Draco-PUC, Experiências e Resultados de Re-Engenharia de Software

Julio Cesar Sampaio do Prado Leite* Antonio F. Prado†
Marcelo Sant'Anna‡
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente, 225.
Rio de Janeiro 22453

Resumo

A re-engenharia de software é um processo distinto da engenharia tradicional porque parte de um desenho já existente. Tendo como ponto de partida um desenho, a re-engenharia pode explorar oportunidades de reutilização. Como normalmente não se dispõe de um desenho que reflita o artefato que se quer refazer, a recuperação de desenho ou engenharia reversa passa a ser peça fundamental na re-engenharia. Este artigo relata nossa experiência na re-engenharia da máquina Draco, e concentra sua atenção na re-engenharia do componente analisador sintático da máquina Draco. A máquina Draco é uma instanciação do paradigma Draco, um dos novos paradigmas para desenvolvimento de software. O paradigma Draco distingue se pelo uso de técnicas de transformação aliadas a tecnologias de compilação, procurando focar o desenvolvimento de software na reutilização de componentes.

1 Introdução

Um fato que é consenso na comunidade de engenharia de software é o de que a manutenção de software resulta em comprometimento de grande parte dos recursos alocados ao ciclo de vida de um software. A literatura de manutenção de software, conforme relata Souza [Souza 91], é unânime nesta constatação. Em função desta realidade, propostas que possam reduzir estes custos têm grande importância.

O problema da manutenção agrava-se com o passar do tempo, na medida em que mais sistemas entram em manutenção e na medida em que cada vez mais cresce a demanda por recursos computacionais de suporte a sistemas de informação. Em função da necessidade de prover métodos eficazes para o desenvolvimento de software, e da dificuldade desse problema, o aspecto específico de manutenção é, muitas vezes, relegado a um segundo plano [Parikh 88, Vallabhaneni 87]. O interesse crescente por métodos de apoio à manutenção, fez

*Apoio da SCT-PR e do Cnpq
†Apoio do IME, Ministério do Exército
‡Apoio do CNPq Bolsa de Iniciação Científica

com que a comunidade de pesquisa procurasse investigar este aspecto. A maior parte destas pesquisas utiliza se de alguma forma de engenharia reversa e reutilização para aumentar a produtividade da tarefa de manutenção [Biggerstaff 89], [Chikofsky 90], [Rugaber 90] e [Baxter 90].

Neste artigo descrevemos um caso de re-engenharia, suas particularidades, interessantes por si próprias, e a estratégia de recuperação utilizada. Esta estratégia proposta por Prado [Prado 92] apresenta características inovadoras principalmente porque se utiliza dos conceitos do paradigma Draco. Através deste artigo reportamos a estratégia de re-engenharia e descrevemos detalhes de implementação da máquina Draco, principalmente no que se refere a parte de tecnologia de compilação. O artigo está organizado em cinco Seções. Na Seção 2 apresentamos um breve sumário sobre o paradigma Draco e sobre a máquina Draco. Na Seção 3 descrevemos a estratégia utilizada na re-engenharia. Na Seção 4 detalhamos como foi possível adaptar a forma LALR, do Yacc/Bison [Bison 90], na máquina Draco, através da re-engenharia do sistema analisador da Draco-PUC. Concluímos, Seção 5, ressaltando o que aprendemos na condução do estudo de caso, bem como delineando trabalhos futuros na área de re-engenharia.

2 Draco, O Paradigma, e a Máquina Draco-PUC

O paradigma Draco é um paradigma para a construção de software. Este paradigma fundamenta-se na reutilização de componentes através de um esquema baseado em conhecimento. Sua principal característica é a distinção e identificação, não só dos diferentes tipos de conhecimento, bem como dos diferentes níveis de abstração presentes no processo de construção de software. Em Draco a reutilização ocorre no nível de modelagem do problema, já que estes são especificados por linguagens próprias do problema (domínio).

O paradigma Draco foi originalmente concebido por Jim Neighbors [Neighbors 84], utilizando-se de idéias de reutilização e de transformação de programas. As proposições de Neighbors se materializaram em um protótipo de nome Draco, que por muito tempo se confundiu com o paradigma. Pesquisas subsequentes realizadas pelo grupo do professor Peter Freeman na Universidade da California, Irvine ampliaram os conceitos presentes em Draco e difundiram o que hoje identificamos como o paradigma Draco [Freeman 87], [Arango 88].

