [1] apresenta um conjunto de ações com uma semântica bem definida. No contexto de fluxo de dados uma ação pode ser classificadas 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 possue 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.

Tabela 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		

- *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 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.

3.3 Tratamento de exceções no AD

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 [27].

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 6 é mostrada a classificação das exceções 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

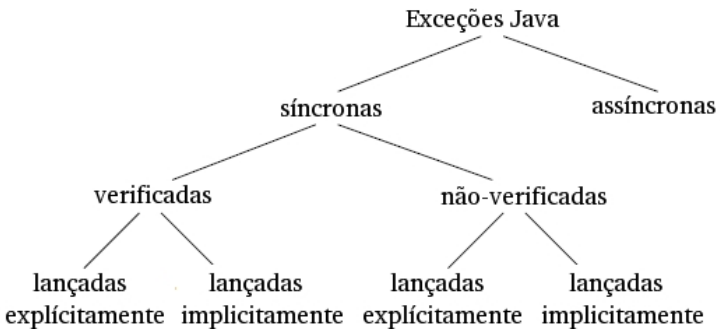


Figura 6: Classificação dos Tipos de Exceção do Java

modeladas explicitamente.

Fazendo uma analogia com a linguagem de programação Java, a ação *RaiseExceptionAction* 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 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.

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.

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 7 um 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.

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

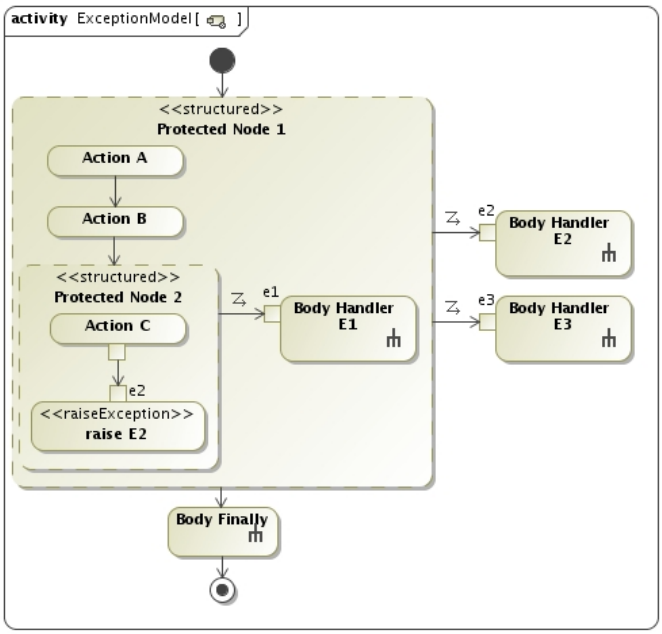


Figura 7: Modelo de Exceções do DA

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 um execução normal ou excepcional. No exemplo da Figura 7 a ação *"Body Finally"* é utilizada para modelar um bloco de finalização.

4 Obtenção do Grafo de Fluxo de Dados

Para realizar uma análise de fluxo de dados no modelo, é preciso transformar o Diagrama de Atividades em um grafo de fluxo de dados (vide seção 2.2). Essa transformação é feita em três passos: (i) criação do Grafo de Atvidades, um grafo auxiliar utilizado para representar cada operação (procedimento) especificado no diagrama em um único modelo; (ii) criação do Grafo de Fluxo de Controle a partir do Grafo de Atividades e (iii) criação do Grafo de Fluxo de dados, que é uma extensão do Grafo de Fluxo de Controle contendo as informações sobre o fluxo de dados. Em seguida apresentaremos os detalhes de cada um destes passos.

4.1 Criação 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. 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 de Intercalação) 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. Esta representação intermediária é utilizada apenas durante o processo de construção do GFD, e será apresentada neste documento para demonstrar este processo. As principais justificativas para a utilização deste grafo 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.

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 em um DA. O algoritmo mostrado na Figura 8 é utilizado para realizar esta ordenação. O resultado do algoritmo é um grafo direcionado, onde cada nó representa uma atividade ou ação 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 e atividades. Para começar a executar, uma ação deve saber quando começar e quais são as suas entradas. Estas condições são 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 ordem de execução possível 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 recursivo mostrado na Figura 10 é o responsável pela geração dos subgrafos de todos os DA e 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 9 mostra os nós e arestas auxiliares utilizados para representar o fluxo de controle

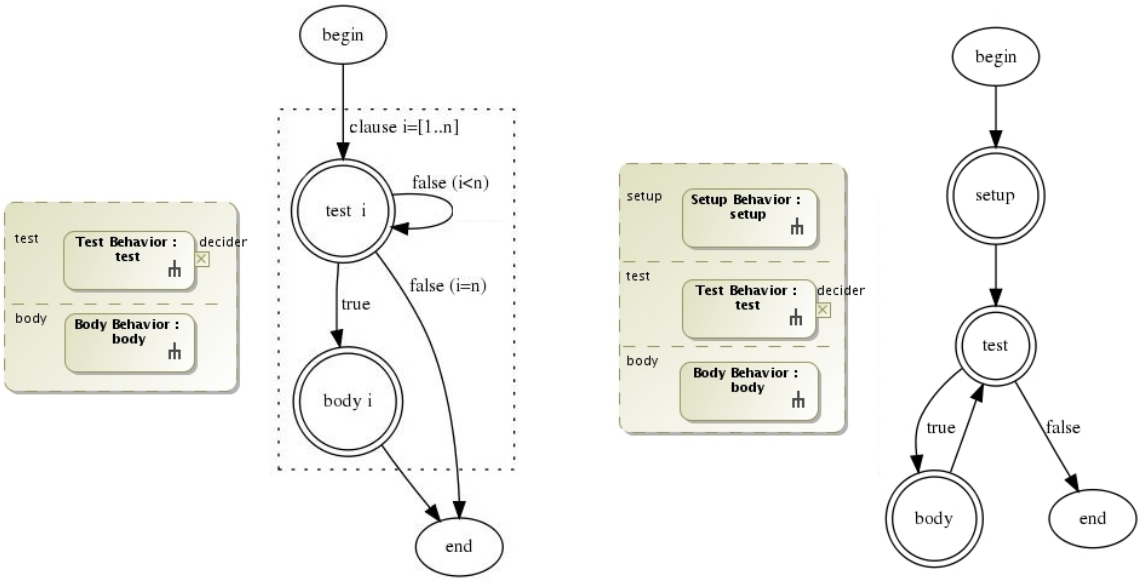
algoritmo: ***orderingActivityNodes***
entrada: *node* : nó do diagrama de atividades
entrada: *edge* : aresta do diagrama de atividades
entrada: *predecessor* : nó predecessor
entrada/saída: *graph* : grafo de execução das atividades

```
1  begin
2     edges = all edges of node
3     if node is not visited
4        incomings = all incomings control edges of node
5        outgoings = all outgoings control edges of node
6     if node is not Pin
7        if incomings > 1
8           incomings = incomings - edge
9           if incomings == 0
10             node is visited
11             for each incoming edge of node do
12                sourceNode = incoming edge source
13                graph add edge(sourceNode, node)
14                graph add node
15                for each edge of node do
16                   nextNode = target node of edge
17                   call orderingActivityNodes(graph,nextNode,edge,node)
18       if outgoings > 1
19          sourceNode = incoming edge source
20          graph add edge(sourceNode, node)
21          for each edge of node do
22             nextNode = target node of edge
23             call orderingActivityNodes(graph,nextNode,edge,node)
24       else
25          node is visited
26          graph add edge(predecessor, node)
27          if node is Action
28             for each OutputPin of node do
29                nextNode = OutputPin
30                call orderingActivityNodes(graph,nextNode,null,node)
31    if node is InputPin
32       ownerNode = owner of node
33       if ownerNode is Action
34          if ownerNode is not visited
35             inputs = all input pins of ownerNode
36             inputs = inputs - node
37             node is visited
38          else
39             inputs = inputs - node
40          if inputs == 0
41             graph add edge(predecessor,ownerNode)
42             graph add ownerNode
43             call orderingActivityNodes(graph,node,null,ownernode)
44    else
45       for each edge of edges do
46          nextNode = target node of edge
47          call orderingActivityNodes(graph,nextNode,edge,predecessor)
48 end
```

Figura 8: Algoritmo para obter a ordem de execução das atividades

dessas atividades.

