UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DEPAnalyzer: um Modelo de Análise Estática de Dependências para Programas Orientados a Objetos

por

SILVANA CAMPOS DE AZEVEDO

Dissertação submetida à avaliação, como
requisito parcial para a obtenção do
grau de Mestre em Ciência da Computação

Prof. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, janeiro de 2002

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Azevedo, Silvana Campos de

DEPAnalyzer: um Modelo de Análise Estática de Dependências para Programas Orientados a Objetos / por Silvana Campos de Azevedo. – Porto Alegre: PPGC da UFRGS, 2002.

72p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Geyer, Cláudio Fernando Resin.

1. Análise Estática. 2. Análise de Dependências. 3. Programas Orientados a Objetos. I. Geyer, Cláudio Fernando Resin. II. Título.

Agradecimentos

Ao Conrado, por seu incentivo, carinho e pela sua presença em minha vida.

À minha família, pelo apoio e por estarem comigo nesta caminhada.

Ao meu orientador, prof. Cláudio Geyer, pelos seus conhecimentos, sua compreensão e sua amizade.

A Patrícia, pois sua colaboração e sua amizade foram essências na realização deste trabalho.

Ao Adenauer, por sua paciência, sua psicologia e seu constante incentivo.

A Débora por estar sempre presente, com sua compreensão e sua amizade.

Ao Jorge, por seus ensinamentos e por seu incentivo.

A todos os meus amigos, em especial ao Edvar, Flávia e Simone por dividirem comigo as angustias e as vitórias desta luta.

Sumário

Lista de Abreviaturas

API	Application Programming Interface
CE	Conditionally Extant
CFA	Control Flow Analysis
CHA	Class Hierarchy Analysis
CPA	Cartesian Product Algorithm
CPU	Control Processing Unit
EXEHDA	EXecution Environment for High Distributed Applications
GPPD	Grupo de Processamento Paralelo e Distribuído
ID	Identifier
ISAM	Infra-estrutura de Suporte às Aplicações Móveis
JavaCC	Java Compiler Compiler
JVMPI	Java Virtual Machine Profile Interface
PAG	Program Analyzer Generator
PIFG	Partial Interprocedural Flow Graph
PPGC	Programa de Pós-Graduação em Computação
RTA	Rapid Type Analysis
RMI	Remote Method Invocation
UCE	Unconditionally Extant
UCNE	Unconditionally Non-Extant
UFRGS	Universidade Federal do Rio Grande do Sul

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta um modelo de análise estática de programas orientados a objetos, o qual se denomina DEPAnalyzer (*DEPendencies Analyzer*). O modelo realiza a análise das dependências entre as classes de um programa, ou seja, os relacionamentos estabelecidos entre estas. As classes representam as entidades estáticas, as quais em tempo de execução darão origem a conjuntos de objetos.

Através da simulação do programa o modelo consegue obter as informações sobre quem se relaciona com quem e qual é a intensidade destes relacionamentos. Estas informações visam auxiliar no processo de escalonamento de um programa em uma arquitetura distribuída. Para expressar estes relacionamentos podem ser gerados dois grafos, o grafo de dependências e o grafo de invocações. A geração de um ou de ambos depende do propósito de utilização, ou seja, alocação inicial ou redistribuição dos objetos.

O grafo de dependências apresenta uma versão resumida dos relacionamentos. Este adequa-se a auxiliar no processo de distribuição inicial por propiciar um panorama geral dos relacionamentos sem considerar a ordenação de ocorrência das ações. Por sua vez o grafo de invocações tem como propósito a discriminação da ordem de ocorrência das ações de uma aplicação. Viabilizando a utilização deste por parte do processo de redistribuição dos objetos.

Palavras-Chave: Análise Estática, Análise de Dependências e Programas Orientados a Objetos

TITLE: "DEPANALYZER: A MODEL OF STATIC ANALYSIS OF DEPENDENCIES TO OBJECT ORIENTED PROGRAMS."

Abstract

This work presents a model of static analysis of object oriented programs named DEPAnalyzer (which stands for DEPendencies Analyzer). Such model carries through the analysis of inter-class dependencies, i.e. it aims to extract the relationships between these. Classes are static entities which will originate sets of objects at execution time.

Through simulation of the program, the model not only extracts information about what class depends on each other, but also estimates how strong are these relationships. This information may be used later to assist program scheduling on distributed architectures. Two graphs, one of dependencies and other of invocations, may be generated to express the inter-class relationships. The generation of one of these graphs or both then depends on its intended usage: initial allocation or redistribution of objects.

The dependencies graph presents a summarized view of the relationships. This work can help in the process of entities initial distribution, for offering a general vision of the relationships, without considering the ordination of the actions occurrence. In turn, the invocations graph is specifically targeted to discriminate the order in which actions take place when the application is running. Such approach makes it very helpful in object re-placement and migration decisions.

Keywords: Static Analysis, Dependencies Analysis and Object Oriented Programs

1 Introdução

1.1 Tema do Trabalho

O tema deste trabalho é a inferência estática de informações de dependências de programas orientados a objetos. Dentro deste tema abordam-se estudos sobre programação orientada a objetos, análise estática e a utilização da análise estática neste paradigma. Como resultado deste estudo é proposto um modelo de análise estática de programas Java, com o objetivo de inferir informações sobre os relacionamentos entre as entidades do programa. Estas informações visam auxiliar o escalonamento destes programas em um ambiente distribuído.

1.2 Motivação

O paradigma orientado a objetos é uma ferramenta adequada à estruturação de problemas do mundo real [AMB 98]. Esta metodologia facilita a modelagem de problemas complexos. A união da facilidade de modelagem e de características como reusabilidade e herança, torna a utilização deste paradigma atrativa. Uma das linguagens provenientes deste paradigma é a linguagem Java [SUN 2001].

Duas razões parecem ter contribuído para que Java se tornasse tão popular mesmo não sendo substancialmente original. A primeira é o fato de Java apresentar uma sintaxe parecida com a sintaxe da linguagem C, a qual é bastante conhecida. A segunda, e possivelmente a principal razão, é a disponibilidade de recursos previstos para suporte à programação distribuída. Além disso, sua elevada portabilidade de código se mostra muito conveniente para aplicações que serão executadas em redes de computadores inerentemente heterogêneas.

O preço pago pela portabilidade da linguagem Java, decorrente do uso da interpretação em uma máquina virtual, recai no desempenho da execução dos seus programas. Diversos esforços têm sido feitos procurando aumentar o desempenho da execução de programas Java, a exemplo, a proposta de uso da compilação *Just-in-Time* [SUN 2001].

Deste modo, apesar da portabilidade degradar o desempenho, ela é um fator imprescindível para a execução de aplicações em uma arquitetura distribuída. Arquiteturas distribuídas estão se popularizando no meio comercial sendo normalmente heterogêneas, por isso o programa deve ter a capacidade de rodar a mesma aplicação em ambientes distintos. Tais considerações também são relevantes devido ao crescimento da adoção de arquiteturas de memória distribuída como plataforma para processamento de alto desempenho como ilustrado pelas estatísticas do TOP500 (indicativo dos quinhentos computadores com maior desempenho) [TOP 2001].

Outro aspecto é que com o emprego de arquiteturas de memória distribuída, novas questões em nível de processamento estão surgindo. Uma destas questões é a localidade das entidades de um programa na arquitetura. A localidade caracteriza quais

nós contêm quais componentes. Este é um dos fatores determinantes do desempenho da aplicação, uma vez que componentes que interagem com freqüência devem ser prioritariamente alocados no mesmo nó (local).

Os nós de um sistema distribuído são conectados por uma rede de intercomunicação, o que torna mais lenta a comunicação inter-nós do que a comunicação intra-nó. As comunicações inter-nós referem-se às comunicações entre entidades alocadas em nós distintos e as comunicações intra-nó referem-se as comunicações entre entidades alocadas no mesmo nó. Neste contexto fica evidente que quanto menos comunicações nó-nó ocorrerem, o desempenho da aplicação como um todo atingirá um melhor resultado.

Particularmente para um ambiente de programação distribuída, baseado em uma linguagem orientada a objetos como Java, objetos que compartilham dados, ou que apresentam um perfil de intensa troca de mensagens precisam ficar o mais "próximo possível". Isto significa que devem ser alocados no mesmo nó processador, ou conectados através de um canal de comunicação rápido. Esta localidade pode ajudar a melhorar substancialmente o desempenho das aplicações.

Para identificar os componentes de um programa e os seus relacionamentos é necessário a sua análise. Esta análise pode ser feita estaticamente, durante a compilação, ou dinamicamente, em tempo de execução. A análise estática obtém informações sobre o comportamento do programa a partir do estudo das ações descritas textualmente no código fonte, as quais representam a semântica ou o significado dos comandos da aplicação. A análise dinâmica obtém informações sobre o comportamento do programa a partir da execução deste.

Através da análise estática pode-se realizar a simulação da execução de um programa [AZE 99], [AZE 98], [COR 97], [DAM 97]. Esta técnica prove informações aproximadas, pois estas são aplicáveis a todas as possíveis execuções de um programa, sendo portanto abrangentes. No entanto, mesmo com a possível amplitude dos resultados desta análise, estas informações podem auxiliar na distribuição dos programas.

Em contra partida a análise dinâmica apresenta resultados precisos, pois estes se encontram vinculados a uma execução em particular do programa. O fato das informações geradas dinamicamente estarem vinculadas a uma determinada execução torna este tipo de análise necessário toda vez que em uma execução o comportamento do programa for alterado.

Outro fator importante para a distribuição dos componentes é a previsão das comunicações que ocorrem ao longo do processamento. Isto pode contribuir não somente para uma alocação inicial das entidades mas para uma alocação que minimize as comunicações remotas ocorridas durante toda a execução. Neste caso, pode ser incluída a previsão da necessidade de replicações e/ou mobilidade dos componentes.

Tendo em vista o auxílio ao escalonamento este trabalho apresenta um modelo de análise estática das dependências (comunicações) entre as entidades de um programa orientado a objetos, o qual denomina-se DEPAnalyzer (*DEPendencies Analyzer*). As dependências são os relacionamentos entre suas entidades de um programa, ou seja, os relacionamentos entre os conjuntos de objetos do mesmo. Este relacionamento é estabelecido através das invocações de variáveis e de métodos entre as classes.

Juntamente com o DEPAnalyzer mais dois trabalhos ([ARA 2001], [SIL 2001]) desenvolvidos no grupo OPERA do GPPD (Grupo de Processamento Paralelo e Distribuído) do Instituto de Informática da Universidade Federal do Rio Grande do Sul (II-UFRGS), oferecem uma contribuição para o mapeamento de objetos distribuídos em Java.

Em [ARA 2001] é descrito um modelo de monitoração no nível da aplicação, o qual utiliza a JVMPI (*Java Virtual Machine Profiler Interface*) afim de poder realizar uma seleção dinâmica dos eventos de interesse da aplicação (por exemplo, ativação, interrupção e tempo de espera de métodos) que devem ser monitorados. Além de informações para consumo dinâmico durante a execução, este trabalho compõem um arquivo de rastro da execução que pode ser utilizado por ferramentas de visualização *post-morten*.

Em [SIL 2001] é descrito um modelo de monitoração no nível de sistema, o qual oferece dois tipos principais de métricas: (*i*) uma para caracterização do *workload* dos *hosts* e (*ii*) outra para construção de perfis de comunicação entre objetos Java. Os mecanismos para capturar os perfis de comunicação entre os objetos são integrados com a API RMI de Java, preservando a compatibilidade com a semântica nativa da linguagem. Este trabalho também contempla a especificação de uma primitiva para migração de objetos.

A contribuição oferecida por estes modelos objetiva auxiliar um outro grupo de projetos, também desenvolvidos no contexto do grupo OPERA, os quais são: o Holoparadigma [BAR 2001], o EXEHDA [YAM 2001] e ISAM [AUG 2001]. Estes projetos têm seu ambiente de execução baseado para Java. Desta forma, prover informações que auxiliem no entendimento do comportamento da aplicação, pode contribuir diretamente para o mapeamento de entidades no ambiente de execução destes projetos.

Embora não seja o enfoque deste trabalho, as informações sobre os relacionamentos podem auxiliar no processo de desenvolvimento de software, pois evidenciam o grau de acoplamento entre as entidades da aplicação. A teoria de desenvolvimento de software orientado a objetos sugere que a modelagem das aplicações leve a um baixo acoplamento entre as entidades que compõem o sistema [AMB 98]. Isto proporciona uma maior reusabilidade dos componentes da aplicação, pois podem ser incorporados a outros sistemas com nenhum ou pouco esforço de modelagem. Desta forma, detectar o grau de acoplamento pode auxiliar na evidência de um mal planejamento de modelagem.

1.3 Objetivos do Trabalho

O objetivo geral deste trabalho é proporcionar um sistema de análise de dependências que auxilie na distribuição de programas orientados a objetos. Como objetivos específicos, pode-se salientar:

- realizar um estudo sobre características das linguagens orientadas a objetos;
- realizar um estudo sobre os conceitos de análise estática;
- realizar um estudo sobre analisadores estáticos de programas orientados a objetos;
- aplicar este modelo a uma linguagem específica;
- realizar um estudo sobre analisadores estáticos aplicados à linguagem na qual o modelo será aplicado;
- modelar um sistema de análise estática que vise o auxílio ao escalonamento das entidades de uma aplicação analisada em uma arquitetura distribuída;
- classificar as informações de dependências visando a necessidade do processo de escalonamento;
- modelar e implementar um protótipo para a validação da proposta.

1.4 Contribuição do Autor

A realização deste trabalho desencadeou o alcance de várias contribuições, dentre as quais destacam-se:

- A disponibilização de um resumo das principais técnicas de análise estática de programas orientados a objetos. Este resumo tem o objetivo de proporcionar uma visão da problemática do processo de análise estática e mais precisamente da problemática do processo de análise estática no paradigma orientado a objetos, o qual provê características dinâmicas às suas linguagens;
- Uma comparação entre trabalhos de análise estática aplicados à linguagem Java, a qual é o alvo de aplicação para o modelo desenvolvido neste trabalho;
- A concepção de um modelo de análise estática aplicável ao paradigma orientado a objeto. Além disso, como contribuição adicional tem-se o emprego deste modelo à linguagem Java. Para tanto considerou-se alguns aspectos específicos de Java. No entanto o modelo pode ser aplicado a qualquer linguagem deste paradigma, através de pequenas modificações, requeridas pelo fato de estar se empregando um modelo geral para uma linguagem específica;
- Outra contribuição deste trabalho é o direcionamento da análise para prover informações que auxiliem no processo de escalonamento. Este objetivo foi alcançado através da modelagem de um domínio abstrato, o qual retrata o escopo em que determinada dependência ocorre. Esta informação pode auxiliar no processo de escalonamento uma vez que o grafo resultante torna explícitas as dependências interativas e condicionais, provendo uma maior liberdade ao ambiente de escalonamento. A liberdade é decorrente do conhecimento das circunstâncias em que determinada dependência ocorreu estaticamente;
- Descartar a necessidade de se ter o conhecimento de toda a hierarquia das classes envolvidas na computação da aplicação analisada. Esta característica é decorrente do modelo considerar que em um ambiente distribuído o ambiente Java se fará presente em todos os nós da arquitetura, fato que torna desnecessária a análise de relacionamento entre classes primitivas e classes definidas pelo programador;
- A obtenção de um grafo direcionado à apresentação de uma visão simplificada, propícia para o emprego em uma alocação inicial dos componentes no ambiente distribuído;
- Obtenção de informações que podem auxiliar no processo de desenvolvimento de software, através da detecção do grau de acoplamento das entidades em decorrência de determinação dos relacionamentos das mesmas.

1.5 Estrutura do Texto

A dissertação está organizada em cinco capítulos. O segundo capítulo realiza uma revisão bibliográfica sobre modelos de análise estática em linguagens orientadas a objetos. O terceiro capítulo apresenta a estrutura do modelo DEPAnalyzer enfatizando os detalhes de cada módulo deste. Além disso, este capítulo descreve o relacionamento entre o DEPAnalyzer e os modelos que compõem o contexto atual de trabalho do grupo onde este trabalho se encontra. O quarto capítulo apresenta o protótipo desenvolvido. Finalmente o quinto capítulo aborda as conclusão obtidas a partir da realização deste trabalho bem como indica trabalhos futuros.

2 Análise Estática de Programas Orientados a Objetos

Este capítulo descreve o emprego da análise estática em linguagens orientadas a objetos. O capítulo encontra-se organizado da seguinte forma: a seção 2.1 aborda aspectos gerais do uso da análise estática em linguagens orientadas a objetos. A seção 2.2 apresentada características da análise estática de tipos em programas tipados dinamicamente. A seção 2.3 aborda aspectos relevantes da análise estática de tipos em linguagens tipadas estaticamente. Já a seção 2.4 mostra trabalhos de análise estática aplicados a linguagem Java. Além disso, ao final da seção 2.4 é apresentada uma comparação entre os trabalhos abordados nesta seção. A seção 2.5 descreve as considerações finais do capítulo.

2.1 Aspectos Gerais

O código fonte de um programa é uma representação textual das ações realizadas durante a execução do mesmo. Então em nível de código fonte pode-se detectar certas propriedades, que determinadas ações ou elementos assumem. A análise estática é a área de pesquisa dedicada a este tipo de tarefa, ou seja, determinar em tempo de compilação comportamentos que o programa desempenhará em tempo de execução.

Neste sentido a análise estática tem sido amplamente empregada em linguagens imperativas [COH 96], [PER 98] e em linguagens declarativas [CAS 97], [COR 97], [DAM 97], [LAB 96]. O intuito de realizar este tipo de análise é a otimização da execução dos programas. Esta otimização deve-se à detecção prévia (antes da execução) de fatos que ocorrerão durante a computação.

Um dos conflitos, nesta área, é que as informações geradas pela análise estática são geralmente aproximadas, o que leva a assumirem um caráter impreciso. Deve-se ressaltar que a imprecisão não esta vinculada com a geração de resultados errôneos, mas sim com a geração de resultados conservativos (abrangentes).

O estudo estático de programas orientados a objetos tem sido intensamente pesquisado. Isto deve-se a várias características deste paradigma como herança, polimorfismo e ligação tardia. Estas características conferem poder aos programas mas reduzem a eficiência dos mesmos [JEN 2000]. Deste modo um dos enfoques dado aos analisadores estáticos de programas orientados a objetos é a previsão do destino das invocações de métodos. Uma divisão neste contexto refere-se a métodos monomórficos e métodos polimórficos. Quando a invocação recai sobre um método monomórfico a análise estática consegue identificar a versão que está sendo chamada e consequentemente saber o caminho do fluxo de controle após a invocação. Isto deve-se ao fato de métodos monomórficos possuírem, como o próprio nome sugere, somente uma versão (implementação). Já métodos polimórficos, métodos que possuem mais de uma versão, causam uma indefinição sobre o caminho do fluxo de controle, em tempo de compilação, impedindo, muitas vezes, a análise estática de prover informações precisas neste sentido.

O polimorfismo pode ser classificado como: sobrecarga e reescrita. A sobrecarga refere-se à situação onde existem dois métodos, da mesma hierarquia, com o mesmo identificador mas com assinaturas diferentes. A reescrita refere-se à situação onde existem dois métodos, da mesma hierarquia, com o mesmo identificador e com a mesma assinatura. Por exemplo, a reescrita de métodos permite que métodos de uma classe possam ser redefinidos em suas subclasses [BAI 97]. Com isto, somente em tempo de execução, através da verificação dinâmica de tipos, muitas invocações são resolvidas, ou seja, sabe-se qual versão do método que está sendo invocada. A verificação dinâmica de tipos degrada o desempenho das aplicações. A baixa no desempenho torna necessário o surgimento de alternativas que possam contornar esta situação, isto é, aumentar o desempenho de tais programas. Então para se detectar qual método (versão) está sendo invocado em determinado ponto, da aplicação analisada, deve-se obter informações sobre o tipo (classe) do objeto invocador e dos argumentos recebidos pelo método. Como pode-se perceber para solucionar, estaticamente, uma determinada invocação, a análise dos tipos dos elementos (variáveis) de um programa deve ser realizada.

2.2 Análise Estática em Programas Tipados Dinamicamente

Em [AGE 95] e [AGE 94] pode-se encontrar uma ótima contribuição no tocante à inferência de tipos em programas orientados a objetos. Agesen elucida a complexidade de se inferir os tipos das variáveis de um programa com a presença do polimorfismo paramétrico. Este tipo de polimorfismo é caracterizado pela capacidade de existência de dois ou mais métodos com o mesmo nome (identificador) mas com assinaturas diferentes.

Outro tipo de polimorfismo, caracterizado por Agesen, é o polimorfismo de dados. Este tipo representa a habilidade de dois ou mais métodos terem a mesma assinatura mas possuírem implementações diferentes.

Como pode-se perceber a classificação de Agesen para os tipos de polimorfismo é análoga à classificação citada anteriormente, ou seja, o polimorfismo paramétrico corresponde ao polimorfismo de sobrecarga e o polimorfismo de dados corresponde ao polimorfismo de reescrita.

O foco do trabalho de Agesen é a análise do polimorfismo paramétrico, pois as linguagens analisadas são linguagens não tipadas. Neste tipo de linguagem pode-se verificar a complexidade de análise do polimorfismo paramétrico. No entanto, a complexidade de análise também mostra-se presente em linguagens tipadas, como Java, pela presença do polimorfismo de dados.

Em [AGE 95] são abordados três algoritmos para a inferência de tipos. O primeiro é o algoritmo básico, descrito por Palsberg. Este algoritmo essencialmente simula a execução de um programa. Para realizar está tarefa o algoritmo tem como primeiro passo a expansão das classes, ou seja, o corpo de uma subclasse é a soma dos métodos herdados e dos métodos declarados por esta.

A expansão das classes parece contrária a teoria do paradigma orientado a objetos, no qual a herança tem como principal vantagem proporcionar um maior grau de reutilização do código. No entanto, Palsberg se baseia na seguinte observação: as classes herdam implementações e não especificações [PAL 91].

Após esta expansão é gerado um grafo de traço do programa. Para isso primeiramente é associada uma variável de tipo a cada *slot* ou expressão do programa

analisado. Uma variável de tipo contém ao longo do processamento da análise os possíveis tipos ao qual a expressão ou *slot* que esta representa pode assumir.

Uma vez que o primeiro passo somente vincula uma variável de tipo a cada *slot* e expressão do programa, o segundo passo tem a função de capturar o estado inicial dos tipos das variáveis do programa analisado.

Após isto, conecta-se as variáveis de tipos sobre uma rede por adição de setas dirigidas entre algumas delas. Na conclusão deste passo, as variáveis de tipos podem constituir os nodos no grafo dirigido. As setas podem ser as restrições. Por exemplo, se o programa executa uma atribuição ($a = b$) o fluxo da variável que está sofrendo a atribuição passa a referenciar os valores da variável que lhe foi atribuída. Quando o algoritmo encontra tais fluxos de dados, isto pode adicionar uma seta da variável de tipo de *b* para a variável de tipo de *a*, estabilizando o fato que $tipo(b) \subseteq tipo(a)$ [AGE 95].