No paradigma Draco, o "conhecimento" pode ser dividido em duas grandes categorias: conhecimento do domínio em questão e conhecimento sobre o processo de desenvolvimento de software. Abstraindo se que o conhecimento do processo também pode ser expresso por domínios, têm-se basicamente que o conhecimento do domínio é embutido na descrição sintática da linguagem (gramática) e na descrição da semântica da linguagem (componentes), enquanto o conhecimento da engenharia de software está embutido em transformações e nas táticas para aplicação dos refinamentos entre domínios. O ganho de produtividade prometido pelo paradigma se fundamenta na reutilização de descrições em alto nível de abstração, isto é do domínio. Reutilizando se descrições de alto nível e mantendo uma separação entre conhecimento do domínio e conhecimento da engenharia de software, obtém se o reuso da definição, do desenho e do código. Base para esta reutilização em larga escala é o conceito de **rede de domínios**, isto é, uma quantidade de domínios suficiente para que cada novo domínio possa utilizar conhecimentos já previamente codificados sob a forma de domínio.

A máquina Draco é uma instância do paradigma. Esta máquina pode ser vista como

um gerador de geradores de aplicação, isto é, fornece facilidades para a definição de um gerador e para o seu uso. Domínios em Draco são vistos como linguagens com semântica e sintaxe bem definidas. A semântica é implementada por um conjunto de componentes e a sintaxe é definida por uma gramática própria da linguagem. Para possibilitar a definição de domínios a máquina Draco dispõe de subsistemas que constroem: um analisador sintático, um formatador de árvore interna, uma biblioteca de transformações, e uma biblioteca de componentes.

Dispondo-se da descrição de um domínio (gerador) composto basicamente por estas quatro partes acima, a aplicação deste domínio, isto é, a produção de sistemas de software, é feita por quatro subsistemas a saber:

- Análise Sintática: o subsistema de análise sintática, ao analisar um programa, cria a árvore interna, representação base em Draco.
- Formatação: o subsistema que exibe uma árvore interna na forma original da linguagem do domínio.
- Transformação: o subsistema que efetua as transformações horizontais sobre a árvore interna. Estas transformações têm o objetivo de manipular as expressões escritas em um domínio.
- Refinamento: o subsistema que efetua as transformações verticais sobre a árvore interna. Estas transformações permitem que se mude de nível linguístico, ou seja através delas se descreve parte de um domínio na linguagem de um outro domínio da rede de domínios.

Concebida e implementada por Jim Neighbors em UCI Lisp [Neighbors 84], a máquina Draco foi posteriormente portada para o Franz Lisp [Arango 86]. De forma a possibilitar a recuperação do desenho de Draco [Leite 91], construímos a primeira versão, 0.1, da máquina Draco-PUC em Scheme (Common Lisp). Esta versão reproduzia fielmente a máquina original, só com pequenas mudanças na parte de interface. Na tarefa de recuperação do desenho e na própria utilização da versão 0.1 foram observadas vários pontos problemáticos com a máquina. Um deles era o sistema de análise sintática, extremamente lento. Com base nessas observações e utilizando-se uma estratégia para re-engenharia [Prado 92], fez-se a re-engenharia da máquina Draco, que resultou na Draco-PUC, versão 1.0. Esta nova versão combina um pacote *off-the-shelf* com a linguagem C e ainda mantém partes em Lisp (Scheme). Este artigo reporta, em detalhes, a re-engenharia da parte que cuida das gramáticas de domínios.

3 Re-Engenharia, Definições e Estratégia

A re-engenharia de um artefato de software é composta de duas partes distintas:

- recuperação do desenho e
- reutilização do desenho na tarefa de re-construção do software.

A recuperação do desenho, segundo Biggerstaff [Biggerstaff 89], recria abstrações de desenho segundo uma combinação de: código, documentação existente, experiência pessoal, e conhecimento geral do problema e da aplicação. A recuperação de desenho caracteriza a engenharia reversa, isto é, a arquitetura de um sistema é montada a partir de um processo