A Figura 9a 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, a área pontilhada representa as cláusulas condicionais que podem variar de 1 até *n*. Uma cláusula é formada por um par de seções: *test* e *body*. Os nós *test i* e *body i* representam respectivamente as seções *test* e *body* de uma cláusula *i*. A aresta (*test i*→*test i*) representa o fluxo de controle quando a condição da ação *test i* é falsa e ainda existe outra cláusula no nó condicional para ser testada. Quando a condição da ação *test i* é verdadeira o fluxo de controle segue pela aresta (*test i*→*body i*). A aresta (*test i*→*end*) representa o fluxo de controle quando a última condição testada é falsa, e por fim a aresta (*body i*→*end*) representa o final da execução de um nó condicional quando o fluxo de controle sai da ação *body i* e segue para o nó *end*.



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

Figura 9: Fluxo de Controle das Atividades Estruturadas

A representação gráfica de um Laço *LoopNode* e o seu respectivo fluxo de controle são exibidos na Figura 9b. 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 9b, representa um laço com a propriedade *isTestedFirst* setada com valor true. Esta propriedade booleana 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 9 as seções são representadas por um círculo duplo, para indicar que estes nós serão substituídos posteriormente pelo subgrafo da atividade vinculada.

algoritmo: ***processAD***
entrada: *activity* : diagrama de atividades
entrada/saída: *graphs* : coleção de grafos de atividades (GA)

```
1  void processAD(uml2.Activity activity, List graphs) {
2     ActivityGraph graph = new ActivityGraph(activity);
3     nodes = activity.getNodes();
4     edges = activity.getEdges();
5     structuredNodes = activity.getStructuredNodes();
6     /* processa cada no da atividade */
7     for(uml2.ActivityNode n : nodes) {
8        /* associa o no do GA com uma atividade UML */
9        ActivityGraphNode activityGraphNode = new ActivityGraphNode(n);
10       graph.addActivityGraphNode(activityGraphNode);
11       /* efetua o mapeamento do fluxo de dados */
12       dataFlowMapping(n,activityGraphNode,graph);
13    }
14    /* processa cada atividade estruturada aninhada */
15    for(uml2.StructuredActivityNode structured : structuredNodes) {
16       ActivityGraphNode activityGraphNode = new ActivityGraphNode(structured);
17       graph.addActivityGraphNode(activityGraphNode);
18       /* efetua o mapeamento do fluxo de dados */
19       dataFlowMapping(structured,activityGraphNode, graph);
20       if (structured instanceof uml2.ConditionalNode) {
21          /* cria nos auxiliares para cada secao do no condicional */
22       } else if (structured instanceof uml2.LoopNode) {
23          /* cria nos auxiliares para cada secao do laco */
24       } else { /* outra atividade estruturada    */
25          /* chamada recursiva do algoritmo */
26          processAD(structured);
27       }
28    }
29    /* processa cada aresta da atividade */
30    for(uml2.ActivityEdge e : edges) {
31       graph.addEdge(e);
32    }
33    graphs.add(graph);
34 }
```

Figura 10: Algoritmo da transformação de um DA em um Grafo de Atividades

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 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 9 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.

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

4.2 Criação do Grafo de Fluxo de Controle

No segundo passo, o GA gerado anteriormente 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.

Durante a construção do GFC alguns nós do GA devem ter um nó exclusivo 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. Embora nossa abordagem seja limitada ao fluxo de controle e de exceções intraprocedimental, o nó 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, facilitando a extensão da abordagem para tratar a análise interprocedimental. No final deste processo cada operação (procedimento) existente no modelo terá o seu GFC gerado. A Figuras 11 e 12 mostram os algoritmos utilizados para a construção do GFC.

algoritmo: ***generateCFG***
entrada: *aGraphs* : coleção de grafos de atividades dos procedimentos
saída: *cfgs* : coleção de grafos de fluxo de controle e dados (GFD)

```
1  List generateCFG(List aGraphs) {
2     List cfgs = new ArrayList();
3     /* processa cada grafo de atividades */
4     for(ActivityGraph ag : gGraphs) {
5        /* cria um novo grafo de fluxo de controle */
6        ControlFlowGraph cfg = ControlFlowGraph();
7        /* cria um novo no */
8        Node node = new Node();
9        cfg.add(node);
10       /* pega o primeiro no do grafo de atividades */
11       ActivityGraphNode first = ag.getFirstNode();
12       /* gera o grafo de fluxo de controle */
13       processCFG(first, node, cfg);
14       cfgs.add(cfg);
15    }
16    return cfgs;
17 }
```

Figura 11: Algoritmo para a geração da lista de Grafos de Fluxo de Dados

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:

$$CFGNodes = \langle N, iN, fN, EN \rangle$$

onde: N conjunto de nós do GFC
iN, fN nó inicial e nó final do GFC
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: ***processCFG***
entrada: m : nó do GA e n : nó do CFG
entrada/saída: *cfg* : Grafo de Fluxo de Controle

```
1  void processCFG(AGNode m, Node n, CFG cfg) {
2     /* recupera a atividade UML associada ao no do GA */
3     uml2.ActivityNode a = m.getActivity();
4     Node predecessor = n;
5     if ((a instanceof uml2.CallOperationAction)||
6         (a instanceof uml2.DecisionNode)||
7         (a instanceof AuxiliaryNode)) {
8        Node node = new Node();
9        cfg.addNode(node);
10       cfg.addEdge(predecessor, node);
11    } else {
12       n.addNodeActivity(m);
13    }
14    /* as definicoes e usos do no do GA sao capturados pelo no do GFC */
15    node.setUses(m.getUses());
16    node.setDefinitions(m.getDefinitions());
17    /* percorre cada aresta do GA */
18    for (GenericEdge se : m.getEdges()) {
19       AGNode s = (AGNode)se.getTarget();
20       processCFG(s,cfg, n); // recursive call
21    }
22 }
```

Figura 12: Algoritmo recursivo utilizado para a geração do GFC

4.3 Criação do Grafo de Fluxo de Dados

A geração do Grafo de Fluxo de Dados (GFD) está implícita nos dois passos anteriores. No primeiro passo, durante a geração do GA (algoritmo da Figura 10) cada ação do DA é avaliada e todas as informações referentes ao fluxo de dados são capturadas. 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 pode representar uma agrupamento de nós do GA. Todas as definições e usos desse conjunto de nós são propagadas para o nó do GFC.

A Figura 13 mostra o diagrama de classes utilizado para modelar o Grafo de Atividades e o Grafo de Fluxo de Controle. Mostra também as classes criadas para representar as informações de fluxo de dados e o seu relacionamento com os nós do GA e GFC.

