Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos

Ana Paula Fukuda
Elisângela Sato de Jesus
Antonio Francisco do Prado
e-mail: {apfukuda, esato, prado}@dc.ufscar.br

Universidade Federal de São Carlos
Departamento de Computação
C.P. 676 - 13565-905 - São Carlos (SP)

Resumo

Este artigo apresenta uma estratégia de refinamento automático de sistemas legados implementados em linguagem procedural para sistemas orientados a objetos distribuídos. Partindo do código fonte de um sistema legado, em uma linguagem procedural, faz-se a sua reengenharia, organizando-o segundo os princípios da orientação a objetos, usando técnicas da abordagem Fusion/RE. O código organizado, ainda na mesma linguagem do código fonte do sistema legado, é denominado código fonte segmentado. Em seguida faz-se a transformação do código fonte segmentado para descrições em MDL, que é a linguagem para a persistência das técnicas Unified Modeling Language (UML), na ferramenta Rational Rose. Estando o sistema descrito em MDL, na ferramenta Rational Rose, faz-se a recuperação do projeto atual do sistema. Uma vez recuperado o projeto atual, ainda usando a ferramenta Rational Rose, faz-se a distribuição dos seus objetos, segundo técnicas do método Catalisys/UML. Por último, faz-se a reimplementação do sistema, descrito em MDL, para uma linguagem orientada a objetos. Com o objetivo de explorar novas idéias na área de geradores de software, adotou-se uma abordagem de refinamento automático baseada em transformação de software, utilizando o sistema transformacional Draco-PUC como principal tecnologia deste projeto.

Palavras-chave: Sistema Legado, Sistema Orientado a Objetos Distribuído, Sistema Transformacional Draco-PUC e Linguagem de Modelagem MDL.

Abstract

This paper presents an strategy of the automatic refinement of legacies systems implemented in procedural language to distributed object oriented systems. By the way of the source code of the a legacy system, in a procedural language, somebody can do your reengineer, organizing the source code according the principles of object oriented, using techniques Fusion/RE approach. The organized code, still in the same language of the source code of the legacy system, is called segmented source code. Following somebody can do the transformation of segmented source code to the MDL description, that is the language to persistence of the Unified Modeling Language (UML) techniques, in Rational Rose tool. After the system described in MDL, in Rational Rose tool somebody can do the recovery actual project of system. At once recovered the actual project, still using the Rational Rose tool, somebody can do the distribution of the actual project's objects, using techniques of the Catalysis/UML method. Finally, somebody can do the re-implementation of system, described in MDL, to object oriented language. Intending to explore new ideas in the area of software generators, we adopted a automatic refinement approach based software transformation, using the Draco-PUC transformational system as main technology for this project.

keywords: *Legacy Program, Distributed Objects Oriented Program, Transformational System Draco-PUC and MDL Modelling Language.*

1. Introdução

Atualmente, um grande número de empresas continuam trabalhando com sistemas implementados em linguagens de programação orientadas a procedimentos, cujas manutenções são onerosas. Linguagens de programação evoluem constantemente e requerem modificações dos sistemas, sejam para corrigir erros, melhorar desempenho ou adicionar novos requisitos. Sistemas escritos em linguagens antigas ainda são de muita utilidade aos seus usuários e muitas vezes a reengenharia é a solução para reutilizá-los evitando a construção de um novo sistema, bastando reconstrui-lo para as modernas plataformas de hardware e software.

Com a disseminação de CPUs cada vez mais poderosas e de custo cada vez mais baixo e com a evolução na natureza das aplicações para uma maior autonomia local na definição e automação dos processos, prevê-se que os ambientes futuros de processamento da informação vão consistir de uma vasta rede de recursos heterogêneos, pertencentes a indivíduos ou organizações diferentes, caracterizando assim os sistemas distribuídos. Em um sistema distribuído os recursos são integrados de forma que as aplicações e os dados possam ser acessados por múltiplos processadores em um ambiente de hardware e software heterogêneo, cuja localização física é irrelevante.

Várias técnicas que vêm sendo utilizadas com sucesso na especificação de sistemas distribuídos estão baseadas nos princípios da orientação a objetos. A observação de princípios como encapsulamento, herança, agregação e polimorfismo, aumentam o reuso do sistema, melhoram sua segurança e facilitam a distribuição dos seus objetos.

Com o objetivo de pesquisar a reengenharia de sistemas de software antigos, possibilitando suas execuções em plataformas de hardware e software distribuídas, segundo o paradigma da orientação a objetos, este artigo apresenta uma estratégia de Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos.

O refinamento automático do código fonte do sistema legado é obtido pela aplicação de transformações usando um sistema transformacional. Sistemas transformacionais como: Refine [Rea92], Popart [Wie93], Tampr [Boy89], TXL [Cor93], RescueWare [Faq98], DMS [Bax97] e Draco-PUC [Nei84, Pra92, Lei94], têm sido usados em diferentes áreas, destacando-se a de geração automática de código. O sistema transformacional Draco-PUC [Pra92, Lei94] foi adotado neste trabalho como a principal tecnologia responsável pelo refinamento automático, transformando o código fonte do sistema legado em código fonte segmentado, de acordo com os princípios da orientação a objetos. Em seguida, o código fonte segmentado é transformado em descrições MDL, que são importadas pela ferramenta *Rational Rose* para recuperar o projeto do sistema atual, segundo técnicas do método UML [Uml98]. Após recuperar o projeto atual do sistema na ferramenta *Rational Rose*, o sistema é reprojetado com a distribuição dos seus objetos, utilizando técnicas do método *Catalysis/UML* [Dso98]. Técnicas *Catalysis/UML* suportam a modelagem distribuída dos objetos do sistema para que o mesmo seja executado em diferentes plataforma de hardware e software. Num último refinamento, faz-se a reimplementação do sistema modelado em *Catalysis*/UML, em uma linguagem orientada a objetos, usando o Draco-PUC.

