Um Modelo de Memória Virtual em Java para Sistemas de Transformação de Software

Edson Murakami
Antônio Francisco do Prado
{murakami, prado}@dc.ufscar.br
Departamento de Computação
Universidade Federal de São Carlos
Via Washington Luiz, km 235.
São Carlos - SP - CEP 04499-610

Resumo

Sistemas de Transformação (ST) têm sido usados na engenharia de software para construção de compiladores, desenvolvimento e reengenharia de sistemas de software. Durante o processo de transformação toda a memória principal pode ser ocupada pela Abstract Syntax Tree (AST) do programa alvo da transformação. Esse grande consumo de memória tem causado problemas aos ST, particularmente quando se trata dos compiladores e grandes sistemas de software. Este artigo apresenta um modelo de memória virtual em Java que utiliza bancos de dados relacionais para armazenar a AST, estendendo a memória principal utilizada pelos ST durante o processo de transformação. Este modelo possui um gerenciador de memória principal e um serviço de persistência que recupera os nós da AST do banco de dados sob demanda. Este modelo está sendo implementado no ST SpinOff.

Palavras-Chave: Sistemas de Transformação de Software, AST, Gerenciador de Memória Cache, Serviço de Persistência e Banco de Dados Relacional.

Abstract

Transformation Systems (ST) have been used in software engineering for construction of compilers, development and reengineering of software systems. During the transformation process all main memory can be busy for the Abstract Syntax Tree (AST) of target program of transformation. This great consumption of memory has caused problems to the ST, particularly when it is about the compilers and great software systems. This article presents a model of virtual memory in Java that uses relational databases to store the AST, extending the memory used for the ST during the transformation process. This model has a main memory manager and a persistence service that retrieve AST nodes of database on demand. This model is being implemented in the ST SpinOff.

Keywords: Software Transformation System, AST, Cache Memory Management, Persistence Service and Relational Database.

1 Introdução

O uso de Sistemas de Transformação (ST) nas importantes tarefas da engenharia de software, como construção de compiladores, desenvolvimento e reengenharia de sistemas de software tem sido objeto de estudos em diversas universidades e empresas que desenvolvem software. Ferramentas tais como *Tampr* [Boy89], *Refine* [Rea92], TXL [Cor93], *Popart* [Wil93], Draco-PUC [Lei94], DMS [Bax97] e *RescueWare* [Faq98] são alguns exemplos de ST.

A experiência da comunidade de sistemas de transformação de software tem mostrado que as *Abstract Syntax Trees* (AST) são estruturas de dados que podem ser muito grandes consistindo de milhões de nós [Bax98], uma vez que o tamanho da AST é diretamente proporcional ao programa manipulado pelo ST. O potencial de desperdício da memória principal disponível num sistema operacional durante o processo de transformação é muito alto. As implementações de memória virtual de sistemas operacionais não são aceitáveis, porque as experiências mostram que elas conduzem a um esgotamento de memória principal muito rápido durante o processo de transformação [Bax98]. Esse esgotamento ocorre quando o ST alcança um nó da AST e carrega suas subárvores na memória principal. Uma vez carregadas na memória, as subárvores permanecem durante todo o processo de transformação. Um coletor de lixo não pode eliminá-las porque estão referenciadas e também porque elas podem ser reutilizadas, a qualquer momento, pelo ST.

Dadas essas considerações, uma solução é mapear a estrutura hierárquica da AST para uma estrutura relacional, estendendo a capacidade de armazenamento da AST em bancos de dados relacionais persistidas em memória secundária. Com essa solução resolve-se não apenas o problema de desperdício de memória, mas também ganha-se um serviço de persistência. Esse serviço de persistência é importante porque, estando a AST armazenada no banco de dados, esta pode ser reutilizada ou compartilhada por outros transformadores.

Este artigo apresenta um modelo de memória virtual em Java que mantém os nós da AST em banco de dados relacional como tuplas de uma tabela e somente materializa estes nós quando necessários. Apenas os nós mais prováveis de serem usados são materializados e mantidos na memória principal. Neste modelo um nó materializado é um objeto AST. Como mostra a Figura 1 o modelo de memória virtual possui um subsistema gerenciador de memória *cache* e um subsistema de persistência que suporta *lazy materialization* [Lar98]. Os subsistemas baseiam-se nos padrões de design *Cache Management* e *Virtual Proxy* [Gam95, Gra98, Lar98].