O grafo é composto de subgrafos, os quais são criados um para cada método. Estes subgrafos são constituídos de:

- nodos (variáveis de tipo) correspondentes para expressões, variáveis locais e argumentos formais do método;
- setas (restrições) originadas destes nodos.

Como dito anteriormente, cada método cria um subgrafo no grafo. No entanto, pode ocorrer do programa possuir duas invocações do mesmo método com parâmetros de tipos diferentes. Neste caso o algoritmo básico realiza a união dos tipos dos parâmetros. Esta união causa uma imprecisão nas informações resultantes.

Para melhorar a precisão é necessário separar totalmente os subgrafos criados pelas invocações de métodos. Isto é realizado por um algoritmo de melhoramento do algoritmo básico [OXH 92]. No algoritmo de melhoramento são incluídas várias características não tratadas pelo algoritmo básico como: análise de estruturas complexas tipo listas, redução da complexidade exponencial do algoritmo básico para uma complexidade polinomial no algoritmo de melhoramento.

Para realizar a análise de estruturas complexas como listas o algoritmo de melhoramento se utiliza da técnica de duplicação das classes (ou código). Para isto o algoritmo realiza a duplicação de determinada classe *A* diante do comando *A new*, criando uma instância desta classe [OXH 92].

Para contornar o problema de imprecisão este algoritmo (algoritmo de melhoramento) cria subgrafos separados para métodos polimórficos. Esta atitude auxilia na obtenção de informações mais precisas. No entanto, o algoritmo realiza isto para todo método invocado, mesmo que este já tenha sido analisado, causando assim um problema de redundância [AGE 95].

A solução ideal para o problema de redundância e para o problema de precisão requer a criação de um subgrafo para cada caso diferente de invocação de método. Segundo Agesen esta solução necessita do conhecimento prévio dos tipos envolvidos na computação do programa.

O algoritmo iterativo, descrito em [PLE 94], soluciona ambas as questões de precisão e redundância. Na primeira iteração do algoritmo iterativo é aplicado o algoritmo básico, para se ter uma visão dos tipos envolvidos na computação do programa. As iterações seguintes usam este conhecimento (visão) para aplicarem o algoritmo ideal, ou seja, para realizar uma análise incremental nos pontos de imprecisão das informações de tipos. Então, duas invocações de método podem compartilhar o mesmo subgrafo, se os tipos dos parâmetros forem iguais. No entanto, o algoritmo iterativo tem um custo computacional, pelo fato de realizar iterações, o que causa um

overhead, além de tratar com questões complicadas, como quando parar as iterações [AGE 95].

O algoritmo do produto cartesiano (CPA) descrito em [AGE 94] e [AGE 95] não particiona as invocações e sim torna cada caso de invocação em um caso de análise. Dado uma invocação a ser analisada, o CPA computa o produto cartesiano dos tipos dos parâmetros atuais. Cada tupla no produto cartesiano é analisada como um caso independente. Isto torna a informação de tipo exata para cada caso diminuindo a necessidade de iteração. O algoritmo pressupõe que as classes/tipos envolvidas na computação do programa são conhecidas.

O algoritmo do produto cartesiano consegue inferir tipos precisos, no entanto, a sua aplicação em programas com alto nível de polimorfismo eleva a sua complexidade, tornando a sua utilização, nestes casos, pouco atraente.

Como pode-se perceber a análise de tipos em programas orientados a objetos é uma tarefa complexa, pois trata com questões como: conhecimento prévio da hierarquia das classes envolvidas no programa, ligação dinâmica, etc. Por este motivo [STE 99] cita que a analise estática de programas polimórficos tem uma complexidade elevada.

2.3 Análise Estática de Programas Tipados Estaticamente

2.3.1 Análise da Hierarquia das Classes

A análise da hierarquia das classes (*Class Hierarchy Analysis*) [DEA 94], ou simplesmente CHA, visa a detecção dos tipos das expressões de invocação dos métodos. Estas informações podem auxiliar o processo de ligação dinâmica dos métodos.

O funcionamento desta técnica baseia-se em determinar as classes envolvidas na computação de determinado programa e assim prover o conhecimento sobre a hierarquia destas classe. Para tal são recolhidas do código fonte as informações sobre as classes que compõem a aplicação, suas descendências e os métodos que cada classe implementa. A análise despreza a implementação (corpo) dos métodos inferindo somente as informações sobre as suas assinaturas.

Como pode-se perceber através da análise das classes envolvidas na compilação de um programa, a CHA prove uma visão de toda a hierarquia destas classes. Este fato é compatível com a informação prévia dos tipos referenciada por Agesen [AGE 95].

Além disso, esta análise baseia-se nas informações sobre as declarações das variáveis [BAC 96]. Ressalta-se que, em um programa orientado a objetos, a declaração de um objeto de determinada classe informa que este pode ao longo da aplicação comportar-se como um objeto da classe declarada ou como um objeto das subclasses desta. Tendo-se estas informações (hierarquia das classes + tipo da declaração) a análise da hierarquia das classes consegue resolver várias invocações de métodos, ou seja, determinar estaticamente qual versão do método irá responder a determinada invocação. Este resultado é alcançado porque em muitos casos a hierarquia das classes revela que não existe mais de uma implementação de determinado método ou que o conjunto de versões é pequeno ou ainda que o tipo declarado só pode referenciar a uma das versões do método invocado.

Outra característica desta técnica é que preferencialmente deve-se conhecer (ter acesso) todas as classes que compõem o programa, pois tendo-se a hierarquia completa destas classes pode-se prover informações sobre todas as possíveis versões de um

método. Esta preferência é indicada pelo fato desta análise basear-se no grafo de hierarquia para determinar as versões (implementações) que determinado método implementa.

Esta análise pode ser aplicada a linguagens tipadas dinamicamente. Este processo é feito através da inclusão de uma classe de tratamento de erro, a qual é adicionada ao conjunto de possíveis tipos que podem responder a determinada invocação. No entanto, o enfoque explorado neste trabalho será a aplicação desta técnica em linguagens tipadas estaticamente [DEA 94].

Outro aspecto importante é que esta análise é insensível ao fluxo (*flow insensitive*), ou seja, a CHA trata os comandos de um programa como um conjunto de instruções desprezando a seqüência destes [HAS 98]. Este tipo de análise possui a vantagem de ser menos complexa e consequentemente mais rápida no processamento das informações. No entanto, desprezar a seqüência dos comandos pode acarretar, algumas vezes, em informações menos precisas.

2.3.2 Análise Rápida de Tipos

A análise rápida de tipos (*Rapid Type Analysis*) ou RTA [BAC 96] visa detectar as versões dos métodos invocados através da análise dos tipos das variáveis do programa. Para isto esta técnica propõe a geração de um grafo das chamadas (invocações) do programa através da análise da hierarquia das classes. Posteriormente, utilizando-se o grafo inicial, considera-se as questões sobre as instanciações dos objetos declarados. Isto é feito em decorrência da instanciação vincular um determinado tipo ao objeto instanciado, fato que não ocorre se considera-se somente a declaração do objeto. O vínculo de determinado tipo a um objeto auxilia na redução do conjunto de possíveis versões que podem atender a uma certa invocação, ou seja, reduz os caminhos possíveis que o fluxo de controle pode assumir diante de uma chamada de método.

Então, se uma invocação é realizada, através da informação sobre o tipo que a expressão de invocação assumiu devido a sua instanciação, é possível determinar, algumas vezes precisamente, que versão do método que está sendo chamada.

Como a análise rápida de tipos baseia na análise da hierarquia das classes, esta possui as limitações desta, ou seja, a análise é propícia para as situações onde sabe-se informações sobre todas as classes da aplicação. Outra questão interessante é que, como a análise da hierarquia das classes, a análise rápida de tipos é insensível ao fluxo (*flow insensitive*) [BAC 96].

2.3.3 Análise de Tipos das Variáveis

A análise de tipo das variáveis utiliza o nome das variáveis como sua representação [SUV 98]. Então o grafo de propagação dos tipos é construído levando-se em conta os seguintes critérios:

- para todo atributo de classe, o qual corresponde a um objeto, cria-se um nodo no grafo;
- para todo parâmetro de método, o qual corresponde a um objeto, cria-se um nodo no grafo;
- para toda variável local de cada método, a qual representa um objeto, cria-se um nodo no grafo.

Depois de criar todos os nodos referentes ao programa deve-se propagar os tipos destes. Para isso analisa-se os comandos de atribuição, os quais representam o fluxo dos dados do programa.

Então para cada comando de atribuição simples da forma $x = y$, deve-se propagar o tipo de y para x. Esta propagação é representada através de uma seta, a qual acompanha o sentido do fluxo dos dados. Já para comandos de atribuição resultantes de uma invocação de método e para invocações de métodos sem retorno deve-se verificar primeiramente quais versões deste método podem responder a esta invocação e a partir daí começar a inserir setas para cada uma das correspondentes versões. Além disso deve-se inserir uma seta das variáveis que serão passados como argumentos para a entrada do método invocado, o mesmo procedimento deve ser feito inversamente em relação ao nodo representante do retorno se o método invocado possui retorno.

Uma questão abordada em [SUV 98] é sobre a análise de *aliases*, ou seja, duas variáveis são *aliases* se apontam para o mesmo objeto ou região de memória. Segundo Sundaresan todas as regras de declarações assumem que uma variável (objeto) e todas as suas *aliases* são representadas pelo mesmo nodo no grafo de propagação de tipos. Além disso, os atributos de instâncias são representados por um único nodo, mesmo que existam várias (número indefinido) de instâncias de uma determinada classe. No entanto para vetores estas regras não se verificam, pois identificar todas as variáveis que apontam para uma mesma estrutura (vetor) é difícil. Para contornar este problema deve-se assumir que as setas de atribuição para vetores devem ser bidirecionais, assegurando assim que ambos os nodos conterão a mesma informação.

A propagação de tipos tem início com a análise dos comandos de instanciação, ou seja, de criação de objetos. Posteriormente são identificados os componentes com alto grau de relacionamento. Estes tipos de componentes se caracterizam por se atribuírem mutuamente os seus valores. Posteriormente são propagados os tipos. Um exemplo desta análise é dado na figura 2.1.

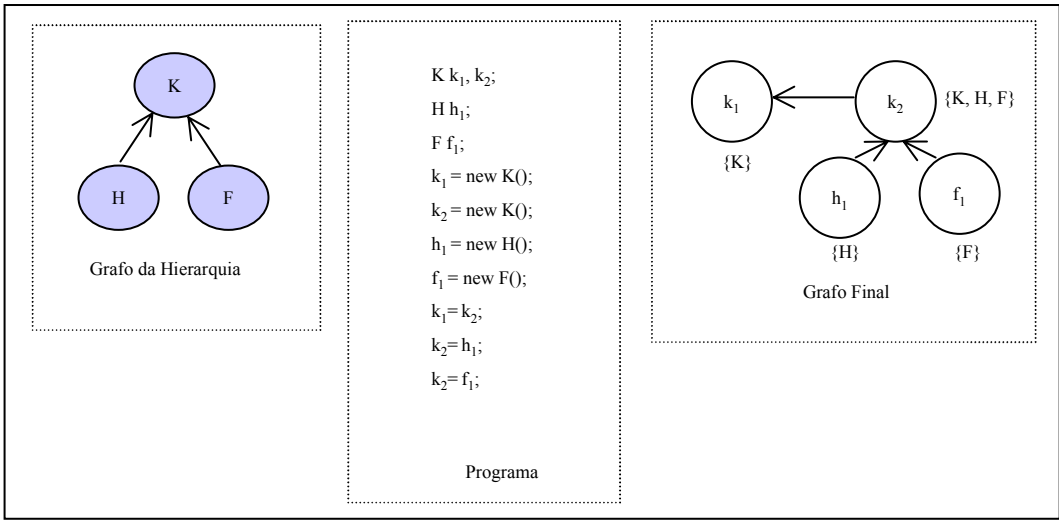


FIGURA 2.1 - Resultado da Análise do Programa Exemplo, baseado em [SUV 98]

Como pode-se perceber o grafo construído não considera a ordem de ocorrência do fluxo, portanto a análise dos tipos das variáveis é uma análise insensível ao fluxo [SUV 98].

2.3.4 Análise de Tipos Declarados

A análise de tipos declarados procede como a análise dos tipos das variáveis. No entanto, o grafo se diferencia, pois os nodos representam as classes declaradas e não as variáveis. Então cada nodo representa o conjunto de variáveis de determinado tipo. As setas representam o relacionamento de atribuição entre estas [SUV 98].

O grafo gerado a partir da análise de tipos declarados é menor que o grafo gerado pela análise de tipo das variáveis. Além, disso, o grafo da análise de tipos declarados pode ser considerado como uma representação resumida do grafo gerado pela análise de tipo das variáveis. Outra característica é que este grafo, também, é insensível ao fluxo. Para elucidar o grafo gerado pela análise apresenta-se este na figura 2.2, considerando-se o mesmo programa exemplo mostrado na figura 2.1.

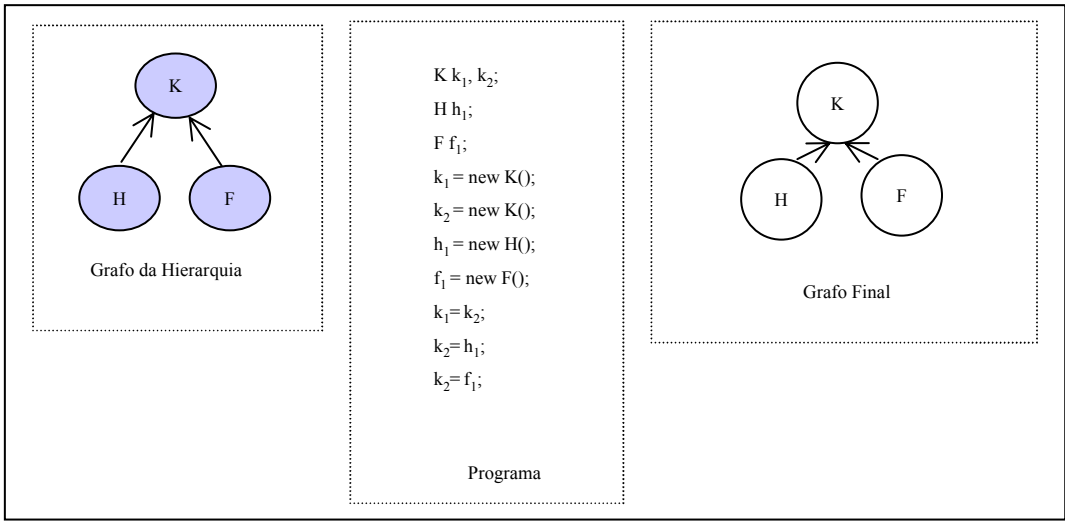


FIGURA 2.2 - Resultado da Análise do Programa Exemplo, baseado em [SUV 98]

2.4 Análise Estática de Programas Java

2.4.1 Grafo de Fluxo Interprocedural Parcial (PIFG)

O algoritmo descrito por Agrawal visa gerar um grafo de chamadas de um programa por demanda. A análise por demanda alivia a necessidade da exaustiva computação das informações de fluxo de dados em todo o programa [AGR 99], ou seja, somente o que realmente irá interferir no processamento da análise será analisado.

Outro fator que incentiva o uso de análise por demanda é que em muitos casos como: ambientes de testes, ambientes de desenvolvimento de software, etc, é requisitada a análise ou verificação de uma ou de algumas partes especificas do programa [AGR 2001].

Esta análise tem como linguagens alvo C++ e Java, ou seja, linguagens tipadas estaticamente. Além disso, algumas considerações são feitas quanto ao modelo de linguagem [AGR 99]:

- assume-se que não existem variáveis globais;

- não há tratamento de herança múltipla, ou seja, um método descende diretamente de uma única classe;
- tipo de um objeto é definido através de um comando de instanciação e somente pode ser alterado através de outro comando de instanciação.

O algoritmo de análise desempenha sua função em passos (fases), os quais podem ser sumarizados como segue:

- Primeiramente é construído um grafo conservativo, o qual utiliza as informações da análise da hierarquia das classes [DEA 94] para realizar tal tarefa. Posteriormente é construído o grafo parcial através do grafo conservativo. Esta última fase leva em consideração a análise dos nodos que influenciam no processamento dos elementos que estão sendo analisados.
- Tendo-se identificado o conjunto de nodos que influenciam na construção do grafo interprocedural parcial, propaga-se os dados utilizando-se as informações de tipos referentes às definições encontradas em determinado ponto. Isto tem a finalidade de prover algum refinamento, se possível, através da propagação dos tipos.

O processo de identificação dos nodos influentes é feito juntamente com a criação do grafo parcial. Este processo utiliza duas estruturas de dados, denominadas *Workset* e *Procset*. A estrutura *Workset* contém os nodos cujos nodos predecessores não foram analisados ainda. A estrutura *Procset* contém os nodos que foram identificados como relevantes, ou seja, os nodos que terão suas porções presentes no grafo parciais [AGR 99].

O processo de análise retira um nodo da *Workset*. O nodo removido é analisado, ou seja, verifica-se se este é um nodo de chamada (invocação) ou um nodo de saída. Isto serve para se fazer o *link* entre nodo de entrada e nodo de chamada ou entre nodo de saída e nodo de retorno, pois estes (entrada e retorno) também serão adicionados a *Workset*. A figura 2.3 apresenta um grafo envolvendo nodos de chamada, entrada, saída e retorno. O conjunto de métodos que podem responder a determinada invocação é conhecido pela análise da hierarquia das classes.

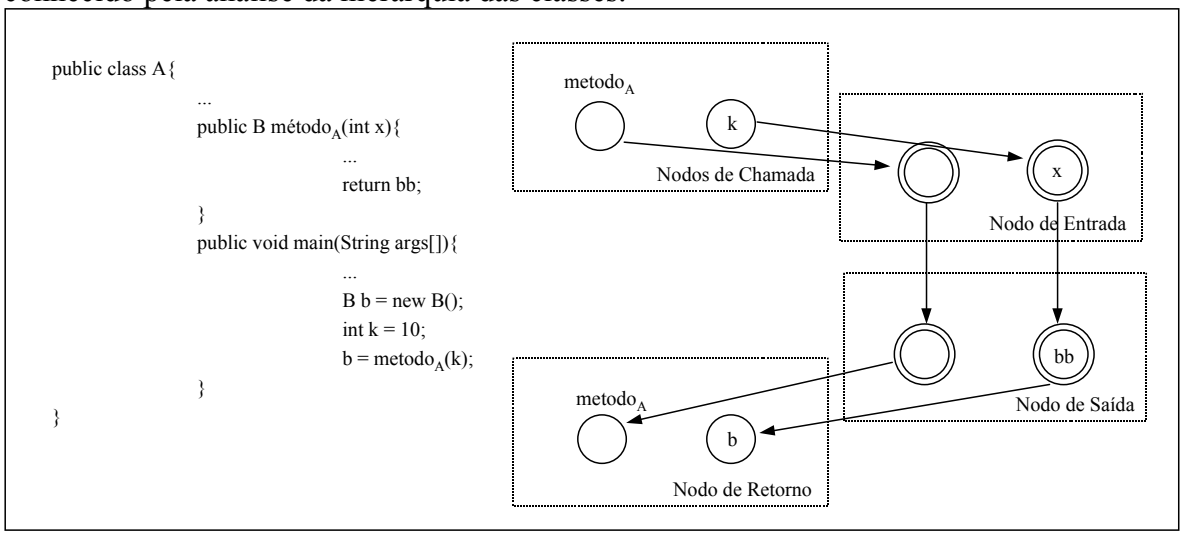


FIGURA 2.3 - Grafo Contendo Nodos de Chamada, Entrada, Saída e Retorno

Então, através deste processo identifica-se todos os nodos que influenciam no processamento de determinada chamada. Estes nodos, após a identificação, tem suas porções construídas no grafo parcial.

Após a inserção dos nodos relevantes, para a análise no grafo parcial, realiza-se a análise de propagação dos tipos sobre este grafo. Isto é feito pelo fato do grafo parcial ser baseado nas informações da análise da hierarquia das classes, a qual considera todas as versões de um método que podem atender a determinada invocação, sem considerar o tipo da expressão invocadora nem os tipos dos argumentos naquele ponto. A tarefa de propagação das informações de tipos é realizada da seguinte maneira [AGR 99]:

- valor inicial de TYPES(v) (sendo *v* um nodo) é computado utilizando-se as informações da hierarquia das classes;
- se o nodo é de chamada ou de saída a informação de tipo não se altera, o que ocorre é a união da informação de tipo correspondente ao nodo predecessor, ou seja, o nodo de retorno ou entrada do método invocado, com a informação de tipo do nodo de chamada ou saída, respectivamente.
- se o nodo é de entrada deve-se considerar a informação sobre as possíveis versões deste método que podem atender a determinada invocação. Por este motivo, se *p* é um nodo de chamada, o qual é predecessor do nodo *v* de entrada, deve-se propagar a informação de tipo do nodo predecessor *p* para o nodo *v*, que está sendo analisado.
- se o nodo é de retorno, deve-se realizar o mesmo procedimento realizado para o nodo de entrada. No entanto, o nodo predecessor neste caso é o nodo de saída do método invocado, e é esta informação que é propagada para o nodo de retorno.

Um aspecto que se evidencia é que a análise considera que o tipo de uma determinada variável só pode se modificar através de um novo comando de instanciação. Isto descarta a possibilidade real das linguagens de se alterar o tipo de uma variável através de um comando de atribuição simples como:

x=y;

no qual a variável *x* foi previamente instanciada com um determinado tipo *A*, do qual o tipo *B* descende. Já a variável *y* também foi previamente instanciada com o tipo *B*. Após a realização deste comando (*x=y*) as variáveis *x* e *y* conterão o mesmo objeto e consequentemente o mesmo tipo.

A desconsideração deste fato facilita a propagação dos tipos, tornando este algoritmo mais rápido e com menos necessidade de processamento. A rapidez de um processo de análise estática é uma característica importante e ressaltada como uma das metas deste tipo de modelo.

2.4.2 Análise de Fluxo de Controle - CFA_0

A análise clássica de fluxo de controle (CFA) computa para cada subexpressão a função que está pode avaliar. Através disso é possível saber para onde o fluxo de controle irá ser transferido. Para linguagens orientadas a objetos a análise de fluxo de controle traça os conjuntos de tipos (classes) que uma variável/objeto pode apontar em tempo de execução [PRO 2001].

Para realizar a análise de fluxo de controle o *framework* descrito por Probst, o qual realiza a análise denominada CFA_0, necessita rotular os parâmetros de métodos,

variáveis e finais de métodos com um único identificador o qual é pertencente a um conjunto de identificadores denominado *Lab*. O subconjunto da linguagem Java usado possui as seguintes construções sintáticas.

rtype	Definição de método
rtype	Definição de método
$v_1 = v_2$	Atribuição
v = new C()	Criação de objeto
v = o.m(a)	Invocação de método
return v	Retorno

O valor abstrato propagado pela análise são elementos do conjunto de classes denominado *Val*, o qual é o conjunto de classes. A função,

$$\text{Cache} = \text{Lab} \rightarrow \text{Val}$$

fornece como valor o mapeamento de um identificador (símbolo) para o seu tipo(s) (classe(s)) correspondente. O resultado da análise de fluxo é um elemento pertencente aos resultados da função *cache*, ou seja, um mapeamento entre um determinado identificador e seu tipo correspondente.