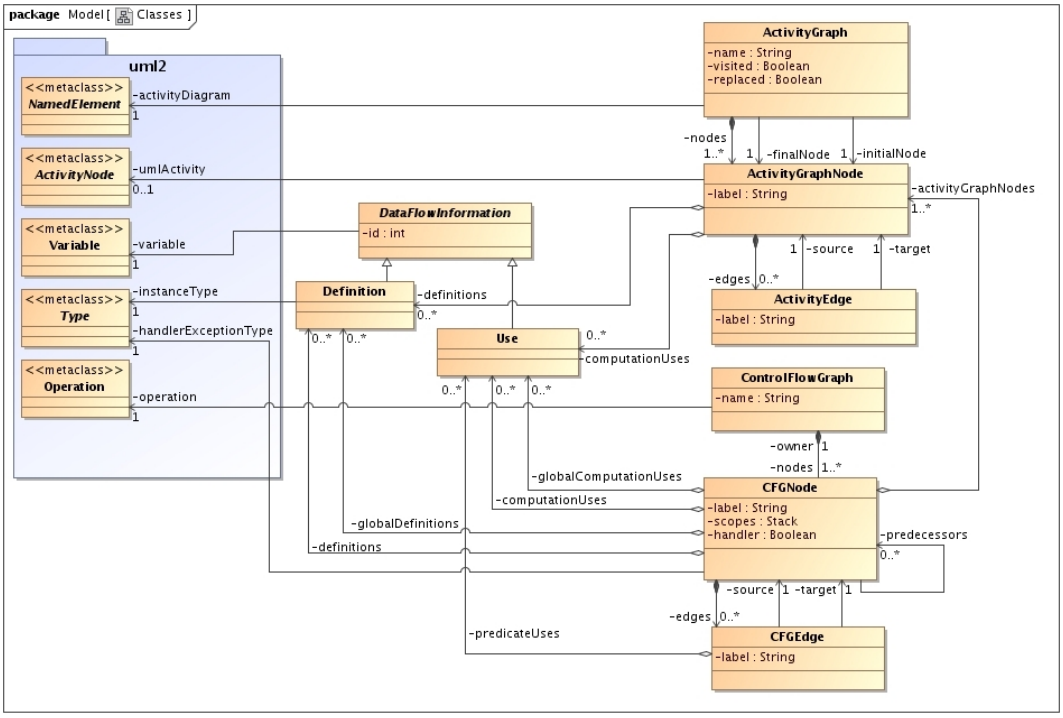


Figura 13: Diagrama de classes que modelam os grafos

As classes `Definition` e `Use` representam, respectivamente uma definição ou uso de uma variável contida no modelo. A classe abstrata `DataFlowInformation` representa a generalização das classes anteriores. Esta classe possui o relacionamento com a classe `Variable`, definida pelo meta modelo da UML para representar uma variável. A classe `Definition` também possui um relacionamento com a classe `Type`, especificada no meta-modelo da UML para representar o tipo de dados de uma variável, de um objeto ou de uma característica estrutural (como por exemplo, o atributo de uma classe). Este relacionamento define o atributo *instanceType*, que determina o tipo de dado da instância do objeto referenciado pela variável.

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

O GA é composto pelas seguintes classes: i) `ActivityGraph`: classe principal que representa o próprio GA; ii) `ActivityGraphNode`: representa um nó do GA; iii) `ActivityEdge`: representa uma aresta entre dois nós do GA.

A classe `ActivityGraph` possui um relacionamento com a classe `ActivityGraphNode` que define o conjunto de nós do GA denominado *nodes*. Esta classe também possui um relacionamento com a classe `NamedElement` que define o atributo *activityDiagram*. Este atributo vincula o GA com um diagrama de atividades ou uma atividade estruturada, este vínculo é utilizado durante o processo de geração do GA.

A classe `ActivityGraphNode` possui um relacionamento com a classe `ActivityEdge` que define o conjunto de arestas do nó do GA. Cada nó do GA pode estar relacionado a uma ação ou nó de atividade, a classe `ActivityNode`, definida no meta-modelo da UML, representa ambos os casos. O relacionamento com esta classe define o atributo *umlActivity*, que permite ao nó do GA fazer acesso ao elemento vinculado do DA. O nó do GA também possui dois conjuntos de informações de fluxo de dados. O conjunto *computationUses* é definido pelo relacionamento com a classe `Use`, representa os c-usos efetuados pelo nó. O conjunto *definitions* é definido pelo relacionamento com a classe `Definition`, que representa as definições efetuadas pelo nó. Os conjuntos são alimentados pelo mapeamento de fluxo de dados da ação UML vinculada ao nó do GA, de acordo com a Tabela 2.

O GFC é modelado pelas classes: i) `ControlFlowGraph`: classe principal que representa o próprio GFC; ii) `CFGNode`: representa um nó do GFC; iii) `CFGEdge`: representa uma aresta entre dois nós do GFC.

A classe `ControlFlowGraph` possui um relacionamento bi-direcional com a classe `CFGNode` que define o conjunto de nós do GFC denominado *nodes* do lado da classe `ControlFlowGraph` e o atributo *owner* do lado da classe `CFGNode`. A classe `ControlFlowGraph` possui também um relacionamento com a classe `Operation`, definida no meta-modelo da UML para representar uma operação de uma classe (método). A classe `CFGNode` possui um relacionamento com a classe `CFGEdge` que define o conjunto de arestas do nó do GFC. Cada nó do GFC pode agrupar um conjunto de nós do GA, o atributo *activityGraphNodes* representa o relacionamento do nó do GFC com a classe `ActivityGraphNode`.

A classe `CFGNode`, também possui um auto-relacionamento que define o atributo *predecessors*, este conjunto armazena todos os predecessores do nó. Já as informações de fluxo de dados do nó do GFC são representadas por quatro conjuntos. Os dois primeiros conjuntos *computationUses* e *globalComputationUses* são relacionamentos com a classe `Use` que definem respectivamente o conjunto de c-usos e c-usos globais. Já os dois últimos conjuntos *definitions* e *globalDefinitions* são relacionamentos com a classe `Definition` que definem respectivamente o conjunto de definições e definições globais. Além disso, quando um nó do GFC é considerado um tratador de exceções, o atributo *handler* é marcado como verda-

deiro e o atributo *handlerExceptionType*, definido pelo relacionamento com a classe `Type`, identifica o tipo de exceção que o tratador é capaz de capturar.

A classe `CFGEdge` possui um relacionamento com a classe `Use` que define o conjunto dos usos predicativos (p-usos) do nó do GFC. Os *p-usos* são identificados durante o processo de transformação do GA→GFC. As *definições* e *c-usos* contidas nos nós do GA são transferidos para o seu respectivo nó do GFC. Caso o nó do GFC seja um nó predicativo, ou seja, possui mais de uma aresta de fluxo de controle partindo dele, os *c-usos* do nó do GA são considerados predicativos e são inseridos no conjunto *p-uses[n,m]* de cada uma das arestas que partem do nó correspondente no GFC.

Para completar a geração do Grafo de Fluxo de Dados (GFD), os conjuntos de definições e usos do GFC são processados para encontrar as *definições globais* e *c-usos globais*, conforme a definição da seção 2.2.

4.4 Criação do Fluxo de Exceções Intraprocedimentais

Neste tópico apresentaremos o processo de criação do fluxo de exceções intraprocedimentais a partir de uma análise de fluxo de dados sobre o GFD. Esta análise consiste no cômputo das definições incidentes (reaching definitions). As definições incidentes são utilizadas para identificar quais são os tipos de exceção que podem ser lançados em um determinado nó do GFD.

Definição: 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 [8].

Para computar as definições incidentes é utilizado o clássico algoritmo iterativo proposto por Aho et al. [8]. A seguir são apresentados os conjuntos e a equação de fluxo de dados utilizados pelo algoritmo:

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

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.

Além do cômputo das definições incidentes, também precisamos inferir quais são os tipos de exceções que podem ser lançados por um determinado nó do GFD e quais são os possíveis tratadores de exceção que podem capturar essas exceções. Para isso, utilizamos o resultado da análise de definições incidentes e também informações do contexto das regiões interruptíveis, das atividades estruturadas e informações dos tratadores de exceção extraídas do modelo.

No caso 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 um objeto do tipo `Exception` ou uma referência de qualquer umas das exceções que são subclasses de `Exception`. O problema a ser resolvido é identificar, em um dado nó do GFD 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.

O que determina se um nó do GFD 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 no tópico 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 ao nó correspondente ao tratador de exceções compatível No caso da exceção não ser tratada, uma aresta de exceção irá ligar este nó até o nó correspondente à saída excepcional do procedimento.

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 incluída 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.

A análise de fluxo de dados auxilia não só na criação do GFD com fluxo de exceções, mas auxilia também para determinar as anomalias nesse fluxo. Assim, por exemplo, o modelador pode ser informado sobre as exceções que não são tratadas na operação, e determinar se elas serão lançadas para serem tratadas pela operação que efetuou a sua chamada. Outra alternativa de validação é comparar os tipos de exceções que são efetivamente lançados, com os tipos de exceções lançados na modelagem das operações.

5 Ferramenta XMI2Graph

Os algoritmos de transformação apresentados nas figuras 10, 11 e 12 foram implementados na linguagem Java[1]. Esta decisão foi tomada com base na facilidade encontrada para converter um modelo UML exportado no formato XMI (XML Metadata Interchange) [15],

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

em instâncias de objetos que representam esse modelo em total conformidade com a especificação da UML 2.0. Para realizar esta conversão foram utilizados os recursos disponíveis no Eclipse Modeling Framework (EMF) [16]. Este framework provê a implementação de todas as classes do meta-modelo da UML 2.0, bem como os recursos necessários para manipulação do arquivo XMI e instanciação dos objetos de acordo com a sua especificação.