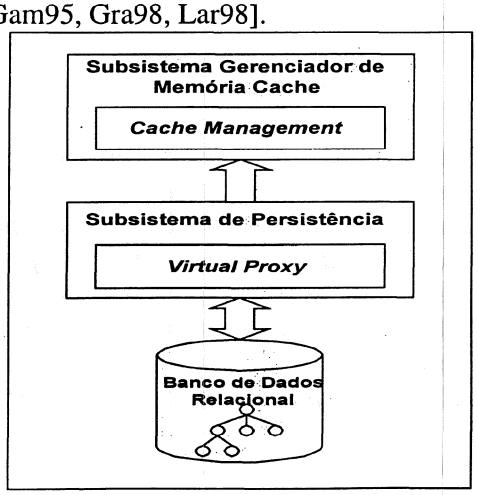


FIGURA 1 - MODELO DE MEMÓRIA VIRTUAL

O Subsistema Gerenciador de Memória Cache mantém na memória principal os nós mais prováveis de serem usados pelo ST durante o processo de transformação. O Subsistema de Persistência recupera os nós do banco de dados quando necessários. O Banco de Dados Relacional é usado para armazenar os nós da AST.

A opção pela utilização de Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDR) para persistir ASTs, deve-se à maturidade desses gerenciadores. Os bancos de dados relacionais atualmente são utilizados predominantemente pela indústria, para armazenamento de grandes quantidades de dados.

Este modelo de memória está sendo implantado no novo ST Draco-PUC [Lei94], denominado SpinOff [San99], em desenvolvimento no Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) em conjunto com pesquisadores do Departamento de Computação da Universidade Federal de São Carlos (UFSCar).

Este artigo está organizado da seguinte forma: na seção 2 são descritas as tecnologias de transformação de software e a integração com Bancos de Dados Relacionais; na seção 3 é apresentado o modelo de memória virtual em Java; na seção 4 é apresentado um protótipo do modelo; e finalmente na seção 5 são apresentadas as conclusões e trabalhos em desenvolvimento.

2 Tecnologia de Transformação de Software e a Integração com Bancos de Dados Relacionais

O SpinOff é uma evolução do sistema de transformação Draco-PUC, com especial ênfase na solução transformacional baseada em componentes. No SpinOff o conhecimento transformacional é capturado em um conjunto de componentes interoperáveis e reutilizáveis diferentemente da maioria dos ST que são tipicamente sistemas monolíticos que trabalham como caixas pretas com muitas limitações de interfaces. O SpinOff utiliza os conceitos das ferramentas de projeto de circuitos eletrônicos *Very Large System Integration* (VLSI) para projetar seus Circuitos Transformacionais (CT). Os CTs são componentes, que integrados, formam os transformadores de software. O SpinOff fornece suporte ao projeto e simulação dos transformadores que podem ser visualizados através de uma interface gráfica do usuário (GUI). No SpinOff são construídas partes transformacionais reutilizáveis que podem ser integradas a outros componentes em um barramento de software compartilhado.

O SpinOff está organizado em camadas como mostra a Figura 2. A primeira camada é a **AST**, sobre a qual todas as outras camadas foram construídas. Essa camada é o subsistema AST do Draco-PUC que foi portado para Java para manipular ASTs, é nessa camada que o modelo de memória virtual está sendo implantado. A Segunda camada é o ***Mutant***, um *framework* para o desenvolvimento de interpretadores AST extensíveis. O *Mutant* é usado para coletar informações ou fornecer semântica operacional a uma AST. Este *framework* é chamado de *Mutant*, por causa da sua natureza adaptativa de fornecer extensibilidade a linguagens quando está interpretando uma AST. A terceira camada é uma máquina virtual no topo do *Mutant* que fornece a funcionalidade essencial para o SpinOff. Essa máquina virtual é dirigida por uma linguagem chamada *AST-based Simple Component Interconnection Language* (**SCIL**), uma linguagem de interconexão de componentes simples baseada em AST. Além das três camadas que foram mencionadas, existe outra camada do SpinOff relacionada a uma linguagem muito específica para descrição de transformadores. Essa linguagem é chamada de *Transformer Description Language* (**TDL**) e é a linguagem utilizada pelo usuário final na descrição de CTs.

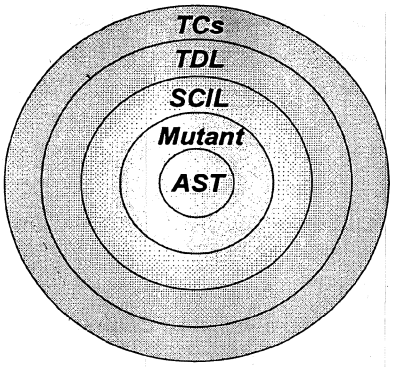


FIGURA 2 - CAMADAS DO SPINOFF

O SpinOff está firmemente baseado nas idéias do paradigma Draco [Nei80] e nos experimentos com o sistema de transformação Draco-PUC.

