Um Processo de Reengenharia Econômico e Eficaz

Luiz Carlos Guedes *Arndt von Staa*
(e-mail: gmcgued@bruff.bitnet
e-mail: arndt@inf.puc-rio.br)

PUCRioInf - MCC 11/93

Abstract

In this paper we propose a re-engineering process. Using this process, source programs are restructured in order to reduce maintenance costs and, simultaneously, increase their quality. Several computer aided activities are performed during this process. Roughly speaking, this process uses sintax directed reverse engineering to transport the original source code to a CASE repository. It is a basic requirement that the CASE tool be capable of generating (linearize) source code from the contents of the repository. Once disponible in the repository, the CASE tool is used to re-engineer the program. The process may be adapted to several different source languages by simply defining the grammar and lexical analysers of these languages.

KEYWORDS: LaR*S, Maintenance, Syntax driven analysis, Re-engineering, Repositories, Restructuring, Reverse engineering,

Resumo

Neste artigo é proposto um processo de reengenharia visando a reestruturação de programas de modo a reduzir os custos de manutenção e, simultaneamente, aumentar a sua qualidade, em particular a confiabilidade, a manutenibilidade e a economicidade. O processo a ser descrito obtém, a partir do programa fonte original existente, um projeto modular estruturado, devidamente documentado e fácil de ser mantido. O processo é formado por diversas tarefas. Todas estas tarefas recebem apoio computacional. Em linhas gerais, o processo utiliza engenharia reversa baseada em analisadores sintáticos para tornar o programa disponível no repositório de uma ferramenta CASE. A seguir a ferramenta é utilizada para efetuar a reengenharia. O processo pode ser adaptado a diversas linguagens de programação, através da definição da gramática destas linguagens.

PALAVRAS CHAVE: Análise dirigida por sintaxe, Engenharia reversa, LaR*S, Manutenção, Reengenharia, Reestruturação, Repositórios

1. O problema a resolver

Há muito tempo vem se afirmando que mais de 70% dos custos de um CPD são devidos à manutenção de software[1,2]. Outros afirmam que os custos da manutenção vão aumentando em virtude da gradativa deterioração da estrutura do software provocada pela própria manutenção[8]. Por outro lado, é economicamente inviável reconstruir uma parcela significativa dos sistemas deteriorados que se encontram em uso. Para "salvar" tais sistemas, dando-lhes uma expectativa de vida mais longa, tem sido proposta a engenharia reversa seguida da reengenharia[14].

Em sua forma típica, a engenharia reversa procura criar a documentação de engenharia do sistema a partir do código fonte. Em geral, este é o único documento técnico confiável que resta do sistema. De posse desta documentação reorganiza-se o projeto do sistema. A partir do projeto reorganizado, altera-se o código fonte, a seguir testa-se o novo sistema e, finalmente, passa-se a mantê-lo. Esta etapa de reorganizar é chamada de reengenharia. É claro que este processo é muito caro e demorado e, conseqüentemente, pouco trilhado. Além disso, tem se a impressão que tudo é inútil, uma vez que, algum tempo mais adiante, tem-se que reorganizar tudo de novo, pois é altamente provável que o programa tenha se deteriorado devido a sucessivas manutenções e que o código fonte seja o único documento confiável que reste...

Temos que encontrar uma solução mais barata, mais eficaz e mais duradoura.

Este artigo propõe um *processo de reengenharia* visando a reestruturação de programas de modo a reduzir os custos de manutenção e, simultaneamente, aumentar a qualidade destes programas, em particular, a confiabilidade, a manutenibilidade e a economicidade. Entendemos por *processo* uma rede (ou seqüência) de atividades que, quando executada ("enacted"), gera resultados previsíveis. O processo descrito é genérico e amparado por ferramentas. O objetivo do processo é obter, a partir do programa fonte original de um sistema existente, um projeto modular estruturado, devidamente documentado e fácil de ser mantido. O processo é formado por diversas tarefas. Todas estas tarefas recebem apoio computacional. A figura 1 mostra os componentes automatizados que

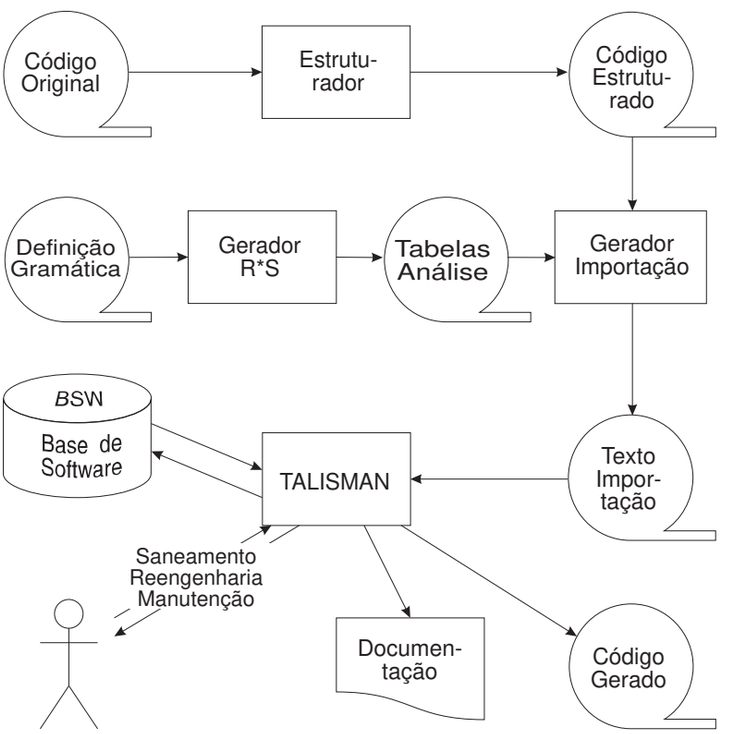

Figura 1. Componentes automatizados utilizados

```
#include <stdio.h>
#define max 10
    int A[max+1];
    void BubbleSortRec( int i, int n )
    {
        int j,w;
        if( i<n ) BubbleSortRec(i+1,n);
        for( j=1; j; j++ )
            if( A[i]<A[j] )
            {
                w=A[i]; A[i]=A[j]; A[j]=w;
            }
    }
    void main(void)
    {
        int k;
        k=1;
        while( k<=max )
        {
            scanf("%d",&A[k]); fflush(stdin);
            k=k+1;
        }
        BubbleSortRec(1,max);
        k=1;
        while( k<=max )
        {
            printf("%d ",A[k]);
            k=k+1;
        }
    }
```