Então para cada comando da sublinguagem considerada definiu-se um regra para a análise de fluxo de controle (CFA) determinar o valor abstrato deste. O funcionamento destas regras é:

- variável — Esta regra mapeia o identificador dado a uma variável, em um ponto do programa, para o tipo desta no ponto correspondente.
- Atribuição — A regra para um comando de atribuição realiza a união do tipo da expressão do lado esquerdo (do comando) ao tipo da expressão do lado direito (do comando).
- Criação (instanciação) — Para o comando de criação (instanciação) a função *cache* mapeia o identificador da instância para o tipo que está sendo vinculado a esta. Nota-se que este tipo tem que estar contido no tipo de declaração do objeto, ou seja, ser o próprio tipo da declaração ou algum dos subtipos deste.
- Retorno — A função *cache* para um ponto de retorno mapeia o identificador deste para o tipo retornado pelo método invocado.
- Sem argumentos — A função *cache* para uma invocação de método sem argumentos mapeia o identificador do objeto que realiza a invocação e identificador de retorno do método invocado para os correspondentes tipos para detectar qual versão esta sendo chamada.
- Argumentos — A função *cache* para uma invocação de método sem argumentos mapeia o identificador do objeto que realiza a invocação, o identificador de retorno do método invocado e os identificadores dos argumentos para os correspondentes tipos para detectar qual versão esta sendo chamada.

- Método sem argumentos: A função *cache* mapeia o conjunto de restrições para uma invocação de método, ou seja, o conjunto de métodos (versões) que podem atender a determinada invocação. Estas restrições levam em conta os tipos que o objeto que realiza a invocação pode assumir
- Método com argumentos: A função *cache* mapeia o conjunto de restrições para uma invocação de método. Estas restrições levam em conta os tipos que os argumentos podem assumir e os tipos que o objeto que realiza a invocação pode assumir

Antes de gerar as restrições na propagação dos tipos o analisador realiza a computação, para cada variável em cada ponto, dos identificadores (rótulos) de definição da variável que são visíveis neste ponto. A análise de variáveis globais também é realizada para cada ponto do programa. Proporcionando assim o conhecimento do tipo das variáveis globais em cada ponto [PRO 2001].

O analisador gera restrições intraprocedurais, ou seja, realiza a análise interna dos métodos do programa. As restrições intraprocedurais são geradas conforme estas regras. Isto é realizado para que posteriormente durante a análise interprocedural do fluxo, cada método invocado já esteja previamente analisado, facilitando a propagação das informações de tipos. Para realizar a análise interprocedural deve-se, além das regras de identificação,

- realizar a unificação dos argumentos atuais, que estão sendo passados para um determinado método, com os parâmetros formais deste.
- a unificação do final do procedimento com a expressão que causou a invocação, ou seja, o tipo retornado pela invocação deve ser atribuído a expressão que causou esta.

No analisador é adotado o conceito de atribuição de restrições por demanda, ou seja, só são atribuídas restrições quando estas são necessárias. Para isto se introduz uma macro restrição denominada $CALL_m$, onde *m* é o método invocado [PRO 2001].

- rótulo l_v da variável que contém o objeto que realiza a invocação;
- rótulo l_r para o qual o resultado da invocação deve ser atribuído;
- rótulo l_i de todos os argumentos que são passados para o método invocado.

Então o formato da macro restrição é: $CALL_m(l_v, l_r, [l_i])$, sendo que se o método invocado não possui argumentos a restrição é: $CALL_m(l_v, l_r)$.

Depois de terem sido criadas todas as restrições para todos os métodos no programa, deve-se resolver estas restrições. A resolução das restrições é feita sobre um grafo, o qual representa o programa que está sendo analisado. Neste grafo os nodos representam as variáveis e as invocações de métodos. As setas representam as restrições entre os nodos. Para cada nodo *nl* um campo $D[n_l]$ é associado, o qual computa o conjunto de tipos de n_l. Então se *nl* é um nodo de invocação de método deve-se analisar o corrente conjunto de tipos, que o objeto que causa a invocação pode assumir (se o método possuir argumentos estes também devem ser avaliados). Tendo-se esta informação deve-se procurar quais versões do método invocado podem responder a invocação em questão. Este processo é feito através da análise do grafo da hierarquia das classes [DEA 94].

2.4.3 Grafo de Chamadas em Java

O analisador proposto por Dewes tem o objetivo de computar estaticamente o tipo dinâmico das expressões de invocações de métodos [DEW 2001].Para tal finalidade o analisador utiliza um gerador de analisadores denominado PAG (*Program Analyzer Generator*), o qual tem a finalidade de definir a análise de fluxo de dados.

O tipo da expressão de invocação, muitas vezes não pode ser obtida com precisão. No entanto, através de uma análise como a descrita em [DEW 2001] pode-se obter resultados mais precisos e com isso aumentar a precisão do grafo de chamadas correspondente ao programa analisado.

A realização da análise proposta por Dewes conta com questionamentos a respeito das estruturas da linguagem que devem ser analisadas em busca das informações de tipos das expressões de invocações. Neste sentido o trabalho encontra-se voltado para a utilização da análise na linguagem Java.

Dentro do contexto da linguagem Java pode-se perceber que as sintaxes que uma expressão de invocação pode assumir, no formato mais geral, são: *MethodName(arguments)*; *Primary.Identifier(argments)*; *Super.Identifier(arguments)*; *ClassName.super.Identifier(arguments)*.

Nota-se que nas situações aonde a palavra reservada *super* aparece o processo de análise não encontra maiores dificuldades, pois só necessita procurar na primeira das suas superclasses que implementa o método invocado e repassar está invocação como uma chamada direta de procedimento. Já quando trata-se com o estilo de sintaxe aonde a palavra reservada *super* não se faz presente o processo de análise necessita avaliar outras construções sintáticas para determinar o tipo da expressão que está invocando o método. Estas expressões são: as variáveis, os atributos de classe, os atributos de instâncias, os vetores e as invocações de métodos.

Para realizar a análise do fluxo dos dados deve-se ter o grafo do fluxo dos dados referente a aplicação. Este último é construído a partir das informações geradas pela análise rápida de tipos [BAC 96], a qual leva em consideração as instanciações dos objetos do programa para determinar o tipo, conservativamente, das expressões de invocações dos métodos.

Além disso o analisador em questão faz uso de um domínio funcional, onde cada alocação é mapeada para um conjunto de referências O identificador da alocação consiste de uma *string* única e um *flag* indicando se a alocação é uma variável local ou um campo. A referência consiste de quatro partes:

- A primeira parte é o ID do nodo onde a referência foi declarada;
- Número de contexto (representa o contexto de invocação, pilha); A análise usa estes números para distinguir variáveis criadas na mesma função mas em contextos diferentes desta.
- ID do nodo aonde a referência foi criada;
- tipo dinâmico, ou seja, o nome da classe da referência.

Uma característica ressaltada em [DEW 2001] é que a perda de precisão nas informações sobre os tipos das expressões de invocações de métodos deve-se ao fato de algumas construções da linguagem inviabilizarem o acompanhamento do fluxo de dados individualmente, ou seja, algumas estruturas dificultam a identificação e consequentemente o tratamento individual de cada variável do programa. Uma destas estrutura são os atributos de instâncias, pois para se ter um tratamento individual destas estruturas deve-se ter um tratamento individual dos objetos do programa. O tratamento individual de cada objeto é uma tarefa altamente complexa pois implica em se ter um

controle sobre quais variáveis apontam para o mesmo objeto e quais variáveis com o mesmo nome apontam para objetos distintos. Outra estrutura que dificulta o tratamento individual são os vetores, pois estes podem conter elementos de tipos diferentes que no entanto fazem parte de uma mesma estrutura.

Uma forma de tratar estes pontos de imprecisão é fazer com que estas sejam controladas através do número de contexto destas estruturas, do ID, ou seja, o identificador do nodo de criação da variável e do tipo de criação desta. Já para os vetores se considera que estes são um único objeto e o seu tratamento procede como para atributos de variáveis.

Depois de realizar a análise das referências abstratas das variáveis do programa realiza-se a análise interprocedural, a qual permite que se obtenha informações sobre os valores retornados de uma invocação de método ou dos valores enviados para uma invocação de método. Esta fase conta com o auxilio da ferramenta PAG, utilizada pelo analisador. O algoritmo necessita, somente decidir quais informações serão enviadas sobre as setas do grafo. Estas informações são as referências abstratas ou locações, as quais são enviadas sobre as setas de uma chamada de método se e somente se esta locação (variável) é visível para o método que está sendo invocado [DEW 2001]. Como pode-se perceber esta análise baseia-se na informação do fluxo de dados da aplicação ou seja é uma análise sensível ao fluxo, além disso, a análise também considera o contexto das variáveis, ou seja, a análise também é sensível ao contexto.

2.4.4 Análise de Existência

Em [SRE 2000] é proposto um *framework* de análise denominado *extant analysis* (análise de existência). Este tem o objetivo de realizar a análise interprocedural de programas que suportam carga dinâmica de classes.

A análise baseia-se no fato que em um programa Java classes podem ser carregadas dinamicamente, o que dificulta a otimização dos programas utilizando-se o principio da análise de todo o programa. Levando-se em conta este aspecto, o *framework* propõem a detecção de invocações de métodos que possam ser seguramente convertidas para invocações de métodos das classes que o analisador tem acesso. Este conjunto de classes acessíveis é denominado mundo fechado.

Tendo-se este objetivo o *framework* trabalha sobre um *lattice*, ou seja, sobre um conjunto, o qual representa a classificação das informações coletadas pela análise. A tabela 2.1 apresenta o *lattice* utilizado.

TABELA 2.1 - *Lattice* Utilizado pela Análise de Existência

SIGNIFICADO	SÍMBOLO
Desconhecido (*Unknown*)	$\top$
Incondicionalmente *extant*	UCE
Condicionalmente *extant*	CE ou $\bot$

Quando a análise consegue determinar que determinada expressão de invocação de método nunca irá apontar para um objeto de algum dos tipos que compõem o mundo fechado diz-se que este é UCNE, ou seja, incondicionalmente não existente. Se há a possibilidade determinada expressão apontar para um objeto existente (objeto de algum dos tipos do mundo fechado) diz-se que está é condicionalmente existente. Já quando existe a certeza, estática, de que determinada expressão será de algum dos tipos que

compõem o mundo fechado está é incondicionalmente existente, ou seja, UCE [SRE 2000].

Baseando-se nestas informações o *framework* pode especializar o código quando isto for útil para o melhoramento do desempenho da aplicação.

A primeira ação é determinar o mundo fechado, ou seja, as classes que compõem este universo. Após isso é feita uma análise de fluxo para determinar os pontos onde uma determinada invocação é condicionalmente existente, incondicionalmente existente ou incondicionalmente não existente. Após inferir estas informações o programa pode ser especializado se necessário ou melhor se conveniente.

A especialização do programa realiza a transformação deste para que em tempo de execução, conforme a informação de tipo, possa-se agilizar o processo de ligação dinâmica dos métodos. A transformação inclui ao código testes existência, os quais representam as versões dos métodos que podem ser invocados. Então durante a execução pode-se saber exatamente o tipo vinculado a determinado objeto invocador e assim seguir o fluxo do teste que corresponde a tal informação.

Esta técnica é propícia para a inferência precisa das informações de tipos, o que é uma característica dinâmica, pois combina análise estática com análise dinâmica. Além disso, a presença de testes durante a execução, que apontam para as possíveis versões que determinada invocação pode assumir, ajuda a diminuir o *overhead* decorrente da ligação dinâmica de chamadas.

2.4.5 Análise Estática de Tipos em Programas JavaParty

Esta análise visa a distribuição de programas em arquiteturas paralelas com memória distribuída. Neste contexto, as informações relevantes são as que propiciam um auxilio no escalonamento dos objetos, o qual deve preocupar-se com a minimização das comunicações remotas.

A linguagem JavaParty objetiva ocultar o endereçamento e os mecanismos de comunicação do usuário. Então, embora objetos de classes remotas, as quais são uma extensão da classe *thread* do Java, possam residir em diferentes máquinas seus métodos e variáveis podem ser acessados da mesma maneira como no puro Java [PHI 97].

O fator da localidade é importante pois as invocações em Java são síncronas. Isto conduz ao seguinte cenário: se um objeto alocado em um nodo n_1 invoca um método de um objeto alocado em um nodo n_2, a execução deste método (atendimento da invocação) torna-se concorrente com o fluxo de execução atual do nodo n_2. Esta situação leva a uma perda de desempenho por parte da aplicação [PHI 2000].

Como a explicação anterior sugere um dos desafios seria determinar qual método está sendo chamado. Para isso deve-se realizar a análise de tipos. Além disso, deve-se ter conhecimento das ações de cada *thread* separadamente. Visando isso Philippsen cita duas técnicas para prover a precisão das informações de tipos:

- Clonagem de métodos: a clonagem de métodos acontece quando em uma invocação de método o(s) argumento(s) deste possuem informações imprecisas quanto ao(s) seu(s) tipo(s). Para contornar esta imprecisão clona-se o método e assume-se que cada versão deste irá receber um dos tipos possíveis que o(s) argumento(s) pode assumir. Com esta ação os fluxos podem seguir independentes e a imprecisão é contornada, ou seja, é propaga de forma individual, possibilitando a visão independente de cada uma das versões que podem ser invocadas.

- Clonagem de classes: a clonagem de classes acontece quando em uma invocação de método a expressão invocadora possui informação imprecisa sobre a sua tipagem. Neste caso, clona-se a variável para que os seus clones possam assumir os tipos referentes em sua informação de tipos. Assim, pode-se prever quais versões dos métodos podem ser chamadas. Isto, também, proporciona uma visão individual das versões que determinada invocação de método pode assumir.

Sendo assim, através da clonagem de métodos e de classes o analisador pode prover informações sobre cada fluxo de cada *thread* individualmente. Além disso, é utilizado uma técnica de identificação individual de cada linha de execução criada. Isto permite o tratamento individual de cada uma destas, o qual é o objetivo da análise. Este objetivo é visado pelo fato das ações, invocações de métodos entre as *threads*, determinarem ou pelo menos levarem a uma conclusão da melhor localidade que esta deve assumir. No entanto, o analisador não se preocupa em inferir estas informações fazendo o papel de prover informações para auxiliar na tomada de decisões sobre a localidade.

Então, toda vez que a análise depara-se com uma informação imprecisa, seja de argumentos ou de expressões invocadoras, esta realiza a técnica de clonagem apropriada. Isto leva a se ter um retrato de todos os caminhos possíveis individualmente.

2.4.6 Análise e Comparação dos Modelos Apresentados

Está seção apresenta uma análise e uma comparação entre os analisadores estáticos de programas Java apresentados nas subseções anteriores.

Através dos trabalhos relatados nesta seção pode-se perceber várias características em comum e divergentes entre estes. Uma das características em comum é a análise de programas Java. Além disso, os trabalhos visam o auxilio destes programas, seja no processo de desenvolvimento ou no processo de execução dos mesmos. Isto ressalta o fato da análise estática estar vinculada a obtenção de informações que desencadeiem uma otimização da aplicação.

O grafo de fluxo interprocedural parcial (PIFG) visa auxiliar no processo de desenvolvimento de software, através da detecção do grafo de fluxo de partes do programa, as quais são requeridas pelo usuário.

Já o algoritmo desenvolvido por Probst visa a geração de um grafo de chamadas total. Esta característica vai ao encontro do propósito da análise do algoritmo CFA_0, o qual é o auxilio no processamento dos programas através da conversão de invocações dinâmicas em chamadas diretas a procedimentos. Este também é o propósito do trabalho descrito por Dewes.

No entanto para realizar isto o algoritmo CFA_0 utilizar como ponto de partida a análise da hierarquia das classes e o algoritmo de Dewes utilizar a análise rápida de tipos. Estas análises são utilizadas em ambos para prover uma visão das classes envolvidas na computação. Este aspecto é interessante pois estas duas análises (CHA e RTA) são análises insensíveis ao fluxo, as quais são utilizadas pelos três primeiros trabalhos apresentados nesta seção para gerar um ponto de partida para a análise de tipos.

Deve-se ressaltar que estas análises iniciais são importantes para prover condições de se aplicar o algoritmo ideal descrito por Agesen, o qual necessita do conhecimento prévio dos tipos que participam da computação.

Já o trabalho descrito por Sredall enfoca o aspecto da carga dinâmica de classes durante a execução de um programa Java. Outro aspecto interessante deste trabalho é o vinculo entre análise estática e análise dinâmica, o qual é, muitas vezes, necessário devido ao fato das invocações dinâmicas só serem resolvidas dinamicamente.

Phillipsen enfoca a análise estática de programas Java concorrentes. Para isto este trabalho realiza a análise de tipos visando obter informações que contribuam para uma melhor alocação das *threads* que compõem o programa em um ambiente distribuído.

2.5 Considerações Finais

A análise estática visa otimizar a execução das aplicações. Para isto, esta técnica baseia-se nas informações textuais de um programa, as quais representam as ações dinâmicas desempenhadas por este. Então através da simulação da execução de uma aplicação pode-se obter informações sobre o comportamento que esta terá durante a execução.

Em programas orientados a objetos existem vários aspectos dinâmicos, os quais são inerentes deste paradigma, como: verificação dinâmica de tipos, ligação dinâmica de invocações, etc. Neste contexto, a análise estática pode contribuir através da verificação estática dos tipos. Este processo consegue, muitas vezes, determinar estaticamente os tipos das variáveis, o que propicia a resolução estática de várias invocações. Esta contribuição diminui o *overhead* causado pela verificação dinâmica de tipos e conseqüentemente aumentar o desempenho das aplicações. Além disso, a análise estática de tipos representa a base para outras análises, as quais objetivam outros empregos como: desenvolvimento de *software*, escalonamento, ect.

3 O Modelo DEPAnalyzer

Este capítulo apresenta a descrição do modelo proposto neste trabalho. A seção 3.1 aborda aspectos relacionados à concepção do modelo. Já a seção 3.2 introduz a visão geral da estrutura deste. As seções 3.3 e 3.4 oferecem, respectivamente, uma visão detalhada do módulo de coleta das informações e do módulo de análise. A seção 3.5 apresenta uma comparação entre o modelo proposto e os modelos abordados na seção 2.4. A seção 3.5 mostra as considerações finais deste capítulo.

3.1 Aspectos Conceituais

Os relacionamentos entre as entidades de um programa (conjuntos de objetos) representam as dependências entre estes. Estes relacionamentos são estabelecidos através das invocações de atributos e de métodos entre as classes. O DEPAnalyzer (*DEPendence Analyzer*) é um modelo de análise estática de programas orientados a objetos que visa capturar estes relacionamentos.

As dependências revelam o grau de relacionamento entre as classes do programa. Esta informação pode ser útil no processo de desenvolvimento de *software*, pois detecta o grau de acoplamento das entidades. Na área de desenvolvimento de *software* orientado a objetos ressalta-se que as classes, as quais representam uma abstração das entidades reais (de tempo de execução), devem ser coesas. Isto confere uma maior possibilidade de reuso [AMB 98]. Assim, o relacionamento entre as classes pode auxiliar no processo de desenvolvimento de aplicações verificando o acoplamento existente entre as entidades modeladas.

Neste sentido o DEPAnalyzer poderia prover informações para uma ferramenta de engenharia reversa. No entanto, a principal diferença entre estes universos é que as ferramentas de engenharia reversa normalmente tratam de aspectos sintáticos sem considerar aspectos dinâmicos da aplicação [FUJ 2001], [WEB 2001]. Além disso, o DEPAnalyzer também se diferencia de ferramentas como [VER 2001], as quais capturam aspectos dinâmicos como instanciações de objetos, pelo fato de simular a execução de um programa considerando-se o método principal da aplicação analisada.

No entanto, é no suporte ao escalonamento em uma arquitetura distribuída que as informações geradas pelo DEPAnalyzer visam auxiliar. Considerando que as dependências revelam comunicações entre classes, é possível separar estes conjuntos de objetos em grupos comunicantes, isto é, que possuem dependências. Esta ação leva a um menor custo geral de processamento, pois visa reduzir as comunicações inter-nodos, as quais são um dos fatores que degradam o desempenho da aplicação.

Além disso, a análise de dependências pode auxiliar, não somente em um processo inicial de alocação, mas também, no processo de redistribuição das entidades da aplicação no decorrer da execução da mesma. Isto é possível pelo fato da análise de dependências percorrer as possíveis invocações ocorridas no programa a fim de detectar as dependências entre as classes analisadas. Esta ação pode gerar o grafo das invocações feitas durante a simulação da execução do programa.

Tendo em vista este objetivo, o modelo baseia-se na abordagem do pior caso. Nesta se uma dependência pode ocorrer em tempo de execução, mas sua ocorrência não

é confirmada em tempo de compilação, considera-se que esta irá ocorrer. A abordagem do pior caso evita que ocorra em tempo de execução algum relacionamento não considerado pelo escalonamento, o que pode levar a comunicações entre os nós da arquitetura, isto é, a um custo adicional de processamento. No entanto, as informações geradas por esta abordagem podem levar a uma limitação na exploração do paralelismo da aplicação. A redução é causada pela proposta pessimista de que todas as possíveis ocorrências de relacionamento serão verificadas como verdadeiras. No entanto, isto prevê uma concepção conservativa no sentido de escalonar as entidades somente quando estas verificam uma baixa dependência.

Outro aspecto do modelo é que os programas analisados não possuem interação com o usuário, ou seja, as aplicações analisadas devem possuir um comportamento *CPU bound*. Neste tipo de programa uma vez disparada a execução da aplicação esta não necessita ser alimentada com informações durante o processamento. Este comportamento facilita a análise, pois não necessita simular os padrões de interações que ocorreriam se o programa tivesse esta característica.

A princípio a modelagem do DEPAnalyzer [AZE 2001], visava somente a análise de programas com polimorfismo de sobrecarga. Isto se deve ao fato do polimorfismo de sobreescrita requerer uma análise do fluxo dos tipos para sua detecção, conferindo assim um maior grau de complexidade a análise. No entanto, a não avaliação de programas com polimorfismo de sobreescrita por parte do sistema reduz o escopo das aplicações analisadas uma vez que o processo de desenvolvimento de *software* orientado a objetos incentiva a utilização deste recurso.

Além disso, o polimorfismo como um todo é uma característica que confere poder [STE 99] as aplicações desenvolvidas neste paradigma. Em decorrência disso, o DEPAnalyzer, atualmente [AZE 2001_a], analisa programas com polimorfismo tipo reescrita.