O Draco-PUC foi reconstruído no Departamento de Informática da PUC-RJ por Prado [Pra92], a partir do sistema transformacional Draco apresentado por Neighbors [Nei80, Nei84, Nei89]. O Draco-PUC objetiva desenvolver, colocar em prática e testar o paradigma transformacional para o desenvolvimento de software orientado a domínios. O Draco-PUC propõe um modelo para o processo de produção de software, em que uma nova especificação pode ser obtida através da aplicação de regras de transformação, descritas formalmente, sobre a especificação de entrada. As regras de transformação são responsáveis pela automatização do processo de construção de software [Fre87, Lei95].

As principais partes que compõem o subsistema AST do Draco-PUC são:

1- ***Parser***: define a sintaxe da linguagem na qual são escritas as especificações do domínio e é responsável por gerar a AST, representação interna utilizada no sistema Draco-PUC.

2- ***Prettyprinter***: responsável por escrever representações em sintaxe abstrata para a sintaxe concreta da linguagem, ou *seja*, a partir de uma AST escreve novamente a especificação na linguagem do domínio.

3- **Transformações**: são componentes contendo regras que transformam a AST. Uma regra de transformação é essencialmente formada por dois padrões: LHS (*Left Hand Side*) que descreve o padrão que deve ser reconhecido nas especificações e RHS (*Right Hand Side*) que descreve o padrão de rescrita que irá substituir a parte da especificação instanciada. Para que uma regra de transformação seja aplicada, o sistema procura na AST pelo padrão de reconhecimento definido (LHS), e ao encontrá-lo, realiza a rescrita do mesmo de acordo com o padrão de substituição desejado (RHS).

Na Figura 3 pode-se observar a ligação entre essas partes. Um programa escrito na linguagem A de um determinado domínio A é analisado pelo *parser* desse mesmo domínio, gerando uma representação interna AST1. Sobre essa AST1 pode ser aplicado o *prettyprinter* do domínio ou transformações intra e inter domínios. O *prettyprinter* é usado para exibir novamente o programa a partir da AST1. Aplicando-se transformações intra-domínio, o resultado é uma nova AST2 representando um novo programa, porém do mesmo domínio. Se as transformações aplicadas forem

inter-domínios, o resultado é também uma nova AST3 já no novo domínio. A partir da AST3, usando o *prettyprinter* desse novo domínio, pode-se obter o programa na linguagem B do domínio B equivalente ao programa inicial escrito na linguagem A.

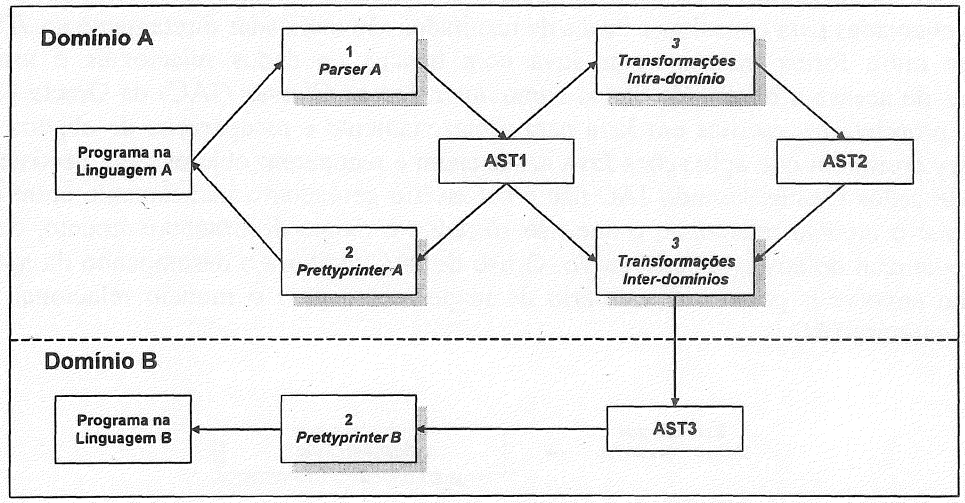


FIGURA 3 - PARTES DE UM DOMÍNIO NO DRACO-PUC

A camada AST do SpinOff é o subsistema AST do Draco-PUC que foi portado para Java. Um dos principais requisitos para essa reengenharia foi a criação de um serviço de persistência que utiliza bancos de dados relacionais para armazenar a AST.