Figura 2. Código fonte original

apóiam as tarefas do processo. Em linhas gerais, o processo utiliza engenharia reversa para tornar o programa disponível no repositório de uma ferramenta CASE, no caso a *base de software* de Talisman[17]. Qualquer que seja a ferramenta CASE escolhida, é fundamental que seja capaz de gerar (*linearizar*) código a partir do conteúdo do repositório que ela mantém. A seguir, a ferramenta é utilizada para efetuar a reengenharia e a manutenção. Este processo tem diversas vantagens:

1- a documentação técnica não é perdida, uma vez que faz parte do repositório mantido pela ferramenta CASE.

2- a documentação técnica é confiável e consistente com o código, uma vez que a ferramenta CASE gera (lineariza) o código fonte através da linearização do projeto.

3- a reengenharia pode ser realizada incrementalmente, *sob demanda*, a medida que forem surgindo solicitações de manutenção. Assim, somente serão renheiradas as porções de programas que são alvos de alguma atividade de manutenção.

2. O formato de entrada

O programa fonte original a ser renheirado é, tipicamente, um programa desenvolvido sem regras conhecidas. Este programa é transformado em um programa estruturado, usando algum programa estruturador disponível no mercado. Existem diversos estruturadores automáticos de código[5] para as linguagens tradicionais (COBOL, FORTRAN, ...). Lano e Haughton[11] descrevem técnicas para extrair a funcionalidade de programas não estruturados visando sua estruturação e, posterior, geração de projeto estruturado.

O objetivo de um estruturador de programas é obter um programa fonte estruturado equivalente ao programa fonte original. Aqui equivalência significa que o programa estruturado possui as mesmas propriedades funcionais, gerando os mesmos acertos e erros, e possui as mesmas interfaces do programa original. Cabe observar que muitos dos

programas desenvolvidos na última década já satisfazem os requisitos de estruturação tradicionais[6], prescindindo desta etapa de estruturação.

Obviamente, após a estruturação do programa, resta um longo esforço a ser realizado para se obter um programa projeto passível de sofrer manutenção, uma vez que facilidade de ser mantido (manutenibilidade) não se resume à codificação estruturada como, há muito tempo, já é sabido[7].

Na figura 2 temos um exemplo de código fonte original que utilizaremos para ilustrar o processo de reengenharia. Trata-se de uma versão recursiva do algoritmo de ordenação pelo método da bolha. A simplicidade do exemplo deve-se à falta de espaço neste artigo, não comprometendo, porém, a apresentação do processo.

3. O formato de saída

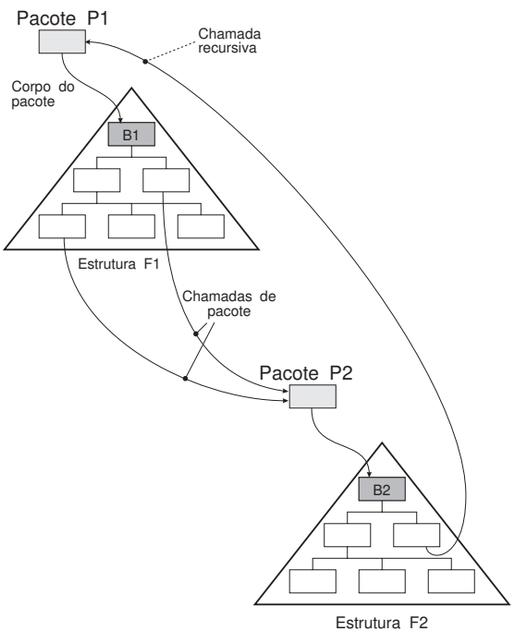

Figura 3. Ilustração parcial de uma estrutura parte de um programa projeto

A ferramenta Talisman oferece um editor de estrutura. Com este editor editam-se e mantém-se estruturas organizacionais de dados e de código. Estas organizações constituem o *programa projeto*. Um programa projeto corresponde, tipicamente, ao resultado da etapa de projeto detalhado nos ciclos de vida tradicionais.

O editor de estruturas mantém, na *base de software* todos os fatos correspondentes ao projeto detalhado. O editor de estruturas opera com as seguintes classes de objetos: *módulos*, *pacotes* (em C correspondem a funções), *blocos programa*, *tipos* (em C correspondem a **struct**'s, ou **typedef**'s) e *blocos de dados*. Todos estes objetos podem ser inter-relacionados por intermédio de *relações* (referências cruzadas). A figura 3 ilustra uma estrutura de pacotes.

O editor de estrutura permite associar fragmentos de texto aos objetos. Em particular, permite associar fragmentos de texto de código aos objetos. Utilizando um *linearizador*, a organização de objetos contida na base de software é percorrida e os fragmentos de código são externados para um arquivo seqüencial, na ordem e no formato requeridos pelo correspondente compilador. Os editores de especificações de objetos e os linearizadores são definidos por meio de um programa escrito numa linguagem interna a Talisman, a linguagem de programação de formulários. Modificando-se esses programas, pode-se adaptar os editores e os linearizadores a diferentes linguagens fonte. O uso de linearizadores para converter *programas projeto* em *programas fonte*, assegura que os programas fonte correspondam fielmente ao projeto contido na base de software.