Porém, deve-se ressaltar que este tipo de aplicação pode apresentar, em nível estático, uma certa indefinição nos possíveis métodos que podem responder a determinada invocação, tornando necessário à análise de todas as possíveis implementações. Isto causa uma exponencialização no processo de análise. Devido a este fato o modelo, adota a sugestão dada por Agesen ([AGE 95]) aos programas submetidos ao algoritmo do produto cartesiano, ou seja, sugere que os programas submetidos ao DEPAnalyzer possuam baixo grau de polimorfismo. Isto facilita o processo de análise uma vez que a exponencialização causada também será baixa, não conferindo um aumento indesejável na complexidade do processo desempenhado pelo sistema.

O grau de polimorfismo depende de vários fatores como: quantidade de métodos reescritos presentes nas classes da aplicação, da incidência de invocações destes métodos, do número de implementações que podem responder a determinada invocação e da precisão da informação de tipo, referente ao objeto invocador, no momento da chamada. Por exemplo, se um programa apresenta vários métodos reescritos, os quais são invocados constantemente no código da aplicação, mas a informação sobre o tipo do elemento que esta realizando a invocação é uma informação precisa, estas invocações não vão causar uma exponencialização no processo de análise. Já se existe poucos métodos reescritos, mas suas ocorrências são inúmeras e a informação sobre o tipo da variável invocadora é impreciso a análise pode atingir um nível alto de exponencialização.

3.2 Visão Geral do Modelo

O modelo em seu mais alto nível de abstração recebe um programa fonte Java e gera o grafo de dependências ou o grafo de invocações correspondente, podendo gerar ambos os grafos se for requerido pelo usuário.

Um programa Java é formado por várias classes cada uma das quais, normalmente, é armazenada em um arquivo separado. Portanto para realizar a análise todas as classes envolvidas no programa, e consequentemente todos os arquivos, devem ser submetidas ao DEPAnalyzer.

O modelo é organizado, internamente, em dois módulos, o módulo de coleta das informações e o módulo de análise. A figura 3.1 apresenta esta estrutura interna.

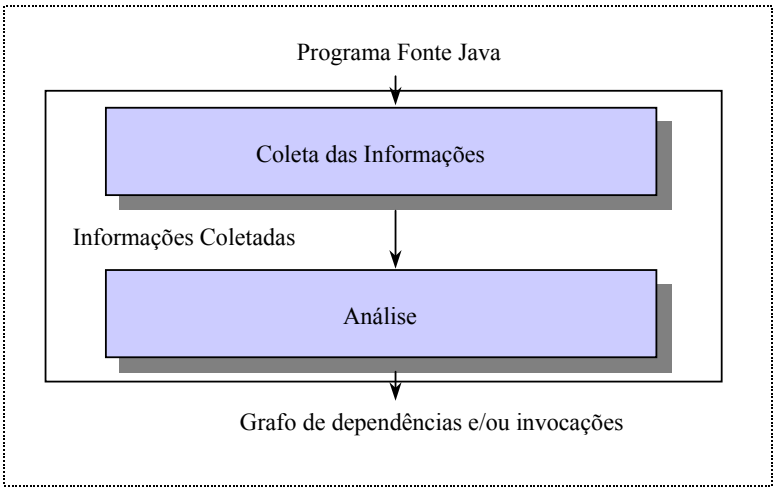


FIGURA 3.1 - Estrutura do DEPAnalyzer

O módulo de coleta das informações realiza a análise das classes do programa. Esta análise serve para prover o conhecimento sobre as classes declaradas no programa. Através destas informações pode-se realizar um mapeamento da hierarquia destas classes. A coleta das informações gera uma estrutura de dados, a qual retrata esta hierarquia juntamente com os elementos (atributos e métodos) declarados em cada classe. Nesta fase também é realizada a análise intraprocedural [DIW 96] dos métodos declarados. Isto serve para auxiliar o processo de propagação dos tipos durante a simulação da execução da aplicação. A análise interna também obtém informações que são depositadas na estrutura de dados gerada por este módulo.

O módulo de análise realiza a análise do fluxo das invocações com o intuito de detectar os relacionamentos entre as classes do programa. Como resultado desta análise tem-se o grafo das dependências entre as classes declaradas pelo programador e/ou o grafo de invocações entre estas classes. O usuário deve optar por obter ambos os grafos ou somente um dos dois, dependendo do propósito de aplicação das informações (alocação inicial, realocação ou ambos).

3.3 Coleta das Informações

No desenvolvimento de um programa orientado a objetos, as classes são abstrações dos componentes da aplicação existentes em tempo de execução. Um sistema é dividido em entidades que através de suas funcionalidades apresentam uma solução

para um determinado problema. Em uma linguagem tipada como Java, cada variável (atributo) recebe a indicação de qual é o tipo esperado. Em Java existem alguns tipos de dados primitivos da linguagem como *int* e *float*. No entanto, para modelar um sistema é necessária a definição de entidades mais complexas, as quais definem partes do sistema. Estas entidades mais complexas são as classes que também são consideradas como tipos. De fato, um programa Java somente vai existir se pelo menos uma classe for definida. Note que Java também possui um conjunto de classes primitivas que compõem a sua API (*application programming interface*) tais como *String* e *Vector* que estão disponíveis para o programador incluir nos seus programas.

O conhecimento das classes envolvidas na computação de um programa antes da simulação do fluxo das invocações é sugerido por [AGE 95]. Segundo Agesen, este conhecimento prévio facilita a análise de tipos em programas orientados a objetos. Além disso, muitos trabalhos de análise das invocações realizam a análise da hierarquia das classes ou análises derivadas desta como: a análise rápida de tipos (RTA) [BAC 96], para prover esta visão e assim auxiliar a análise de tipos e consequentemente a análise das invocações [AGR 99], [DEW 2001], [PRO 2001].

No entanto, como salientado em [SRE 2000], durante a execução de programas orientados a objetos classes ou métodos são carregados dinamicamente, o que torna difícil à análise estática de todas as classes envolvidas na computação. O fato do ambiente Java oferecer um conjunto de classes predefinidas faz com que as classes de um programa Java possam ser divididas em duas categorias: as classes/tipos primitivas (classes da biblioteca Java) e as classes/tipos definidas pelo programador. Considerando-se que para ocorrer a distribuição de um programa Java em uma arquitetura multiprocessada deve-se ter o ambiente Java, juntamente com sua biblioteca de classes, em cada nó de processamento da arquitetura. Isto torna os relacionamentos entre as classes primitivas e as classes definidas pelo programador relacionamentos locais.

Tendo em vista estes conceitos a coleta realiza a análise das classes definidas pelo programador a fim de obter uma visão prévia destas e assim detectar dependências entre estes conjuntos de objetos.

A análise estática de um programa necessita da utilização de um analisador léxico-sintático (*parser*). Este é composto de um analisador léxico e de um analisador sintático. O analisador léxico tem a função de ler os caracteres de um programa e produzir uma seqüência de *tokens*, a qual é utilizada pelo analisador sintático. A análise sintática verifica se a seqüência de *tokens* é gerada por alguma das regras de produção da gramática da linguagem fonte [PRI 2000].

O programa Java (todos os arquivos ".java" que compõem o programa) é recebido pelo módulo de coleta das informações. A leitura do código fonte (análise léxica) começa juntamente com a identificação das estruturas sintáticas (análise sintática). Quando a regra gramatical da declaração de uma classe é atingida a coleta das informações tem início.

As informações sobre o tipo que está sendo declarado são depositadas em uma estrutura de dados denominada *DataClass*. Esta ação é realizada porque além de representarem os novos tipos definidos pelo programador as classes são as entidades estáticas do programa, nas quais estão estabelecidos todos os possíveis relacionamentos ou dependências dos conjuntos de objetos (entidades dinâmicas).

Como pode-se perceber o código fonte é uma representação textual das ações que ocorrerão em tempo de execução. A figura 3.2 mostra a estrutura de dados utilizada para armazenar o resultado da coleta.

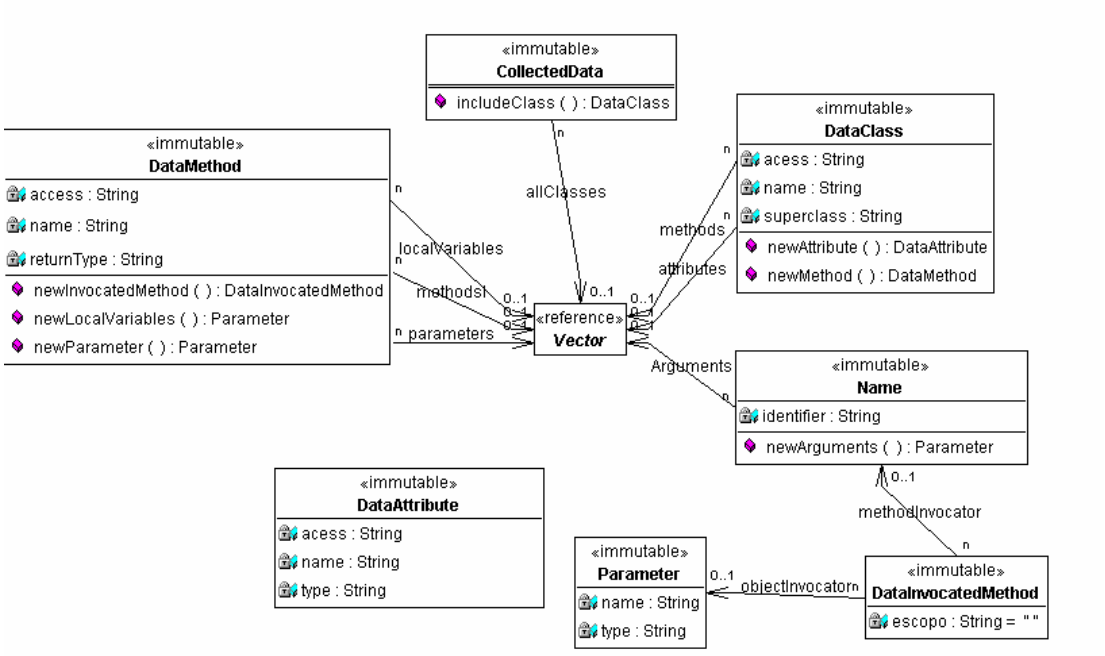


FIGURA 3.2 - Estrutura *CollectedData*

Cada classe definida pelo programador é um elemento da estrutura de dados *CollectedData*, ou seja, um objeto deste tipo. Os elementos desta estrutura contêm as informações *access, name*, *superclass*, *attributes* e *methods*.

O *access* refere-se a visão que esta classe tem perante as outras entidades da aplicação. Esta visão ou visibilidade refere-se aos quatro tipos possíveis de restrição de acesso ao qual classes, atributos e métodos estão sujeitos conforme é detalhado na descrição do atributo *attributes*. O *name* representa o identificador desta, ou seja, um novo tipo definido pelo programador. O atributo *superclass* refere-se ao identificador da superclasse, se esta existir. Quando a descendência não é explícita atribui-se a este campo a descendência implícita *Object*, da qual toda classe Java descende. A informação sobre as superclasses das classes definidas no programa possibilitam o mapeamento da hierarquia destas classes.

O atributo *attributes* é uma estrutura de dados onde cada elemento é a descrição de um atributo declarado na classe. Cada elemento desta estrutura contém as seguintes informações: *access*, *type* e *name.* O *access* refere-se ao tipo de visão que este atributo terá perante as outras entidades (classes) do programa. Em Java pode-se atribuir quatro tipos de acessos, o acesso público (*public*), o acesso privado (*private*), o acesso protegido (*protected*) e o acesso sem especificação (*package*). O acesso público é quando os atributos e métodos são acessíveis por todos os métodos em todas as classes. O acesso privado é quando os atributo e métodos são acessíveis somente nos métodos da própria classe. O acesso protegido é quando os atributos e métodos são acessíveis aos métodos da própria classe e das subclasses desta. O acesso sem especificação é quando os atributos e métodos são acessíveis somente nos métodos das classes que pertencem ao mesmo pacote (*package*). Um pacote é definido pela palavra reservada *package* no início do código e serve para indicar um grupo de classes relacionadas.

O *type* representa o tipo da variável declarada. Como Java é uma linguagem tipada as variáveis de um programa possuem necessariamente um tipo vinculado a sua declaração. O *name* representa o identificador da variável declarada.

O atributo *methods* é uma estrutura de dados onde cada elemento é uma descrição de um método da classe declarada. Cada elemento da estrutura de dados m*étodos* contém as seguintes informações: *acess*, *name*, *returnType, parameters*, *localVariables* e *methodsInvocated*.

O *access* representa o tipo de visão que as outras entidades (classes) do programa analisado terão deste método. O *name* representa o identificador do método declarado. O *returnType* representa o tipo/classe da variável retornada pelo método. Cada elemento da estrutura *localVariables* é uma variável local declarada no método analisado, o qual contém as informações *name* e *type*. Cada elemento da estrutura de dados *parameters* é um dos parâmetros recebidos pelo método, sendo que estes elementos possuem como informações *name* e *type*. A estrutura de dados *methodsInvocated* é composta pelos métodos invocados dentro do método declarado. Cada elemento desta estrutura possui as seguintes informações: *objectInvocator*, *methodInvocated* e *escopo*. O *objectInvocator*, representa a instância que invocou o método dentro do corpo do método declarado, esta informação contém o nome e o tipo do objeto invocador. O *methodInvocated* representa a assinatura do método invocado, ou seja, (1) o identificador do método juntamente com, (2) os argumentos formados por nome e tipo, recebidos por este método e em qual ordem, (3) o *escopo*, que representa o contexto da invocação que está sendo detectada, isto é, o escopo refere-se à presença de contextos interativos. Por contexto interativo entende-se que invocação é disparada dentro do escopo de um comando interativo (laço).

A fim de ilustrar as informações coletadas considera-se o programa *ThesisNeurosis*, o qual foi desenvolvido por Mangan [MAN 2001]. Esta aplicação encontra-se disponível em: http://www.cos.ufrj.br/~mangan/thesisneurosis/ThesisNeurosisApplet.html e possui a hierarquia de classes apresentada na figura 3.3.

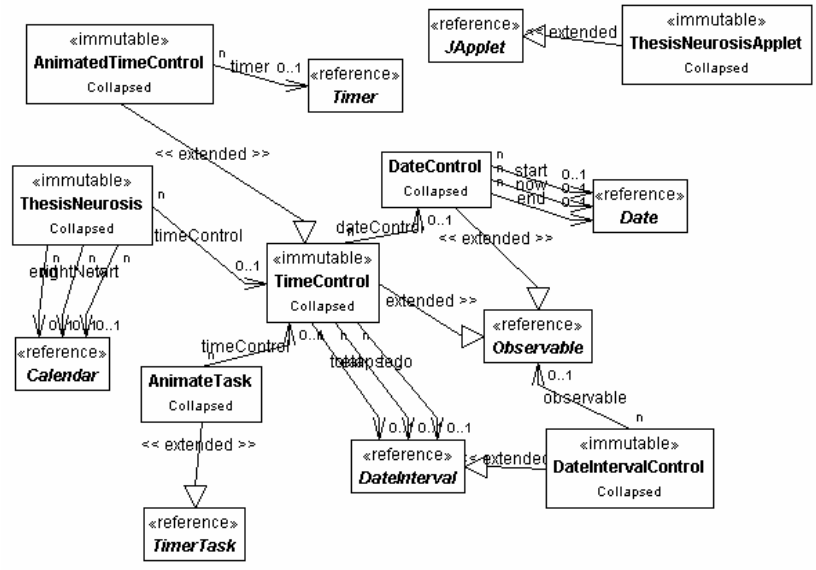


FIGURA 3.3 - Hierarquia de Classes Definidas na Aplicação *ThesisNeurosis*

As classes *Observable*, *JApplet*, *DateInterval* e *TimerTask* não pertencem ao conjunto de classes submetidas ao modelo, isto é, são classe que pertencem à API Java. A classe principal desta aplicação é a *ThesisNeurosis*.

Diante desta hierarquia a figura 3.4 mostra as informações coletadas, para a classe *ThesisNeurosis*, referentes a descendência, atributos e assinaturas dos métodos declarados por esta. Todas as demais classes definidas pelo programador também são analisadas e informações análogas são coletadas.

```
• ThesisNeurosis
        o superclass: Object
        o attributes:
                ▪ [(acess: protected, type: Calendar, name: rightNow), (acess:
                  protected, type: Calendar, name: start), (acess: protected,
                  type: Calendar, name: end), (acess: protected, type:
                  TimeControl, name: timeControl)]
        o methods
                ▪ name: getTimeControl, acess: public, returnType: TimeControl,
                  parameters: sem parâmetros
                ▪ name: setup, acess: public, returnType: void, parameters: sem
                  parâmetros
                ▪ name: setupDates, acess: public, returnType: void, parameters: sem
                  parâmetros
                ▪ name: setupTimeControl, acess: public, returntype: void,
                  parameters: sem parâmetros
                ▪ name: main, acess: public static, returnType: void, parameters:
                  [(type: String, name: args)]
```

FIGURA 3.4 - Informações do objeto *DataClass* referente a classe *ThesisNeurosis*

3.3.1 Análise de Tipos Intraprocedural

Para se identificar o tipo do *objectInvocator* e o tipo dos argumentos recebidos por um método invocado é necessário a realização de uma análise dos tipos das variáveis locais do método declarado. Está análise é feita durante a coleta das informações sobre os métodos invocados (*methodsI*). Então quando um método é declarado sabe-se, através da coleta das informações, os tipos e os respectivos identificadores dos parâmetros formais recebidos pelo método. Além disso, as variáveis declaradas no corpo deste método, suas respectivas instanciações e os métodos invocados. A ação de instanciação (invocação do método construtor) de uma variável/objeto no corpo de um método desencadeia a criação de uma estrutura de dados, a qual irá refletir as variáveis instanciadas no método e os seus respectivos tipos. Esta tarefa pode ser vista como uma análise intraprocedural. Com o intuito de elucidar a análise interna de cada método, considera-se o programa apresentado na figura 3.5.

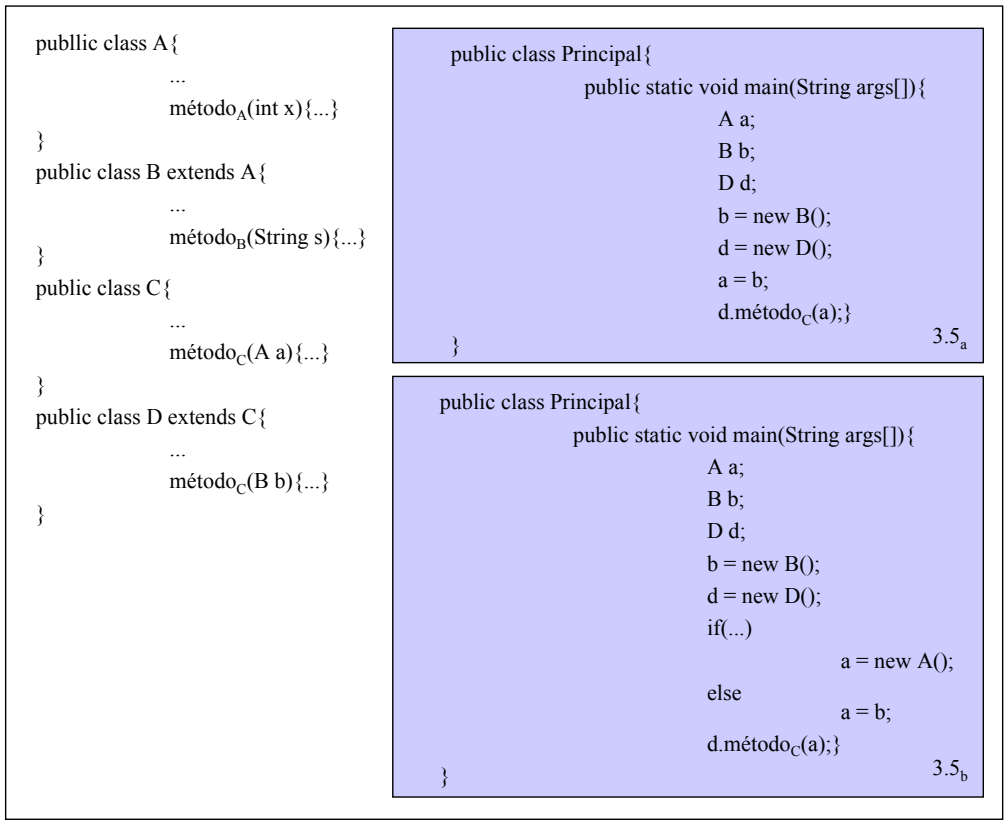


FIGURA 3.5 - Programa Exemplo da Análise Interna dos Métodos

O método *main*, da figura 3.5$_a$, possui como parâmetro formal um *String*. Este é o método padrão de início de execução de um programa do tipo aplicação em Java. No corpo do método obtém-se a informação sobre a instanciação da variável *b*, a qual é do tipo *B*. Esta informação é obtida pela análise através da identificação da invocação do método construtor da classe *B*. Após esta ação obtém-se a informação sobre a instanciação da variável *d*, a qual é da classe/tipo *D*. Na seqüência encontra-se, através da leitura do código, a ação de atribuição *a=b*, a qual reflete o fato da variável *a* passar a conter o mesmo dado contido na variável *b*, ou seja, uma instância da classe *B*. Então na invocação do método *método$_C$()*, detecta-se que o tipo da instância que invoca este método, ou seja, *objectInvocator* é da classe *D*, e a versão do método que está sendo invocada é a versão declarada na classe *D*, pois recebe como argumento uma instância da classe *B* (*a*).

Nota-se que para realizar a análise intraprocedural dos métodos declarados deve-se ter um controle dos fluxos dos dados (tipos dos dados) durante a leitura do código fonte. Para tal é criada uma instância da classe *TypeAnalysis*, apresentada na figura 3.6, para cada método declarado em cada classe.

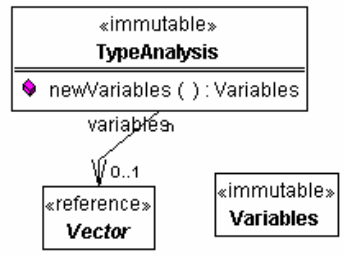


FIGURA 3.6 - Estrutura de Dados da Análise de Tipos Intraprocedural

Ao detectar a estrutura sintática de declaração de um método, ocorrem as seguintes ações:

- os parâmetros formais (nome e tipo) do método declarado são colocados na estrutura *TypeAnalysis*;
- as variáveis locais ao método declarado também são colocadas na estrutura *TypeAnalysis*;
- fluxo dos dados é acompanhado pela análise de tipos. Então diante de um comando de atribuição (x=y) é propagado o tipo da variável *y* para o tipo da variável *x*. Isto reflete na entrada da estrutura *TypeAnalysis*, a qual refere-se a variável *x*;

Como percebe-se a análise de tipos acompanha o fluxo interno do método declarado sem propagar o fluxo para os métodos invocados no método declarado. Então, o campo *nome*, o qual refere-se a assinatura do método invocado no método declarado possui a informação sobre o identificador do método invocado seus argumentos, em ordem, e os possíveis tipos destes. Para ilustrar este campo considera-se o programa da figura 3.5_a e o programa da figura 3.5_b. Na invocação do método $método_C()$ na figura 3.5_a, tem-se a seguinte informação sobre os tipos do *objectInvocator* e sobre os argumentos recebidos por este:

- *objectInvocator*: identificador: *d*, tipo: *D*;
- *identifier*: identificador do método: $método_C$, argumentos: *name*: *a*, *type*: *B*.