Este artigo está organizado da seguinte forma: na seção 2 é apresentada uma visão geral dos Sistemas Orientados a Objetos Distribuídos e Técnicas de Transformação. Na seção 3 é descrita a construção e utilização da estratégia de Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos, objetivo deste artigo. Na seção 4 é apresentado um estudo de caso utilizando a estratégia proposta e finalmente, na seção 5 são apresentadas as conclusões.

2. Sistemas Orientados a Objetos Distribuídos e Técnicas de Transformação

Atualmente, a arquitetura Cliente/Servidor é adotada para substituir os computadores de grande porte. Nesta arquitetura o servidor não tem a única função de servir dados, mas também a de solicitar dados a outros servidores.

Além da arquitetura Cliente/Servidor, outra realidade é a da Orientação a Objetos. Várias ferramentas usadas no processo de desenvolvimento de software são otimizadas para trabalhar com os conceitos de Orientação a Objetos.

A integração das tecnologias, Sistemas Distribuídos e Orientação a Objetos, dá origem à área de Objetos Distribuídos (OD), responsável pela intercomunicação entre os vários componentes distribuídos em uma rede.

O uso de sistemas distribuídos, em que seus componentes podem estar localizados em qualquer máquina de uma rede, é hoje uma realidade devido às tecnologias disponíveis para objetos distribuídos. Atualmente existem três arquiteturas predominantes com tecnologias para objetos distribuídos: A *Object Management Architecture-OMA* que trabalha na especificação de uma arquitetura para a administração de objetos distribuídos. A Microsoft, incorporou em seus sistemas operacionais Windows NT/95 o software para objetos distribuídos conhecido como *Distributed Component Object Model – DCOM* [Dco96]. Finalmente, a filial da SUN, *JavaSoft*, estendeu a linguagem de programação Java com uma arquitetura completa para objetos distribuídos: *Remote Method Invocation – RMI* [Sun98].

Outra tecnologia utilizada no Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos é o sistema transformacional Draco-PUC [Nei84, Lei91, Pra92] que baseia-se na implementação por transformação orientada a domínios. Um primeiro protótipo do sistema transformacional, foi construído por Neighbors [Nei84]. Posteriormente, esse Sistema transformacional foi reconstruído na PUC-RJ [Pra92], usando novas linguagens em plataformas modernas de hardware e software. Várias pesquisas [Fre87, Nei89, Lei91, Lei94, Lei95] para mudanças e melhorias [Fre96] resultaram em novas versões do sistema transformacional Draco-PUC, cada vez mais eficientes e poderosas.

Pela estratégia proposta por Prado [Pra92], é possível a reconstrução de um software pelo "porte" direto do código fonte para linguagens de outros domínios. Um domínio, de acordo com o paradigma Draco, é constituído de três partes:

- Uma **linguagem**: definida por um *parser* que é responsável por analisar os sistemas da linguagem e gerar a representação interna do Draco, uma *Abstract Syntax Tree (AST)*, denominada no Draco de DAST.
- Um ***PrettyPrinter*** ou ***unparser***: que realiza a formatação da DAST, tornando-a novamente textual na linguagem do domínio. Baseado nas definições das regras das gramáticas das linguagens o Draco-PUC gera, automaticamente, os respectivos *PrettyPrinters* dos domínios.
- Um ou mais **Transformadores**: que mapeiam estruturas de uma linguagem para estruturas na mesma linguagem do domínio, chamadas de transformações intra-domínio, e que mapeiam as aplicações descritas em uma linguagem de um domínio para descrições de linguagem de outro domínio, chamadas transformações inter-domínios.

O sistema transformacional Draco-PUC propõe um modelo para o processo de produção de software, no qual uma nova especificação pode ser obtida através da aplicação de **regras de transformação.** As regras de transformação são responsáveis pela automatização ou semi-automatização do processo de construção de sistemas. Uma regra de transformação é essencialmente formada por dois pontos de controle:

- Lado esquerdo *(Lhs - Left hand side)*: descreve o padrão que deve ser reconhecido nas especificações escritas na linguagem do domínio, antes da aplicação da regra de transformação.
- Lado direito *(Rhs - Right hand side)*: descreve o padrão de reescrita que substitui a parte da especificação reconhecida no lado esquerdo.

As transformações são armazenadas em transformadores. Cada transformador pode conter seções para declarações, inicializações globais e um ou vários conjuntos de transformações. Cada conjunto de transformações é formado por uma ou mais transformações que possuem os pontos de controle ***Lhs*** e ***Rhs***. As transformações podem ainda disparar eventos, ou alterar o fluxo de controle da aplicação das transformações através de outros pontos de controle, suportando a solução de problemas relacionados com as diferenças sintáticas e semânticas das linguagens usadas na implementação e reimplementação dos sistemas.

3. Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos

O Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos suporta a geração automática de código orientado a objetos distribuídos a partir de sistemas legados implementados em uma linguagem procedural, do qual dispõe-se do código fonte e, caso estejam disponíveis, das informações que documentam o sistema.

Para viabilizar a estratégia foi construído, no sistema transformacional Draco-PUC, um transformador intra-domínio, que organiza o código fonte do sistema legado a ser reconstruído, segundo os princípios da orientação a objetos. O código fonte organizado, ainda na mesma linguagem do código fonte do sistema legado, é denominado código fonte segmentado. A segmentação do sistema é orientada pela abordagem de engenharia reversa Fusion/RE [Pen96], que faz a recuperação dos sistemas de acordo com os princípios da orientação a objetos, sem que os mesmos tenham sido desenvolvidos utilizando essa tecnologia. Esta abordagem recupera os modelos de projeto do sistema, baseados no método Fusion [Col94], e os utiliza para orientar na segmentação do código. A abordagem Fusion/RE é utilizada no transformador intra-domínio para identificar as estruturas do código fonte do sistema legado e organiza-las de acordo com o paradigma da orientação a objetos.

O transformador intra-domínio, construído no sistema transformacional Draco-PUC, utiliza uma Base de Conhecimento, na qual são armazenados fatos obtidos durante a análise do código fonte do sistema legado, para consultas posteriores para resolução de problemas semânticos quando são aplicadas as transformações.