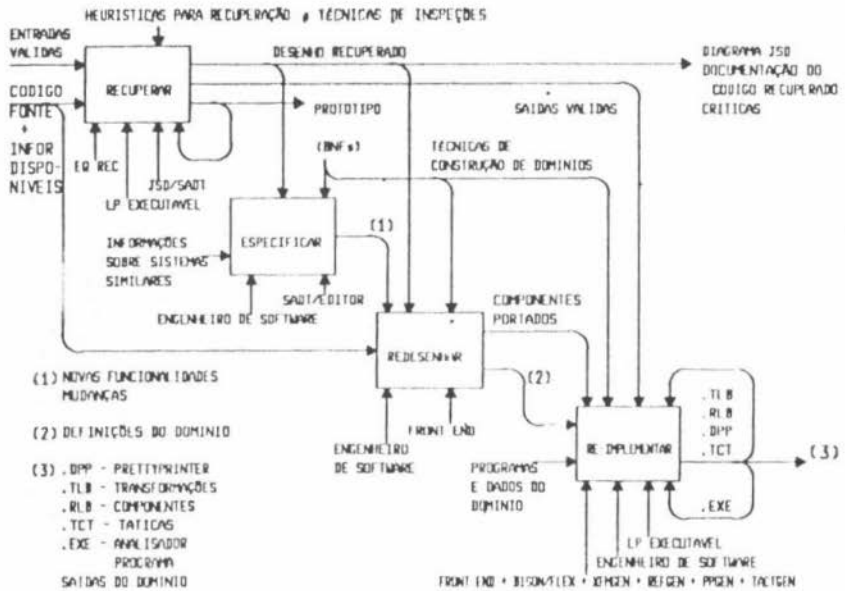

Figura 1: Estratégia de Re-engenharia de Software Orientada a Domínios

ão arqueologia. Este processo é em grande parte do tipo *bottom-up* onde aspectos de modelagem referentes a agregação, generalização e associação estão envolvidos. Normalmente, a principal fonte de informação é o código, sendo que, a disponibilidade de outras fontes é dependente de cada caso em particular.

A reutilização do desenho no processo de re construção do artefato acontece de acordo com as necessidades de nova funcionalidade do artefato. Esta reutilização não necessariamente implica na reutilização total da arquitetura recuperada, nem do código que a implementa. A reutilização a nível de desenho permite uma maior liberdade na escolha das implementações, isto é, o código existente pode ser inteiramente reutilizado, ou modificado, ou substituído por novo código.

3.1 Visão Geral da Estratégia

A Figura 1, um modelo SADT [Ross 77], apresenta a Estratégia de Re Engenharia de Software Orientada pelo Paradigma Draco. Esta estratégia proposta por Prado é composta de quatro atividades, recuperar, especificar, redesenhar, e re implementar.

Partindo do código fonte de um sistema, a estratégia combina a técnica de inspeções com

técnicas de modelagem, como JSD [Jackson 83] e SADT, para recuperar o desenho. Utilizando a versão recuperada, observando críticas ao desenho recuperado e fazendo-se um estudo mais detalhado do domínio do problema, pode se, então, especificar mudanças a serem introduzidas no desenho recuperado. Em seguida, utilizando estas especificações, processa-se o redesenho, definindo as partes do problema segundo os conceitos do paradigma Draco. Finalmente, o novo desenho é re-implementado utilizando-se uma liguagem executável, que pode ser uma linguagem diferente da linguagem original.

3.2 Recuperação do Desenho

No passo recuperar faz-se a recuperação do desenho do software a partir do seu código fonte e de outras informações disponíveis, usando uma estratégia [Leite 91] que combina técnicas de inspeções, prototipação e Jackson System Development - JSD [Jackson 83] e SADT para representar o desenho. A prototipação pode ser usada neste passo caso o software não esteja operacional para o ambiente de software e hardware existente. O objetivo principal da recuperação é obter o desenho do código que servirá para conhecer seu funcionamento e orientar a especificação de mudanças e novas funcionalidades e o seu redesenho. O conhecimento do desenho do software permitirá a deteção de falhas e problemas existentes no código que está sendo recuperado.

O processo é iniciado dividindo-se o sistema a ser recuperado em subsistemas. Esta divisão facilita a compreensão e entendimento do código fonte, de onde serão retiradas as informações do desenho em todos os níveis de detalhes exigidos para sua re-engenharia. Os subsistemas e suas partes podem ser identificados através de uma referência cruzada dos procedimentos que compõem o sistema que está sendo recuperado. Um *cross-analyzer* mostra quais procedimentos são chamados por cada procedimento e quais chamam determinado procedimento. Embora o escopo de cada subsistema não fique totalmente claro inicialmente, a referência cruzada, complementada pelo conhecimento sobre algumas entradas e saídas do sistema, permite uma primeira divisão do problema em subsistemas. Desta maneira é dado início ao processo de recuperação. Tão logo vai-se mergulhando mais a fundo no código pode se obter um maior conhecimento sobre o papel de cada subsistema. Para recuperar cada subsistema, executa-se a inspeção do código, prototipação para operacionalizar o código caso este não esteja operacional e validação do desenho recuperado.