Já a mesma informação para o programa da figura 3.5_b é:

- *objectInvocator*: identificador: *d*, tipo: *D*;
- *identifier*: identificador do método: $método_C$, argumentos: *name*: *a*, *type*: *{A, B}*.

A indefinição sobre o tipo do argumento *a*, vem do fato desta invocação depender do resultado do comando condicional que a antecede, tornando, a nível estático, necessário considerar os dois ramos do comando condicional. Então a informação guardada na estrutura *TypeAnalysis*, pode ser um conjunto, como resulta a análise da figura 3.5_b;

Outro comando analisado pela análise de tipos é o comando de instanciação *new*. Diante deste é inicializado um elemento da estrutura *TypeAnalysis* com o nome da instância criada e o tipo desta.

Em cada invocação de método o(s) tipo(s) do elemento que invoca (*objectInvocator*) e dos argumentos passados são obtidos através da procura destas entidades na estrutura *TypeAnalysis* gerada durante a leitura do método que está sendo declarado. A procura é feita levando-se em conta a última definição da variável com o identificador referenciado. Isto se deve ao fato de um método poder redefinir as variáveis no decorrer de seu código.

Além disso, se não existe a presença de uma variável invocadora ou esta é a palavra reservada *this,* deve-se colocar o tipo que está sendo analisado como tipo do *objectInvocator* e o nome recebe *this.* O mesmo procedimento deve ser efetuado quando encontra-se diante de uma invocação realizada pela palavra reservada *super*.

Quando se tem uma variável referenciada pela palavra *this* (*this.a*) deve-se procurar por esta nos atributos declarados na classe analisada. Se esta não se encontrar neste contexto deve-se atribuir o nome de referência desta ao atributo *nome* e ao atributo *type* do campo *objectInvocator*, ou seja, *this.nome_da_variável*. Esta situação também deve ser adotada quando existe a presença da palavra super. Nota-se que o

atributo *type* não recebe um tipo e sim um nome de variável antecedido da palavra reservada *this* ou *super*. Posteriormente o sistema de análise buscará o tipo referente a esta variável na hierarquia de classes analisadas

Além disso, quando uma variável local ao método analisado recebe uma variável referenciada por *this* (*a = this.cor*) e esta última não se encontra no escopo dos atributos declarados na classe deve-se colocar a palavra reservada *this* juntamente com nome da variável no campo tipo da estrutura *TypeAnalysis*. Este comportamento reflete na estrutura *DataClass* se a variável *a* invocar algum método na seqüência do fluxo do método analisado, o tipo referente a variável que realiza a invocação conterá a informação *this.cor*.

Este problema provem do fato da análise intraprocedural ocorre juntamente com a obtenção das informações sobre as classes em uma única leitura do código fonte. Isto impede que no momento da análise interna dos métodos se tenha acesso a informações das outras classes presentes na hierarquia de tipos analisados. Já o processamento do módulo de análise possui o mapeamento completo das informações sobre as classes analisadas, podendo assim identificar o tipo do objeto que realiza esta invocação se este se encontrar presente nestas classes.

3.3.2 Análise de Dependências sem Simulação

Pode-se obter o grafo de dependências a partir das informações inferidas na estrutura de dados *DataClass* sem tomar nenhum método como ponto de partida para a simulação da execução do programa. No entanto, este grafo é altamente conservativo, pois leva em consideração apenas informações estáticas.

Para o escalonamento informações conservativas podem levar a situações desfavoráveis à distribuição. As informações de dependências obtidas somente em nível estático, ou seja, em nível sintático apresentam, muitas vezes, dependências que não ocorrem em determinada implementação. Por exemplo, considera-se o código da figura 3.7.

```
public class A{
        public void mA(){}
}
public class B{
        public void m( ){
                ...
                A a = new A();}
}
public class K extends B{
        public void m(){...}
}
```

```
public class Principal{
        public static void main (String args []){
                B b = new B();
                b.m();}
}
```

FIGURA 3.7 - Programa Exemplo

Através das informações das classes a dependência entre a classe *K* e a classe *A*, também seria verificada, pois somente as informações estáticas estariam sendo levadas em consideração. Assim todas as dependências estabelecidas com uma superclasse, também, são estabelecidas com a subclasse.

No entanto, se considerarmos o fluxo de execução do programa, o qual é expressado através do método principal, pode-se refinar esta análise considerando-se somente o que realmente causa dependência durante o fluxo. A análise de fluxo também é conservativa, pois considera, em um ramo condicional, os dois caminhos verdadeiros.

Porém, a fato de basear-se na simulação do fluxo de execução, que é uma característica dinâmica, pode levar à inferência de informações mais precisas.

Por este motivo é realizada a análise de dependências, a qual leva em consideração características dinâmicas como o fluxo de execução do programa. Isto torna o resultado da análise menos conservativa levando, em muitos casos, a uma maior possibilidade de exploração do escalonamento.

3.4 Análise

A análise de dependências visa gerar um grafo representando o relacionamento entre os conjuntos de objetos de um programa. Para isso a análise pode gerar o grafo de dependências e/ou o grafo de invocações. O grafo de dependências é uma versão resumida dos relacionamentos entre as classes, sem a especificação de quem, ou seja, qual método dispara este relacionamento. No entanto este pode ser utilizado em um processo de alocação inicial visando assim reunir em um mesmo nó conjuntos de objetos que estabelecem um maior grau de dependências. Além disso, a análise pode disponibilizar o grafo das invocações ocorridas na simulação do programa. Este último prove uma visão mais detalhada dos relacionamentos uma vez que proporciona a informação de qual método dispara o relacionamento. O grafo de invocações, também, permite o conhecimento de quando estes relacionamentos irão se estabelecer ao longo do processamento da aplicação. Os dois grafos são gerados pelo mesmo processo de análise, não implicando assim um *overhead* (custo adicional) decorrente de uma análise distinta para os dois grafos. A figura 3.8 apresenta a estrutura interna deste módulo.

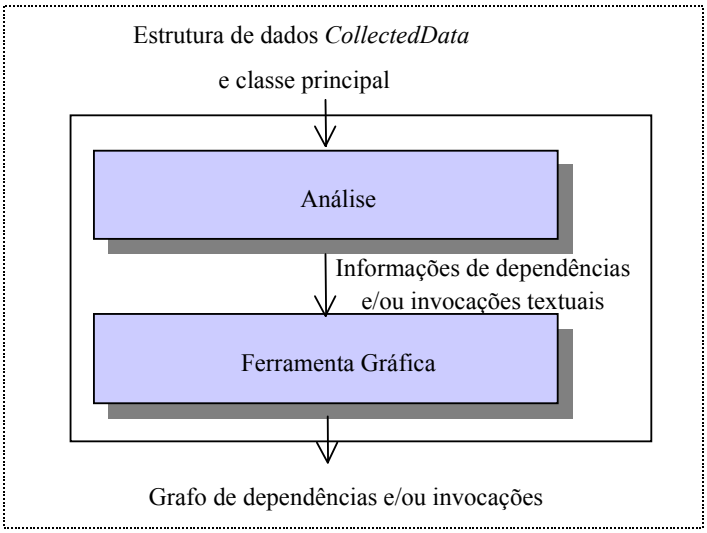


FIGURA 3.8 - Estrutura Interna do Módulo de Análise

O módulo de análise de dependências recebe as informações coletadas na estrutura de dados *CollectedData* e o nome referente a classe principal da aplicação analisada. A análise tem início na leitura dos métodos invocados (*methodsI*) no método principal. O método principal é responsável por desencadear o fluxo de execução da aplicação. Durante a análise do programa é gerado um arquivo contendo as informações solicitadas, ou seja, de dependências e/ou invocações em um formato textual. Este arquivo é recebido por uma ferramenta gráfica, a qual converte as informações em formato textual para informações em formato gráfico. A visualização gráfica dos grafos

pode auxiliar no seu entendimento e consequentemente no processo de extração das informações apresentadas por estes.

Como citado anteriormente um programa Java é composto por varias classes, as quais geralmente encontram-se em arquivos independentes. Além disso, cada classe pode possuir um método principal, o qual desencadeia o seu fluxo de execução. Perante estas características é necessário que o usuário informe ao sistema qual classe representa a classe principal, para que a análise tenha início.

Como exemplo das informações visualizadas pelo processo de análise considera-se as duas primeiras invocações do método *main* do programa *ThesisNeurosis*, o qual tem sua estrutura hierárquica de classes apresentada na figura 3.3 A figura 3.9 apresenta o resultado desta análise.

```
·   localVariables
    ·   type: ThesisNeurosis, name: tn;
    ·   type: JTimeControlPanel, name:timeControlPanel1;
    ·   type: JdateControlPanel, name:dateControlPanel;
    ·   type: Jframe, name: frame;
    ·   type: Dimension , name:d.
·   methodsInvocated
    ·   objectInvocator: método construtor,
        ·   name:
            ·   identifier: ThesisNeurosis,
            ·   arguments: sem argumentos.
    ·   scope: sem escopo
    ·   objectInvocator: ThesisNeurosis (tipo), tn(nome),
        ·    name:
            ·   identifier: setup,
            ·   arguments: sem argumentos.
     ·  scope: sem escopo
```

FIGURA 3.9 - Análise Interna do Método *main* da Classe *ThesisNeurosis*

O funcionamento da análise baseia-se no fato que em uma invocação de método existe um método invocador (ou que causa a invocação) e um método invocado. A análise utiliza este conceito para simular o fluxo de execução do programa.

Considerando-se o fluxo de invocações apresentado na figura 3.9, no momento inicial o método invocador é o método *main* da classe *ThesisNeurosis* (*ThesisNeurosis::main*) e o primeiro método invocado é o método *ThesisNeurosis*, isto é, o construtor, o qual após a invocação passa a ser o método invocador. Como se pode perceber após a invocação de um método, este assume a condição de invocador, pois o fluxo de execução passa para o seu interior. O fluxo de execução retorna ao contexto anterior somente depois de realizar todas as ações descritas no corpo (código) do método invocado.

3.4.1 Grafo de Dependências

No grafo de dependências gerado pela análise os nodos representam as classes declaradas no programa, as setas (arestas) representam as dependências entre a classe representada pelo nodo origem em relação à classe destino. As dependências podem ser classificadas conforme mostra a tabela 3.1.

TABELA 3.1 - Classificação das Dependências

Sem dependência	$\perp$
Dependência	n°
Dependência indeterminada	$\top$

O símbolo $\perp$ representa a ausência de dependência. O *n*° representa a quantidade de vezes que determinadas classes estabelecem dependência. Estas dependências são livres de qualquer escopo, ou seja, independente de laços. O símbolo $\top$ representa a dependência indeterminada, ou seja, a dependência estabelecida no contexto de um ou mais laço(s).

A quantidade de vezes que se estabelece determinada dependência é quantificada por um número seqüencial, o qual declara este grau de relacionamento entre as mesmas. Uma classe *A*, hipotética, e uma classe *B*, também hipotética, podem estabelecer quatro dependências, as quais são estabelecidas fora de qualquer contexto interativo. O modelo também oferece a possibilidade do usuário estabelecer um limite, no qual uma determinada quantidade de dependências torne-se um relacionamento de dependência indeterminada. Isto quer dizer que, levando-se em conta, um parâmetro dado pelo usuário o DEPAnalyzer pode quantificar um determinado número de dependências que estão fora de qualquer contexto e a partir de determinada quantidade tornar este relacionamento como um relacionamento indeterminado.

As dependências indeterminadas evidenciam o fato da indefinição sobre a quantidade de vezes que a comunicação irá ocorrer. No auxílio ao escalonamento, o fato de não se saber quantas vezes irá ocorrer determinada comunicação/dependência é um impasse para a tomada de decisões. O modelo DEPAnalyzer sugere que quando isto ocorrer o escalonador assuma a abordagem do pior caso, na qual é baseado. Está abordagem parte do pressuposto que estas comunicações/dependências irão acontecer um número máximo de vezes, ou seja, um valor *upper bound*, o que representaria para o escalonamento que estes conjuntos de objetos não devem se encontrar em nós distintos de processamento. A separação destes conjuntos causaria um número desconhecido de comunicações remotas o que tende a ser muito custoso e consequentemente degradar o desempenho do sistema como um todo.

Inicialmente cria-se o nodo referente à classe do método principal. Diante de uma invocação no método principal, a análise de dependências detecta o escopo desta, ou seja, se esta ocorreu dentro de um comando interativo (laço) ou não. Após isto é verificado o tipo/classe do elemento que está realizando a invocação. Este elemento pode ser uma classe, o que representa uma invocação de classe ou pode ser uma instância (variável/objeto).

Além disso, como citado na seção 3.3.1 o tipo desta variável pode conter o valor *this.** ou *super.**. Quando isto ocorrer deve-se buscar na hierarquia de classe referente a classe do método invocador qual é o tipo da variável, uma vez que está foi definida em alguma superclasse. Na primeira ocorrência desta variável deve-se atribuir o seu tipo ao campo tipo e seguir com o processo de análise. Ressalta-se que esta busca é feita a partir da classe do método invocado em direção a superclasse desta e assim por diante. Este processo é necessário pois a detecção sobre o tipo desta variável só pode ocorrer depois que todas as classes analisadas forem conhecidas pelo sistema, ou seja, durante o processo de análise.

Depois de identificar o tipo do método invocado deve-se comparar este com o tipo do método invocador. Caso exista uma indefinição no tipo do elemento que causa a invocação, os possíveis valores que este pode assumir são considerados de forma

conservativa. Se os tipos forem iguais não se estabelece nenhuma dependência, ou seja, não é criado nenhum nodo e nenhuma seta no grafo de dependências.

Quando os tipos diferem o tipo da variável que invoca o método é comparado com os tipos definidos pelo programador. Este processo é realizado para se verificar se esta dependência é entre a classe do método invocador e alguma das classes primitivas dos pacotes Java, ou seja, se esta dependência é entre duas classes declaradas no programa. Esta comparação descarta as dependências entre as classes definidas no programa e as classes primitivas, pois estas são consideradas comunicações locais.

Quando o tipo da variável é um tipo definido pelo programador se estabelece uma dependência entre a classe do elemento que realiza a invocação e a classe do método invocador. Neste caso, é criado o nodo correspondente a classe do elemento que invoca o método, se este ainda não existir, e a seta que representa a dependência entre as classes.

Depois de verificar o tipo da variável que realiza a invocação é necessário analisar os tipos dos argumentos recebidos pelo método invocado e a ordem destes. Isto se deve a presença do polimorfismo tipo sobrecarga, o qual permite que existam dois métodos com o mesmo nome mas com assinaturas diferentes. Uma vez identificado os tipos dos argumentos e sua respectiva ordenação procura-se na hierarquia de classes da própria classe da variável invocadora a declaração deste método. Neste ponto deve-se levar em conta que em alguns casos as informações de tipos podem ser indeterminadas e obviamente estas devem ser tratadas conservativamente. Então, se existir mais de uma versão (implementação) do método invocado que possa responder a determinada invocação estas são consideradas como se acontecessem em tempo de execução. No entanto sabe-se que durante a execução somente uma das versões do método invocado irá responder a esta invocação.

Tendo-se encontrado a(s) definição(ões) para o método invocado dentro da hierarquia declarada no programa, percorrer-se o fluxo de execução deste método. Então o fluxo de controle passa para dentro do corpo do método invocado e neste são verificados as novas invocações de métodos.

Caso a definição do método não esteja presente na hierarquia de classes definidas, isto representa um método definido no escopo das classes primitivas. Impossibilitando assim do fluxo ser percorrido através da análise estática. Esta situação detecta pontos de abertura [SRE 2000], ou seja, neste contexto pontos de abertura são pontos em que as classes do programa se comunicam com a biblioteca do ambiente Java.

O escopo de invocação do método é propagado juntamente com o fluxo de execução para dentro do corpo do método invocado. É importante salientar que inicialmente, antes da primeira invocação do método principal, não existem dependências. Após a primeira invocação, mesmo que está não cause dependência, se o escopo desta for iterativo esta informação é propagada para as invocações decorrentes da primeira invocação. Isto quer dizer que sendo $D = \{\bot, n^{\circ}, \top\}$, o domínio abstrato [COU 76] das dependências e sendo um elemento deste conjunto, ou seja, $K \in D$, $K \wedge \top = \top$, $K \wedge \bot = K$. Isto que dizer que a ordem de crescente de precedência deste domínio abstrato é a própria ordem em que este foi apresentado, ou seja, $\bot$, n°, $\top$.

Se algum dos tipos dos argumentos do método invocado possui um tipo diferente do tipo da variável que o invoca, compara-se novamente este tipo com os tipos definidos pelo programador. Como pode-se perceber as formas de relacionamento entre as entidades de um programa estão expressas nas estruturas de comportamento

(métodos). Em um programa orientado a objetos existem três formas de se estabelecer uma dependência:

- quando uma variável ou método de classe é invocado por um método de outra classe;
- quando um de método de instância é invocado por um método de uma instância de outra classe;
- quando um objeto é passado como argumento em uma invocação de método.

As dependências relacionadas com os argumentos recebidos pelo método que está sendo invocado são entre o tipo (classe) do argumento e a classe do método invocado. Então na detecção de argumentos com: tipo diferente ao tipo do método invocado, e igual a algum dos tipos definidos pelo programador, é estabelecida uma dependência. Para esta é criado um nodo, representando o tipo do argumento (se este não existir), e uma aresta, representando a dependência entre o tipo do método invocado e o tipo do argumento. Outro aspecto relevante da análise é a detecção de invocações recursivas, as quais produzem dependências indeterminadas.

Para exemplificar o funcionamento do processo de análise descrito, considera-se novamente, o programa *ThesisNeurosis*, mais precisamente o fluxo desencadeado pela segunda invocação disparada pelo método *main*.

O segundo método invocado é o método *setup*, o qual é invocado pelo objeto *tn* do tipo *ThesisNeurosis*. Este método não recebe nenhum argumento de entrada, verificando assim nenhuma dependência no momento da sua chamada. Ao passar para dentro do fluxo deste método detecta-se que este invoca dois métodos da própria classe principal, os quais são: *setupDates* e *setupTimeControl*.

O método *setupDates* não recebe nenhum argumento de entrada, verificando a inexistência de relacionamentos (dependências) quando da sua chamada. Este método desencadeia um fluxo de invocações de métodos da classe *Calendar*, a qual não esta presente no conjunto de classe submetidas ao modelo. Isto inviabiliza o modelo de percorrer o fluxo desencadeado por este método. Além disso, o fato da classe *Calendar* não pertencer ao conjunto de classes acessíveis pelo DEPAnalyzer verifica a ausência de dependências por parte do fluxo do método *setupDates*.

O método *setupTimeControl* desencadeia o seguinte fluxo de invocações: *AnimatedTimeControl* e *setNow*. O método *AnimatedTimeControl* corresponde ao método construtor desta classe, a qual faz parte do conjunto de classes submetido ao modelo. Isto verifica um relacionamento entre a classe principal e a classe *AnimatedTimeControl*, o qual é apresentado na figura 3.10. Além disso, esta invocação recebe dois argumentos como parâmetros de entrada, isto distingue qual dos dois métodos *AnimatedTimeControl* está sendo invocado.

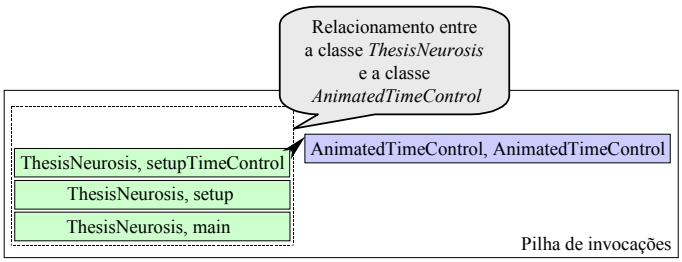


FIGURA 3.10 - Relacionamento entre a Classe *ThesisNeurosis* e a Classe *AnimatedTimeControl*

Sabendo-se qual método está sendo invocado e tendo-se acesso a este, pode-se simular o fluxo interno deste método, o qual desencadeia a seguinte seqüência de invocações: *super* e *animate*. A palavra reservada *super* refere-se à invocação de um método declarado na superclasse da classe do método que o invoca. Neste caso, mais precisamente, trata-se da invocação do método construtor da classe *TimeControl*, a qual é superclasse da classe *AnimatedTimeControl*, como pode ser verificado pela figura 3.3. O método construtor recebe como parâmetros duas variáveis do tipo *Date*, fato que distingue qual versão deste método esta sendo invocada. O fluxo de invocação desta versão do método construtor invoca a outra versão do mesmo, a qual recebe como argumentos três variáveis do tipo *Date*.

O fluxo de invocações deste último método citado é: *DateControl*, *DateInterval*, *DateInterval* e *DateInterval*. A primeira invocação corresponde à chamada do método construtor da classe *DateControl*, a qual faz parte do conjunto de classes analisadas. Esta estabelece um relacionamento, apresentado na figura 3.11, entre a classe *AnimatedTimeControl* e a classe *DateControl*. O fluxo de invocações desencadeado por esta é nulo, pois este método realiza somente atribuições.

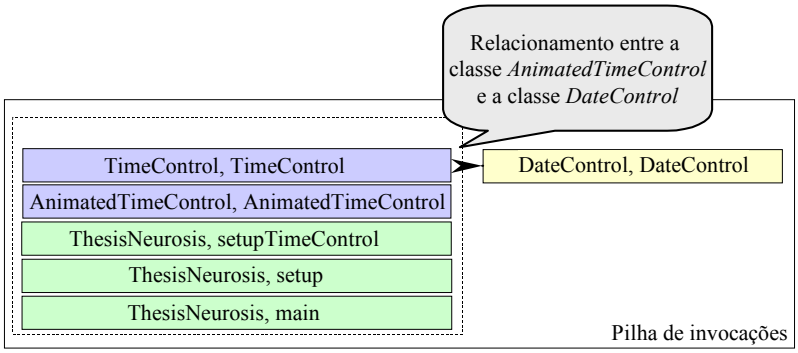


FIGURA 3.11 - Relacionamento entre a Classe *AnimatedTimeControl* e a Classe *DateControl*

As outras três invocações desencadeadas pelo método *TimeControl* referem-se à chamada do método construtor da classe *DateInterval*, a qual não faz parte do conjunto de classes analisadas.

O segundo método invocado por *AnimatedTimeControl*, ou seja, *animate* pertence à própria classe *AnimatedTimeControl* e desencadeia o seguinte fluxo de invocações: *java.util.Timer*, *AnimatedTask*, *setTimeControl* e *schedule*. A invocação *java.util.Timer* refere-se a um método da biblioteca Java conferindo assim um relacionamento local. A invocação *AnimatedTask* refere-se ao método construtor da classe com o mesmo nome, a qual pertence ao conjunto de classes submetido ao sistema. Esta invocação estabelece um relacionamento entre a classe *AnimatedTimeControl* e a classe *AnimatedTask*, representado pela figura 3.12. No entanto, o método *AnimatedTask* não encontra-se presente entre os métodos definidos nesta classe, identificando que esta invocação chama o método construtor da classe primitiva *Object*, inviabilizando assim o DEPAnalyzer de percorrer o seu fluxo.