O modelo de memória virtual em Java foi proposto para atender o requisito de persistência da AST em banco de dados. Uma das formas usadas para integrar bancos de dados relacionais usando Java é através das classes *Java Database Connectivity* (JDBC) [Ham98, Ora98, Syb98]. JDBC é uma interface de programação de aplicações (API) para acesso a bancos de dados incluída no *Java Development Kit* (JDK) a partir da versão 1.1. Essa API é um conjunto de classes que possibilita a adição de comandos SQL (*Structured Query Language*) em aplicações Java. A Figura 4 mostra a arquitetura do JDBC.

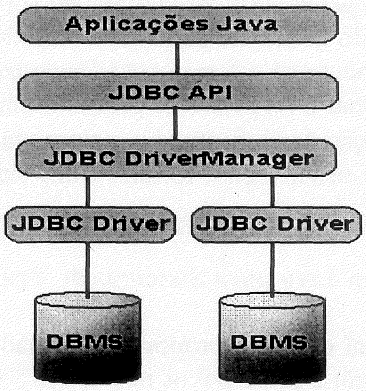


FIGURA 4 - ARQUITETURA DO JDBC

O *JDBC DriverManager* é utilizado para abrir uma conexão com um banco de dados através de um *JDBC driver* que precisa estar registrado antes da conexão. Quando uma conexão é iniciada, o *JDBC DriverManager* escolhe um de uma lista de *JDBC drivers* disponíveis para efetuar a conexão, dependendo do *Uniform Resource Locator* (URL) especificado. Uma vez estabelecida a conexão com sucesso, as chamadas para consultas e busca de resultados são realizadas diretamente no *JDBC driver*.

Uma outra forma para integrar Java com bancos de dados relacionais é usando classes proprietárias de acesso a bancos de dados como *Java Access Classes* (JAC) da Oracle [Ora98]. JAC são classes proprietárias escritas em Java para armazenamento e recuperação de objetos persistentes. Essas classes permitem que aplicações Java armazenem e recuperem objetos Java persistentemente de um banco de dados Oracle. Usando JAC não é necessário gerenciar o mapeamento entre o modelo de objetos Java e o modelo relacional, o que não só reduz o tempo de desenvolvimento, como também diminui o potencial de erros de codificação. O uso de JAC melhora o desempenho da aplicação, uma vez que não envolve o passo intermediário de mapeamento para o modelo relacional. A Figura 5 mostra a arquitetura JAC.

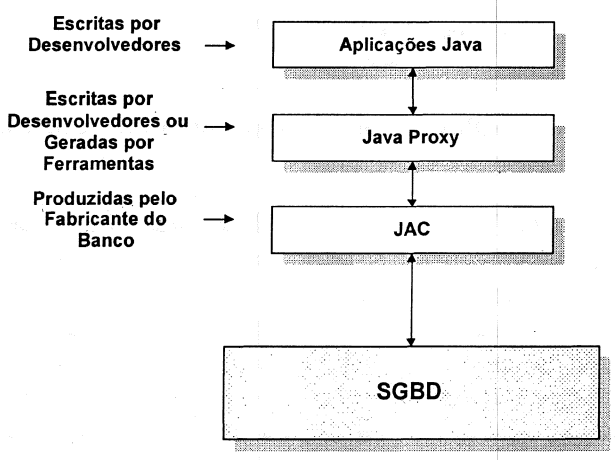


FIGURA 5 - ARQUITETURA JAC

As *Aplicações Java*, escritas pelos desenvolvedores de software, utilizam as classes *Java Proxy* para recuperar objetos persistentes do banco de dados. As classes *Java Proxy* são classes persistentes que podem ser geradas automaticamente pela ferramenta *Designer* da Oracle [Ora98]. As classes *Java Proxy* se comunicam com JAC, o qual é responsável pela comunicação com o sistema gerenciador de banco de dados (SGBD) e manipulação das informações armazenadas no banco de dados.

As tecnologias apresentadas nesta seção, foram utilizadas no modelo de memória virtual que será apresentado na próxima seção.

3 Modelo de Memória Virtual em Java para Sistemas de Transformação de Software

O modelo de memória virtual em Java proposto é formado por um subsistema gerenciador de memória *cache* que mantém na memória principal os nós da AST mais prováveis de serem usados e um

subsistema de persistência para recuperar esses nós sob demanda, a partir de um banco de dados relacional. Este subsistema baseia-se no *lazy materialization* [Lar98].

Para representar o modelo de memória virtual usou-se técnicas do método UML (*Unified Modeling Language*) [UML98]. As classes do modelo de memória virtual em Java foram especificadas em dois diagramas de classes que representam os subsistemas gerenciador de memória *cache* e de persistência. A Figura 6 mostra o diagrama de classes do Subsistema Gerenciador de Memória Cache.