Além do transformador intra-domínio também foram construídos, no sistema transformacional Draco-PUC, dois outros transformadores inter-domínios. Um que faz a transformação do código fonte segmentado, para descrições da linguagem de modelagem MDL, usada pela ferramenta *Rational Rose* para descrever as técnicas de especificação de requisitos de sistemas em UML [Uml98]. As especificações dos requisitos de um sistema em UML, na ferramenta *Rational Rose*, são persistidas em descrições textuais na linguagem de modelagem MDL. Uma vez transformado o código fonte segmentado para a linguagem de modelagem MDL, pode-se usar a ferramenta *Rational Rose* para recuperar o projeto atual do sistema e reprojetá-lo fazendo a distribuição dos seus objetos, segundo técnicas do método *Catalysis/UML* [Dso98]. Estas técnicas suportam a modelagem distribuída dos objetos do sistema para serem executados em diferentes plataformas de hardware e software. Um outro transformador inter-domínios faz a reimplementação do sistema, reprojetado em *Catalysis/*UML, para uma linguagem orientada a objetos.

Os transformadores construídos no Draco-PUC, são utilizados na estratégia de **Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos**, que parte do código fonte do sistema legado, e tem como saídas o novo projeto do sistema orientado a objetos distribuído e sua reimplementação em uma linguagem orientada a objetos, como mostra a Figura 1. A estratégia é realizada em quatro passos: **Segmentar**, **Transformar**, **Distribuir** e **Reimplementar** Sistema.

No primeiro passo do refinamento, **Segmentar Sistema**, o Engenheiro de Software utiliza o Transformador Intra-Domínio, no sistema transformacional Draco-PUC, para segmentar o Código Fonte do Sistema Legado, ou seja, organizar o código segundo os princípios da orientação a objetos. Como saídas deste passo tem-se o Código Fonte Segmentado e uma Base de Conhecimento, gerada a partir de informações obtidas durante a análise do Código Fonte do Sistema Legado. A Base de Conhecimento é utilizada neste mesmo passo, para a resolução de diferenças semânticas do Código Fonte do Sistema Legado para o Código Fonte Segmentado. Embora utilize o sistema transformacional Draco-PUC e uma Base de Conhecimento, este passo é semi-automático, pois o conhecimento do engenheiro de software é requerido para organizar partes do código legado que não são identificados no transformador.

O segundo passo da estratégia, **Transformar Sistema**, tem como entrada o Código Fonte Segmentado, obtido no passo anterior. Sobre o Código Fonte Segmentado o Engenheiro de Software, utilizando o sistema transformacional Draco-PUC, aplica o Transformador Inter-Domínios da Linguagem do Código Fonte do Sistema Legado para a Linguagem de Modelagem MDL, para transformar o Código Fonte Segmentado em Descrições na Linguagem de Modelagem MDL. A MDL permite representar os elementos que irão compor o modelo orientado a objetos, na ferramenta *Rational Rose*, mas não suporta a representação do comportamento destes objetos, cuja descrição é feita através de métodos. Como o comportamento dos elementos é de grande

importância para a reutilização do sistema, a Linguagem Básica de Ensino (LBE) foi embutida na linguagem de modelagem MDL, a fim de permitir a representação dos métodos. A Base de Conhecimento, gerada no passo Segmentar Sistema, é consultada para solução de diferenças semânticas entre o Código Fonte Segmentado e as Descrições na Linguagem de Modelagem MDL e na LBE.

O terceiro passo, **Distribuir Sistema**, é suportado pela ferramenta *Rational Rose*, na qual as Descrições na Linguagem de Modelagem MDL são importadas, permitindo que o Engenheiro de Software obtenha o projeto atual do sistema. Uma vez obtido o projeto atual, o Engenheiro de Software o reconstrói, usando técnicas do método *Catalysis*/UML [Dso98], para atender novos requisitos, corrigir erros, melhorar desempenho ou mesmo para adaptar às novas plataformas de hardware e software, como por exemplo, reprojetar para uma plataforma distribuída. O método *Catalysis* preocupa-se com a distribuição dos objetos, que é caracterizado pela partição dos dados e das funcionalidades do sistema em uma arquitetura Cliente/Servidor Uma vez distribuídos os objetos do sistema, com as técnicas de *Catalysis*/UML, tem-se o Projeto Orientado a Objetos Distribuído e as Novas Descrições na Linguagem de Modelagem MDL, geradas pela ferramenta *Rational Rose*, para persistência das especificações em Catalisys/UML.