Os programas fonte são sempre gerados a partir da base de software. Isto permite concluir que a base de software é, efetivamente, o formato de saída almejado pelo processo de reengenharia.

Em adição a organizar algoritmos e estruturas de dados, o editor de estruturas permite editar e manter o código de programas, utilizando princípios semelhantes a Modula-2, mesmo quando escritos em outra linguagem de programação, como no nosso

```
DA "Pacotes programa" "BubbleSortRec" Nome
DA "Blocos programa" "Corpo de BubbleSortRec" Nome
DA "Blocos programa" "Bloco 1_ 0 de BubbleSortRec" Nome
DA "Blocos programa" "Bloco 1_ 1 de BubbleSortRec" Nome
DA "Blocos programa" "Bloco 2_ 0 de BubbleSortRec" Nome
DA "Pacotes programa" "BubbleSortRec" Relação 61
Corpo de BubbleSortRec
#@#
DA "Blocos programa" "Corpo de BubbleSortRec" Relação 57
Bloco 1_ 0 de BubbleSortRec
Bloco 1_ 1 de BubbleSortRec
#@#
DA "Blocos programa" "Corpo de BubbleSortRec" Texto 18
void BubbleSortRec( int i, int n )
#@#
DA "Blocos programa" "Corpo de BubbleSortRec" Texto 19
int j,w;
#@#
DA "Blocos programa" "Bloco 1_ 0 de BubbleSortRec" Relação 70
BubbleSortRec
#@#
DA "Blocos programa" "Bloco 1_ 0 de BubbleSortRec" Texto 14
if(i<n)
#@#
DA "Blocos programa" "Bloco 1_ 0 de BubbleSortRec" Texto 17
BubbleSortRec(i+1,n);
#@#
DA "Blocos programa" "Bloco 1_ 1 de BubbleSortRec" Texto 13
for(j=1;j<i;j++)
```

Figura 4. Exemplo de texto de importação

caso: a linguagem C. Ou seja, os módulos são linearizados em duas formas: *módulo de implementação*, que contém o código, e *módulo de definição*, que contém as declarações de interface.

O uso deste editor de estruturas tem se mostrado extremamente conveniente para construir e manter programas. A título de ilustração, todos os módulos que constituem Talisman são gerados com o uso deste editor de estruturas. Assim, tornou-se óbvio o desejo de converter programas existentes em programas projetos que possam ser renheirados e mantidos utilizando o editor de estruturas.

4. A conversão de programas fonte em programas projeto

A primeira etapa na obtenção de um programa projeto a partir de um programa fonte é a criação de uma base de software contendo um programa projeto equivalente ao código fonte original. O programa projeto é equivalente ao programa fonte original se, eliminando todos os comentários e formatações do programa gerado pelo linearizador e do programa fonte original, os códigos forem iguais. A conversão manual de um programa fonte para um programa projeto foi considerada inviável, ou, no mínimo, anti-econômica. Como Talisman permite gerar e/ou atualizar bases de software a partir de textos de importação, procurou-se construir um gerador de arquivos de texto de importação.

Para ser importado, cada módulo (unidade de compilação) deve ser convertido para o formato de importação. No texto de importação ocorrem diretivas de acréscimo (DA) de seus elementos. A figura 4 mostra uma parte do texto de importação do pacote BubbleSortRec.

Com a importação obtém-se automaticamente o programa projeto, no qual cada pacote corresponde a um diagrama de estrutura organizacional, estando suas estruturas de controle de execução associadas a blocos programa. Estes blocos programa são as caixas encontradas nos diagramas de estrutura organizacional.

Cada diretiva define um atributo de um objeto a ser importado. O primeiro símbolo é o nome da classe de objetos, o segundo símbolo é o nome do objeto, o terceiro símbolo é o nome do atributo. Quando o atributo possui várias ocorrências, o quarto símbolo é o índice da ocorrência. Este índice possui um significado semântico bem definido. No caso de textos, o significado é definido pelos programas de formulários utilizados na especificação e na linearização. Por exemplo, o fragmento de texto de índice 18 contém o código fonte cabeçalho de uma função. No caso de relações, o significado é definido pela linguagem de representação. Por exemplo, a relação 61 define um vínculo de um pacote para um bloco programa, onde este vínculo tem o significado de *corpo de pacote*. Caso o atributo possua um valor, este vem na linha a seguir da diretiva de adição. Textos e relações admitem diversas linhas. Neste caso, a seqüência de caracteres "#@#" determina o término do texto ou da relação.

5. A organização do arquivo contendo texto de importação

A proposta de engenharia reversa e reengenharia aqui apresentada, se caracteriza por atuar sobre cada módulo do programa de cada vez. Desta forma, se as interfaces e a organização interna dos módulos forem preservadas pelo processo, o programa prosseguirá funcionando de forma igual à sua versão original.

Cada objeto possui um *nome* e pode conter duas outras classes de atributos, os *textos*, que servem para compor o código fonte, e as *relações*, que servem para reconstruir a estrutura a que pertencem e manter referências cruzadas sobre a utilização de membros do programa.

Para assegurar a equivalência entre programas tal como definida acima, é necessário seguir uma certa ordem na importação. Em particular, devem ser importadas primeiro as informações organizacionais e, depois, as de código fonte. Em adição, cada atributo importado deve designar o nome do objeto no qual é atributo. Como é desconhecido o significado de cada porção de código, os nomes dos blocos são meramente nomes estruturais sem semântica associada. Os nomes de blocos são gerados na forma "Bloco *X*_*Y* de *NomePacot*e", onde*X* é a profundidade e *Y* é o índice de irmão. Os nomes de pacotes são os nomes de código da função correspondente. Faz parte do trabalho de reengenharia transformar estes nomes estruturais em nomes com sentido semântico.