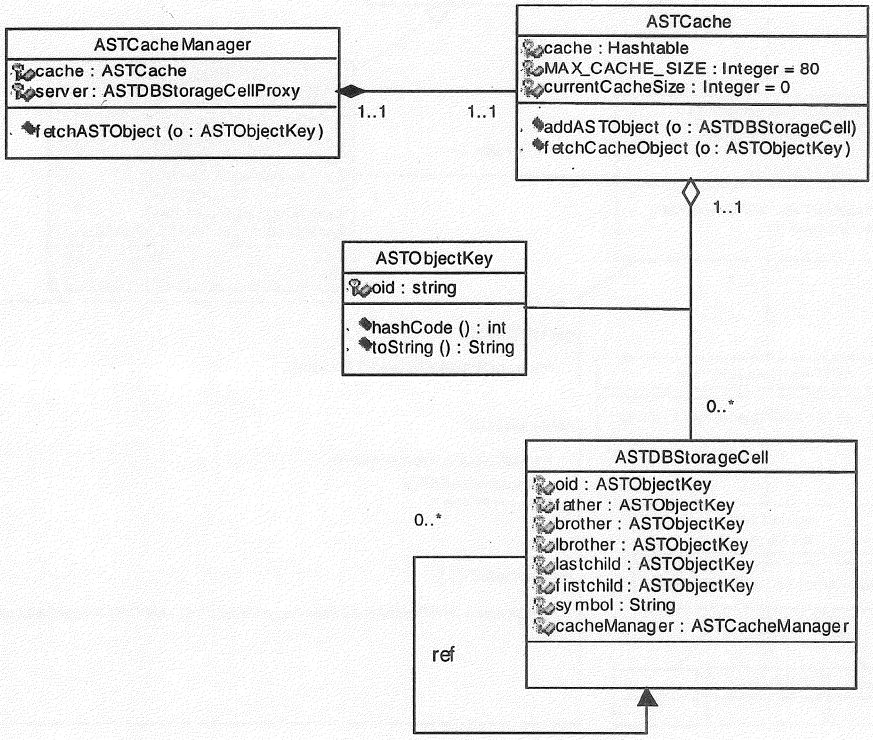


FIGURA 6 – SUBSISTEMA GERENCIADOR DE MEMÓRIA CACHE

Durante o processo de transformação, nós de uma AST são requisitados a um objeto da classe `ASTCacheManager`. Esse objeto é composto de uma memória *cache* representada por um objeto da classe `ASTCache`. A memória *cache* é composta pelos nós de uma AST, representados pelos objetos da classe `ASTDBStorageCell`. Os nós da AST são identificados na memória *cache* pelo atributo *oid* da classe `ASTObjectKey`.

Quando um objeto da classe `ASTCacheManager` recebe uma requisição de um nó, ele tenta recuperá-lo da memória *cache*. Se obtiver sucesso, retorna o nó requisitado.

O objeto da classe `ASTCache` é responsável em gerenciar a memória *cache* mantendo os nós utilizados mais recentemente. Quando o limite da memória *cache* for atingido o objeto da classe `ASTCache` deve descartar o nó utilizado menos recentemente.

A classe `ASTDBStorageCell` possui um auto-relacionamento, denominado *ref*, que faz o encadeamento dos nós da AST.

A Figura 7 mostra o diagrama de classes do subsistema de persistência. As classes `ASTCacheManager` e `ASTDBStorageCell` foram apresentadas na Figura 6.