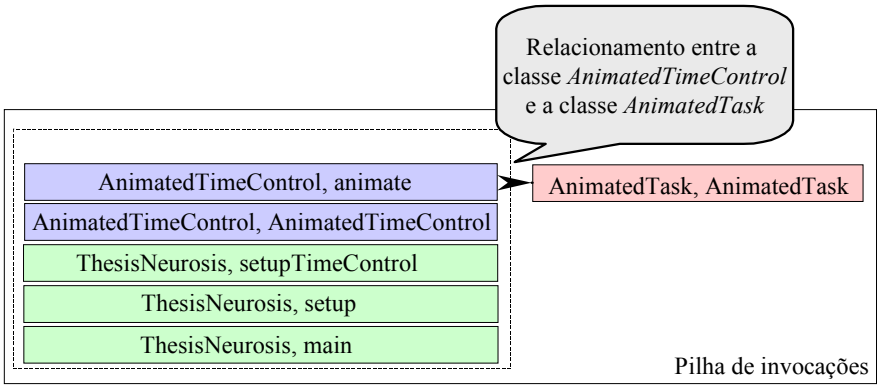


FIGURA 3.12 - Relacionamento entre a Classe *AnimatedTimeControl* e a Classe *AnimatedTask*

Sendo assim o fluxo de controle passa para o penúltimo método invocado em *animate*, ou seja, *setTimeControl* da classe *Animatedtask*. Isto estabelece outro relacionamento entre a classe *AnimatedTimeControl* e a classe *AnimatedTask*. Além disso, esta invocação recebe como argumento de entrada um objeto do tipo *TimeControl*, estabelecendo assim um relacionamento entre a classe *AnimatedTask* e a classe *TimeControl*. Outra característica é que este método não realiza nenhuma invocação. Estes relacionamentos são expressos através da figura 3.13.

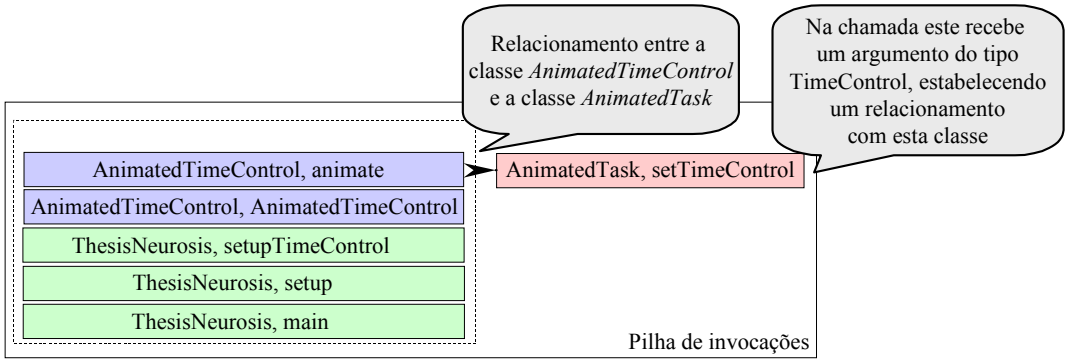


FIGURA 3.13 - Relacionamentos entre a Classe *AnimatedTimeControl* e a Classe *AnimatedTask* e entre a Classe *AnimatedTask* e a Classe *TimeControl*

O último método invocado por *animate* é da classe *Timer*, a qual não faz parte do conjunto de classes analisadas. Após simular o fluxo de invocações do método *animate* retorna-se ao fluxo desencadeado por *AnimatedTimeControl*. O retorno verifica que o fluxo deste chegou ao fim, o que faz o fluxo de controle retornar ao método *setupTimeControl*. Este dispara mais uma invocação, a qual refere-se ao método *setNow*, da classe *AnimatedTimeControl*, sem parâmetros de entrada, o qual expressa um relacionamento entre a classe *ThesisNeurosis* e a classe *AnimatedTimeControl* apresentado na figura 3.14. Este invoca um método da mesma classe e com o mesmo identificador, porém, o qual recebe um argumento, do tipo *Date*, como parâmetro de entrada.

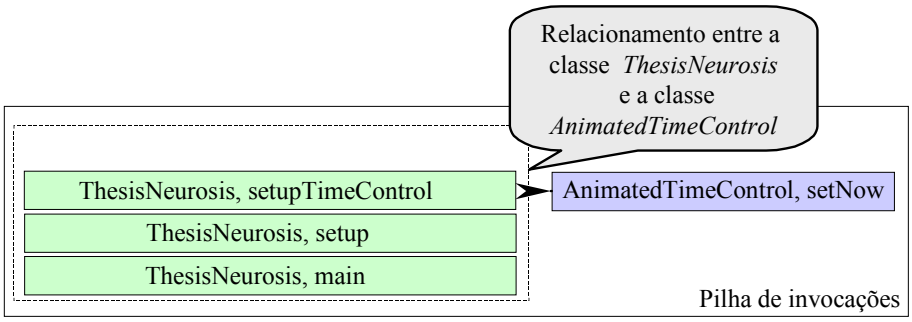


FIGURA 3.14 - Relacionamento entre a Classe *ThesisNeurosis* e a Classe *AnimatedTimeControl*

Primeiramente este método desencadeia duas invocações não acessíveis pelo modelo, pois representam chamadas a métodos declarados em classes não submetidas ao DEPAnalyzer. Posteriormente invoca os métodos *setNow* da classe *DateControl* e *update* (da própria classe *AnimatedTimeControl,* ou mais especificamente da superclasse desta *TimeControl*). O primeiro da origem a um fluxo inacessível pelo analisador, por referir-se a métodos de classes que estão fora do escopo submetido ao modelo, no entanto este estabelece um relacionamento entre a classe *AnimatedTimeControl* e a classe *DateControl*, o qual é mostrado na figura 3.15.

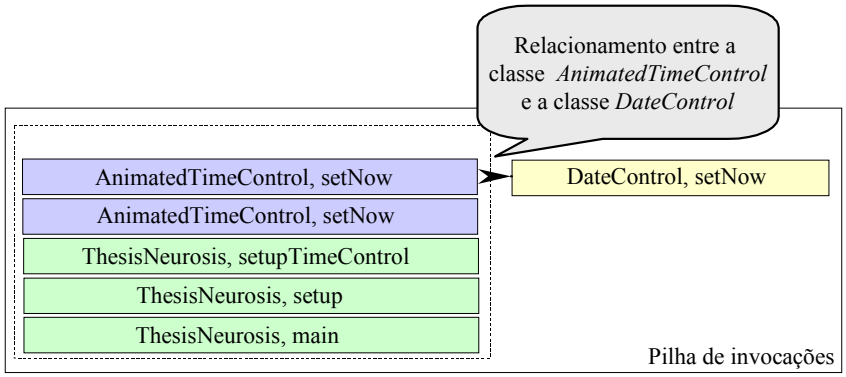


FIGURA 3.15 - Relacionamento entre a Classe *AnimatedTimeControl* e a Classe *DateControl*

A invocação do método *update*, herdado pela classe *AnimatedTimeControl*, dispara a seguinte seqüência de invocações: *getStart*, *getEnd*, *getNow*, expressado pela figura 3.16, os quais são da classe *DateControl*, estabelecendo assim três relacionamentos entre estas classes. As invocações subsequentes do método *update* referem-se a métodos inacessíveis pela análise.

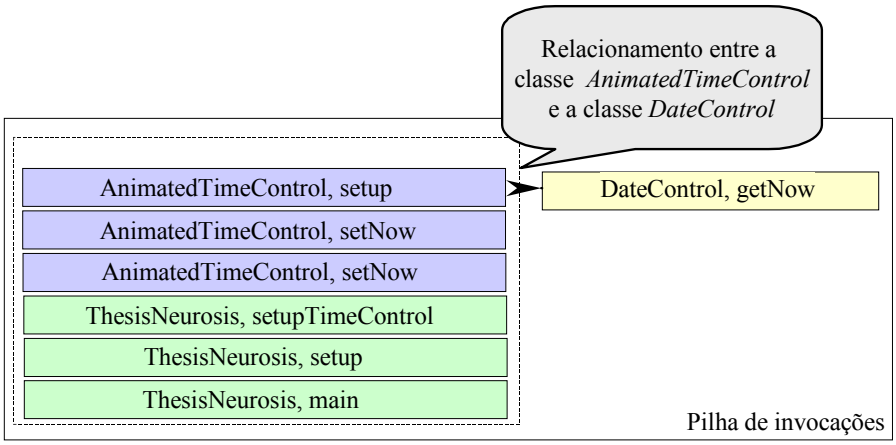


FIGURA 3.16 - Relacionamento entre a Classe *AnimatedTimeControl* e a Classe *DateControl*

Após isso o fluxo de invocações do método *setupTimeControl* chega ao final, propiciando o retorno do fluxo de controle ao método *setup*. Este, por sua vez, também verifica o final do seu fluxo, retornando para o método principal.

O grafo de dependências é representado textualmente por uma matriz de adjacência [TEN 95], a qual contém o mesmo número de colunas e de linhas, ou seja, o número de classes definidas (C_D) na aplicação. Esta configuração facilita a visualização, mesmo sendo textual, das informações de dependências.

Para o programa *ThesisNeurosis* o grafo de dependências, gerado a partir da segunda invocação, é mostrado na figura 3.17 juntamente com a matriz textual correspondente, a qual é apresentada na figura 3.18.

Nota-se que as invocações ocorridas não estão vinculadas a nenhum escopo e não apresentam um caráter recursivo. Este aspecto faz com que as dependências não sejam indeterminadas.

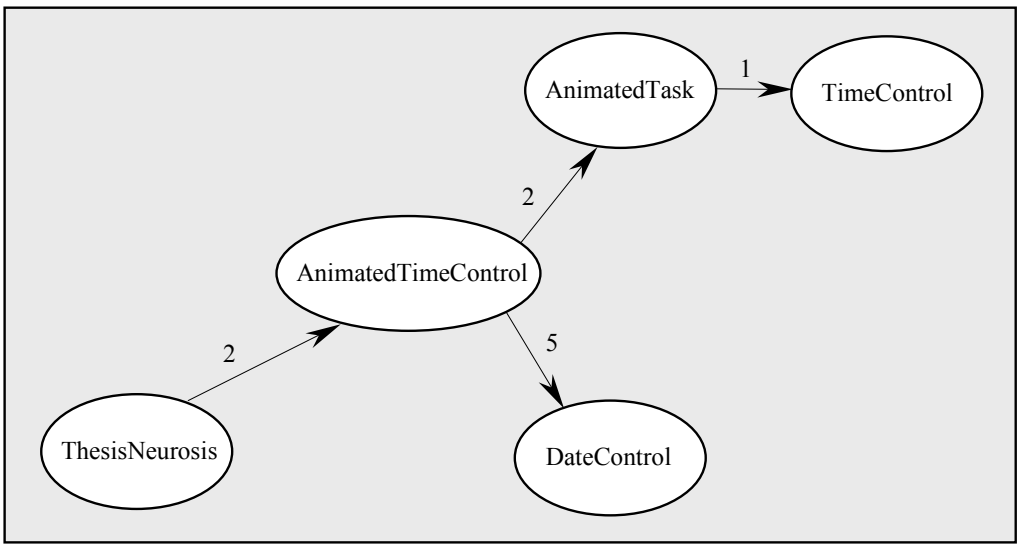


FIGURA 3.17 - Grafo de Dependências Referente a Segunda Invocação do programa *ThesisNeurosis*

LEGENDA:

1. ThesisNeurosis (TN);
2. TimeControl (TC);
3. DateControl (DC);
4. ThesisNeurosisApplet (TNA);
5. AnimatedTask (AT);
6. AnimatedTimeControl (ATC);
7. DateIntervalControl (DIC).

	1	2	3	4	5	6	7
1						2	
2							
3							
4							
5		1					
6			5		2		
7							

FIGURA 3.18 - Matriz Correspondente ao grafo de dependências referente a segunda invocação do programa *ThesisNeurosis*

A primeira coluna da matriz representa a entidade que dispara a dependência e a primeira linha representa as entidades que são invocadas, ou seja, que prestam serviço às referentes entidades que os solicitam. Neste contexto as entidades referentes à primeira coluna são dependentes das entidades referentes a primeira linha, isto é, necessitam de informações providas por estas últimas.

Através da análise destas informações detecta-se que a segunda invocação da aplicação não faz referência às classes *ThesisNeurosisApplet* e *DateIntervalContro,* sendo assim os nodos referentes a estas ausentes no grafo.

Outro aspecto relevante de ser ressaltado é o relacionamento entre as entidades do tipo *AnimatedTimeControl* e *DateControl*, o qual é de cinco dependências. Considerando-se este fragmento de fluxo como um todo, poderia se dizer que os conjuntos de objetos destas classes não devem se encontrar em nós distintos da arquitetura distribuída. No entanto, tal decisão deve ser tomada levando-se em conta a quantidade de dependências estabelecidas durante todo o fluxo da aplicação, visando um menor custo geral de comunicações interclasses.

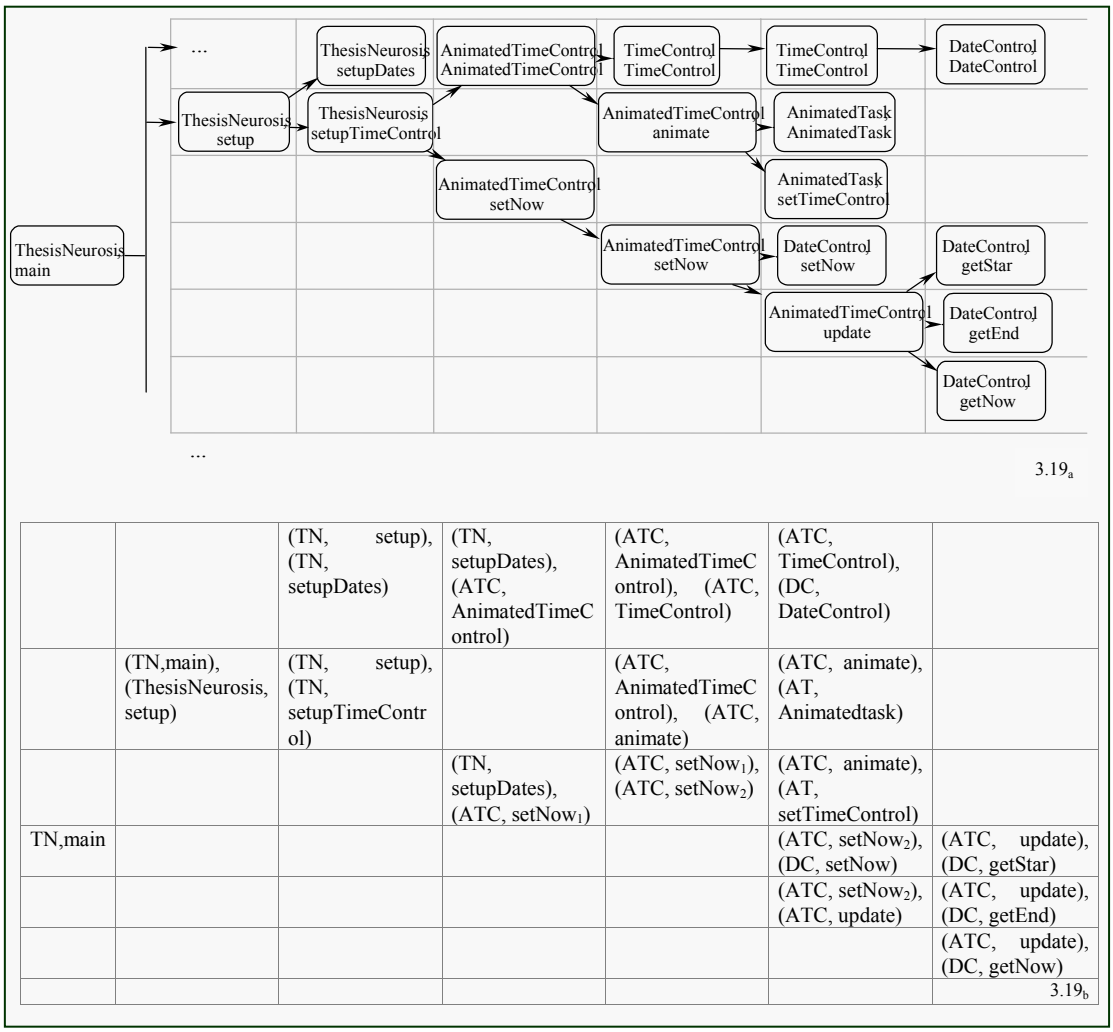
3.4.2 Grafo de Invocações

O grafo de invocações apresenta uma versão mais detalhada dos relacionamentos entre as classes declaradas no programa. Para tal os nodos deste grafo representam os métodos invocados e suas respectivas classes. As arestas representam o fluxo de invocação entre estes.

O formato textual destas informações é apresentado em uma matriz a qual depende da quantidade de métodos invocados e da profundidade destes fluxos, ou seja, da quantidade de invocações que uma determinada invocação desencadeia. No entanto,

quanto ao formato destas informações pode-se dizer que cada entrada na matriz contém a classe e o método de origem e a classe e o método destino de determinada invocação.

Para elucidar o grafo de invocações considera-se o fluxo de invocações desencadeado pela segunda invocação da aplicação *ThesisNeurosis*, o qual é apresentado na figura 3.19_a e o seu respectivo padrão textual na figura 3.19.



		(TN, setup), (TN, setupDates)	(TN, setupDates), (ATC, AnimatedTimeControl)	(ATC, AnimatedTimeControl), (ATC, TimeControl)	(ATC, TimeControl), (DC, DateControl)	
	(TN,main), (ThesisNeurosis, setup)	(TN, setup), (TN, setupTimeControl)		(ATC, AnimatedTimeControl), (ATC, animate)	(ATC, animate), (AT, Animatedtask)	
			(TN, setupDates), (ATC, $setNow_1$)	(ATC, $setNow_1$), (ATC, $setNow_2$)	(ATC, animate), (AT, setTimeControl)	
TN,main					(ATC, $setNow_2$), (DC, setNow)	(ATC, update), (DC, getStar)
					(ATC, $setNow_2$), (ATC, update)	(ATC, update), (DC, getEnd)
						(ATC, update), (DC, getNow)

3.19_b

FIGURA 3.19 - Grafo de Invocações Referente a Segunda Invocação do Programa *ThesisNeurosis* (3.19_a) Matriz Correspondente (3.19_b)

3.5 Modelo Proposto X Modelos Relacionados

Esta seção tem o objetivo de explicitar as caraterísticas do modelo proposto e realizar uma comparação deste com os trabalhos apresentados na seção 2.4. Dentre estas características encontram-se:

- **Domínio abstrato (1):**

O domínio abstrato refere-se a classificação das informações geradas pela análise. Conforme o enfoque da análise deve-se adotar um domínio abstrato que ressalte os aspectos de comportamento desejados;

- **Grafo de invocações parcial ou total (2):**

  Um grafo pode ser construído parcialmente ou totalmente. Então devido a este aspecto os sistemas analisados podem ser classificados quanto a parcialidade ou totalidade dos grafos;

- **Grafos gerados (3):**

  Outro aspecto relevante para a análise entre os sistemas é avaliá-los segundo os tipos de grafos que estes geram. Esta característica identifica diferentes propósitos, seja de visualização ou de informação, dependendo das opções de grafos oferecidas;

- **Aplicação das informações (4):**

  Este tópico tem a finalidade de classificar os sistemas avaliados segundo a aplicabilidade das informações geradas por estes.

A tabela 3.2, apresenta a comparação entre o DEPAnalyzer e os analisadores estáticos descritos na seção 2.4.

TABELA 3.2 - Tabela de Comparação entre o Modelo Proposto e Modelos Relacionados

	DEPAnalyzer	**PIFG**	**CFA_0**	**Grafo de Chamadas**	**JavaParty**	***Análise de Existência***
1	relacionamentos	tipos	tipos	tipos	tipos	verifica se o tipo pertence ou não ao conjunto de classes anlisado (mundo fechado)
2	parcial	parcial	total	total		
3	sim	não	não	não	não	não
4	escalonamento	desenvolvimento de *software* (depuração)	verificação estática de tipos	verificação estática de tipos	escalonamento	verificação estática de tipos

Como se percebe, através da tabela 3.2, tanto o sistema de análise estática do ambiente JavaParty quanto a análise de existência não são comparados em relação ao grafo que geram, pois este não é o objetivo deste sistemas.

Tanto o DEPAnalyzer quanto o analisador estático PIFG geram grafos parciais. No entanto, a parcialidade destes grafos provem de naturezas diferentes. A parcialidade dos grafos gerados pelo DEPAnalyzer provem do desprezo pelas ações realizadas pela API, já que estas são entidades locais presentes em todos os nós da arquitetura. Já a parcialidade do grafo gerado pela analisador PIFG provem do fato deste analisar uma determinada região de código. Esta análise distinta de uma região de código pode não atingir o comportamento de determinado grupo de entidades, como o feito pelo

DEPAnalyzer. Por este motivo, a utilização das informações geradas pelo PIFG no processo de redistribuição e relativa a região de código que deve ser analisada.

Outro aspecto interessante é que tanto o analisador estático do ambiente JavaParty como o DEPAnalyzer estão voltados para o escalonamento. No entanto, estes sistemas diferem por seus domínios abstratos. Isto deve-se ao fato do sistema de análise estática do ambiente JavaParty tratar as entidades (objetos) individualmente. Esta característica difere da concepção do modelo DEPAnalyzer, o qual visa detectar o relacionamento entre as classes de um programa. Além disso, o domínio abstrato do sistema DEPAnalyzer baseia-se no escopo em que os relacionamentos ocorrem para determinar a intensidade destes.

3.6 Considerações Finais

O DEPAnalyzer é um analisador estático de dependências para programas orientados a objetos. Neste paradigma as classes (entidades estáticas) provem em tempo de execução conjuntos de objetos. O modelo proposto visa mapear os relacionamentos entre estes conjuntos. A finalidade deste mapeamento é auxiliar no processo de escalonamento. No entanto, as informações geradas podem ser utilizadas para outros propósitos como: engenharia reversa, depuração de programas, etc.

O DEPAnalyzer atinge o seu objetivo através da geração de dois grafos, o grafo de dependências e o grafo de invocações. Os grafos expressam os relacionamentos segundo a classificação do seu domínio abstrato quantitativo. Isto tem como objetivo apresentar ao escalonador a intensidade destes relacionamentos. O grafo de dependências visa auxiliar no processo de alocação inicial. Já o grafo de invocações pode ser utilizado no processo de redistribuição das entidades durante o processamento.

4 Protótipo do modelo DEPAnalyzer

Este capítulo apresenta a implementação do modelo DEPAnalyzer (*DEPendencies Analyzer*). A seção 4.1 descreve as características gerais do protótipo. A seção 4.2 apresenta o protótipo do módulo de coleta das informações. Já a seção 4.3 descreve o protótipo desenvolvido para o módulo de análise. A seção 4.4 apresenta as considerações finais deste capítulo.