O modelo UML utilizado na transformação pode ser elaborado por qualquer ferramenta UML que atenda os seguintes requisitos: a) Conformidade com a especificação da UML 2.0 ou versão superior; b) Suporte a semântica das ações UML; c) Funcionalidade de exportação para o formato XMI versão 2.1. Os exemplos apresentados neste relatório foram modelados pela ferramenta MagicDraw[2].

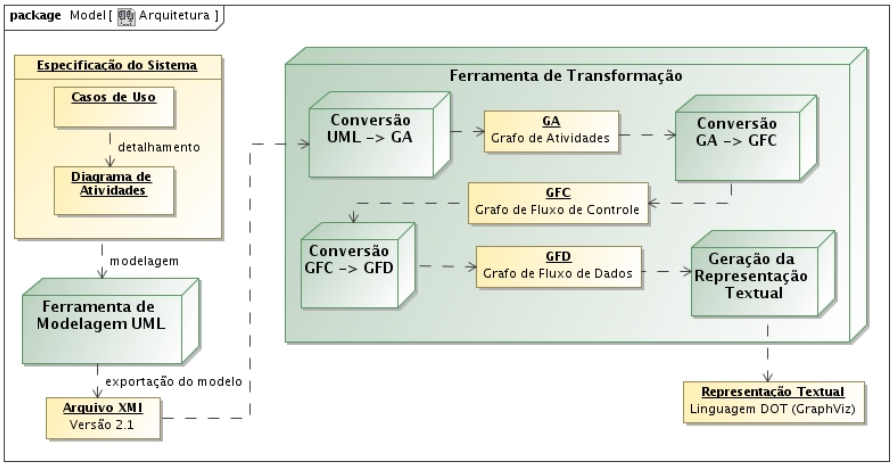


Figura 14: Processo de Transformação

O processo de transformação (Figura 14) tem início com a leitura do arquivo XMI gerado a partir da exportação do modelo UML pela ferramenta de modelagem. A ferramenta de transformação lê o arquivo XMI e utiliza os recursos do framework EMF para instanciar os objetos que representam os componentes contidos no modelo, como por exemplo: pacotes, interfaces, classes, atividades, ações, etc. O primeiro algoritmo de transformação (Figura 10) inspeciona os objetos do modelo e extrai as informações necessárias para construir o Grafo de Atividades (GA) de acordo com as regras de transformação mostradas na seção 3. A segunda transformação é a conversão do GA no Grafo de Fluxo de Dados (GFD), essa tarefa é realizada pelos algoritmos das Figuras 11 e 12. Como resultado final do processo a ferramenta produz como saída uma descrição textual do grafo no formato aceito pela ferramenta de visualização Graphviz[3].

[2]MagicDraw: http://www.magicdraw.com
[3]Graphviz: http://www.graphviz.org/

5.1 Estudo de Caso

Para ilustrar a semântica de ações da UML, bem como a captura do fluxo de dados, será utilizado como exemplo a especificação do método *readFile()* da classe `MyReadFile`[4]. A Figura 15 mostra o diagrama de classes desse exemplo, no qual o método *readFile()* é a única operação pública desta classe. Este método recebe como parâmetro um objeto do tipo `MyFile`, que representa um arquivo qualquer, e encapsula as chamadas aos demais métodos privados definidos na classe. A operação utiliza o método *openFile()* para abrir o aquivo, em seguida utiliza o método *sizeFile()* para verificar o tamanho do arquivo em bytes. Na sequência utiliza o método *loadFile()* para ler o conteúdo do arquivo, e por fim utiliza o método *closeFile()* para fechar o arquivo completando a operação. Apesar das chamadas interprocedimentais, serão considerados somente o fluxo de dados e fluxo de exceções gerados pela operação *readFile()*.

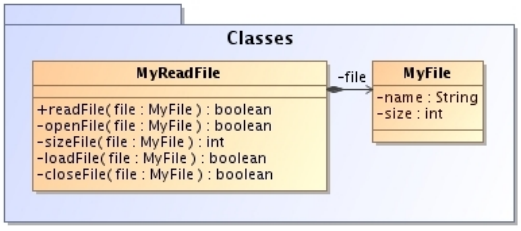


Figura 15: Diagrama de Classes do Exemplo `readFile()`

A modelagem do comportamento do método *readFile()* é feita por um conjunto de Diagramas de Atividades mostrados nas Figuras 16 e 17. Os diagramas utilizam atividades estruturadas para representar o fluxo de controle e tratamento da exceções. Para determinar as definições e usos dos dados, utilizamos a semântica das ações da UML 2.0 de acordo com a classificação mostrada na Tabela 2. A seguir apresentamos alguns exemplos da utilização dessas ações.

A atividade principal é representada pelo diagrama *readFile* (Figura 16a). Esta atividade tem um parâmetro de entrada, que representa o parâmetro formal *file* do método *readFile(file:Myfile*), este parâmetro é considerado como uma definição de *file*. A ação *"write myFile"* do tipo *AddStructuralFeatureValueAction* faz uso do parâmetro *file* e faz uma definição do atributo *myFile*. A referência do objeto utilizada por esta ação, recebida como parâmetro no pino de entrada *object*, é recuperada pela ação `this` do tipo *ReadSelfAction*. Esta ação recupera o objeto hospedeiro do método *readFile* que é do tipo `MyReadFile` e disponibiliza a referência deste objeto no pino de saída *result*.

Um exemplo da ação *CallBehaviorAction* é a ação *"test"* do nó condicional *if1* (Figura 16a), esta ação faz referência ao diagrama de atividades *if1Test* (Figura 16b). Uma ação deste tipo pode aceitar argumentos para a sua execução (pinos de entrada) e parâmetros de saída (pinos de saída). A ação *"ifBody"* (Figura 16a), por exemplo, possui um pino de saída chamado *return* que armazena o valor retornado pela atividade *if1Body* (Figura 17a). Uma ação do tipo *CallBehaviorAction* é um recurso de modelagem que permite abstrair

[4]Baseado no tutorial: http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html

um comportamento detalhado em outra atividade. No caso da ação *CallOperationAction*, tomamos como exemplo a ação ‘‘`openFile()`’’ (Figura 16b). Esta ação faz uma chamada ao método *openFile()*. Como o método não possui parâmetros de entrada, a ação tem somente um pino de entrada *target* que recebe a referência do objeto alvo e um pino de saída *result* que representa o valor booleano do retorno do método.