A inspeção do código é executada em 4 ações:

- Visão Geral e Preparação

  Os participantes da inspeção, auxiliados pela documentação existente, executam a árdua tarefa de ler e entender o código existente. Para entendimento do código usa-se um processo tipicamente *bottom-up* artesanal que parte do nível de simples comandos e estruturas que constituem sua arquitetura, até um entendimento global de cada procedimento. O desenho recuperado dos procedimentos é expresso por Diagramas de Estrutura JSD (Figura 2). Este processo de recuperação é auxiliado pela análise detalhada do código, como, por exemplo, a expressa na Figura 3.

- Inspecionar

  Na atividade Inspecionar descreve-se cada procedimento do subsistema ou parte que está sendo recuperada, e formulam-se questões, baseadas em checklist e heurísticas de recuperação, sobre as falhas, erros ou dúvidas encontradas na Preparação.

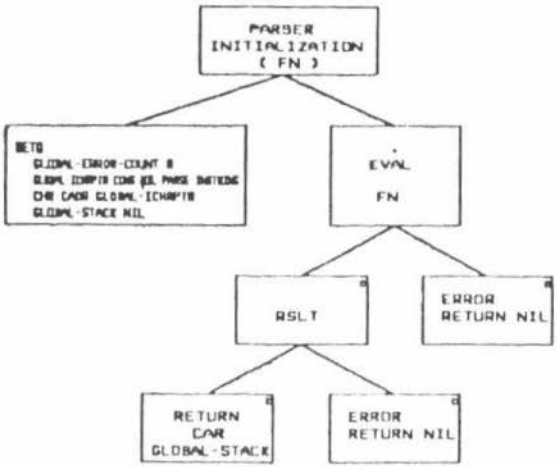

Figura 2: Diagrama de Estrutura JSD de procedimento recuperado

VERSÃO 0

PARSER-SCANCHR

```
(De Parser-Scanchr NIL
    (Setq Global Ichrptr (Cdr Global Ichrptr))
    (Or (Cdr Global Ichrptr) (Rplacd Global Ichrptr
                                (Parse Instring 1 )))
    (Setq Chr (Cadr Global Ichrptr))
    (Cond ((Zerop (*Dif Chr 10 )) (Parser-Get Line)))
    T)
```

```
(Define (Parser Scanchr)
  (Begin
    (Set' Global Ichrptr (Cdr Global Ichrptr))
    (If (Null' (Cdr Global Ichrptr))
        (Set Cor' Global Ichrptr
        (Parse Instring Global Input Port)))
    (Set' chr (Cadr Global IchrPtr))
    (If (Eq' Chr 10)
        (Parser Get Line))
    = 1))
```

* Documentação

Este procedimento pega o proximo caracter
na variável Global Ichrptr e
coloca-o na variavel Chr.

Figura 3: Exemplo de análise de código

- Resolver Erros ou Problemas

  Corrigem-se os erros relatados na Inspeção, gerando uma nova versão do desenho recuperado.

- Validar Versão

  Verifica-se então se todos os erros e problemas descobertos foram resolvidos corretamente. O resultado deste passo é uma versão validada do desenho do código. O desenho do código neste subpasso corresponde a uma representação textual e gráfica de cada procedimento que compõe o código.

Uma vez estando validados os desenhos obtidos de cada procedimento (processo) constrói-se o SSD(System Specification Diagram), ou seja, o diagrama de rede da metodologia JSD. Este diagrama registra o entendimento global do subsistema pela integração dos vários procedimentos. Os vários processos podem ser ligados por fila de mensagens ou por vetores de estado. Argumentos são considerados fila de mensagens e as variáveis globais são transformadas em um entidade denominada **globalvar**, a partir da qual todas as referências são descritas por filas de mensagens.

3.3 Especificação de Mudanças e Novas Funcionalidades

No passo **especificar** descrevem-se textualmente e graficamente, usando SADT, as mudanças e melhorias a serem introduzidas no redesenho do software recuperado, em função das críticas observadas em **recuperar** e do desenho recuperado. O estudo de sistemas similares ao que está sendo recuperado alimenta o processo com novas informações, e as técnicas de construção de domínios orientam o engenheiro de software na especificação correta das mudanças e novas funcionalidades do sistema.