4.1 Aspectos Gerais

Com o propósito de propiciar a validação dos principais pontos do modelo proposto, algumas características do modelo foram desconsideradas. No entanto estas mesmas podem ser incluídas com relativa facilidade ao protótipo sem afetar a estrutura geral do mesmo.

Uma das limitações diz respeito à análise intraprocedural dos tipos, ou seja, a propagação interna, de cada método, dos tipos das variáveis. Isto reduz o escopo de análise do protótipo a somente programas com polimorfismo de sobrecarga. No entanto, mesmo sem a análise de aplicações com polimorfismo de reescrita o protótipo alcança o seu propósito, pois verifica os relacionamentos entre as entidades de um programa.

Outra característica importante é que o protótipo gera somente o grafo de dependências. No entanto, esta limitação pode ser contornada através do recolhimento, durante o processamento do algoritmo de análise, de informações como: “quem esta invocando quem?”. Percebe-se que as informações necessárias para a geração do grafo de invocações já estão sendo produzidas, no entanto, como o objetivo é a obtenção do grafo de dependências estas não estão sendo armazenadas.

O protótipo segue a estruturação básica do modelo DEPAnalyzer, ou seja, é composto de dois módulos, o módulo de coleta das informações e o módulo de análise.

4.2 Módulo de Coleta das Informações

Como apresentado na seção 3.3 o módulo de coleta das informações necessita da utilização de um *parser* para coletar as informações relevantes para a análise. O *parser* utilizado é denominado JavaCC, o qual é uma ferramenta geradora de analisadores léxico-sintáticos [ENS 2000].

Este sistema foi desenvolvido pela Sun [SUN 2001] e encontra-se provido de um suporte de manutenção conferido pela WebGain (http://www.webgain.com). A versão do JavaCC utilizada pelo DEPAnalyzer é a 2.0 e encontra-se disponível em: http://www.webgain.com/products/java_cc/. Além disso, o JavaCC possui um repositório de gramáticas, no qual se encontra a gramática Java1.1, da linguagem Java. Esta gramática foi desenvolvida por Sriram Sankar e é sobre a qual o protótipo reconhece os programas analisados.

Para se recolher as informações contidas na estrutura *CollectedData*, a gramática Java1.1 foi instrumentada com ações semânticas que disparam um processo de coleta toda vez que uma estrutura sintática relevante é reconhecida pela gramática. As ações

semânticas foram implementadas em Java [SUN 2001_a], como todo o resto do protótipo. A estrutura *CollectedData* é apresentada na figura 4.1.

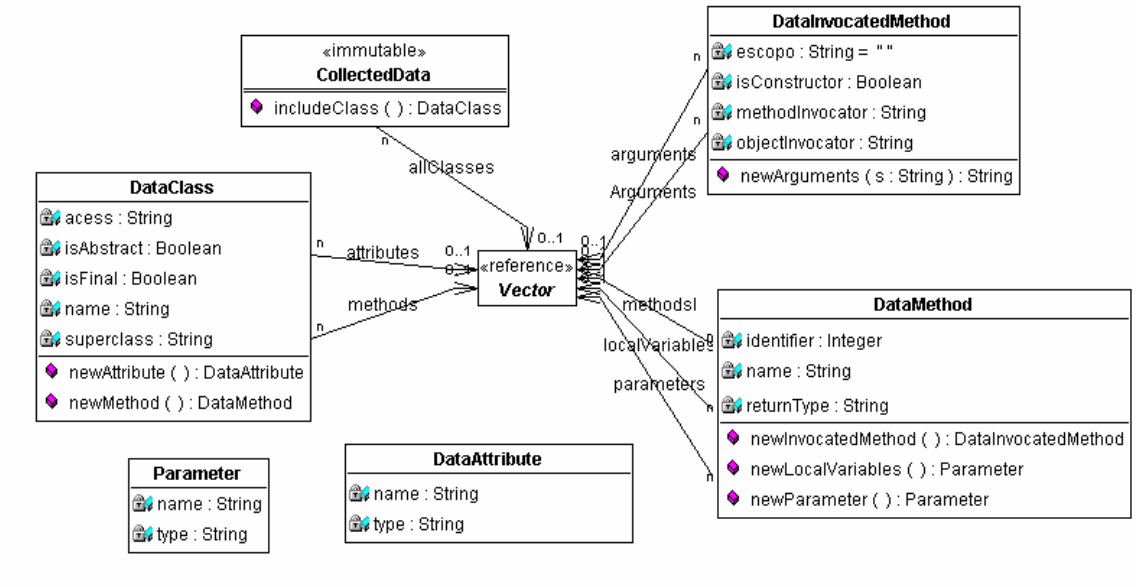


FIGURA 4.1 - Estrutura *CollectedData*

A diferença entre a figura 4.1 e a figura 3.3, ou seja, a estrutura *DataInvocatedMethod*, deve-se ao fato do protótipo não realizar a análise interna do fluxo dos dados (tipos) dentro de cada método declarado. No entanto, o protótipo consegue realizar a análise interna dos tipos vinculados às variáveis, ou seja, quando uma variável é instanciada o módulo de coleta das informações consegue capturar o tipo que está sendo vinculado à variável em questão. Este processo juntamente com a análise das atribuições do programa pode prover a análise intraprocedural necessária para se analisar programas com polimorfismo de reescrita.

Outra característica é que em nível de protótipo é importante saber se um método é construtor ou não. Esta necessidade vem do fato deste tipo de método ser invocado sem a presença de um objeto (*objectInvocator* igual a ***null***), uma vez que a sua invocação é disparada a partir da palavra reservada *new*. Esta identificação permite que se saiba o escopo de procura, na hierarquia de classes analisadas, da declaração do método.

Então como pode-se perceber o atributo *objectInvocator* é uma String, a qual contém o nome do objeto que realiza a invocação. Se este atributo contiver o valor **null**, isto significa que o método invocado é da própria hierarquia de classes do seu método invocador, ou seja, o tipo do método em que ocorre a invocação sem objeto é o mesmo do método que está sendo invocado diretamente (sem objeto). Outra opção para o valor ***null*** no atributo *objectInvocator* é que este método seja construtor.

Se o valor de *objectInvocator* é diferente de ***null*** este pode ser subdivido em: ***super***, ***this*** e **nome da variável/objeto**. Quando é a palavra reservada ***super*** significa que esta invocação refere-se a um método declarado nas superclasses da classe do método invocador. Quando o valor é ***this*** significa que está se invocando um método ou atributo da própria classe do método invocador ou de alguma de suas superclasses. Quando o valor é o **nome da variável/objeto** deve-se procurar pela declaração desta variável primeiramente no método invocador depois na classe deste.

Além disso, o protótipo necessitou da identificação única dos métodos declarados nas classes analisadas. Isto é alcançado através da numeração seqüencial, dentro de cada classe, dos métodos declarados. A identificação permite que o sistema de controle da simulação do fluxo de invocação ao verificar invocações recursivas não necessite acessar a estrutura *CollectedData*. Isto torna o processamento desta tarefa mais rápido.

Para exemplificar as informações coletadas considera-se a classe *DateIntervalControl*, a qual é apresentada na figura 4.2_a e sua corresponde coleta de informações mostrada na figura 4.2_b.

```
public class DateIntervalControl extends DateInterval {
protected Observable observable = new Observable();

  public void addObserver(Observer observer) {
observable.addObserver(observer);
  }


  public void setInterval(int days, int hours, int minutes, int seconds) {
      super.setInterval(days, hours, minutes, seconds);
  }
}
```

4.2_a

```
[CLASS DateIntervalControl ]
    Superclass = DateInterval
    Acess = public
    Attributes = observable Observable
    Methods =
    <Method Name>  addObserver
    return type: void
    identifier: 1
    parameters (Observer  observer)
    local variables
    method invocated (observable) (addObserver)  (false)
    arguments (observer)
    <Method Name>  setInterval
    return type: void
    identifier: 2
    parameters (int  days)  (int  hours)  (int  minutes)  (int  seconds)
    local variables
    method invocated (super) (setInterval)  (false)
    arguments (days)  (hours)  (minutes)  (seconds)
```

4.2_b

FIGURA 4.2 - Informações Coletadas da Classe *DateIntervalControl*

Na figura 4.2_a, a invocação do método *setInterval*, ocorrida no corpo do método declarado *setInterval*, possui como valor de *objectInvocator* ***super***, isto caracteriza a invocação do primeiro método com a mesma assinatura encontrado entre as classes superiores da hierarquia de *DateIntervalControl*.

Outro exemplo de coleta é apresentado nas figuras 4.3_a e 4.3_b. O código da classe *AnimatedTimeControl* é apresentado na figura 4.3_a e a respectiva coleta é mostrada pela figura 4.3_b.

```
public class AnimatedTimeControl extends TimeControl {
  protected Timer timer = null;
  public AnimatedTimeControl(Date start, Date end, Date now) {
    super(start, end, now);
    animate();
  }

  public AnimatedTimeControl(Date start, Date end) {
    super(start, end);
    animate();
  }

public void animate() {
    timer = new java.util.Timer();
    AnimateTask animate = new AnimateTask();
    animate.setTimeControl(this);

   timer.schedule(animate,
                       0, 1*1000);
  }
}
```

4.3_a

```
[CLASS AnimatedTimeControl ]
     Superclass = TimeControl
     Acess = public
     Attributes = timer Timer false
     Methods =
     <Method Name>  AnimatedTimeControl
     return type: null
     identifier: 1
     parameters (Date  start)  (Date  end)  (Date  now)
     local variables
     method invocated (null)(super)  (false)
     arguments (start)  (end)  (now)
      (null)(animate)  (false)
     arguments
     <Method Name>  AnimatedTimeControl
     return type: null
     identifier: 2
     parameters (Date  start)  (Date  end)
     local variables
     method invocated (null)(super)  (false)
     arguments (start)  (end)
     (null)(animate)  (false)
     arguments
     <Method Name>  animate
     return type: void
     identifier: 3
     parameters
     local variables (Timer  timer)  (AnimateTask  animate)
     method invocated (null)(Timer)  (true)
     arguments
     (null)(AnimateTask)  (true)
     arguments
    (animate)(setTimeControl)  (false)
     arguments
    (this)(this)  (false)
    arguments (this)
    (timer)(schedule)  (false)
    arguments (animate)  (0)  (1*1000)
```

4.3_b

FIGURA 4.3 - Informações Coletadas da Classe *AnimatedTimeControl*

Este exemplo consegue elucidar o fato de um método ser invocado sem *objectInvocator* e ser um método construtor. Os métodos *Timer* e *AnimateTask* invocados no terceiro método (*identifier* 3) declarado na classe *AnimatedTimeControl* são exemplos de invocações de construtores. Isto pode ser verificado através do valor do atributo *isConstructor*, o qual apresenta o seu valor após o nome do método invocado, ou seja, nos casos citados este valor é ***true***.

Outra característica interessante, ressaltada pela figura 4.3$_b$, é a possibilidade do atributo *methodInvocated* ser ***super***. Quando isto acontecer o algoritmo de análise deve procurar o construtor da superclasse da classe onde ocorre esta invocação. No caso da figura 4.3$_b$ deve-se procurar nos métodos construtores declarados na classe *TimeControl*, a qual é *superclasse* de *AnimatedTimeControl*.

O valor de *methodInvocated* também pode ser ***this***, o que representa a invocação de um método com o mesmo identificador do método invocador da referente hierarquia de classes onde ocorre esta invocação. Isto também deve ser distinguido pelo sistema de análise, pois tanto a palavra reservada ***super*** quanto a palavra reservada ***this*** não encontram-se no escopo de métodos declarados nas classes analisadas.

O processo de coleta necessita de uma varredura do código fonte. Sabe-se que esta tarefa é uma ação *I/O bound*, o que geralmente é mais demorada que uma ação *CPU bound* [OLI 2001]. Por este motivo durante o desenvolvimento do protótipo DEPAnalyzer teve-se a preocupação de coletar todas as informações importantes para a análise com somente uma leitura do código fonte. Este aspecto foi alcançado, tornando o processo de coleta mais rápido se comparado com um coletor que recolhe as suas informações realizando duas ou mais passagens pelo código.

4.3 Módulo de Análise

Como previsto pelo modelo o módulo de análise recebe a estrutura *CollectedData* e a referente classe principal da aplicação. A primeira ação realizada pelo processo de análise é a verificação da variável booleana *End*, a qual identifica se o processo de simulação chegou ao final. Então se esta variável possui o valor ***true*** o processo chegou ao fim, se não deve-se continuar o processo de simulação. Inicialmente esta variável possui o valor ***false***. Então partindo deste pressuposto (*End* igual a ***false***) o processo de análise realiza a criação de um objeto tipo *Tupla*, o qual tem sua estrutura mostrada na figura 4.4.

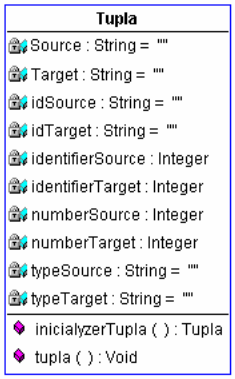


FIGURA 4.4 - Classe *Tupla*

A classe *Tupla* refere-se à estrutura que mapeia o relacionamento entre método invocador e método invocado, ou seja, compõe um dos princípios básicos no qual o algoritmo de análise se fundamenta.

Os atributos **Source* (ou seja, *idSource*, *typeSource*, *Source*, *numberSource identifierSource*) referem-se ao método invocador, os atributos **Target* (ou seja, *idTarget*, *typeTarget*, *Target*, *numberTarget*, *identifierTarget*) referem-se ao método invocado. O atributo *id** contém o nome do método. O atributo *type** contém a classe a que o método pertence. Os atributos *Source* e *Target* referem-se respectivamente ao escopo de invocação do método invocador e ao escopo de invocação do método invocado. Já o atributo *number** refere-se ao número seqüencial do método dentro de seu contexto de invocação, ou seja,

idSource = m_A.
typeSource = A.
numberSource = 1.
idTarget = m_B.
typeTarget = B.
numberTarget = 5.

significa que o método m_A da classe *A* é o primeiro método invocado por seu método invocador, e que o método m_B da classe *B* é o quinto método invocado por m_A.

O atributo *identifier** contém o identificador do método. Esta informação é necessária para se distinguir entre invocações de métodos com o mesmo nome mas com assinaturas distintas.

Após a criação da *Tupla* inicial, a qual recebe o nome da classe principal nos campos *typeSource* e *typeTarget* e recebe o valor "*main*" nos campos *idSource* e *idTarget*, a simulação do fluxo de invocações do programa tem início, através da invocação do método *simulation*, o qual realiza a simulação das invocações ocorridas no corpo do método *main*.

Este método utiliza uma estrutura de controle das invocações, a qual recebe as *tuplas*. Esta estrutura simula a pilha de invocações do programa refletindo em cada momento da simulação o atual traço de invocações desencadeado pela aplicação.

A primeira ação realizada pelo método *simulation* é a criação de uma nova *Tupla* a qual irá atribuir valores aos seus atributos conforme o valor contido na variável *Return*. Esta variável tem a função de identificar quando um método não dispara invocações, seja porque este chegou ao final da sua seqüência de invocações ou porque este não as realiza (não possui invocações ou suas invocações são de métodos declarados em classes que se encontram fora do contexto de análise).

Inicialmente a variável *Return* se encontra com o valor ***false***, indicando que ocorrerá uma nova invocação a partir daquele ponto do programa. Então o processo de simulação começa atribuindo os campos "*.Target*" da *Tupla* inicial para os campos "*.Source*" da nova *Tupla*. A figura 4.5 sumariza esta ação.

```
if(Return == false)
{
        newTupla.idSource = oldTupla.idTarget;
        newTupla.typeSource = oldTupla.typeTarget;
        newTupla.Source = oldTupla.Target;
        newTupla.numberSource = oldTupla.numberTarget;
        newTupla.identifierSource = oldTupla.identifierTarget;
}
```

FIGURA 4.5 - Ação Feita pelo Método s*imulation* Quando *Return* e *End* são Falsos

Após esta ação deve-se obter o próximo método invocado pelo método invocador (inicialmente o método *main*). Isto é alcançado através de uma procura na estrutura *CollectedData* em busca do próximo método invocado por *idSource*, da nova *Tupla*. Se este for ***null*** deve-se verificar se o método contido em *idSource* (da nova *Tupla*) é o método *main*. Se este fato for verdadeiro detecta-se que o método *main* chegou ao final da sua seqüência de invocações, ou seja, a variável *End* recebe a valor ***true*** indicando que o processo de simulação do programa terminou. A figura 4.6 mostra esta situação.

```
if(nextM.methodInvocated == null){
        if(t.idSource == "main"){
                End = true;
        }
        else{
                ...
        }
}
```

FIGURA 4.6 - Teste para Verificar se a Simulação Chegou ao Final

Se o método contido em *idSource* não for o método *main* deve-se retornar o fluxo de simulação ao método invocador, ou seja, o método invocador de *idSource*. Então o processo de análise faz *Return* igual a ***true*** e invoca o método *simulation*.

O fato de retornar ao método invocador significa que o traço de invocações retornado será retirado da estrutura de controle, fazendo com que a estrutura reflita este retorno e se encontre pronta para representar o novo traço que será disparado. A figura 4.7 apresenta as ações feitas nesta situação.

```
if(nextM.methodInvocated == null){
        if(t.idSource == "main"){
                ...
        }
        else{
                Tupla Pt = new Tupla();
                do{
                        Pt = control[pop--];
                        controlPut[p] = Pt;
                        p++;
                } while (t.idSource != Pt.idTarget);
                Return = true;
        }
}
```

FIGURA 4.7 - Retorno Provocado por *Return* ser Verdadeiro

A estrutura *control* representa a estrutura de controle das invocações, ou seja, a estrutura aonde estão as *Tuplas* invocadas naquele ponto da simulação da aplicação. Como percebe-se, retira-se as *Tuplas* da estrutura *control* até que se ache a *Tupla* correspondente à invocação do método que chegou ao final da sua seqüência de invocações.

Se o próximo método invocado por *idSource* for diferente de ***null*** deve-se identificar a classe da nova invocação, para que a nova *Tupla* obtenha o valor do atributo *typeTarget*. Isto necessita da análise do tipo do elemento que realiza a invocação, ou seja, o tipo da variável contida em *objectInvocator* do método invocado.

Como citado na seção 4.2 o valor de *objectInvocator* pode ser ***null***, ***this***, ***super*** ou **nome de uma variável/objeto**. Ressalta-se que semanticamente o valor ***null*** e ***this*** são iguais. No entanto, o protótipo realiza este tratamento em nível de análise e não em

nível de coleta. Isto requer que o processo de análise considere os dois casos distintamente, pois só assim a análise pode conferir a mesma semântica aos dois.

Se o valor for igual a ***null*** pode ser a invocação de um método construtor ou de um método declarado na hierarquia de classes da classe invocadora (tipo do método invocador). Para se saber se o método é construtor deve-se verificar o atributo *isConstructor* do método invocado. Se este for verdadeiro significa que o método é construtor. Então o processo de análise atribui o valor contido em *methodInvocated* para *typeTarget*.

Se o método não é construtor deve-se atribuir o valor contido em *typeSource* a *typeTarget*. No entanto, se o valor de *objectoInvocator* é ***null*** e o valor de *methodInvocated* é igual a ***super*** significa que se está diante da invocação do método construtor da superclasse da referente classe, ou seja, referente a *typeSource*. Neste caso, *typeTarget* também recebe *typeSource*. O mesmo processo pode ser verificado quando *objectInvocator* é igual a ***null*** e *methodInvocated* é igual a ***this***.

Quando o valor de *objectInvocator* é igual a ***this*** ou a ***super*** significa que o método invocado está presente no conjunto de métodos visíveis pela classe invocadora. A visibilidade refere-se aos métodos herdados pela referida classe. Nos dois casos o valor de *typeTarget* é igual ao valor contido em *typeSource*.

Se o valor de *objectInvocator* é diferente de ***null***, ***super*** e ***this***, realiza-se a verificação do tipo da variável/objeto que realiza a invocação. Este processo é realizado pelo método *typeVerification*. Este realiza uma busca nos atributos da classe referente às variáveis locais do método invocado. Se esta busca não obtiver sucesso busca-se a referente informação entre os atributos declarados na classe igual a *typeSource* ou nas superclasses desta. A procura nas superclasses decorre do fato desta variável não se encontrar dentre os atributos declarados na classe e sim dentre os atributos herdados por esta.

A tabela 4.1 apresenta um sumário das combinações possíveis entre *objectInvocator* e *methodInvocated*, para que se possa esclarecer mais o que o processo de análise faz em cada um dos casos.

TABELA 4.1 - Combinação dos Valores de *objectInvocator* e *methodinvocated*

objectInvocator	***methodinvocated***	**Ação**
null	super	typeTarget = typeSource
null	this	typeTarget = typeSource
null	nome do método	typeTarget = typeSource, se isConstructor é igual a ***false***
null	nome do método	typeTarget = methodInvocated, se isConstructor é igual a ***true***
this	nome do método	typeTarget = typeSource
super	nome do método	typeTarget = typeSource
nome da variável/objeto	nome do método	typeTarget = tipo da variável que realiza a invocação

Posteriormente a este processo de verificação de tipos realiza-se a atribuição do valor contido em *Target* para *Source*. Isto representa a propagação do escopo das invocações.

Na seqüência compara-se o valor contido em *typeTarget* com os tipos submetidos ao modelo. Se isto for verdadeiro deve-se obter o identificador do método invocado. Se o tipo do método invocado não se encontrar entre as classes analisadas deve-se fazer *Return* igual a ***true*** e invocar o método *simulation*.

Para se obter a informação sobre o identificador do método invocado leva-se em consideração o nome do método invocado e sua seqüência de argumentos recebidos. Então quando se encontra diante das duas primeiras opções da tabela 4.1 torna-se o valor da variável booleana *Super* (diante da primeira opção) ou da variável booleana *This* (diante da segunda opção) ***true***. Isto é necessário pelo fato citado anteriormente, ou seja, não existe um método declarado com o nome ***this*** ou com o nome ***super***. Estes valores representam referencias a métodos construtores de determinada hierarquia de classes.

Então o processo de identificação primeiramente avalia estas duas variáveis (*Super* e *This*) se estas se encontram com valores iguais a ***false*** procede-se com a procura pelo método com o mesmo nome do método invocado na classe referenciada por este (ou seja, *typeTarget*). Se este não se encontra neste escopo deve-se procurar na superclasse desta. É claro que isto só é possível se a superclasse se encontra presente no conjunto de classes submetido ao protótipo. Se o método não se encontra no conjunto de classes analisado tornar-se *Return* igual a ***true*** e invocar-se o método *simulation*. Se existe um método com o mesmo nome na hierarquia desta classe submetida ao modelo, analisa-se os tipos dos argumentos recebidos por este e a ordem, para se verificar que se trata do mesmo método (devido a presença do polimorfismo de sobrecarga).

Quando se trata do mesmo método deve-se recolher a informação sobre o *identifier* deste no atributo *identifierTarget*. Se não existe um método com esta assinatura na hierarquia visível pelo sistema deve-se tornar *Return* igual a ***true*** e invocar *simulation*.