Para assegurar que, ao linearizar, a ordem com que os objetos serão externados seja a mesma que a ordem em que figuram no texto fonte estruturado, devem ser obedecidas algumas regras básicas. A organização geral do texto de importação deve ser a seguinte:

1- texto de importação do módulo.

2- para cada tipo, o texto de importação do tipo.

3- para cada pacote, o texto de importação do pacote.

Dentro de cada um destes 3 grupos, devem-se organizar as diretivas de importação da seguinte forma:

1- definição do nome do objeto corrente, se ainda não foi definido.

2- definição dos nomes dos componentes relacionados ao objeto corrente.

3- definição das relações do objeto corrente para os demais objetos, sempre em uma ordem da relação mais abrangente para a menos abrangente. Por exemplo, módulos relacionam os pacotes que contém e relacionam os pacotes que exportam. Estes últimos formam um sub-conjunto dos primeiros, assim a diretiva da relação de objetos exportados deve aparecer depois da diretiva de pacotes que compõem o módulo. As relações devem relacionar os nomes na ordem em que aparecem no texto fonte.

4- definição dos fragmentos de texto do objeto corrente.

A figura 4 ilustra a ordem de importação de pacotes descrita a seguir:

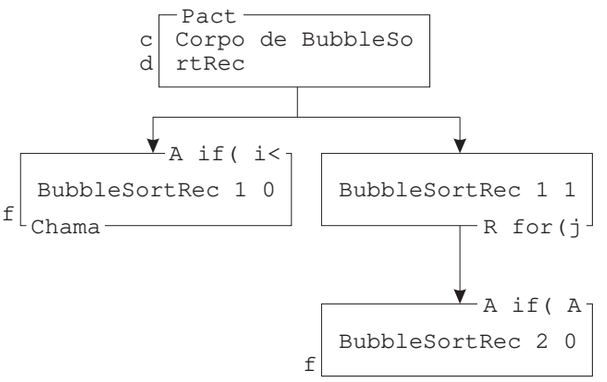

Figura 5. Estrutura organizacional exibida para o programa tal como importado

1- definição do nome do pacote.

2- definição dos nomes de todos os blocos programa que fazem parte do corpo do pacote, em ordem de caminhamento pre-fixado pela esquerda.

3- para cada bloco, em ordem de caminhamento prefixado pela esquerda as relações deste bloco para outros objetos. Estes são ou pacotes chamados pelo bloco, ou blocos filho. Após às relações, os textos fonte do bloco.

A cada pacote estará associado um diagrama de estrutura organizacional que, se linearizado, gera o código fonte do pacote. Este diagrama de estrutura organizacional está associado ao pacote via o bloco programa raiz do seu corpo. Na figura 5 é ilustrada a estrutura organizacional correspondente ao projeto programa tal como importado.

Os blocos programa, apresentam como componentes principais elementos que representam as estruturas de controle de execução e os trechos de código seqüencial controlados pelas mesmas. Também apresentam as relações que permitem a construção e navegação por toda a estrutura. Sem preocupação com a ordem de apresentação no arquivo de texto de importação, os atributos de um bloco são os seguintes:

1- cabeçalho da função, caso o bloco seja raiz do corpo de um pacote.

2- texto de declaração dos dados utilizados na estrutura.

3- relação dos blocos estrutura de dados declarados na estrutura. Contém referências a estruturas formadas por blocos de dados.

4- condição de ativação do bloco. Contém o código de comandos condicionais, por exemplo: **if** (*expressão*), **else**, ou **case** *valor***:**.

5- texto fonte antes de ativar filhos. Contém zero ou mais linhas de código estritamente seqüencial.

6- condição de repetição dos filhos. Contém o código de comandos de repetição, por exemplo: **while** (*expressão*), **for** (; ;).

7- condição de seleção dos filhos. Contém o cabeçalho de comando de seleção **switch**.

8- relação dos blocos filho.

9- texto fonte após ativar filhos.

10- relação dos pacotes chamados pelo código do bloco.

Assim como os pacotes programa, os tipos de dados apresentam seu corpo descrito em uma estrutura, formada por blocos estrutura de dados. Os blocos estruturas de dados ocorrem nas descrições dos tipos de dados e das estruturas de dados propriamente ditas dos módulos. Seus componentes têm de prever a descricão de estruturas de dados homogêneas ou não. As estruturas de dados homogêneas (*arrays*) são representadas pela repetição de um dado elementar, enquanto que as heterogêneas (*structs*) demandam a criação de blocos filhos para repesentarem cada dado elementar.

```
StatList      = StructStat
          ! SeqStats ;
[ExternarSeq]: SeqStats = [InicSeq] SeqStat ;
SeqStat       = +[ Expression ';' [NovoStat] ] ;
StructStat    =  IfStat
          ! WhileStat
          ! DoWhileStat
          ! ForStat
          ! SwitchStat
          !  CaseStat ;
IfStat    = 'if' [InicCond] CondExp [FimCond]
            '{' [GerarFilhoTrue] StatList '}'
            'else' '{' [GerarFilhoFalse] StatList '}' ;
WhileStat = 'while' [InicRepet] CondExp [FimRepet]
            '{' [InicFilRep] StatList [FimFilRep] '}' ;
```

Figura 6. Trecho de uma definição de gramática contendo marcadores semânticos

1- código de declaração do bloco estrutura de dados.

2- relação de blocos estrutura de dados filho.

3- texto de declarações de elementos de dados.

4- código indicativo da condição de repetição do elemento de dados.

5- texto de inicialização do elemento de dados.

6- relação dos tipos referenciados pelo bloco estrutura.

Cabe ao importador de programas a detecção de cada um dos componentes citados acima e a construção da estrutura organizacional referente a cada um.

A organização do arquivo de texto de importação descrita acima, sugere o uso de analisadores sintáticos acoplados a "geradores de código" para criá-los a partir do código fonte. O analisador sintático decompõe o código fonte em parcelas relevantes, e o gerador de código produz as diretivas de importação. Processo semelhante, baseado em um conjunto de padrões, é descrito por Santanu[16].