Para especificar mudanças e novas funcionalidades para o software, que reduzem o uso de CPU, memória e outros recursos computacionais limitados, o engenheiro de software pode se orientar por heurísticas [Prado 92]. Abaixo listamos três destas heurísticas.

a) Procure identificar exatamente quais as partes do sistema que causam as deficiências do seu funcionamento. Simples *traces* e ferramentas de depuração podem ser úteis na identificação dos problemas;

b) Não exagere com as melhorias e com o uso de técnicas altamente sofisticadas. Especificações mais simples em geral produzem melhores resultados.

c) Lembre-se de que sistemas modulares mais coesos e com fraco acoplamento são fáceis de serem entendidos e simplificam a tarefa da manutenção.

3.4 Re-Desenho do Software Recuperado

O redesenho do software é dito orientado a domínios porque faz sua reconstrução representando-o através das partes que compõem um domínio Draco, ou sejam: o analisador, as transformações, os componentes e o prettyprinter. Conforme se pode ver na Figura 1 em **redesenhar**, as novas funcionalidades e mudanças estabelecidas no passo de especificação, as informações obtidas do desenho recuperado e as técnicas de construção de domínios orientam o processo do redesenho, cujas entradas são o código fonte do software e as informações sobre as entradas e saídas disponíveis.

Das quatro partes que representam um domínio, seu analisador, definido através de sua gramática, é sem dúvida a mais importante, pois serve de base para as outras partes.

Também importante é a biblioteca dos componentes envolvidos no processo do redesenho, os quais apesar de terem o mesmo objetivo podem ser construídos e utilizados de formas diferentes. Assim, tem-se componentes que expressam a semântica de um domínio em outro domínio, definidos na sintaxe do Draco.

Eventualmente, pode-se usar o código fonte original como entrada do passo **redesenhar** para executar o **porte** de componentes diretamente para outra linguagem (linguagem alvo) para a qual o software está sendo reconstruído. O **portè** de componentes para linguagens executáveis, é realizado automaticamente ou manualmente usando parte ou toda a equipe de recuperação do desenho. Os componentes portados normalmente residem numa biblioteca de uso comum e podem ser usados para completar a semântica do código que está sendo redesenhado orientado a domínios. Numa forma mais completa o redesenho do software é representado pelas definições de seu analisador, do prettyprinter, das transformações e dos componentes, tarefas estas realizadas em 5 passos.

1. definir analisador
2. definir componentes e táticas
3. portar componentes
4. definir transformações
5. definir prettyprinter

No redesenho, além do próprio domínio do software objeto da recuperação, deve ser definido e construído, caso ainda não esteja disponível, o domínio alvo, para o qual o software será redesenhado e re implementado (por exemplo a linguagem C). O domínio alvo é necessário porque permitirá a construção dos componentes que farão a mudança de domínio e a exibição dos programas refinados para a linguagem deste domínio alvo. São partes obrigatórias do domínio alvo seu analisador e seu prettyprinter. O primeiro para analisar os componentes, cujas semânticas são expressas na linguagem do domínio alvo, e o segundo para imprimir os programas refinados, na linguagem externa do domínio alvo [1].

3.5 Re-Implementação

Conforme se pode ver na Figura 1, no passo **re-implementar** o analisador, o prettyprinter, as transformações de otimização, os componentes e táticas são construídos a partir das definições obtidas no redesenho. As técnicas de construção de domínios juntamente com as BNFs de domínios que implementam a própria máquina Draco orientam o engenheiro de software neste passo. Os editores da interface da Draco PUC facilitam e guiam o processo de construção dos domínios.

Uma vez dispondo do domínio, o engenheiro de software pode construir sistemas escritos na linguagem deste domínio, os quais serão otimizados e refinados para outras linguagens de diferentes domínios já disponíveis. A execução de testes com o software reconstruído na nova linguagem permite validar se a re engenharia orientada a domínios re implementou toda semântica do desenho recuperado.

[1] Vale ressaltar que uma re engenharia feita por porte não necessita recuperar o desenho do software, bastando apenas uma ligação entre o domínio fonte e o domínio alvo.