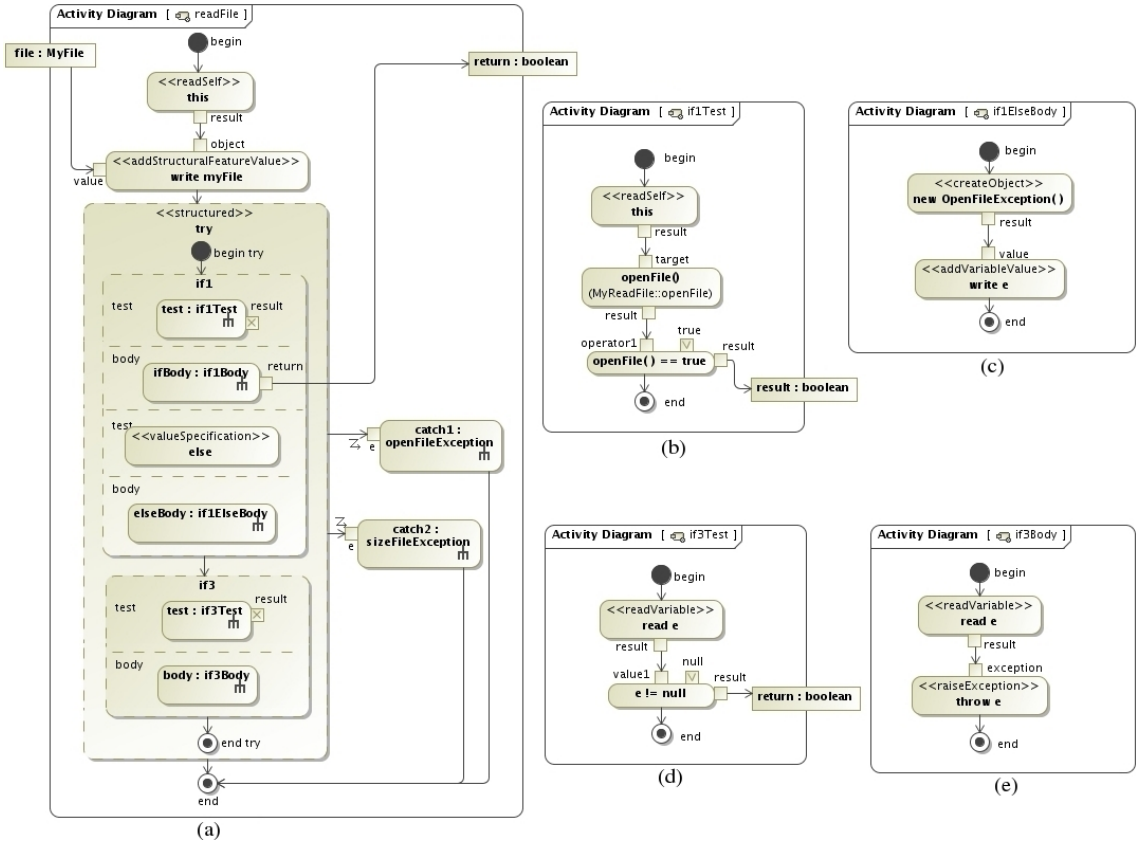


Figura 16: Especificação da operação `readFile()` - Parte 1

Podemos observar a utilização de uma ação opaca *(OpaqueAction)* na ação *“openFile()==true”* (Figura 16b). Esta ação compara o retorno do método *openFile()*, disponibilizado no pino *operator1*, com a constante *true*, disponibilizada pelo outro pino de entrada. O resultado da operação é um valor booleano colocado no pino de saída *result*.

A atividade estruturada *try* é utilizada para encapsular outras duas atividades estruturadas *if1* e *if3*, ambas do tipo *ConditionalNode*. Além disso, a atividade *try* possui duas arestas de exceções (*ExceptionHandler*). A primeira aresta captura exceções do tipo *OpenFileException* e a segunda do tipo *SizeFileException*.

A ação *“if1”* (Figura 16a) é um exemplo da atividade estruturada *ConditionalNode*. Esta atividade representa a modelagem de um nó condicional com duas cláusulas (if-else). Cada cláusula possui duas seções (*test* e *body*) que são modeladas por ações do tipo *CallBehaviorAction*.

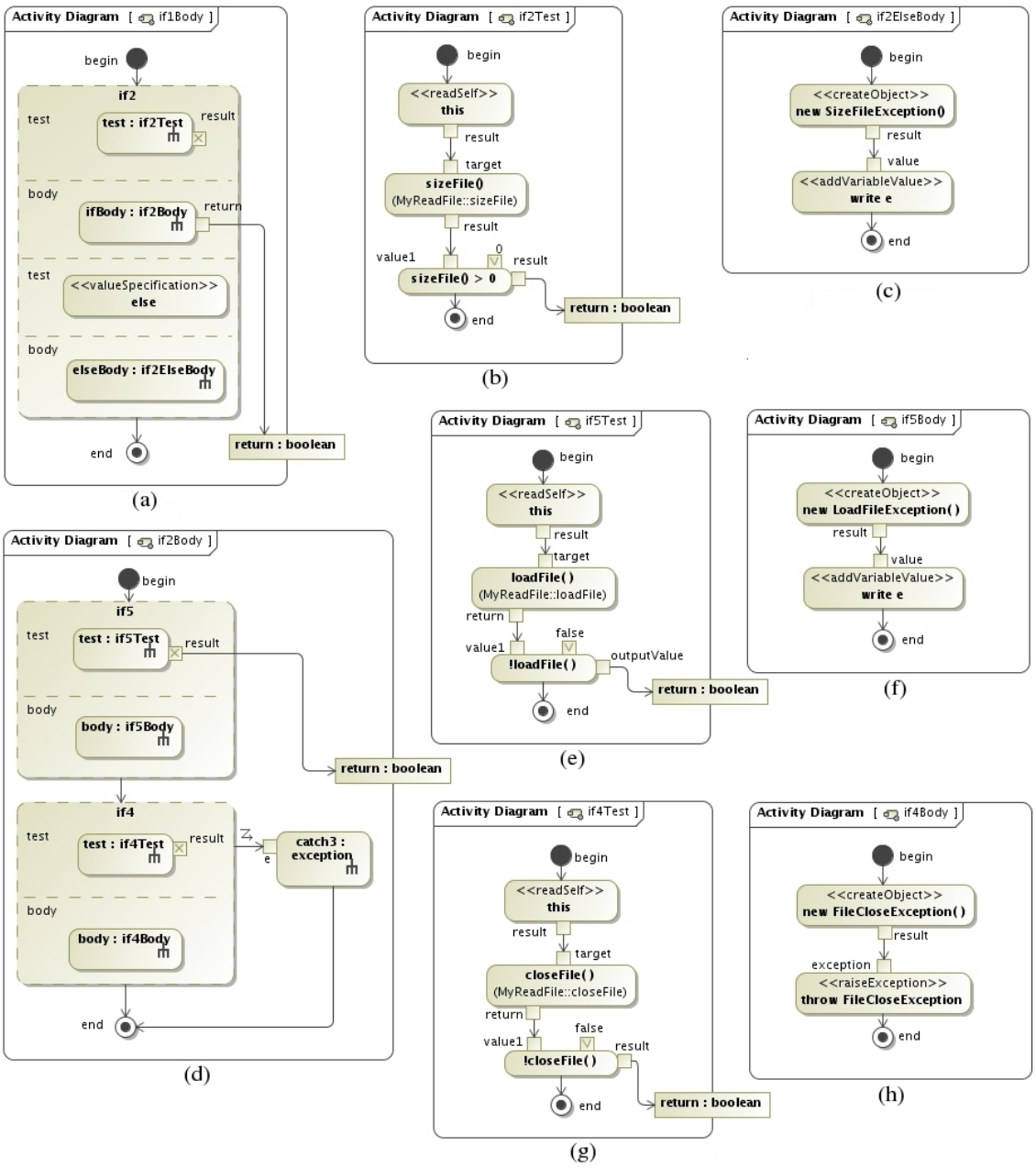


Figura 17: Especificação da operação `readFile()` - Parte 2

Um exemplo da ação *AddVariableValueAction* é a ação *"write e"*. Esta ação escreve um valor em uma determinada variável. Na Figura 16c a referência do objeto disponível no pino de entrada *value* é atribuída à variável *e*. Já uma ação do tipo *ReadVariableAction* lê o conteúdo de uma determinada variável. Na Figura 16d a ação *"read e"* lê a referência do objeto atribuído a variável *e* disponibilizando o seu valor no pino de saída *result*.

A ação *CreateObjectAction* é utilizada para criar um objeto, um exemplo é a ação *"new FileCloseException()"* (Figura 17h). Esta ação cria um objeto de exceção, este objeto é utilizado pela ação seguinte para lançar uma exceção. Por fim a ação *RaiseExceptionAction* é representada pela ação *"throw e"* (Figura 16e). Esta ação lança um dos possíveis tipos de exceção atribuídos a variável *e* em algum percurso do fluxo de controle do método *readFile()*.

Para gerar o Grafo de Atividades (GA) da atividade *readFile*, utilizamos o algoritmo da Figura 10. As atividades estruturadas são transformadas em nós e arestas conforme as regras de transformação das Figuras 9a e 9b. As ações *CallBehaviorAction* contidas nas seções destas atividades são substituídas pelos seus respectivos GAs. Este processo é executado recursivamente até não existir mais nenhum nó, que represente uma seção de atividade estruturada, no GA sem ter sido substituído. No final do processo teremos um único GA para representar o método *readFile()*. No nosso exemplo o GA resultante, mostrado na Figura 18, inclui todos os DAs apresentados nas Figuras 16 e 17. Nesta etapa do processo de transformação os nós que lançam as exceções e os nós que fazem o tratamento ainda não estão ligados. Isto ocorrerá após a geração do GFD e da análise de fluxo de dados.