6. A geração de arquivos de importação dirigida por sintaxe

Para identificar e delimitar as regiões do programa fonte contendo os trechos de código correspondentes aos elementos descritos acima, utilizamos um analisador sintático. Este reconhece programas redigidos na linguagem fonte e ativa as rotinas semânticas geradoras do arquivo de texto de importação. Estas rotinas semânticas agregam porções de código do programa em listas internas e externam essas porções de código código através de diretivas de importação na ordem experada pelo importador.

Para gerar as tabelas de análise sintática utilizamos o gerador de analisadores sintáticos laR*S(1)[9], o qual é uma extensão do gerador de analisadores sintáticos R*S simples[15]. Uma das propriedades do gerador laR*S(1) é a capacidade de associar nomes às regras e de incorporar marcadores semânticos no corpo das produções. Um *marcador semântico* corresponde a uma regra que reconhece a cadeia vazia e que, quando "reconhecida" em um programa fonte, ativa a rotina semântica associada. Um *nome de regra* define uma rotina semântica a ser ativada quando a regra correspondente for reduzida. Desta forma torna-se possível estabelecer um conjunto de marcadores semânticos e de nomes de regras independente da linguagem fonte analisada. Isto, por sua vez, permite criar um processador semântico independente da linguagem fonte e, conseqüentemete, utilizá-lo para gerar textos de importação para diferentes linguagens fonte.

A figura 6 ilustra um trecho de reconhecedor para a linguagem C. Nesta figura os nomes entre colchetes correspondem a nomes de regras e a marcadores semânticos. O nome *[ExternarSeq]* é nome de regra. O nome *[InicSeq]* é um marcador semântico. Ao

analisar o programa fonte fornecido, se o ponto marcado por *[InicSeq]* for encontrado, inicia-se a agregação de linhas de código a uma lista de código fonte estritamente seqüencial. O reconhecimento de código seqüencial termina quando a regra *SeqStats* for reduzida. Quando isto ocorrer, o nome de regra *[ExternarSeq]* ativa o processador semântico para que seja externado o conjunto de linhas de código fonte seqüencial colecionado desde o último *[InicSeq]*.

O marcador semântico *[NovoStat]* cria uma nova linha para cada *statement* lido. Os marcadores semânticos *[InicCond]* e *[FimCond]* delimitam a expressão condicional do *if*. Esta expressão deve ser incluída no fragmento de texto *"Condição de ativação"* do bloco. Para isto é necessário que se crie um novo bloco, irmão dos blocos criados neste nível de abstração. Agora todo o código agregado será incorporado a este novo bloco até que se chegue ao '}' da parte *true*. Existindo *else* a seguir, cria-se um novo bloco, contendo *else* como o texto de condição de ativação.

Deve estar claro agora que, para todas as linguagens com estrutura de código semelhante a C (são exemplos: Algol, Pascal, Módula, etc.) o processo de construção do projeto programa é o mesmo. Desta forma consegue-se uma técnica independente da linguagem fonte, uma vez que os marcadores semânticos podem ser portados para a gramática de qualquer linguagem, sendo esta submetida a um gerador de analisadores sintáticos para a obtenção de um importador para nova linguagem. Conseqüentemente, o analisador, bem como o gerador de texto de importação, é insensível à linguagem, desde que receba a tabela de análise sintática corretamente marcada, bem como um analisador léxico apropriado para a linguagem em questão.

O apêndice 1 apresenta o código linearizado a partir do projeto programa criado pela importação do texto de importação. Note que este código é idêntico ao código do programa fornecido, exceto no que concerne a formatação e os comentários.

7. A transformação em programa bem projetado

Até a conclusão da etapa de engenharia reversa, todas as tarefas foram realizadas automaticamente. Ao final destas tarefas dispõe-se de uma base de software a partir da qual se pode gerar um programa equivalente ao programa fonte original. Para se poder realizar as etapas de engenharia reversa, precisa-se, para cada linguagem fonte considerada:

1- tabela de análise sintática devidamente marcada.

2- tabela de análise léxica adequada à linguagem.

3- processador semântico capaz de gerar o texto de importação a partir do código fonte analisado com as tabelas marcadas (virtualmente independente da linguagem fonte).

4- editor de formulários de especificação (virtualmente independente da linguagem fonte).

5- linearizador (pouco dependente da linguagem fonte).

Uma vez dispondo destes instrumentos, pode-se proceder a engenharia reversa com inúmeros programas. Além disto, conforme já foi mencionado, o esforço de se adaptar o conjunto de instrumentos a diferentes linguagens é pequeno. O formato final atingido pelo programa projeto contido na base de software tal como esta se encontra ao final da engenharia reversa possui as seguintes características:

1- para cada tipo dispõe-se de diagramas de estrutura organizacional dos dados.

2- para cada pacote dispõe-se de diagramas de estrutura organizacional do pacote.

3- conjunto completo de referências cruzadas entre os blocos programa e os pacotes.

4- conjunto completo de referências cruzadas entre os tipos de dados e as estrutras de dados.

5- conjunto completo de referências cruzadas entre os tipos de dados e os pacotes que os utilizam em declarações internas.

No entanto, este programa projeto ainda é insatisfatório. Os nomes dos blocos nada dizem, o código linearizado pode conter partes inúteis, duplicadas, mal organizadas, etc. É necessário, agora, realizar a reengenharia. O objetivo final desta tarefa é obter módulos de qualidade substancialmente melhor que a do módulo original. A primeira etapa da reengenharia, é o *saneamento* do projeto. O resultado final do saneamento é um projeto onde:

1- os nomes de todos os objeto têm significado semântico.

2- todo código inútil foi eliminado.

3- todo código duplicado foi simplificado e/ou fatorado.