Se a variável *This* encontra-se com o seu valor igual a ***true*** a análise verifica o nome do método invocador é procura por este método na referente classe e superclasses. Após encontrar deve-se verificar os tipos dos argumentos e a sua ordem. Após isto procede-se igualmente ao processo requerido para *This* igual a ***false***. O mesmo é feito quando o valor da variável *Super* é ***true***. Ressalta-se que quando *Super* é ***true*** a busca é realizada nas superclasses da classe invocadora.

Posteriormente a determinação do identificador do método invocado pode-se realizar a verificação se esta invocação já ocorreu no traço de invocações presentes na estrutura de controle. Se isto for verdadeiro detecta-se uma invocação recursiva. Diante deste fato o processo de simulação faz *Return* igual a ***true*** e invoca o método *simulation* novamente.

Ao detectar um traço recursivo marca-se todas as *Tuplas* pertencentes a este com o escopo **T**, o qual representa um escopo iterativo. Deve-se também retomar a simulação destas *Tuplas* para verificar as dependências produzidas. Na detecção de alguma dependência produzida pelo traço recursivo marca-se a matriz com o valor **T**.

A matriz representa a estrutura onde as dependências são armazenadas. É também esta o produto final do módulo de análise do sistema DEPAnalyzer.

As dependências são detectadas durante o processo de verificação de tipos. Quando um método possui um tipo diferente do tipo do método invocador, deve-se estabelecer uma dependência entre a classe do método invocador e a classe do método

invocado se esta última se encontra dentro do escopo dos tipos analisados. Se estas condições são verdadeiras então deve-se invocar o método *isDependencies*, o qual marca a respectiva dependência na matriz de dependências. Para realizar esta ação o método recebe os tipos envolvidos no relacionamento é o escopo de ocorrência da invocação, ou seja, do método invocado. Isto serve para que o método *isDependencies* saiba se deve marcar o relacionamento como se este fosse interativo (ocorreu dentro laços) ou como um relacionamento normal, isto é, desprovido de escopos interativos. Quando o escopo é interativo deve-se marcar o relacionamento com o símbolo T, o qual representa relacionamentos indeterminados. Quando o escopo não é interativo deve-se verificar se já existe algum relacionamento entre estes tipos na estrutura matriz e qual a natureza deste. Se o relacionamento existente for indeterminado o valor continua indeterminado. Se o relacionamento existente é não indeterminado deve-se acrescer o número existente de uma unidade para que a matriz passe a representar este novo relacionamento.

Como percebe-se diante da detecção de um traço recursivo as dependências providas por este são tratadas como se fossem indeterminadas. Este fato leva ao requerimento do reprocessamento do traço recursivo em busca das dependências desencadeadas por este.

Um exemplo das informações de dependências geradas pode-se ser dado considerando-se a classe principal do programa *ThesisNeurosis*, o qual foi introduzido no capítulo três. A figura 4.8 apresenta o método principal desta aplicação.

```
public static void main(String[] args) throws Exception {
    ThesisNeurosis tn = new ThesisNeurosis();
    tn.setup();

    JTimeControlPanel timeControlPanel1 = new JTimeControlPanel("Time Control");
    JDateControlPanel dateControlPanel = new JDateControlPanel("Dates");

    timeControlPanel1.setTimeControl(tn.getTimeControl());
    dateControlPanel.setDateControl(tn.getTimeControl().getDateControl());

    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(3);
    frame.setTitle("Thesis Neurosis");
    frame.getContentPane().setLayout(new FlowLayout());
    frame.getContentPane().add(dateControlPanel);
    frame.getContentPane().add(timeControlPanel1);
    frame.setSize(700,320);

    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation((d.width - frame.getSize().width) / 2, (d.height -
frame.getSize().height) / 2);
    frame.setVisible(true);
  }
```

FIGURA 4.8 - Método *main* da Classe *ThesisNeurosis*

A figura 4.8 evidência que somente o fato que somente as duas primeiras invocações disparadas pelo método *main* da classe *ThesisNeurosis* referenciam classes contidas no conjunto de classes submetidas ao sistema. Isto pode ser conferido pela figura 3.3. A matriz de dependências resultante para esta aplicação é a mesma apresentada pela figura 3.17_b.

Como descrito na seção 3.4 o módulo de análise tem como submódulos o submódulo de análise e o submódulo composto por uma ferramenta gráfica. O submódulo de análise gera uma matriz, a qual pode alimentar decisões de escalonamento, ou ser visualizada por uma ferramenta gráfica.

O protótipo do modelo DEPAnalyzer utiliza como ferramenta gráfica o sistema gráfico apresentado em [ERL 97] e disponível em [ERL 2001]. Para que este sistema

reproduzisse a matriz resultante da análise implementou-se uma classe Java denominada *ConvertMatrixToGraph*, a qual lê a matriz de adjacência gerada e converte esta para o grafo apresentado na figura 4.9. Este processo é feito através da criação de um arquivo *.html*, o qual permite a visualização deste grafo.

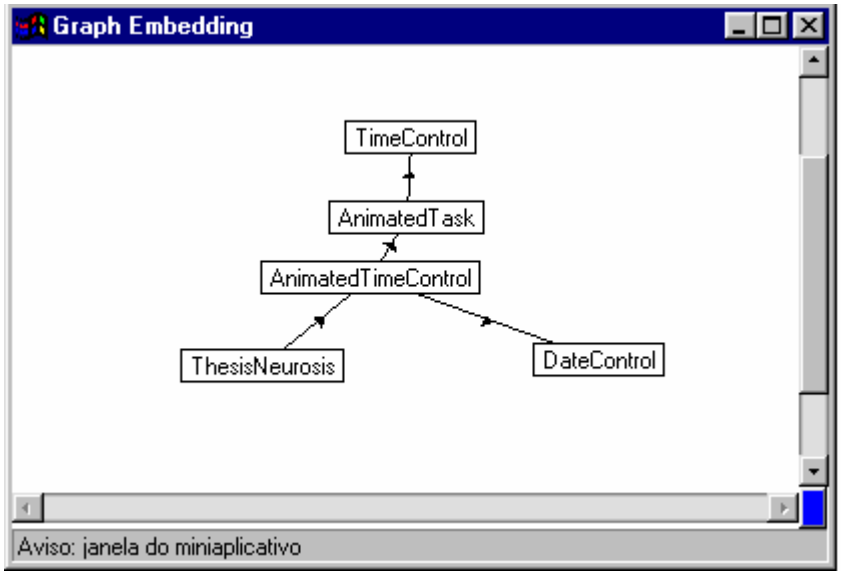


FIGURA 4.9 - Representação Gráfica da Matriz Gerada pela Análise

4.4 Considerações Finais

O protótipo do modelo DEPAnalyzer foi desenvolvido totalmente em linguagem Java. O mesmo emprega o *parser* JavaCC para realizar a análise sintática do código. Os resultados são disponibilizados na forma de uma matriz. Esta matriz contém as informações para o escalonador e também pode ser visualizada através de uma ferramenta gráfica [ERL 2001].

As informações na matriz caracterizam os relacionamentos estabelecidos entre as entidades declaradas na aplicação. Para expressar estes relacionamentos o protótipo gera o grafo de dependências.

Apesar do grafo de invocações entre as entidades não estar sendo disponibilizado nesta versão do protótipo, as informações para a construção do mesmo já estão sendo detectadas.

5 Conclusão

Este capítulo apresenta as conclusões obtidas através da realização deste trabalho, bem como as frentes de trabalhos futuros.

A análise estática visa obter informações sobre o comportamento que um programa terá durante a sua execução. Isto torna a utilização desta técnica em programas orientados a objetos atraente. Estes programas possuem vários aspectos dinâmicos como: verificação dinâmica de tipos, ligação dinâmica de invocações, etc. A verificação dinâmica de tipos causa um *overhead* na execução, o qual pode ser contornado pela análise estática de tipos. Esta última consegue, muitas vezes, determinar estaticamente os tipos das variáveis, contribuindo para diminui o *overhead* causado pela verificação dinâmica de tipos e consequentemente aumentar o desempenho das aplicações.

As informações geradas pelo DEPAnalyzer visam auxiliar no escalonamento das entidades em um ambiente distribuído. Para atingir este objetivo o sistema gera dois grafos, o grafo de dependências e o grafo de invocações. O primeiro objetiva colaborar com o processo de alocação inicial, realizado pelo escalonador. O segundo pode colaborar com o processo de redistribuição dos objetos em uma arquitetura distribuída.

O grafo de dependências oferece uma visão resumida da simulação do processamento da aplicação, no qual o tempo de ocorrência dos relacionamentos é desprezado. No entanto, esta visão simplificada pode ser de grande valia para as primeiras decisões tomadas pelo ambiente de escalonamento.

Já o grafo de invocações pode colaborar com o processo de redistribuição, pois permite a discriminação dos relacionamentos através do tempo. Esta característica permite que com o passar do tempo do processo de execução novas configurações de distribuição sejam atingidas para se obter o menor *overhead* de comunicação.

O modelo classifica as informações apresentadas pelos grafos devido a intensidade de dependência dos relacionamentos. Isto é expressado através do domínio abstrato sob o qual as informações são classificadas. O domínio abstrato utilizado baseia-se na questão do escopo em que ocorrem os relacionamentos. Este fato é decorrente da natureza da análise estática, a qual não propaga valores e sim informações gerais (como tipos) que possam classificar um conjunto de entidades (objetos) da aplicação. A generalização das informações propagadas gera o desconhecimento do número de vezes que a interação de um laço ocorrerá, ou o desconhecimento do caminho que um comando condicional tomará. Levando-se em conta esta característica o modelo propõem uma classificação das informações coletadas através do escopo em que estas ocorrem, verificando assim o levantamento de todas os relacionamentos quantitativamente indefinidos estaticamente.

O modelo trabalha com um conjunto de classes. Este aspecto parte do pressuposto que as classes da biblioteca Java estarão presentes em todos os nós da arquitetura. Assim as comunicações entre classes primitivas e classes definidas pelo programador são comunicações locais. Além disso, este aspecto, muitas vezes, prove grafos parciais. A parcialidade dos grafos refere-se as ações desempenhadas pela API e não as ações desempenhadas pelas classes definidas pelo programador. Então, mesmo tendo a parcialidade como característica os grafos atingem os seus objetivos, ou seja, auxiliar no processo de escalonamento.

O DEPAnalyzer abre frentes de trabalho envolvendo o aprimoramento das informações geradas. Isto permitirá que as mesmas possam cada vez mais auxiliar no processo de escalonamento. Para isso alguns trabalhos futuros destacam-se:

- detecção do comportamento de leitura/escrita de um método: este tipo de informação visa auxiliar, dentre outros trabalhos, no processo de replicação de objetos do ReMMoS [FER 2001]. Este consiste em um ambiente para suporte a replicação de objetos distribuídos móveis. Para que as réplicas sejam criadas, é necessário que o ReMMoS tenha a informação do tipo de comportamento do método. Desta forma, a cada acesso remoto ao objeto, o sistema de replicação controla se o acesso é de leitura ou de escrita. Esta informação é usada também para atualizar e descartar as réplicas, e deve ser obtida por um analisador através da análise dos métodos;
- emprego na alocação de entidades em sistemas distribuídos *wide-area*: a proposta EXEHDA [YAM 2001] realiza a alocação das unidades de modelagem (entes) do Holoparadigma [BAR 2001] no contexto de aplicações móveis distribuídas. A integração das informações do fluxo de invocações do DEPAnalyzer com as informações obtidas dinamicamente pelos trabalhos [ARA 2001], [SIL 2001] poderá contribuir para a otimização das decisões de escalonamento.

Referências

[AGE 94] AGESEN, O. **Constraint-Based Type Inference and Parametric Polymorphism.** 1994. Disponível em: <http://www.sun.com/research/self/papers/sas94.html>. Acesso em: maio 2000.

[AGE 95] AGESEN, O. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. European Conference on Object- Oriented Programming. In: ECOOP: EUROPEAN CONFERENCE ON OBJECT- ORIENTED PROGRAMMING, ECOOP, 9., 1995, Aarhus, Denmark. **Proceedings…** Berlin: Springer, 1995. Disponível em: <http://citeseer.nj.nec.com/agesen95cartesian.html.>.Acesso em: maio 2000.

[AGR 99] AGRAWAL, G. **Simultaneous Demand Drive Data Flow and Call Graph Analysis**. 1999. Disponível em: <http://citeseer.nj.nec.com/417649.html>. Acesso em: dez. 2000.

[AGR 2001] AGRAWAL, G. et al. **Evaluating a Demand Drive Technique for Call Graph Construction**. 2001. Disponível em: <http://citeseer.nj.nec.com/420154.html>. Acesso em: set. 2001.

[AMB 98] AMBLER, S. W. **Análise e Projeto Orientado a Objetos**. Rio de Janeiro: Infobook, 1998. 472 p.

[ARA 2001] ARAUJO, E. B. et al. Uma Proposta de Monitoração para Visualização de Aplicações Distribuídas Java. In: JORNADAS CHILENAS DE COMPUTACIÓN, 2001, Punta Arenas. **Anales…** Punta Arenas: Universidad de Magallaes, 2001.

[AUG 2001] AUGUSTIN, I. et al. Requisitos para o Projeto de Aplicações Móveis Distribuídas. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, CACIC, 8., 2001, Santa Cruz, Ar. **Anales...** [S.l.: s.n.], 2001.

[AZE 98] AZEVEDO, S. C. et al. Integração Party-Granlog: Interpretação Abstrata Aplicada a Paralelização de Programas em Lógica. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, CACIC, 4., 1998, Neuquen, Ar. **Trabajos Selecionados**. Neuquen: Universidad Nacional Del Comahre, 1998. v.1, p.551-559.

[AZE 99] AZEVEDO, S. C. et al. Automatização da Análise Global no modelo GRANLOG. In: CONFERENCIA LATINO AMERICANA DE INFORMÁTICA, 1999, Asunción. **Anales…**[S.l.: s.n.], 1999.

[AZE 2001] AZEVEDO, S. C. et al. DEPAnalyzer: um Analisador Estático de Dependências para Programas Java. In: WORKSHOP EM SISTEMAS

COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 2001, Pirenópolis. **Anais...**[S.l.:s.n.], 2001. p. 135-141.

[AZE 2001$_a$] AZEVEDO, S. C. et al. Análise Estática de Dependências em Programas Java Seqüenciais. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, CACIC, 8., 2001, Santa Cruz, Ar. **Anales...** [S.l.: s.n.], 2001.

[BAC 96] BACON, D. et al. **Fast Static Analysis of C++ Virtual Functions Calls**. 1996. Disponível em: <http://citeseer.nj.nec.com/bacon96fast.html.>. Acesso em: mar. 2000.

[BAI 97] BAIRAGI, D. et al. **Precise Call Graph Construction for OO Programs in the Presence of Virtual Functions.** 1997. Disponível em: <http://www.computer.org/proceedings/icpp/8108/8108toc.htm>. Acesso em: jan. 2001.

[BAR 2001] BARBOSA, J. et al. Using Mobility and Blackboards to Support a Multiparadigm Model Oriented to Distributed Processing. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E PROCESSAMENTO DE ALTO DESEMPENHO, SBAC PAD, 2001, Perinópolis. **Proceedings ...** [S.l.: s.n.], 2001.

[CAS 97] CASTRO, L. F. P. **Um Modelo de Analisador Estático Baseado na Interpretação Abstrata Direcionado à Paralelização de Programas em Lógica.** 1997. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[COH 96] COHEN, A. et al. **Array Data-Flow Analysis for Imperative Recursive Programas**, 1996. Disponível em: <http://citeseer.nj.nec.com/10464.html>. Acesso em: mar. 2001.

[COR 97] CORTESI, A. et al. Complementation in Abstract Interpretation. **ACM Transactions on Programming Languages and Systems**, New York, v.19, n.1, p.7-47, 1997.

[COU 76] COUSOT, P. et al. **Static Determination of Dinamic Properties of Programs**. 1976. Disponível em: <http://www.di.ens.fr/~cousot/COUSOTpapers.shtml>. Acesso em: nov. 1999.

[DAM 97] DAMS, D. et al. Abstract Interpretation of Reactives Systems. **ACM Transactions on Programming Languages and Systems**, New York, v. 19, n. 2, p.253-291, 1997.

[DEA 94] DEAN, J. et al. **Optimization of Object-Oriented programs Using Static class Hierarchy Analysis.** 1994. Disponível em: <http://citeseer.nj.nec.com/dean94optimization.html>. Acesso em: mar. 2001.

[DEW 2001] DEWES, H. et al. Static Method Call in Java. In: JOSES: Java Optimization Strategies for Embedded Systems. Apr. 2001. Disponível em: <http://i44w3.info.uni-karlsruhe.de/~josesworkshop/>. Acesso em: mar. 2001.

[DIW 96] DIWAN, A. et al. **Simple and Effective Analysis of Statically-Type Object-Oriented Programs**. 1996. Disponível em: <http://citeseer.nj.nec.com/159587.html>. Acesso em: maio 2000.

[ENS 2000] ENSELING, O. **Build your own languages with JavaCC**. 2000. Disponível em: <http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html>. Acesso em: set. 2000.

[ERL 97] ERLINGSSON, Ú. et al. Interactive Graph Drawing on the World Wide Web. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, 1997. **Poster Presentation**. Disponível em <http://www.scope.gmd.de/info/www6/posters/703/igdraw.html.>. Acesso em: set. 2001.

[ERL 2001] ERLINGSSON, Ú. et al. **Interactive Graph Drawing Homepage**. Disponível em: <http://www.cs.rpi.edu/projects/pb/graphdraw/>. Acesso em: set. 2001.

[FER 2001] FERRARI, D.N. et al. Um ambiente de suporte à replicação em sistemas com mobilidade explícita de objetos distribuídos. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, 2001, Mérida. **Anales…** [S.l.: s.n.], 2001.

[FUJ 01] FUJUBA 2.5.4 Disponível em: <http://www.uni-paderborn.de/cs/fujaba/>. Acesso em: maio 2001.

[HAS 98] HASTI, R. et al. **Using Static Single Assigment Form to Improve Flow-Insensitive Pointer Analysis.** 1998. Disponível em: <http://citeseer.nj.nec.com/hasti98using.html>. Acesso em: abr. 2001.

[JEN 2000] JENSEN, T. et al. **An Overview of Class Analyses of Object-oriented Languages**. 2000. Disponível em: <http://www.irisa.fr/lande/spoto/papers.html>. Acesso em: mar. 2000.

[LAB 96] LA BANDA, M. et al. Global Analysis of Constraint Logic Programming. **ACM Transactions on Programming Languages and Systems**, New York, v.18, n.5, p.564-614, 1996.

[MAN 2001] MANGAN, M. A. S. **Programa ThesisNeurosis**. Disponível em: <http://www.cos.ufrj.br/~mangan/thesisneurosis/ThesisNeurosisApplet.html>. Acesso em: mar. 2001.

[OLI 2001] OLIVEIRA, R. S. et al. **Sistemas Operacionais.** Porto Alegre: Sagra Luzzatto, 2001. 233 p.

[OXH 92] OXHOJ, N. et al. **Making Type Inference Pratical**. 1992. Disponível em: <http://citeseer.nj.nec.com/oxhoj92making.html>. Acesso em: maio 2000.

[PAL 91] PALSBERG, J. et al. **Object-Oriented Type Inference**. 1991. Disponível em: <http://citeseer.nj.nec.com/palsberg91objectoriented.html>. Acesso em: maio 2000.

[PER 98] PERALTA, J. C. et al. **Analysis of Imperative Programs through Analysis of Constraint Logic Programs**. 1998. Disponível em:

<http://citeseer.nj.nec.com/59770.html >. Acesso em: mar. 2001.

[PHI 97] PHILIPPSEN, M. et al. JavaParty - Transparent Remote Objects in Java. **Concurrency - practice and experience**, New York, v.9, n.11, p.1225-1242, Nov. 1997.

[PHI 2000] PHILIPPSEN, M. et al. Locality optimization in JavaParty by means of static type analysis. **Concurrency - practice and experience**, New York, v.18, n.8, p.613-628, July 2000.

[PLE 94] PLEVYAK, J. et al. **Precise Concrete Type Inference for Object-Oriented Languages**. 1994. Disponível em: <http://citeseer.nj.nec.com/plevyak94precise.html>. Acesso em: mar. 2000.

[PRI 2000] PRICE, A. M. A. et al. **Implementação de Linguagens de Programação:** compiladores. Porto Alegre: Sagra Luzzatto, 2000. 195 p.

[PRO 2001] PROBST, Christian. Flow Sensitive Call Graph Construction For Java. In: JOSES: Java Optimization Strategies for Embedded Systems. Apr., 2001. Disponível em: <http://i44w3.info.uni-karlsruhe.de/~josesworkshop/>. Acesso em: mar. 2001.

[SIL 2001] SILVA, L. et al. Mecanismos de Suporte ao Escalonamento em Sistemas com Objetos Distribuídos Java. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, CACIC, 8, 2001, Santa Cruz. **Anales...** [S.l.: s.n.], 2001.

[SRE 2000] SREEDHAR, V. C. et al. **Framework Optimization in the Presence of Dinamic Class Loading.** 2000. Disponível em: <http://citeseer.nj.nec.com/sreedhar00framework.html>. Acesso em: mar. 2001.

[STE 99] STEINDL, C. **Static Analysis of Object-Oriented Programs**. 1999. Disponível em: <http://www.ssw.uni-linz.ac.at/Research/Papers/Ste99b.html>. Acesso em: set. 2000.

[SUN 2001] SUN MICROSYSTEMS. The Java Hotspot Performance Engine Architecture. Disponível em: <http://java.sun.com/products/hotspot/whitepaper.html.>. Acesso em: mar. 2001.

[SUN 2001_a] SUN MICROSYSTEMS. 2001. Disponível em: <http//java.sun.com/>. Acesso em: mar. 2001.

[SUV 98] SUNDARESAN, V. et al. **Practical Virtual Method call Resolution for Java**. 1998. Disponível em: <http://citeseer.nj.nec.com/137178.html>. Acesso em: mar. 2001.

[TEN 95] TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados Usando C**. São Paulo: Makron Books, 1995. 884 p.

[TOP 2001] TOP 500 Supercomputer site. Lista dos 500 mais poderosos computadores com suas características, principais usos e local de

instalação. Disponível em: <http://www.top500.org/>. Acesso em: 2001.

[VER 2001] VERONESE, G. O. et al. **ARES:** uma ferramenta de Auxílio a Recuperação de Modelos UML de Projetos a partir de Códigos Java. 2001. Trabalho de conclusão (Curso em Ciência da Computação) – Departamento de Ciência da Computalção, UFRJ, Rio de Janeiro.

[WEB 2001] WEBGAIN. **Structure Builder 4.0.** Disponível em: <http//www.webgain.com>. Acesso em: 2001.

[YAM 2001] YAMIN, Adenauer et al. Explorando o Escalonamento no Desempenho de Aplicações Móveis Distribuídas. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 2, 2001. **Anais...** Brasília: UNB, 2001.