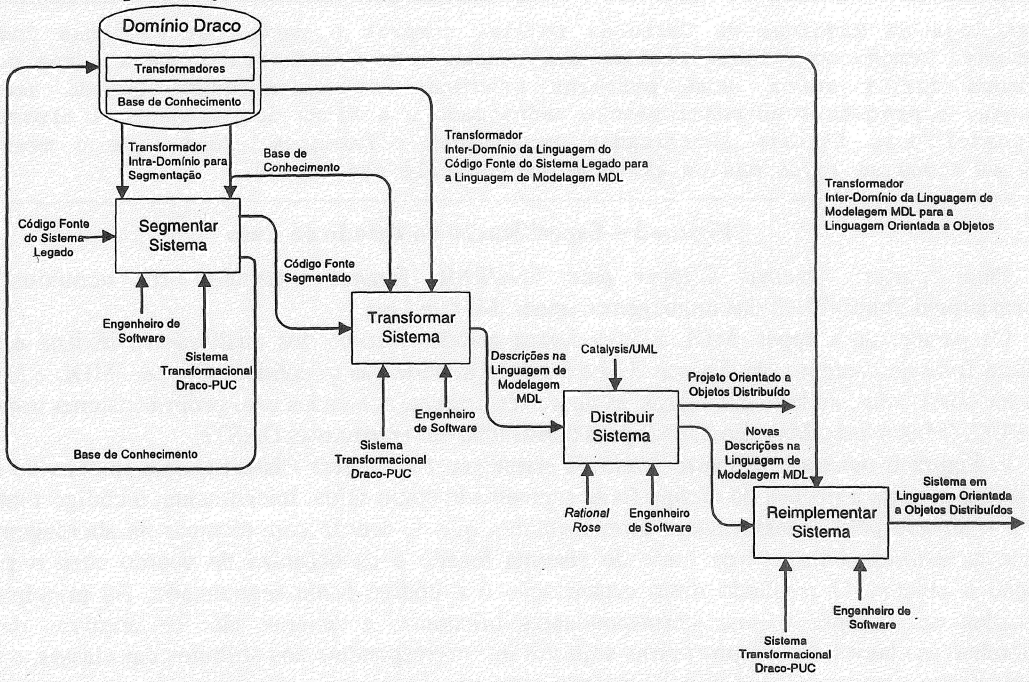

Figura 1 – Refinamento Automático de Sistemas usando Técnicas de Transformação

Finalmente no quarto passo, **Reimplementar Sistema**, o Engenheiro de Software, utilizando o sistema transformacional Draco-PUC, aplica o transformador Inter-Domínios da Linguagem de Modelagem MDL para a Linguagem Orientada a Objetos para transformar as Novas Descrições na Linguagem de Modelagem MDL em uma linguagem orientada a objetos.

Em resumo, tem-se que o Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos, parte do código fonte do sistema legado, para obter um novo projeto orientado a objetos distribuído e a reimplementação em uma linguagem orientada a objetos distribuídos.

4- Estudo de Caso

O Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos foi validado através da reengenharia de sistemas escritos na linguagem Clipper [Vid91, Spe94], transformado-os na linguagem de modelagem MDL e desta para a linguagem Java/RMI [Sun97, Sun98]. A Figura 2 mostra uma descrição resumida do sistema exemplo utilizado na estratégia apresentada. Com a evolução das tecnologias computacionais de hardware e software, foi necessária uma reengenharia deste sistema, para que o mesmo pudesse controlar todas as lojas independente da sua localização física.

A linguagem Clipper foi escolhida para validar a estratégia devido ao grande número de sistemas legados implementados nesta linguagem e por esta não ser orientada a objetos.

Java/RMI foi adotada como linguagem de reimplementação dos modelos orientados a objetos porque: implementa os princípios da orientação a objetos; adequa-se à implementação de sistemas distribuídos e permite a portabilidade entre diversas plataformas, além da crescente popularidade que vem adquirindo junto à indústria.

```
Uma loja de Máquinas de Costuras realiza compras e vendas de máquinas industriais e
domésticas. Quando um cliente realiza uma compra este é cadastrado. Se a forma de pagamento
da compra for a prazo, suas parcelas precisam ser controladas. Quando uma compra é
realizada, o produto é automaticamente cadastrado e a forma de pagamento é armazenada. Esta
loja possui duas filiais localizadas em cidades diferentes. Sendo que o mesmo processo
ocorre em todas as lojas mas os sistemas de controle são locais.
```

Figura 2 – Especificação do Estudo de Caso

Para "portar" sistemas Clipper para Java/RMI, foram construídos três domínios no sistema transformacional Draco-PUC: das linguagens Clipper, MDL e Java.

Os parsers do Clipper, MDL e Java foram gerados através dos analisadores léxicos e sintáticos de gramáticas livre de contexto. As Figuras 3, 4 e 5 mostram parte das gramáticas Clipper, MDL e Java, nas quais podem ser observadas, ao lado das regras, as ações semânticas, colocadas pelo próprio sistema transformacional Draco-PUC, (MakeNode, MakeLeaf, etc.) para construção das respectivas DASTs.

A Figura 6 mostra o primeiro passo da estratégia, **Segmentar Sistema**, com trechos do código fonte legado como entrada e trechos do código fonte segmentado como saída. Inicialmente, o código fonte do sistema legado é analisado pelo Transformador Intra-Domínio, que de acordo com técnicas da abordagem Fusion/RE, identifica as estruturas no código fonte do sistema legado e as organiza de acordo com o paradigma da orientação a objetos. O resultado desta organização é o código fonte segmentado. As principais estruturas identificadas em Clipper e suas correspondentes orientadas a objetos são: os arquivos de dados que correspondem às classes, os campos desses arquivos que correspondem aos atributos das classes, a relação entre os arquivos que corresponde ao relacionamento entre as classes e os procedimentos que correspondem aos métodos.

O comando `USE Cliente` no código fonte do sistema legado, mostrado no lado esquerdo da Figura 6, é usado na abertura do arquivo de dados `Cliente`. Quando este comando é identificado durante o processo de segmentação, é criado um arquivo no código fonte segmentado para armazenar funções que manipulam apenas o arquivo de dados `Cliente`. Assim que este arquivo é criado, são gerados os métodos `Cliente()` e `FinalizaCliente()`, mostrados nos trechos do código do lado direito da Figura 6. O método `Cliente()` faz a declaração e a inicialização das variáveis de memória que irão representar, ao longo do código fonte segmentado, as informações obtidas do arquivo `Cliente`. Já o método `FinalizaCliente()` elimina as variáveis da memória. Outra identificação mostrada na Figura 6 é a do comando `SEEK t_clicod`. Este comando faz a busca por um registro no arquivo de dados `Cliente`, especificado anteriormente pelo comando `SELECT Cliente`, de acordo com o índice de busca especificado pelo comando `SET ORDER TO 1`. Esta identificação dá origem ao método `LocalizaCliente()`, também mostrado no trecho do código do lado

direito da Figura 6, que faz a busca pelo registro que satisfaça o valor da variável `codcli` e retorna a variável `encontrou`, que indica se o registro foi encontrado ou não.

```
%%
  program          : var_decl_star parameter? .nl prog_head+ ;
  { $$=MakeNode("program1", $1, $2, $3, NULL); }
  prog_head        : prog_body
                   { $$=MakeNode("prog_head1", $1, NULL);
                     | '#' .sp diretiva .sp iden .nl
                   { $$=MakeNode("prog_head2", $2, $3, NULL); }
        ...
  statements       : select_stat
                   | use_stat
...
```

```
body_structs       : .slm .lm(+2) body_struct+ (.slm,.slm,)
                   .slm(-2) ;
  decl_func        : FUNCTION' .sp iden '(' ')' .nl
                     var_decl_star parameter?
                     Body_struct? 'RETURN' .sp expr ;
  use_stat         : 'USE' .sp iden? opt_index? alias?
                     use_par? ;
  opt_index        : 'INDEX' .sp id_list .sp ;
  if_stat          : .lm(+2) 'IF' .sp expr .slm body_structs
                   elseif_opt* else_opt? 'ENDIF' .nl
  ...
  %%
```