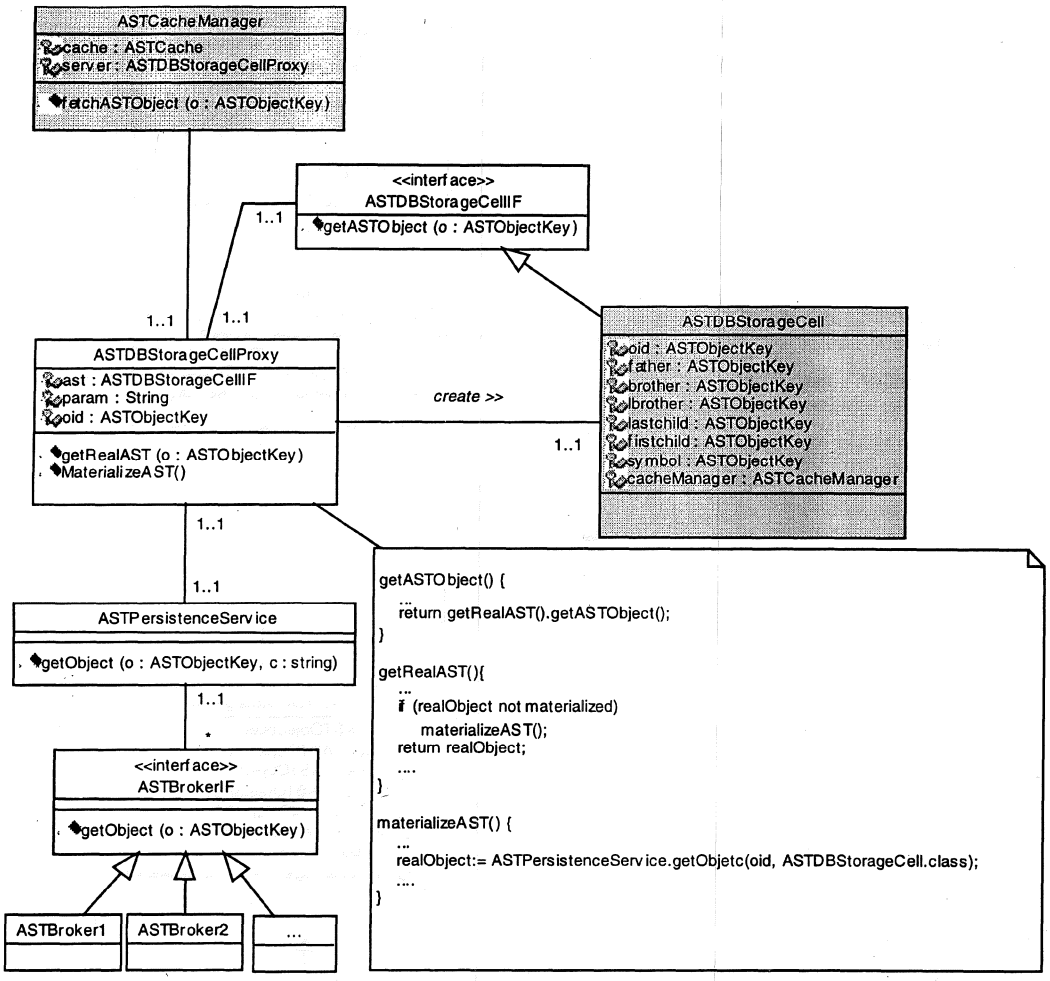


FIGURA 7 – SUBSISTEMA DE PERSISTÊNCIA

Durante o processo de transformação, quando um nó da AST é requisitado a um objeto da classe `ASTCacheManager` e este nó não está disponível na memória *cache*, usa-se o subsistema de persistência para recuperar este nó do banco de dados.

Um objeto da classe `ASTDBStorageCellProxy` é um *proxy* [Gam95]. O *proxy* é usado para encontrar um nó requisitado, testando se este nó foi materializado ou não.

Os nós da AST na memória *cache* são objetos da classe `ASTDBStorageCell`, o qual implementa a interface `ASTDBStorageCellIF`. Através desta interface um *proxy* pode fazer referências dinâmicas aos nós da AST na memória *cache*.

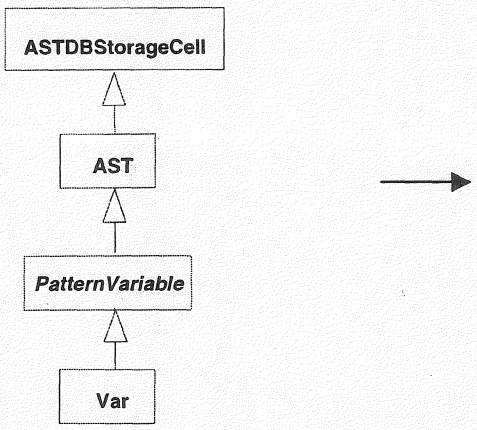
Quando um objeto da classe `ASTCacheManager` não encontra o nó requisitado na memória *cache* ele requisita ao *proxy* para recuperá-lo. O *proxy* testa se o nó requisitado já foi materializado. Se o nó já foi materializado, ele imediatamente o retorna. Se o nó requisitado ainda não foi materializado o *proxy* solicita a um objeto da classe `ASTPersistenceService` para materializá-lo. Esse objeto da classe `ASTPersistenceService` encontra o *broker* [Bro96] responsável pela materialização do nó requisitado, tal como o objeto da classe `ASTBroker1`. A materialização de um novo nó da AST

consiste em instanciar um objeto da classe ASTDBStorageCell com as informações da tupla da tabela do banco de dados correspondente ao nó requisitado. Antes de retornar o novo nó, o *broker* determina que nós filhos sejam criados e conectados ao novo nó. Embora o novo nó esteja materializado, seus nós filhos ainda não estão materializados. Os nós filhos serão lentamente materializados no futuro.