A ação *"write e"* (*AddVariableValueAction*) da atividade *if1ElseBody* (Figura 16c) é uma ação de escrita que realiza uma definição da variável *e* com a referência do objeto criado pela ação *"new OpenFileException()"*. Neste caso a definição gerada foi a seguinte: $\langle d4 : e : Exception : OpenFileException \rangle$, onde: $d4$ é o identificador da definição, *e* o nome da variável definida, $Exception$ o tipo de dado da variável e $OpenFileException$ o tipo de dado do objeto referenciado.

A ação *"read e"* (*readVariableValueAction*) da atividade *if3Body* (Figura 16e) é uma ação de leitura que realiza um uso da variável *e* disponibilizando o seu conteúdo no pino de saída. Neste caso ocorreu um *c-uso* representado como: $\langle u2 : e : Exception \rangle$, onde: $u2$ é o identificador do uso, *e* é o nome da variável utilizada e $Exception$ é o tipo de dado da variável.

O próximo passo é a transformação do GA em um GFC; este processo é demonstrado pelos algoritmos das Figuras 11 e 12. Neste processo os nós do GA que estão em sequência, sem desvio de fluxo de controle, são agrupados e inseridos em um único nó do GFC. Durante o processo de geração do GFC, são capturadas as condições de guarda (Figura 19 nós 2, 3, 5, 9 e 15) são informações coletadas a partir dos nós predicativos, pois geralmente o teste condicional é realizado por uma ação do tipo *OpaqueAction*. Assim, as condições de guarda são extraídas da propriedade *body* deste tipo de ação, onde o modelador tem liberdade para incluir a cláusula que represente a condição de guarda.

Finalmente, para obtermos o GFD é necessário computar as *definições globais* e *c-usos globais*, definidos na seção 2.2. Esses conjuntos são obtidos a partir das definições e usos dos nós do GFC. A Figura 19 exibe o GFD resultante desse processo. Este grafo foi gerado por um protótipo da ferramenta apresentada. No GFD somente as *definições globais* e *c-usos globais* são considerados relevantes, e por isso as *definições locais* e *c-usos locais* não são

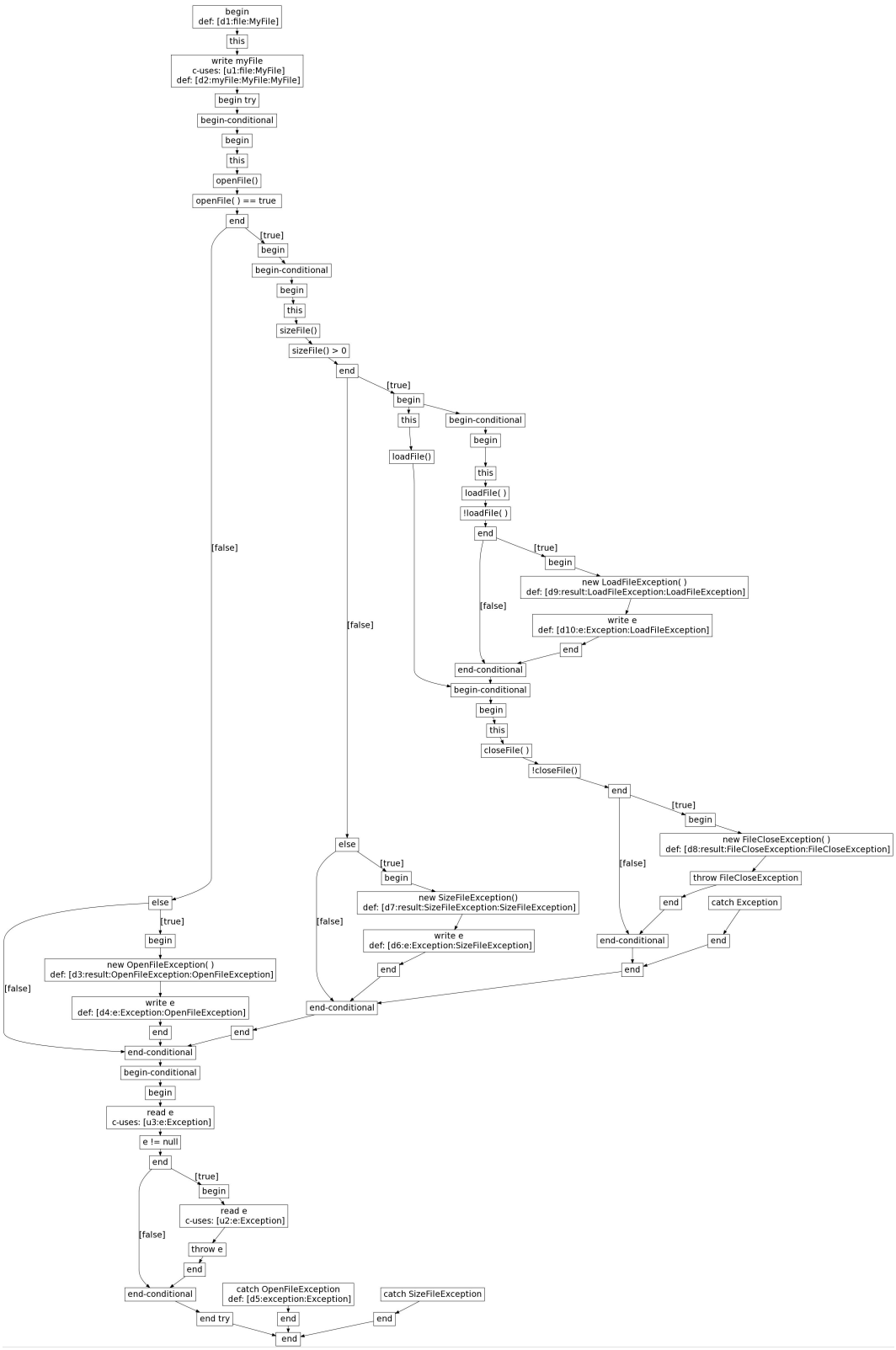


Figura 18: GA da operação `readFile()`

exibidos na Figura 19.

5.2 Exemplo de Análise de Fluxo de Dados

Para ilustrar o uso da análise de fluxo de dados sobre o GFD, apresentamos o resultado do cômputo das definições incidentes (reaching definitions) sobre o GFD gerado anteriormente.

Na Tabela 5 é apresentado um resumo dos conjuntos $def[n]$, $geradas[n]$ e $mortas[n]$ para cada nó do GFD da Figura 19. O resultado do cômputo das definições incidentes é apresentado na Tabela 4.

Tabela 3: *Resumo das Definições do GFD*

n	$def[n]$	$geradas[n]$	$mortas[n]$
1	$\langle d1 : file : MyFile : MyFile\rangle$, $\langle d2 : myFile : MyFile : MyFile\rangle$	$d1$, $d2$	$\varnothing$
5	$\langle d10 : e : Exception : LFE$ [a] $\rangle$, $\langle d9 : result : LFE : LFE\rangle$	$d10$, $d9$	$d4$, $d6$
7	$\langle d8 : result : FCE : FCE$ [b] $\rangle$	$d8$	$\varnothing$
12	$\varnothing$	$\varnothing$	$\varnothing$
13	$\langle d5 : exception : Exception : Exception\rangle$	$d5$	$\varnothing$
14	$\langle d3 : result : OFE : OFE$ [c] $\rangle$, $\langle d4 : e : Exception : OFE\rangle$	$d3$, $d4$	$d10$, $d6$
15	$\langle d6 : e : Exception : SFE$ [d] $\rangle$, $\langle d7 : result : SFE : SFE\rangle$	$d6$, $d7$	$d10$, $d4$

[a] *LoadFileException*
[b] *FileCloseException*
[c] *OpenFileException*
[d] *SizeFileException*

Tabela 4: *Resultado da Análise das Definições Incidentes*

n	$entrada[n]$	$saida[n]$
1	$\varnothing$	$d1, d2$
5	$d1, d2$	$d10, d1$
7	$d10, d1, d2, d9$	$d10, d1, d2, d8, d9$
12	$d10, d1, d2, d3, d4, d6, d7, d9$	$d10, d1, d2, d3, d4, d6, d7, d9$
13	$\varnothing$	$d5$
14	$d1, d2$	$d1, d2, d3, d4$
15	$d1, d2$	$d1, d2, d6, d7$

Para identificar quais são os possíveis tratadores de exceção utilizamos as informações de contexto do modelo. Um nó do GFD pode representar um conjunto de ações do DA que está contido em uma seção de uma atividade estruturada, a atividade *if1Test* (16b) é um exemplo. Este DA corresponde a seção *test* do nó condicional *if1*, que por sua vez está contida na atividade estruturada *try* (16a). Neste exemplo, o nó do GFD possui dois contextos, o primeiro sendo o nó condicional *if1* e o segundo a atividade estruturada *try*, portanto os possíveis tratadores de exceção são representados pelas ações *"catch1"* e *"catch2"*. Essas ações são ligadas ao contexto por arestas que representam um tratador de exceção (*ExceptionHandler*).