Figura 3 - Gramática Clipper

```
%%
oNNexpression:   _lst1_objectOrSet  {
$$=MakeNode("oNNexpression1", $1, NULL); } ;
objectOrSet: '(' objectOrSet ')'   { $$=MakeNode("objectOrSet1",
$2, NULL); }
        | entryPoint          { $$=MakeNode("objectOrSet2", $1,
NULL); };
entryPoint: _DT_OBJECT  objectType {
$$=MakeNode("entryPoint1", MakeLeaf($1), $2, NULL); }
        | list  $$=MakeNode("entryPoint2", $1, NULL); };
objectType: petal          { $$=MakeNode("objectType1", $1,
NULL); }
        | state_diagram{ $$=MakeNode("objectType2", $1, NULL);
}
        | selfMessView { $$=MakeNode("objectType3", $1, NULL);
}
        | design { $$=MakeNode("objectType4", $1, NULL); }
        | class_category{ $$=MakeNode("objectType6", $1,
NULL);}
        | class   { $$=MakeNode("objectType7", $1, NULL); }
        | attribute { $$=MakeNode("objectType8", $1,
...
  %%
```

Figura 4 - Gramática MDL

```
  %%
compilation_unit     : package_statement?  Import_statement*
.(.nl .slm , ..slm)  type_declaration* ;
{ $$=MakeNode("compilation_unit1", $1, $2, $3, NULL); }
type_declaration     : class_declaration
            | interface_declaration  | ';' ;
class_declaration    : modifier* 'class' .sp Identifier
            ( 'extends' .sp class_name )?
                     ( 'implements' .sp interface_name++',' '{' .slm
.lm(+2) field_declaration*    .slm(-2) '}'
{ $$=MakeNode("class_declaration1", $1, MakeLeaf($2),
MakeLeaf($3), $4, $5, $7, NULL); }
declaration          : method_declaration |
constructor_declaration ;
var_decl             : modifier* .(,.sp,.sp) type .sp
          variable_declarator++',' .(,,.sp,) ;
constructor_declaration: modifier* .(,.sp,.sp) Identifier
        '(' parameter_list? ')'
        .sp '{' .slm .lm(+2) statement*.slm(-2) '}' ;
modifier             : 'public'| 'private'  | 'protected'
                     | 'static'  | 'final'  | 'native'
...
%%
```

Figura 5 - Gramática Java

No segundo passo, **Transformar Sistema**, o código fonte segmentado é transformado em descrições na linguagem de modelagem MDL. A transformação do código fonte segmentado em descrições MDL é realizada através da aplicação do Transformador Inter-Domínios, que identifica cada estrutura no código fonte segmentado o seu correspondente na linguagem de modelagem MDL.

A Figura 7 mostra, por exemplo, como o comando de abertura do arquivo `Cliente` no código fonte segmentado, `USE Cliente`, dá origem à especificação em MDL da classe `Cliente`, `object Class "Cliente"`. A declaração do método `LocalizaCliente()`, no código fonte segmentado, dá origem à especificação em MDL do método , `object Operation "LocalizaCliente"`. A inicialização da variável `Clicod`, no código fonte segmentado, dá origem à especificação em MDL do atributo, `object ClassAttribute "CliCod"`. A Figura 7 mostra também o mapeamento do código da função `LocalizaCliente()`, escrito na linguagem Clipper, para descrições em LBE, dentro da declaração do método `LocalizaCliente`.

As especificações na linguagem de modelagem MDL, geradas pela transformação do código fonte segmentado, são importadas automaticamente pela ferramenta *Rational Rose*, que permite recuperar o projeto atual do sistema em Clipper. A Figura 8 mostra, por exemplo, como foi obtida a representação da classe `Cliente`, à direita, com seus atributos e métodos, a partir da especificação em MDL à esquerda.