4 Protótipo do Modelo

O objetivo do protótipo foi validar o modelo de memória virtual. Neste protótipo foi implementado o subsistema gerenciador de memória cache e um subsistema de persistência que utiliza JDBC para acessar o banco de dados. O subsistema gerenciador de memória *cache* foi implementado para manter, na memória principal, os nós das ASTs mais prováveis de serem usados. A memória *cache* possui um número limitado de nós que foi definido através de testes que medem a porcentagem de sucesso a partir das tentativas de recuperação de nós da memória *cache*. A memória *cache* usa a estrutura de dados lista linear duplamente ligada para determinar qual nó remover da *cache* quando um novo nó tem que ser adicionado. A política utilizada para descartar nós da memória *cache* é a LRU (*least recently used*). Toda vez que um nó é removido da memória *cache* imediatamente é gerada uma chamada SQL, pelo serviço de persistência, para inserir ou atualizar o referido nó. Dessa forma, mantém-se a consistência da memória *cache* com a fonte de dados.

Para realizar o mapeamento do modelo de objetos para o modelo relacional foi utilizado o padrão estático do *Crossing Chasms* [Bro96]. Aplicando-se o padrão estático, foi criada uma tabela no banco de dados, denominada *ASTDBStorageCell*, como mostra a Figura 8. Cada tupla dessa tabela armazena um nó da AST, que na memória *cache* corresponde a um objeto da classe `ASTDBStorageCell`. As tuplas têm como chave primária o *oid*. As colunas *father, firstchild, lastchild, rbrother e lbrother* são referências a outras tuplas da tabela *ASTDBStorageCell*, *symbol* é uma *string* que armazena o símbolo encapsulado por um nó da AST. As colunas *varname* e *vartype* são atributos da classe `Var`. Esses atributos são utilizados no reconhecimento e aplicação de padrões durante o processo de transformação. O padrão dinâmico do *Crossing Chasms* foi aplicado para criar uma classe *broker* responsável pela leitura e escrita dos nós das ASTs no banco de dados. Neste protótipo foi utilizado um *driver* JDBC do tipo 1 (*JDBC-ODBC bridge)* [Ham98] para acessar o SGBD *Sybase Anywhere 5.0.*



ASTDBStorageCell	
@oid	integer
father	integer
brother	integer
lbrother	integer
lastchild	integer
firstchild	integer
symbol	varchar
varname	varchar
vartype	varchar

FIGURA 8 - MAPEAMENTO DO MODELO DE OBJETOS PARA O MODELO RELACIONAL.

O protótipo foi testado com uma AST de aproximadamente 17 mil nós de um programa *clipper*. Esta AST gerou 17 mil tuplas no banco de dados. Um computador pessoal *Pentium* 133 Mhz, 16 Mb de memória RAM com 28 processos sendo executados no sistema operacional Windows NT 4.0, foi utilizado nos testes. O protótipo levou 4 minutos para concluir o processo de transformação. Na mesma plataforma, sem persistir os nós em banco de dados, não foi possível realizar o processo de transformação por falta de memória principal. Para programas menores que o tamanho da memória *cache,* 6 mil objetos, o protótipo leva em torno de 10 segundos para processar a transformação.

5 Conclusões e Trabalhos em Desenvolvimento

Este artigo apresentou uma proposta de modelo de memória virtual para sistemas de transformação de software. Este modelo tem como objetivo estender a memória principal utilizada pelos STs durante o processo de transformação, armazenando a AST em banco de dados relacional. O modelo possui um subsistema gerenciador de memória *cache* para manter os nós da AST mais prováveis de serem usados na memória principal e um subsistema de persistência que recupera os nós do banco de dados sob demanda.

Um protótipo do modelo de memória virtual foi desenvolvido e testado. No protótipo foram implementados o subsistema gerenciador de memória *cache* e um subsistema de persistência que utiliza JDBC para acessar o banco de dados. Os testes realizados com o protótipo, apesar do tempo de processamento ainda um pouco elevado, 4 minutos para uma AST de 17 mil nós, apresentaram como principal resultado a solução do problema de esgotamento de memória principal durante o processo de transformação. Com a solução desse problema os STs podem ser utilizados em plataformas de hardware comuns, limitados apenas pelo espaço disponível em memória secundária. Atualmente, as plataformas de hardware necessárias para STs que manipulam programas maiores de 400 K linhas de código necessitam de quantidades de memória principal da ordem de *gigabytes*.