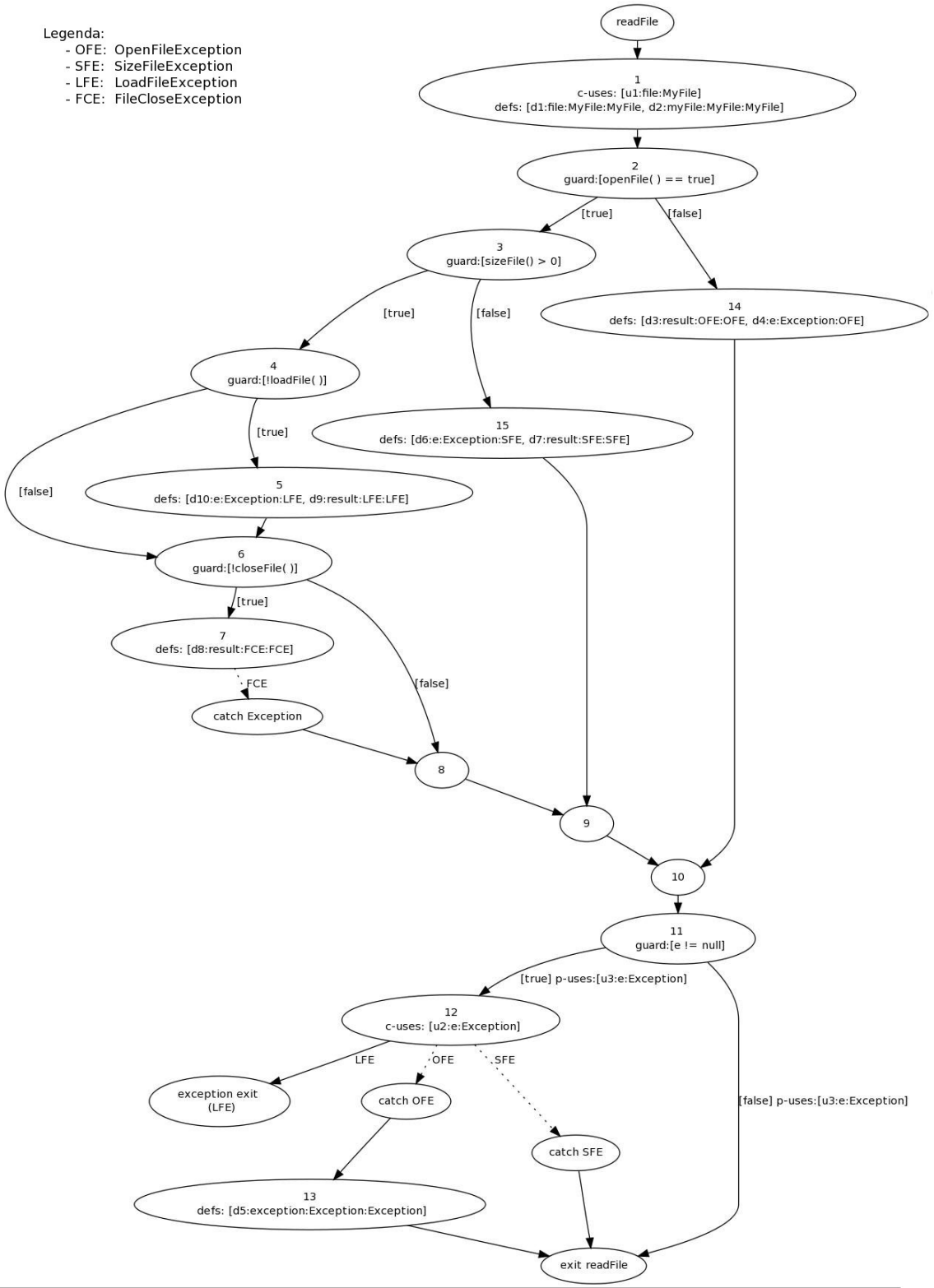


Figura 19: GFD da operação `readFile()`

A Figura 20 mostra a hierarquia das exceções definidas no modelo.

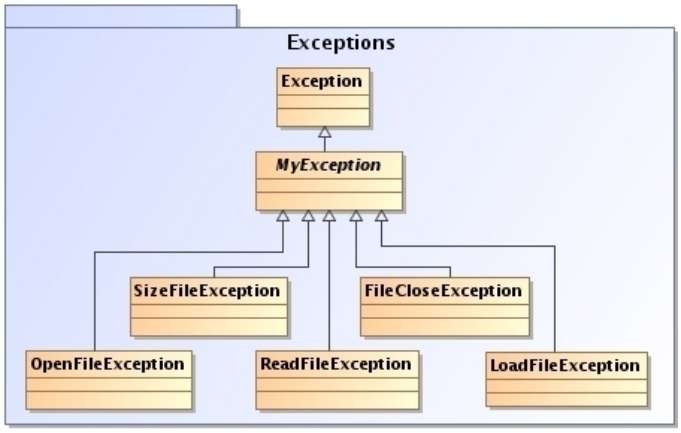


Figura 20: Hierarquia de Exceções do Exemplo `readFile`

O que determina se um nó do GFD lança uma exceção é a ocorrência de uma ação do tipo *RaiseExceptionAction*. No GFD da Figura 19 dois nós lançam algum tipo de exceção. O primeiro é o nó 7, que lança uma exceção do tipo `FileCloseException`. Podemos observar na Figura 17h que a ação *"throw FileCloseException"* faz uso de uma definição local de um objeto de exceção definido pela ação *"new FileCloseException()"* através do pino de saída *result*. Como a definição $\langle d8 : result : FileCloseException : FileCloseException\rangle$não está associada a uma variável, não é preciso considerar as definições incidentes para identificar o tipo de exceção que será lançado. Já o caso do nó 12 é diferente; podemos observar na Figura 16e que a ação *"throw e"* faz uso de uma definição global da variável e para lançar a exceção. Como e é do tipo `Exception`, pode armazenar uma referência de qualquer subclasse de `Exception`, conforme mostra a Figura 15a. Portanto, a variável e poderia conter uma referência de um dos cinco tipos de exceções que são subclasses de `Exception`. Neste caso a análise das definições incidentes é útil para determinar quais são as definições que alcançam o nó 12 e quais são os tipos de objetos referenciados por estas definições. A Tabela 4 exibe o resultado da análise de fluxo de dados; observe-se que o nó 12 possui o seguinte conjunto de definições incidentes: $saida[12] = \{d1, d2, d3, d4, d6, d7, d9, d10\}$. Porém, somente as definições da variável e interessam, são elas:

$$\langle d4 : e : Exception : OpenFileException\rangle$$
$$\langle d6 : e : Exception : SizeFileException\rangle$$
$$\langle d10 : e : Exception : LoadFileException\rangle$$

Estas definições determinam os tipos de exceções lançados pelo nó 12: `OpenFileException`, `SizeFileException` e `LoadFileException`.