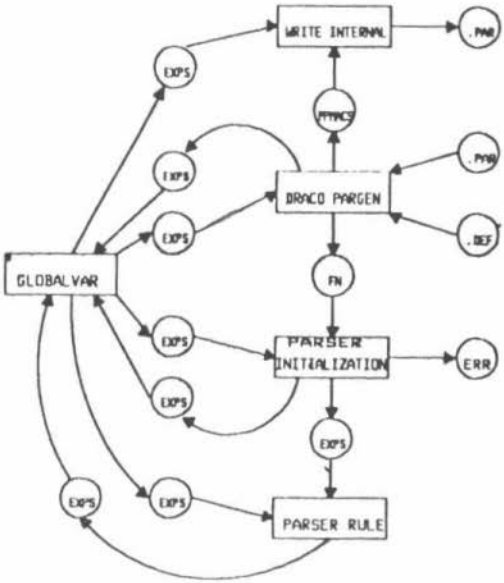

Figura 4: Diagrama SSD/JSD do subsistema PARGEN recuperado

4 Adaptando o Analisador LALR(1) na Arquitetura Draco-PUC, Detalhes da Re-engenharia

A re-engenharia da máquina Draco, cujos subsistemas são usados como mecanismos na estratégia, seguiu o método apresentado acima e renovou partes do Draco-PUC versão 0.1. Algumas dessas partes, como o subsistema construtor de transformações e o subsistema construtor de táticas foram redesenhadas usando na totalidade a representação de domínios Draco.

A re-engenharia do PARGEN (gerador de analisadores da máquina original) e dos *parsers* (subsistema PARSE máquina original, responsável pela análise sintática e léxica de programas de domínio), recuperados em JSD, aplicou a estratégia de uma forma diferenciada. Na especificação utilizou-se de conhecimentos da literatura e do uso de sistemas similares. O redesenho utilizou um sistema já pronto e estabeleceu aspectos de integração, cabendo à re-implementação ligar as implementações do restante da máquina com o pacote analisador. Desta maneira substituiu-se integralmente os subsistemas PARGEN e PARSE.

4.1 Recuperando PARGEN-PARSE

Analisando cada uma das partes da máquina Draco, verificou-se que era possível separar o conjunto de procedimentos de software referentes a cada uma dessas partes. Observou-se, também, que o acoplamento dos módulos que implementam cada parte era muito fraco, podendo ser feito por meios de arquivos, o que tornava o processamento dessas tarefas independente. Esta observação mostra que a máquina Draco original era bem estruturada neste nível de abstração.

As fronteiras do subsistema PARGEN foram delimitadas pelo conjunto de procedimentos responsáveis pela geração de analisadores LL(1), com ações semânticas associadas às regras para construção da forma interna do Draco. Uma vez tendo sido definidos os procedimentos que implementavam o subsistema fez-se a recuperação do seu desenho seguindo os passos da estratégia recuperar, produzindo o diagrama da Figura 4.

4.2 Especificação de Mudanças no Analisador

Na versão inicial da máquina Draco-PUC facilmente pôde-se observar os problemas, tanto de dificuldades para definir gramáticas dos domínios a serem construídos, uma vez que estas estavam limitadas à classe LL(1), como também de performance extremamente fraca dos analisadores construídos pelo PARGEN. Prado [Prado 90] estudou vários analisadores disponíveis, não só para avaliar cada um, como também para caracterizar melhor este domínio.

Baseado neste estudo e na estrutura do desenho recuperado verificou-se a viabilidade de integrar um domínio externo, um gerador de analisador do tipo LALR(1), ao restante da máquina Draco, em substituição ao gerador de analisador *top-down* LL(1). Assim especificou-se como uma das mudanças na reconstrução da máquina a integração dos sistemas geradores de analisadores Yacc/Bison e Lex/Flex ao sistema Draco. Dessa forma não só se substituiu o gerador LL(1) por um LALR(1), como também se dividiu a parte antes única, em duas sub-partes bem distintas, a análise léxica, feita pelos geradores Lex/Flex, e a análise sintática propriamente dita.

Também com base neste estudo [Prado 90] foi especificado uma interface orientada a gramáticas. Semelhantemente a outros projetos de editores orientados a gramáticas [Normark 87], a interface da Draco-PUC foi especificada com o intuito de prover facilidades de edição e verificação de consistência que auxiliam a construção de analisadores.