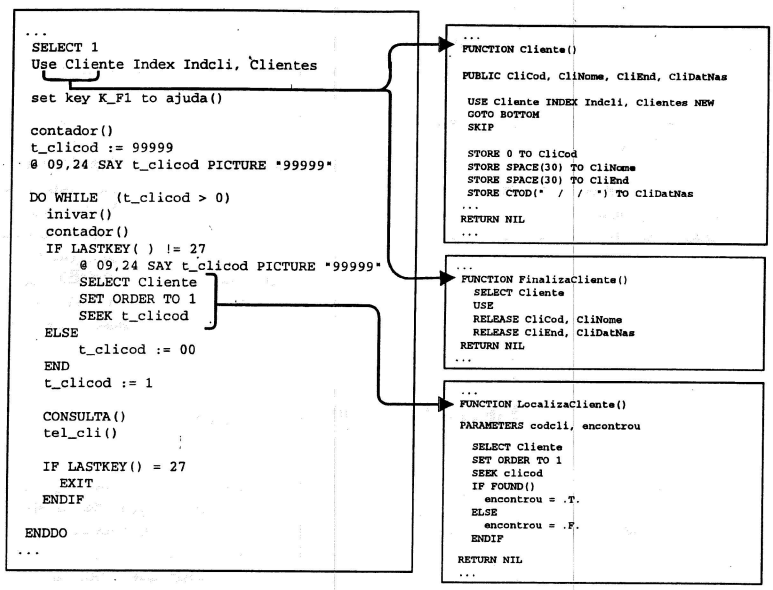

Figura 6 – Segmentação do Código Fonte do Sistema Clipper Legado

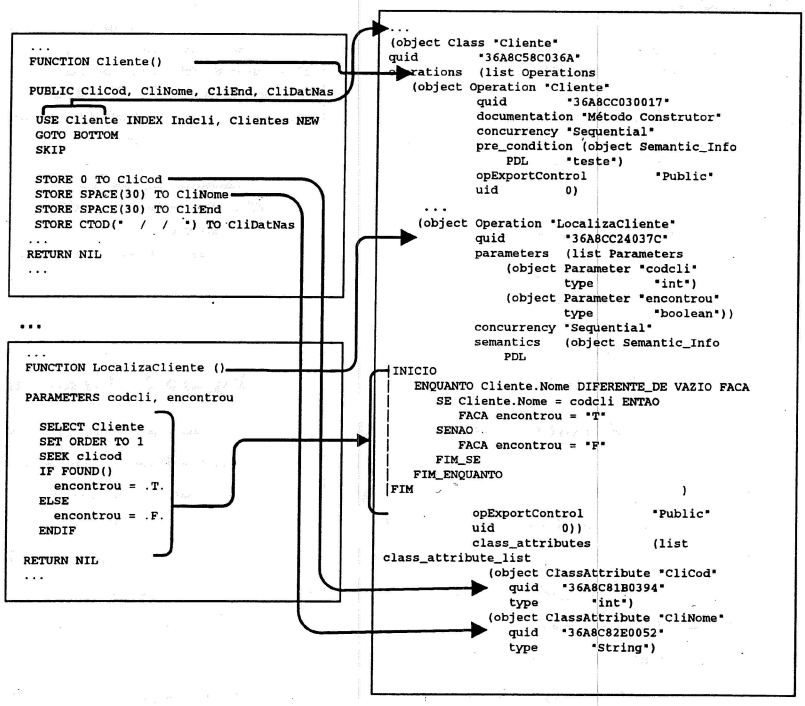

Figura 7 – Transformação do Código Fonte Segmentado para a Linguagem de Modelagem MDL

Em um terceiro passo, **Distribuir Sistema**, o Engenheiro de Software com base no modelo de classes faz o reprojeto do sistema para que as classes sejam distribuídas em um ambiente Cliente/Servidor. As interfaces estendem a interface `java.RMI.Remote` para disponibilizar os métodos remotos das classes servidoras. As classes servidoras, que implementam as interfaces, são conectadas por herança com a classe `java.RMI.server.UnicastRemoteObject`, que gerencia a chamada dos métodos remotos. A Figura 9

mostra o modelo de classes distribuído, onde a classe `ClienteImpl`[1], obtida da classe `Cliente`, implementa `InterfaceCliente`, utilizada na comunicação entre as classes servidora `ClienteImpl` e cliente `CadastrarCliente`.

Ao finalizar este passo, o modelo de classes distribuído é novamente persistido pela ferramenta *Rational Rose* na linguagem de modelagem MDL.

A Figura 10 mostra parte da reimplementação do sistema distribuído na Linguagem Orientada a Objetos Java/RMI da linguagem de modelagem MDL, obtida no passo anterior. No quarto passo, **Reimplementar Sistema**, o transformador Inter-Domínios responsável pela transformação das descrições na linguagem de modelagem MDL para a Linguagem Java/RMI, faz a reimplementação do sistema na linguagem orientada a objetos. Por exemplo, uma classe descrita na linguagem de modelagem MDL pelo comando: `object Class "ClienteImpl"` é transformada para o comando Java/RMI: `public class ClienteImpl`

A linguagem de modelagem MDL permite que especificações sejam inseridas nas pré-condições, pós-condições e semântica dos métodos das classes na ferramenta *Rational Rose*. Para formalizar estas especificações utilizamos a Linguagem Básica de Ensino (LBE) [Pra92, Lim97], que é uma linguagem pseudo-código. Dessa forma, o comportamento detalhado dos métodos em Clipper foram capturados e transformados em LBE. O uso da linguagem LBE combinada com a linguagem de modelagem MDL foi possível, porque o sistema transformacional Draco-PUC permite embutir linguagens em uma linguagem hospedeira, como é o caso da LBE e MDL respectivamente.

```
...
(object Class "Cliente"
quid                "36A8C58C036A"
operations          (list Operations
   (object Operation "Cliente"
        quid                "36A8CC030017"
        ...
        pre_condition       (object Semantic_Info
            PDL             "teste")
        opExportControl     "Public"
        uid                 0)
    ...
    (object Operation "LocalizaCliente"
        quid                "36A8CC24037C"
        parameters          (list Parameters
            (object Parameter "codCli"
                type                "int")
            (object Parameter "encontrou"
                type                "boolean"))
        concurrency         "Sequential"
        semantics           (object Semantic_Info
            PDL                 )
        opExportControl     "Public"
        uid                 0))
        class_attributes            (list
class_attribute_list
          (object ClassAttribute "CliCod"
            quid          "36A8C81B0394"
            type          "int")
          (object ClassAttribute "CliNome"
            quid     "36A8C82E0052"
            type          "String")
          (object ClassAttribute "CliEnd"
            quid     "36A8C83A0168"
            type          "String")
          (object ClassAttribute CliDatNas"
            quid          "36A8C8940117"
            type          "Date")
...
```

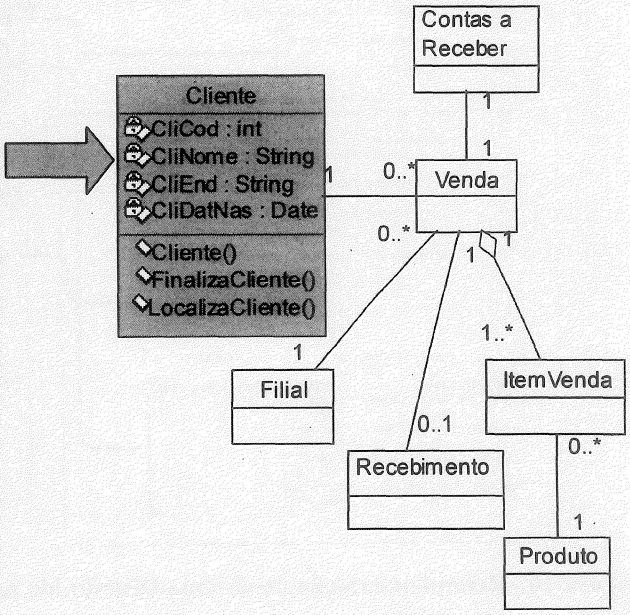

Figura 8– Recuperação do Projeto do Sistema Atual

[1] Sufixo Impl, nomenclatura utilizada por Java/RMI para as classes que implementam uma interface.