4- o módulo possui módulo de definição e módulo de implementação claramente definidos.

5- todos os pacotes possuem especificações de objetivos, dados de entrada, dados de saída e interface com o usuário.

6- todas as estruturas de dados possuem organização aceitável.

7- na medida do possível, as estruturas organizacionais foram simplicadas, sem, no entanto, mudar a funcionalidade.

Um texto de importação obtido a partir da análise sintática do programa fonte, não consegue determinar nomes que caracterizem a função do bloco, ou su-árvore de que o bloco é raiz. Esta limitação ocorre pelo fato de ser um problema indecidível determinar a função executada por um trecho de programa. Assim, o gerador de texto de importação produz nomes estruturais (por exemplo *Bloco 1_ 0 de BubbleSortRec*) sem significado semântico algum. Para obter os nomes apropriados para os diferentes objetos, é necessário explorar e entender as estruturas organizacionais. Ao fazer isto, os nomes estruturais são substituídos por nomes com significado e são adicionados textos de assertivas de entrada e saída sempre que necessário. Um nome é considerado bom, se corresponder precisamente à função do código ou declaração que se inicia naquele ponto. Esta tarefa é um gargalo não automatizável do processo que, apesar disto, apresenta um papel fundamental no saneamento do programa fonte. A figura 7 mostra o programa exemplo linearizado após a correção dos nomes. Deve ser notado que a maioria das tarefas citadas refletirá diretamente na inclusão de comentários ao programa fonte existente.

Ao percorrer as estruturas organizacionais com vistas ao entendimento, realizam-se as demais ações de saneamento. O saneamento sendo realizado de maneira disciplinada, tipo a tipo e, depois, pacote a pacote, permite que se tenha ao final da revisão de cada estrutura um programa projeto estável quase equivalente ao programa original. O programa não será mais equivalente ao programa original, uma vez que defeitos contidos no programa original já podem ter sido sanados. O processo de saneamento pode ser efetuado sob demanda. Ou seja, ao invés de sanear todo o programa projeto de uma vez, efetua-se o saneamento nas partes afetadas ao realizar uma etapa de manutenção. Cabe salientar que, ao efetuar a manutenção, é obviamente imprescindível entender-se corretamente a finalidade de cada linha de código. Isto é, essencialmente, o que se precisa saber para poder rebatizar os nomes dos blocos e para incluir assertivas. Caso, agora outra tarefa de manutenção requeira a modificação de uma estrutura já saneada, o esforço necessário para obter satisfatório entendimento será significativamente menor. Junto cairá a probabilidade de se cometer erros decorrentes do entendimento precário. Isto tudo contribui para um significativo aumento de produtividade e de confiabilidade da manutenção. O saneamento sob demanda reduz os custos da reengenharia, uma vez que somente serão re-organizadas as porções de código efetivamente sujeitas a manutenção.

A revisão da interface dos módulos (módulo de definição) é uma necessidade devida à indisciplina típica dos programadores, que, em geral, não se preocupam em manter bem definida a interface de seus programas. Sabe-se, no entanto, que trabalhar

```
/************************************************************
*
* Pacote: Ordenar utilizando bolha recursiva
*
* Objetivos, função do pacote
*    Reorganizar os dados de um vetor de modo que fiquem
*    em ordem crescente
*    O algoritmo utilizado é o de bolha recursiva
*
************************************************************/

void BubbleSortRec( int i, int n )
{
    /* Dados do bloco: Corpo de Ordenar utilizando bolha r
       int j ,

           Temp ;

/* Corpo de Ordenar utilizando bolha recursiva */

    /* Ordenar a parte final do vetor */

       if( i < n )
       {
          BubbleSortRec( i+1 , n );
       } /* fim ativa: Ordenar a parte final do vetor */

    /* Ordenar todos os elementos da parte inicial */

       for( j=1 ; j<i ; j++ ) {

       /* Assegurar a ordem do elemento correntec */

          if( A[i]<A[j] )
          {
             Temp=A[i];
             A[i]=A[j];
             A[j]=Temp;
          } /* fim ativa: Assegurar a ordem do elemento cor
       } /* fim repete: Ordenar todos os elementos da parte

} /* Fim pacote: Corpo de Ordenar utilizando bolha recursi
```

Figura 7. Código linearizado do pacote de ordenação após o saneamento (truncado a direita)

com interfaces bem definidas é fundamental para qualquer tentativa de trabalho em grupo e/ou de reuso de componentes de software.

A partir de um programa devidamente saneado pode-se iniciar a segunda atividade da reengenharia: a reorganização do programa projeto, com vistas a eliminar deficiências de projeto.

8. Resultados obtidos

O processo aqui apresentado foi testado em dois sistemas considerados representativos.

O primeiro programa em que se aplicou o processo foi o próprio importador que havia sido escrito, originalmente, utilizando editores convencionais. Sua versão inicial se apoiou em um analisador sintático para a linguagem C, a mesma com que foi escrito, permitindo uma espécie de auto-documentação. Após ter sido importado e ter sofrido o saneamento necessário, foi novamente linearizado e compilado, gerando uma nova versão do programa, obtida a partir de um projeto modular bem feito, ou seja, compreensível e

manutenível. O saneamento do importador consumiu aproximadamente 40 horas, metade para reorganizar os dados e metade para reorganizar o código. Este esforço foi considerado pequeno para um sistema de aproximadamente 9000 linhas de código fonte e 180 pacotes distribuídos por 7 módulos.

O segundo programa, ao qual se aplicou o processo, foi o gerador de analisadores sintáticos R*S simples[15]. O sistema, originalmente escrito e mantido na linguagem PASCAL, foi submetido a um tradutor fonte-a-fonte para a linguagem C. Uma vez em C, os algoritmos foram modificados do método simples para o método laR*S(1). Após, o programa foi importado e, a seguir, saneado. O esforço de saneamento do programa consumiu aproximadamente 60 horas de trabalho, das quais 1/3 foram usadas para o saneamento dos dados e 2/3 para o do código. Um tempo considerado pequeno para a compreensão e obteção do projeto de um programa com aproximandamente 20000 linhas de código fonte e 300 pacotes distribuídos por 17 módulos.