4.3 Re-Desenhando PARGEN-PARSE

De acordo com as especificações apresentadas, o gerador dos analisadores, que no Draco é constituido pelo gerador LL(1) PARGEN, é substituido por um gerador LALR(1). Com o intuito de se identificar mais claramente as partes que compõem o domínio redesenhado é apresentada na Figura 5 a gramática utilizada pelo Yacc/Bison.

Os componentes foram definidos através do porte manual dos procedimentos responsáveis pela construção da forma interna Draco. Estes componentes foram portados de Lisp para a linguagem alvo C, usando uma biblioteca de funções escritas em C que simulam a funcionalidade das primitivas Lisp. A figura 6 mostra um destes componentes portados.

A interface, especificada e desenhada em JSD, fez o redesenho não só do código que implementa as entradas e saídas dos subsistemas PARGEN e PARSE bem como também das entradas e saídas de todos os demais subsistemas da máquina Draco PUC.

```
espec : definicoes MARCA regras final ;

final : /* vazio */
      | MARCA ;

definicoes : /* vazio */
           | definicoes definicao ;

definicao : LCHAVE /* ações semânti   */ RCHAVE
          | UNION '{' /* definições de tipos */ '}'
          | START ID
          | EXPECT NUMBER
          | PURE
          | reservada tagdef lista ;

reservada : TOKEN
          | LEFT
          | RIGHT
          | NONASSOC
          | TYPE ;

tagdef : /* vazio */
       | TIPO ;

lista : seqlista
      | lista seqlista
      | lista ',' seqlista ;

seqlista : ID
         | ID NUMBER ;

regras : ID ':' ladodireito precedencia
       | regras regra ;

regra : ID ':' ladodireito precedencia
      | '|' ladodireito precedencia ;

ladodireito : /* vazio */
            | ladodireito ID
            | ladodireito acoes ;

acoes : '{' /* ações semânticas */ '}' ;

precedencia : /* vazio */
            | precedencia ';'
            | PRECID ID
            | PRECID ID acoes ;
```

Figura 5: Gramática do gerador de analisador LALR(1)

```
/* Adiciona as agendas com transformações de otimização
   na forma interna */
void AgendaAdd(code, name, alistc)
lisp expression code, name, alistc;
/* code: Codigo da transformação. Exemplo 99
   name: Transformação. Exemplo: PARPAR
   alistc: Lista com transformações. Exemplo (() (PARGEN ("PVAR" Z)))
*/
{
  declare_lx(tptr), declare_lx(cptr), declare_lx(temp1), declare_lx(temp2),
  arg_lx(code), arg_lx(name), arg_lx(alistc);

  set(temp1, car(alistc));
  set(tptr, assoc(code,temp1));
  if (*nullp(tptr)) {
    if (!atom(name)) {
       set(temp1, cdr(tptr));
       set(temp1, insert(name, temp1, lexorder, 1));
       set_cdr(tptr,temp1);
    }
    else if (*atomp(name)) {
         set(temp1, cdr(tptr));
         set(temp1, cons(temp1, NIL));
         set(cptr,temp1);
         set(temp2, cadr(name));
         set(temp1, car(name));
         AgendaAdd(temp1,temp2,cptr);
         set_cdr(tptr,car(cptr));
    }
  }
  else {
    set(temp2, car(alistc));
    set(temp1, list(code, name, NIL));
    set(temp1, insert(temp1, temp2, appcat, NIL));
    set_car(alistc,temp1);
  }
  * Libera memoria. Retorna com as agendas colocadas em
    alistc. No caso do exemplo: (((99 PARPAR) (PARGEN ("PVAR" Z)))
  */
  free_lx(tptr); free_lx(cptr); free_lx(temp1); free_lx(temp2);
  free_arg_lx(code); free_arg_lx(name); free_arg_lx(alistc);
}
```

Figura 6: Componente portado do domínio PARGEN

4.4 Re-implementando PARGEN-PARSE

Com a biblioteca de componentes de software criada e dispondo se dos sistemas geradores de analisadores Yacc/Bison e Lex/Flex foram então re implementados os subsistemas que constroem analisadores e analisam programas para construção da forma interna Draco.

A interface, redesenhando as entradas e saídas da máquina foi implementada em C, possibilitando a integração entre o pacote Yacc/Bison Lex/Flex e os componentes portados completando o trabalho de re engenharia dos subsistemas PARGEN PARSE.