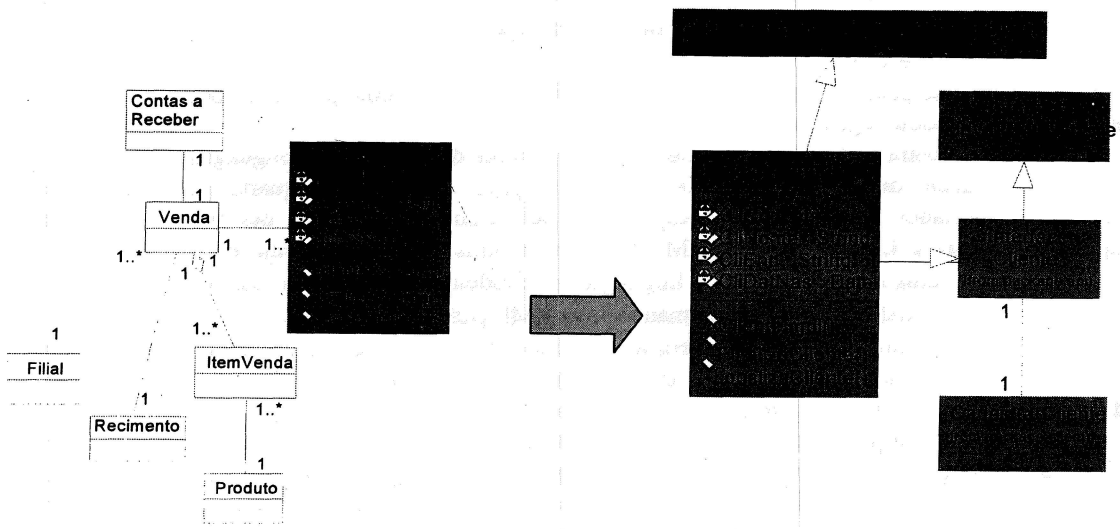

Figura 9 – Reprojeto Orientado a Objetos Distribuídos do Sistema

```
(object Class "ClienteImpl"
 ...
 superclasses(list inheritance_relationship_list
   (object Inheritance_Relationship
            ...
            supplier "java.rmi.server.UnicastRemoteObject"
            ...
            realized_interfaces          (list realize_rel_list
             (object Realize_Relationship
               ...
               supplier  "Cliente"
               ...
 supplier  "Cliente"
 ...
 operations  (list Operations
   (object Operation "ClienteImpl"
            ...
            parameters     (list Parameters
                (object Parameter "CliCod"
                          type          "int")
                (object Parameter "CliNome"
                          type          "String")
                (object Parameter "CliEnd"
                          type          "String"))
...
            semantics      (object Semantic_Info PDL
                           |Inicio
                           |FACA Clicod = Clicod ;
                           |FACA Clinome = CliNome;
                           |FACA CliEnd = CliEnd;
                           |FACA CliDatNas = CliDatNas;
                           |FIM            )
            opExportControl              "Public"
            ...)
 class_attributes          (list class_attribute_list
  (object ClassAttribute "CliCod"
          type           "int")
  (object ClassAttribute "CliNome"
          type           "String")
  (object ClassAttribute "CliEnd"
          type           "String")
  (object ClassAttribute "CliDatNas"
          type           "Date"))
...
```

```
package bemac.server;
import bemac.api.Cliente;
import java.io.Serializable;
import java.util.date;
import java.rmi.server.UnicastRemoteObject;

class ClienteImpl extends UnicastRemoteObject
 implements Cliente, Serializable
{
 private int CliCod;
 private String CliNome;
 private String CliEnd;
 private Date CliDatNas;

 // Construtores
 public ClienteImpl() throws RemoteException
 {
 }

 public ClienteImpl(int CliCod, String CliNome,
String CliEnd) throws RemoteException
 {
  this.CliCod = CliCod;
  this.CliNome = new String(CliNome);
  this.CliEnd = new String(CliEnd);
  this.CliDatNas = new Date();
 }
 public void finalize()
 {
   InterfImplHandler.unregisterImpl(this);
 }
 public Cliente proCliente(int cliCod, boolean
encontrou)
 {
   ...
 }

}
```

Figura 10 –Reimplementação do sistema Distribuído na Linguagem Orientada a Objetos Java/RMI

Por exemplo, a Figura 11 mostra a esquerda uma descrição na linguagem de modelagem MDL, que traz embutido comandos LBE indicados pela "|". Estes comandos LBE são transformados nos comandos Java/RMI mostrados à direita da figura.