9. Alguns trabalhos correlatos

Byrne[3] descreve um processo de engenharia reversa seguido de reharia visando a tradução de programas escritos em FORTRAN para Ada. A única ferramenta utilizada foi uma ferramenta CASE. O processo de captura da documentação de projeto do programa original é totalmente manual e segue um roteiro semelhante ao aqui descrito. A descrição se detém em identificar as propriedades e componentes independentes da linguagem para conseguir realizar a tradução fonte a fonte. O processo narrado é do gênero "tudo ou nada", no sentido que ou todo o programa é recuperado ou nada será recuperado. É evidente que isto é conseqüência direta do fato de se estar realizando tradução fonte a fonte FORTRAN para Ada via engenharia reversa seguida de reengenharia. Finalmente, como a ferramenta não é capaz de linearizar, é difícil avaliar, passo a passo, o resultado (programa Ada sendo gerado).

Canfora, Cimitile e Carlini[4] descrevem um sistema de reestruturação de código seqüencial e manutenção de referências cruzadas exclusivamente para programas desenvolvidos em PASCAL. O sistema é implementado em Prolog. Uma parte inicial, o "Extractor" é essencialmente um processador de linguagem (analisador léxico, analisador sintático, tabelas de símbolos, tabelas de fragmentos de código). A segunda parte, o "Abstractor" controla diversas propriedades, tais como código morto, seqüências de definição e uso etc. Com é redigido em Prolog existe uma certa facilidade de se adicionar regras à medida que se vai adquirindo conhecimento relativo ao processo de engenharia reversa. O nosso gerador de arquivos de exportação é dirigido por um analisador sintático dirigido por por gramáticas formalmente definidas. Em princípio diversas das considerações descritas no artigo em apreço podem ser resolvidas neste nível e de forma independente da linguagem fonte tratada. O arquivo de exportação, ao ser lido para a base de software de Talisman, gera, automaticamente, uma parte significativa das relações mencionadas na seção que trata do "Abstractor". Muitas, mas não todas, das propriedades controladas no artigo podem ser validadas utilizando validadores redigidos na linguagem de formulários de Talisman. Coisa que fizemos em pequena escala durante a etapa que chamamos de *saneamento*. A nossa principal deficiência está na dificuldade de se realizar, de forma automática, a validação de seqüências de definição e uso.

Kozaczynski e outros[10] apresentam um processo para automatizar o recuperação do modelo conceitual do programa. O processo utilizado é semelhante ao descrito por Canfora[4]. Também aqui é utilizado um método baseado em regras para identificar a funcionalidade original dos programas e aplicar transformações que obtenham a funcionalidade desejada.

Leite e outros[13] descrevem um processo utilizado para reconstruir a máquina DRACO-PUC a partir da máquina DRACO. Em outro artigo, Leite e outros[12] descrevem um estudo de caso de reengenharia do sistema Hiperlex. O processo utilizado foi totalmente manual. Em ambos os artigos utiliza-se um processo com as quatro fases tradicionais:

recuperação, re-especificação, redesenho e re-implementação. Em ambos os artigos uma parte substancial das considerações é específica aos programas renheirados, dificilmente podendo ser generalizado para outros programas.

10.Conclusão

Foi descrito um *processo* para realizar a engenharia reversa e a reengenharia de programas com vistas a melhorar a sua qualidade de engenharia, garantindo assim uma expectativa de vida maior ao programa renheirado.

O objetivo deste processo é reestruturar, ou reprojetar, de forma sistemática, sistemas cujo código já esteja disponível. Para poder realizar sistematicamente a engenharia reversa, precisam-se técnicas genéricas e automatizadas, pelo menos até o nível de recuperação da estrutura e dos relacionamentos existentes entre os elementos que formam um dado programa. Como estas propriedades dependem exclusivamente da sintaxe do programa, analisadores sintáticos são ferramentas óbvias. A incorporação de conhecimento sobre o significado dos elementos do programa ainda é fortemente baseado em ação humana. Pode receber apoio a partir de bases de conhecimento, mas, pelo menos por enquanto, dificilmente poderá ser totalmente automático. Finalmente, todo o processo de engenharia reversa e reengenharia atua sobre componentes de programas, alterando-os, dispondo-os em outras ordens, agrupando-os de outras formas etc. É evidente que estas tarefas somente se tornam economicamente viáveis se apoiadas por alguma ferramenta capaz de linearizar o conteúdo de seus repositórios.

O processo descrito é fortemente apoiado por ferramentas e é realizado por intermédio das seguintes etapas:

1- estruturar o código fonte utilizando algum estruturador disponível no mercado.

2- gerar texto de importação utilizando um "compilador". Este compilador possui ações semânticas, em princípio, invariantes com relação à linguagem fonte do programa, e utiliza uma tabela de análise sintática correspondente à linguagem a ser importada. A vinculação entre o analisador sintático e as rotinas semânticas é estabelecida através de marcadores semânticos inseridos na gramática da linguagem. A separação dos processadores sintático e semântico viabiliza, em principio, criar novos geradores rapidamente e a baixo custo.

3- importar o texto, utilizando o importador Talisman. Neste momento, a linearização do programa projeto resulta em um código funcionalmente igual ao programa original.

4- sanear o programa projeto, utilizando o editor de estruturas. Durante o saneamento, são definidos os nomes dos componentes estruturais, eliminado os trechos de código inativos e são ajustadas as estruturas. A linearização do programa projeto saneado resulta em um programa funcionalmente equivalente, no sentido que produz os mesmos resultados para os mesmos dados da entrada.