Na versão 1.0 da máquina Draco PUC já encontramos este sistema híbrido funcionando. Desta forma, um domínio Draco hoje tem sua sintaxe definida pela gramática Yacc/Bison e terá como entrada um padrão já bastante usual na comunidade.

5 Conclusões e Trabalhos Futuros

Nesse trabalho apresentamos uma estratégia de re-engenharia que se utiliza do paradigma Draco para representar novos desenhos em software. Esses novos desenhos são expressos por uma combinação de gramáticas e de componentes que descrevem a semântica dessas gramáticas. Esta estratégia vem sendo usada e aprimorada na própria recuperação da máquina Draco, máquina esta que objetiva implementar este paradigma. Mostramos a recuperação dos subsistemas de Draco, PARGEN e PARSE, e detalhamos de que maneira este foi reconstruído na nova versão do Draco-PUC, versão 1.0.

É importante ressaltar que a presente estratégia, usou como validação a máquina Draco. Portanto, utilizamos um método científico em que há uma auto aplicação dos conceitos. Esta validação é assemelhada a aquelas em que linguagens são usadas para descreverem a si próprias.

Nossos estudos em torno do paradigma Draco e de sua implementação continuam. São muitos ainda os pontos que precisam de mais pesquisas, principalmente na parte de utilização da máquina, isto é na construção de domínios. Uma das tarefas agendadas é a re-engenharia da atual interface para que esta seja organizada também como um domínio Nosso projeto é continuar a melhorar a máquina Draco e desenvolver os domínios básicos para que se possa realmente validar as promessas do paradigma.

Referências

[Arango 86] Arango G., Baxter I., Freeman P., Pidgeon C., *A Transformation-Based Paradigm of Software Maintenance*, IEEE Software, Vol. 3, pp. 27-39, May 1986.

[Arango 88] Arango G., *Domain Engineering for Software Reuse*, PhD. Dissertation, Dept. Of Information and Computer Science, University of California, Irvine, 1988.

[Baxter 90] Baxter, I. *Transformational Maintance by Reuse of Design Histories*. Tese de Doutorado, University of California, Irvine, USA; Nov., 1990.

[Biggerstaff 89] Biggerstaff, T. *Design Recovery for Maintance and Reuse*, IEEE Computer, 22(7); Jul., 1989; pags. 36-49.

[Bison 90] Bison, *Users Guide*, Free Software Foundation, Boston, Mass. 1990.

[Chikofsky 90] Chikofsky, E. e Cross II, J. *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software; Jan., 1990; pags. 222-240.

[Freeman 87] Freeman, P. Software Reusability, IEEE - Computer Society, March 1987.

[Jackson 83] Jackson, M. *System Development*, Prentice Hall International; 1983.

[Leite 91] Leite, J.C.S.P. e Prado, A.F. *Design Recovery - A Multi-Paradigm Approach*, First International Workshop on Software Reusability, Dormund, Germany; Jul., 1991.

[Parikh 88] Parikh, G. *Techniques of Program and System Maintance* (2a. edição), QED Information Sciences, Inc., 1988.

[Neighbors 84] Neighbors J., *The Draco Approach to Constructing Software from Reusable Components*, IEEE Trans. on Software Engineering, SE-10:564-573, September 1984.

[Normark 87] Normark K. *Transformations And Abstract Presentations In A Language Development Environment User Manual*, Aarhus University, Denmark, 1987.

[Prado 90] Prado, A.F., Um Estudo de Analisadôres Sintáticos, *Monografias em Ciência da Computação, Departamento de Informática, PUC/RIO*, 1990.

[Prado 92] Prado, A.F., Estratégia de Re Engenharia de Software Orientada a Dominios, Tese de Doutorado, *Departamento de Informática, PUC/RIO*, 1992.

[Ross 77] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. *In Tutorial on Design Techniques, Freeman and Wasserman (ed.)* IEEE Catalog No. EHQ 161-0(1980), 107-125.

[Rugaber 90] Rugaber, S., Ornburn, S. e Le Blanc Jr., R. Recognizing Design Decisions in Programs. *In IEEE Software*, 7(1), Jan., 1990.

[Souza 91] Souza, A. Manutenção de Software sob Enfoque de Re Engenharia. *Dissertação de Mestrado*, Departamento de Informática - Pontifícia Universidade Católica, Rio de Janeiro, Março, 1991.

[Vallabhanen 87] Vallabhanen, S. *Auditing the Maintance of Software*, Prentice Hall Inc., 1987.