Este exemplo ilustra a aplicação das transformações. Entretanto as transformações são escritas no Draco-PUC, com base nas regras gramaticais das linguagens fonte e alvo das transformações. O sistema transformacional Draco-PUC dispõe de uma linguagem própria para facilitar a escrita das transformações, no

nível das regras gramaticais. Na reengenharia de Clipper para Java/RMI, foram escritas 900 transformações que aplicadas no sistema do Estudo de Caso, com aproximadamente 15 mil linhas de código Clipper, geraram em torno de 14 mil linhas do código Java/RMI.

```
object Semantic_Info PDL
|Inicio
|FACA Clicod = Clicod ;
|FACA Clinome = CliNome;
|FACA CliEnd = CliEnd;
|FACA CliDatNas = DataAtual;
|FIM
```

```
{
 this.CliCod = CliCod;
 this.CliNome = new String(CliNome);
 this.CliEnd = new String(CliEnd);
 this.CliDatNas = new Date();
}
```

Figura 11 – Transformação de LBE para Java/RMI

5 – Conclusões

Este artigo apresentou uma estratégia para transformar código fonte de sistemas legados, orientados a procedimentos, para sistemas orientado a objetos distribuídos. Com a abordagem Fusion/RE organizou-se o código fonte do sistema legado com característica do paradigma da orientação a objetos, sem perder sua funcionalidade. A ferramenta para especificação de sistemas orientados a objetos, já consolidada comercialmente, a *Rational Rose*, foi utilizada para recuperar o projeto atual do sistema legado e para reprojetá-lo fazendo a distribuição dos seus objetos. Esta ferramenta persiste a representação gráfica em descrições textuais na linguagem de modelagem MDL. Partindo destas descrições textuais obteve-se, por transformação, a reimplementação automática em Java/RMI. Com os resultados obtidos comprovou-se a viabilidade de se transformar sistemas legados em sistemas orientados a objetos, através da integração da abordagem Fusion/RE e da ferramenta *Rational Rose* com o sistema transformacional Draco-PUC. Esta estratégia de reengenharia, integrando diferentes tecnologias, auxilia os desenvolvedores de software na manutenção do sistema.

A estratégia pode ser aplicada para outras linguagem fontes e alvos da reimplementação, diferente de Clipper e Java/RMI, dada a capacidade do sistema transformacional Draco-PUC trabalhar com múltiplos domínios e dispor de um mecanismo para reconhecer e transformar diferentes comandos e estrutura de seqüência, seleção e repetição das linguagens, incluindo o tratamento de ponteiros e passagem de parâmetros. O Refinamento Automático de Sistemas Legados para Sistemas Orientados a Objetos Distribuídos além de atualizar o sistema para ser executado numa plataforma atual de hardware e software, possibilita a recuperação do projeto e sua modificação para atender a novos requisitos. A granularidade das transformações, no nível das regras gramaticais, garante que a semântica dos comandos da linguagem fonte da transformação seja mantida na linguagem alvo da reimplementação.

Referências Bibliográficas

[Bla98] BLAHA, M and PREMERLANI, W , *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, 1998.

[Bax97] Baxter, I., Pidgeon, C. *"Software Change Through Design Maintenance"*. In Proceedings of ICSM97.

[Boo94] BOOCH, G. *Object-Oriented Analysis and Design whit Applications*. Benjamin/Cummings. 2 edition, 1994.

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

[Col94] COLEMAN D. et al. *Object Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Cor93] Cordy, J.; Carmichel, I. *The TXL Programming Language Syntax and Informal Semantics*, Techinical Report. Queen's University at Kingston - Canada, 1993.

[Dco96] DCOM/ 1.0 Network Working Group. *Distributed Component Object Model Protocol. Nov 1996*

[Dso98] D'SOUZA D., WILLS A., *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.

[Faq98] FAQs–RescueWare. URL: http://www.relativity.com/products/faqs/index.html, 1998.

[Fre87] FREEMAN, P. A Conceptual Analysis of the Draco Approach to Constructing Software Systems. *IEEE Transactions on Software Engineering.* v.se-13, n.7, pp.830-844. July, 1987.

[Fre96] FREITAS, F.G., LEITE J.C.S., SANT'ANNA M. *Aspectos implementacionais de um Gerador de Analisadores Sintáticos para o Suporte a Sistemas Transformacionais.* I Simpósio Brasileiro de Linguagens de Programação. Belo Horizonte - MG, 1996.

[Lei91] LEITE, J.C.S, PRADO, A.F. Desing Recovery - A Multi-Paradigm Approach. *First International Workshop on Software Reusability.* In proceedings, pp.161-169. Dourtmund, Germany. July, 1991.

[Lei94] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA M. Draco-PUC Machine: A Technology Assembly for Domain Oriented Software Development. *3rd International Conference of Software Reuse.* IEEE Computer Society Press. In proceedings, pp. 94-100. Rio de Janeiro, Rio de Janeiro. 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'95.* Recife, Pernambuco. Anais pp. 397-413. Outubro, 1995.

[Lim97] LIMA, M.A.V. et al. Ambiente "CASE" em Múltiplas Visões de Requisitos e Implementação Automática usando o Sistema Transformacional Draco. *XI Simpósio Brasileiro de Engenharia de Software – SBES'97.* Fortaleza, Ceará. Anais, pp. 65-80. Outubro, 1997.

[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.

[Omg98] OMG Article No. Orbos/98-01-18. *Objects By Value* http://www.omg.org/

[Pen96] PENTEADO, R.D. *Um Método para Engenharia Reversa Orientada a Objetos.* São Carlos, 1996. Tese de Doutorado. Universidade de São Paulo. 251p..

[Pra92] PRADO, A.F. *Estratégia de Engenharia de Software Orientada a Domínios.* Rio de Janeiro, 1992. Tese de Doutorado. Pontífica Universidade Católica. 333p.

[Rat98] RATIONAL SOFTWARE CORPORATRION, *Rational Rose 98 Rose Extensibility User's Guide*, 1998.

[Rea92] Reasoning Systems Incorporated. *Refine User's Guide, Reasoning Systems Incorporated*, Palo Alto, 1992.

[Spe94] SPENCE, R. *Clipper 5.2.* São Paulo: MAKRON Books, 1994.

[Sun97] SUN MICROSYSTEMS. *Tutoriais Java*, www.javasun.com, 1997.

[Sun98] SUNSOFT RMI and IIOP in Java. FAG. http://java.sun.com/pr/1997/juni/statement970626-01.faq.html

[Uml98] URL: http://www.rational.com/uml/references

[Vid91] VIDAL, A.G. *Clipper 5.0.* Rio de Janeiro: Livros Técnicos e Científicos Editora, 1991.

[Wie93] Wile, D. *Popart: Producer of Parsers and Related Tools System Builders Manual.* Techinical Report, *USC/Information Sciences Institute*, 1993.