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 incluída ligando o nó que lança a exceção com o nó que faz o tratamento da mesma. As arestas *(12, "catch*

OFE") e *(12, "catch SFE")* do GFD (Figura 19) são exemplos de aresta de exceção, que representam o fluxo excepcional da operação *readFile()*, que possui tratamento dentro da própria operação. Caso não exista um tratador de exceção equivalente, um nó de saída excepcional é criado. Este caso ocorre com a exceção do tipo `LoadFileException` que não possui um tratador na operação *readFile()*; o nó "exception exit (LFE)" é um exemplo de nó excepcional. Uma aresta de fluxo de controle liga o nó que lançou a exceção com o nó de saída excepcional, para representar que o fluxo de controle será passado para a operação que chamou *readFile()* para tentar tratar a exceção ocorrida. A Figura 16a mostra a atividade estruturada *try* e os tratadores de exceção do tipo `OpenFileException` e `SizeFileException`. As ações *"catch 1"* e *"catch 2"* incluem as atividades que fazem o tratamento das exceções. Os nós iniciais dessas atividades correspondem respectivamente aos nós *"catch OFE"* e *"catch SFE"* do GFD.

6 Trabalhos Relacionados

A obtenção de um GFC a partir do DA não é nenhuma novidade e já foi proposta por outros autores mesmo na versão 1.5 da UML[18]. A abordagem proposta neste trabalho estende a técnica proposta por Perez e Martins[19], 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 relatório, porém abordando somente o fluxo de controle. A abordagem proposta neste relatório estende o 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 [21].

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 [6, 7, 8, 17]. Técnicas de AFD podem ser aplicadas tanto em abordagens baseadas em modelos quanto em abordagens baseadas no código-fonte [25, 26, 21]. Uma abordagem que utiliza modelos foi proposta por Waheed et al. [26], que 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. A técnica apresentada 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[20], 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.

A representação de programas que lançam exceções explícitamente também é abordada por outros trabalhos [22, 28, 23, 24]. Sinha e Harold [22] apresentam uma abordagem para construção de um grafo de fluxo de controle interprocedimental que representa o fluxo de exceções de programas Java. Os autores apresentam técnicas para a representação de programas com ocorrência de exceções que são lançadas explícitamente 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.

Fu et al. [28] 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. [23] 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 [24] 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 idéia de representação reduzida do GFD poderia ser utilizada pela nossa abordagem, caso seja necessário reduzir o tamanho do grafo.

Tabela 5: *Quadro Comparativos dos Trabalhos*

Abordagem	**Fluxo de Controle**	**Fluxo de Dados**	**Fluxo de Exceções**	**Semântica das Ações UML**	**Modelo/Fonte**
Perez e Martins	Sim	Não	Sim	Não	Diagrama de Atividades
Waheed et al.	Sim	Sim	Não	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
Método proposto	Sim	Sim	Sim	Sim	Diagrama de Atividades

7 Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos o fluxo de dados obtidos através da Semântica das Ações da UML. A abordagem apresentada é aplicada para modelos comportamentais, especificamente para o Diagrama de Atividades da UML 2.0. Para representar o fluxo de dados do modelo o DA é transformado em um Grafo de Fluxo de Dados (GFD), modelo amplamente utilizado para diversas aplicações, como: testes baseados no fluxo de dados, program slicing, otimização de compiladores, entre outros. Neste trabalho o GFD foi estendido para demonstrar o fluxo excepcional intraprocedimental; arestas para representar o fluxo de exceções são inseridas no GFD. Estas arestas ligam o ponto onde ocorre o lançamento de uma exceção até o ponto do seu tratamento. No caso de exceções que não são tratadas dentro do procedimento, são inseridos nós de saída excepcional no GFD. Neste caso, as arestas de exceção são ligadas com os nós de saída excepcional. Para cada tipo de exceção não tratada que pode ser lançada, um nó de saída excepcional correspondente é criado. Desta maneira é possível validar o fluxo de exceções intraprocedimental do modelo, identificando quais são os tratadores de exceção que estão sendo utilizados e também quais são as exceções que não estão sendo tratadas. Em muitos casos são utilizados tratadores genéricos que acabam capturando qualquer tipo de exceção. Um tratador desse tipo pode capturar uma exceção que necessita de um tratamento diferencial e que foi modelado incorretamente.

O objetivo deste relatório é apresentar uma técnica de validação do modelo comportamental baseada no fluxo de dados, possibilitando que o modelo seja corrigido antes da sua implementação, seja por uma abordagem dirigida por modelos (MDD) ou por uma abordagem convencional. Para exemplificar apresentamos a validação do fluxo de exceções, em que as informações sobre o lançamento e sobre o tratamento das exceções são capturadas do modelo e uma AFD bem conhecida, as definições incidentes, é utilizada para inferir quais são os tipos de exceções que alcançam o nó onde uma exceção pode ser lançada. O resultado da AFD determina os tipos de exceções que podem ser lançadas por este nó.

Uma vantagem da técnica apresentada é a possibilidade de realizar outras AFD, como por exemplo, uma análise para computar as cadeias de definição-uso das variáveis e características estruturais (Ex: atributos de classe) contidas no modelo. Este tipo de informação pode ser utilizado para a derivação de casos de teste baseados no fluxo de dados, aproximando as técnicas de testes baseados em modelos das técnicas baseadas no código-fonte. Já uma limitação desta abordagem é o fato de tratar apenas do fluxo de dados intraprocedimental. Já está em andamento a extensão do trabalho, que vai estender essa técnica possibilitando o tratamento do fluxo de dados interprocedimental, possibilitando desta forma a obtenção do fluxo de exceções de um programa completo.

Referências

[1] Object Management Group, *UML 2.1.2 Superstructure Specification,* http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/ (2007).

[2] T. Stahl and M. Völter, *M*odel-Driven Software Development: Technology, Engineering, Management, John Wiley and Sons (2006).

[3] A. G. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model-Driven Architecture, Practice and Promise,* Addison-Wesley (2003).

[4] L. Fuentes and P. Sánchez, *Designing and Weaving Aspect-Oriented Executable UML models,* in Journal of Object Technology, vol. 6, pp. 109-136 (2007).

[5] 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, SERP 2005, Las Vegas (2005).

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

[7] P. Ammann and J. Offutt, *Introduction to Software Testing,* Cambridge University Press (2008).

[8] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers-Principles, Techniques, and Tools,* Addison Wesley (1986).

[9] C. Bock, *UML 2 Activity and Action Models,* in Journal of Object Technology, vol. 2, no. 4, pp. 4353 (2003).

[10] C. Bock, *UML 2 Activity and Action Models Part 2: Actions,* in Journal of Object Technology, vol. 2, no. 5, pp. 4156 (2003).

[11] C. Bock, *UML 2 Activity and Action Models Part 6: Structured Activities,* in Journal of Object Technology, vol. 4, pp. 43-66 (2005).

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

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

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

[15] Object Management Group, *XML Metadata Interchange (XMI),* http://schema.omg.org/spec/XMI/2.1/PDF/ (2005).

[16] The Eclipse Foundation, *Eclipse Modeling Framework Project (EMF),* http://www.eclipse.org/modeling/emf/ (2009).

[17] S. Rapps and E. J. Weyuker, *Selecting Software Test Data Using Data Flow Information,* IEEE - Transactions on Software Engineering, SE-11, No. 4. (1985).

[18] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools,* Addison-Wesley Longman Publishing Co., pp. 293. (1999).

[19] 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 (2007).

[20] H. Störrle, *Semantics and Verification of Data Flow in UML 2.0 Activities,* Eletronic Notes in Theorical Computer Science, pp. 35-52. (2005).

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

[22] S. Sinha and M. J. Harrold, *Analysis and testing of programs with exception-handling constructs,* IEEE Transactions on Software Engineering, vol. 26, pp. 849-871, (2000).

[23] B. mo Chang, J. wu Jo, K. Yi and K. moo Choe, *Interprocedural Exception Analysis for Java,* (2001).

[24] 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).

[25] 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).

[26] T. Waheed, M. Z. Z. Iqbal and Z. I. Malik, *Data Flow Analysis of UML Action Semantics for Executable Models,* Springer-Verlag Berlin Heidelberg, vol. 5095, pp. 79-93, (2008).

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

[28] Chen Fu, B. G. Ryder, A. Milanova and D. Wonnacott, *Testing of Java Web Services for Robustness,* In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 23-24, (2004).