Atualmente os trabalhos estão voltados para a implementação do subsistema de persistência que suporta *lazy materialization* [Lar98], para materializar os nós das ASTs somente quando necessários. Apos a implementacao do subsistema de persistencia, testes de desempenho serao realizados utilizando os SGBDRs Oracle, Sybase e DB2.

Referências Bibliográficas

[Bax97] BAXTER, I., PIDGEON, C. *"Software Change Through Design Maintenance"*. In Proceedings of ICSM97, 1997.

[Bax98] BAXTER, I. YAHIN, A. MOURA, L. SANT'ANNA, M. BIER. L. *Clone Detection Using Abstract Syntax Trees.* Proc. ICSM98, 1998.

[Boy89] BOYLE, J. *Abstract Programming and Program Transformations – Na Approach to Reusing Programs.* Software Reusability (Vol 1), Ed. Ted Biggerstaff, ACM Press, pp.361-413, 1989.

[Bro96] BROWN, K. WHITENACK, B. *"Crossing Chasms: A Pattern Language for Object-Relational Integration"*, in Pattern Languages of Program Design 2, Vlissides, Coplien and Kerth, eds., Addison-Wesley, 1996.

[Cor93] CORDY, J. CARMICHAEL, I. *The TXL programming language syntax and informal semantics, Technical Report.* Queen's University at Kingston, Canada,1993.

[Faq98] FAQs–*RescueWare.* URL: http://www.relativity.com/products/faqs/index.html. Consultado em Setembro de 1998.

[Fre87] FREEMAN, P. *A Conceptual Analysis of the Draco Approach to Constructing Software Systems.* IEEE Transactions on Software Engineering. 13,7. July, 1987.

[Gam95] GAMMA, E. HELM, R. JOHNSON, R. VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[Gra98] GRAND, M. *Patterns in Java. A Catalog of Reusable Design Patterns Illustrated with UML.* Wiley, 1998.

[Ham98] HAMILTON, G. CATTELL, R. FISHER, M. *JDBC™ Database Access with Java™. A Tutorial and Annotated Reference.* Addison Wesley Longman, 1998.

[Lar98] LARMAN, C. *Design for Lazy Materialization.* Java Report, april 1998. URL: http://www.javareport.com. Consultado em 2 de julho de 1998.

[Lei94] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA, M. *Draco-PUC Machine: A Technology Assembly for Domain Oriented Software Development.* 3rd IEEE International Conference of Software Reuse. Rio de Janeiro – RJ, ICSR94, 1994.

[Lei95] LEITE, J.C., et al. *O Uso do Paradigma Transformacional no Porte de Programas Cobol.* IX Simpósio Brasileiro de Engenharia de Software (SBES). Recife - PE, Outubro, 1995.

[Nei80] NEIGHBORS, J.M. *Software Construction Using Components.* Thesis (Doctoral) - Information and Computer Science Dept.University of California. Irvine - California, 1980.

[Nei84] NEIGHBORS, J.M. *The Draco approach to Constructing Software from Reusable Components.* IEEE transactions on software engineering. v.se-10, n.5, pp.564-574, September, 1984.

[Nei89] NEIGHBORS, J.M. *Draco: A Method for Engineering Reusable Software Systems.* Software reusability, concepts and models. ACM Press. V.1, pp.295-320, 1989.

[Ora98] ORACLE CORPORATION. *Oracle Lite Java Developer's Guide Release 3.5,* 1998.

[Pra92] PRADO, A.F. *Estratégia de Engenharia de Software Orientada a Domínios.* Tese (Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro. RJ, 1992.

[Rea92] REASONING SYSTEMS. *Refine User's Guide.* Reasoning Systems Inc., Palo Alto, CA. 1992.

[San99] SANT'ANNA, M. *Circuitos Transformacionais.* Tese (Doutorado). Pontifícia Universidade Católica do Rio de Janeiro. RJ, 1999.

[Syb98] SYBASE. *JConnect for JDBC Technical White Paper.* Sybase Inc. URL: http://www.sybase.com:80/products/internet/jconnect/jdbcwpaper.html. Consultada em setembro de 1998.

[UML98] RATIONAL ROSE CORPORATION. URL:http://www.rational.com/uml/references. Consultado em outubro de 1998.

[Wil93] WILE, D. *POPART: Producer of parsers and related tools system builders'manual, Technical Report.* USC/Information Sciences Institute, Los Angeles, CA. 1993.