5- renheirar o projeto, utilizando o editor de estruturas. Nesta fase a estrutura do programa é amplamente revista e otimizada. Deficiências do programa são eliminadas. A linearização do programa projeto resulta em programa funcionalmente "parecido" com o original, uma vez que, agora, pode gerar resultados diferentes para dados iguais.

6- manter o projeto, utilizando o editor de estruturas. Nesta fase o programa é evoluído e adaptado segundo novas necessidades identificadas.

O processo aqui descrito foi utilizado com sucesso em dois sistemas considerados representativos.

11. Referências Bibliográficas:

1- Boehm, B.W.; Brown, J.R.; Lipow, M.; "Quantitative Evolution of Software Quality"; *Proceedings IEEE/ACM 2nd International Conference on Software Engineering*.; Out 1976.

2- Boehm, B. W.; "Software Engineering"; *IEEE Computer* C-25(12); pp. 1226-1241; Dez 1976

3- Byrne, E.J.; "Software Reverse Engineering: A Case Study"; in *Software Practice and Experience* 21(12); dezembro 1991; pags 1349-1364

4- Canfora, G.; Cimitile, A.; Carlini, U.; "A Logic-Based Approach to Reverse Engineering Tools Production"; in *IEEE Transactions on Software Engineering* 18(12); dezembro 1992; pags 1053-1064

5- De Balbine, G.; "Better Manpower Utilization Using Automatic Restructuring"; *Proceedings AFIP 1975 National Computer Conference* 44; Montvale, N.J. AFIPS Press, pp. 319-327

6- Dijkstra, E.; "GOTO Statements Considered Harmful"; *Communications of the ACM* 11(3); Mar 1968

7- Gilb, T.; "Maintainability is More Than Strucutral Coding"; *Computer Weekly*; Jun 1978

8- Glass, R.L.; Noiseaux, R.A.; *Software Maintenance GuideBook*; Prentice-Hall, Inc.; New Jersey, 1981

9- Guedes, L.C.; *O gerador de tabelas de análise sintática laR*S(1)*; Monografia, em preparação

10- Kozaczynski,W., Ning,J., Engberts, A.; "Program Concept Recognition and Transformation"; in *IEEE Transactions on Software Engineering* 18(12); dezembro 1992; pags 1065-1075

11- Lano, K., Haughton, H.; "Extracting Design and Functionality from Code"; in *IEEE*; 1992; pags 74-82

12- Leite, J.C.S.P., Franco, A.P.M.; "Reengenharia de Software, Um Estudo de Caso"; in *Anais V Simpósio Brasileiro de Engenharia de Software*; Ouro Preto, MG; 1991; pags 177-192

13- Leite, J.C.S.P., Prado, A.F., Sant'Anna, M.; "Draco-PUC, Experiências e Resultados de Reengenharia de Software"; in *Anais VI Simpósio Brasileiro de Engenharia de Software*; Gramado, RS; 1992; pags 115-128

14- Miller, J.C.; *Improving Programming Technologies Retrofit*; Relatório interno; Montgomery Wards Corporate Systems Division, 1976

15- Rangel, J.L.; *Manual de operação do sistema de geração de analisadores sintáticos R*S simples*; Monografias em Ciência da Computação; n. 7/88; PUC-RIO, Out 1988.

16- Santanu, P., Prakash, A.; "Source Code Retrieval Using Program Patterns"; *Fifth International Workshop on Computer-Aided Software Engineering*, Proceedings CASE-92; IEEE; 1992; pags 95-105

17- *O Ambiente de Desenvolvimento TALISMAN: Manual do Usuário*; Staa Informática; 1992.

Apêndice 1. Código linearizado final

```
/*****************************************************************
*
* Módulo de implementação: Ordena vetor
*
* Arquivo gerado: ordena.c
*
*****************************************************************/
/******************* Declarações encapsuladas do módulo
/****** Inclusões utilizadas pelo módulo ******/

#include <stdio.h>

/****** Declarações internas ao módulo ******/

#define E_MAX 10
int A[ E_MAX ];


/************Codigo dos pacotes do modulo ******************/
/*****************************************************************
*
* Pacote: Bubble sort recursivo
*
*****************************************************************/
void BubbleSortRec( int i, int n )
{
    /* Dados do bloco: Corpo de Bubble sort recursivo */
       int j,w;

/* Corpo de Bubble sort recursivo */

    /* Posicionar em elemento desordenado */
       if( i < n )
       {
          BubbleSortRec(i+1,n);
       } /* fim ativa: Posicionar em elemento desordenado */

    /* Procurar ponto de substituição */
       for( j=1; j<i; j++ ) {
       /* Efetuar o intercâmbio */
          if( A[i]<A[j] )
          {
             w=A[i];
             A[i]=A[j];
             A[j]=w;
          } /* fim ativa: Efetuar o intercâmbio */
       } /* fim repete: Procurar ponto de substituição */

} /* Fim pacote: Corpo de BubbleSortRec */
/*****************************************************************
*
* Pacote: Função de teste
*
*****************************************************************/
void main(void)
{
    /* Dados do bloco: Corpo de Função de teste */
       int k;

/* Corpo de Função de teste */

    /* Ler dados de teste */
       k=1;
       while( k < E_MAX ) {
```

```
        /* Ler um elemento do vetor de dados teste */
            scanf("%d",&A[k]);
            fflush(stdin);
            k=k+1;
        } /* fim repete: Ler dados de teste */

    /* Exibir o vetor de dados teste ordenado */
        BubbleSortRec( 1 , E_MAX );
        k=1;
        while( k < E_MAX ) {
        /* Exibir um elemento do vetor de dados teste */
            printf("%d ",A[k]);
            k=k+1;
        } /* fim repete: Exibir o vetor de dados teste ordenado */

} /* Fim pacote: Corpo de Função de teste */

/********** Fim do módulo: ordena **********/
```