O Uso do Paradigma Transformacional no Porte de Programas Cobol

Julio Cesar Sampaio do Prado Leite
Marcelo Sant'Anna
Felipe Gouveia de Freitas
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
draco@inf.puc-rio.br

Antonio Francisco do Prado
Departamento de Computação
Universidade Federal de São Carlos
Via Washington Luiz, km 235
draco@power.ufscar.br

Resumo

Sistemas transformacionais são sistemas de manipulação de programas que vêm sendo aplicados em diversas áreas da Engenharia de Software. Neste artigo é descrito o uso da máquina transformacional Draco-PUC no porte de programas Cobol. A estratégia de porte dos programas Cobol para a linguagem C/C++ é descrita e é apresentado um exemplo de conversão sobre um programa para catálogo e busca de radares. A estratégia utilizada apesar de possuir a linguagem C/C++ como alvo não tem por objetivo gerar código segundo os preceitos de programação OO.

Abstract

Transformation systems have been proposed and applied on several Software Engineering tasks. This paper describes the use of the Draco-PUC transformation engine for porting Cobol programs. The porting strategy from Cobol to C/C++ is shown along with a conversion example for a radar cataloging program. Altough using C++ as its target language, the presented strategy does not intend to generate code which strictly follows the object-oriented programming paradigm.

1 Introdução

Uma das preocupações recentes da engenharia de software tem sido com o legado das aplicações existentes. Como conviver com software que funciona, que precisa de manutenção e que não dispõe de representações além do próprio código? De modo a responder a este desafio, vários pesquisadores tem trabalhado na área de re-engenharia e engenharia reversa [Leite 91] [Leite 92] [Guedes 93] [Edwards 93] [Newcomb 93] [Waters 94].

A re-engenharia é uma nova maneira de se fazer manutenção. Utiliza-se o legado, isto é o código existente, como fonte principal de informação, procurando dele extrair, através da engenharia reversa, informações de mais alto nível de abstração. Em nossa pesquisa de re-engenharia [Leite 92] temos utilizado um esquema em que dividimos esse processo em 4 subprocessos: **recuperação, especificação, re-desenho e re-implementação**. Em particular procuramos aproveitar e orientar o processo para o aproveitamento das oportunidades de reuso.

O presente trabalho explora aspectos referentes a manutenção adaptativa, isto é aquela onde a funcionalidade é mantida, mas altera-se a plataforma de suporte (hardware ou software). Na nossa concepção de re-engenharia, o trabalho de porte, que procura atender a manutenção adaptativa, é uma instância do processo onde a ênfase é na re-implementação. Isto acontece porque não há grande preocupação na recuperação e a especificação é dada pelo novo ambiente, no nosso caso por uma nova linguagem de programação.

No que se refere ao re-desenho este fica limitado a decisões pontuais ao nível de implementação, o que certamente é um dos problemas de estratégias como a que apresentamos neste trabalho. No entanto, o que demostramos é justamente a grande vantagem da estratégia de porte quando feita de maneira semi-automática.

O presente trabalho partiu da observação das necessidades crescentes de manutenções adaptativas, principalmente dada a migração de sistemas de grande porte ou de baixa confiabilidade para ambientes Unix ou similar. Partindo-se dessa observação procuramos investigar a propriedade do emprego da máquina Draco-PUC nessa tarefa. Uma das tendências para a tarefas de porte é o emprego de sistemas transformacionais na trabalhosa tarefa de traduzir sistemas de uma linguagem para outra. Trabalhos pioneiros como os de Boyle [Boyle 89] e de Arango [Arango 86] e a utilização comercial de Refine [Reasoning 92] são exemplos do uso de sistemas transformacionais para o porte de software.

Nossa contribuição é pontual. Conseguimos implementar e utilizar transformações complexas de maneira a demonstrar a versatilidade e pontencialidade da máquina Draco-PUC e de seu sistema de transformação, muito mais poderoso do que aquele originalmente implementado por Neighbors e utilizado em [Arango 86]

Nas seções iniciais, o presente artigo oferece uma visão geral do paradigma transformacional, faz um breve apanhado dos principais trabalhos em sistemas transformacionais (Seção 2) e posiciona o projeto Draco-PUC dentro do panorama da área, juntamente com uma introdução ao framework conceitual para descrição de transformações na máquina Draco-PUC (Seção 3).

Nas seções seguintes o artigo expõe o escopo do presente trabalho e a estratégia adotada pelo projeto Draco-PUC para o porte de programas Cobol para C/C++ (Seção 4). Por fim é apresentado um exemplo prático de utilização da estratégia através da conversão de um programa utilizado no catálogo e busca de radares (Seção 5), seguindo-se uma descrição das conclusões alcançadas e dos trabalhos futuros a serem realizados (Seção 6).

2 O Paradigma Transformacional

O paradigma transformacional propõe um modelo para o processo de produção de software em que os passos de desenvolvimento a partir da especificação podem ser descritos formalmente através de regras bem definidas chamadas regras de transformação [Partsch 83] [Baxter 94] [Sant'Anna 93]. Os sistemas transformacionais (ou sistemas de transformação) são as principais ferramentas para o uso deste paradigma, sendo responsáveis pela (semi)-automatização do processo de construção de software.

Em um sistema transformacional descrevemos as regras de transformação que poderão ser aplicadas sobre as entradas que a ele serão submetidas, bem como o utilizamos para a aplicação de cada passo de transformação. O sistema transformacional deve garantir a exatidão com que cada um dos passos está sendo realizado.

Desde que as bases para o uso do paradigma transformacional foram propostas, diversos sistemas transformacionais tem sido implementados. Em particular, tem se desenvolvido e integrado tecnologias que viabilizem o uso prático destes sistemas. Citaremos aqui alguns dos sistemas transformacionais que podem ser utilizados atualmente.

O sistema transformacional **Refine** [Reasoning 92] é um sistema comercial produzido pela firma Reasoning Systems (EUA), baseado nas pesquisas do Kestrel Institute [Smith 85] e que apresenta um grande suporte para a elaboração de sistemas específicos, já que toda a funcionalidade necessária a um sistema transformacional é disponibilizada através de bibliotecas de funções escritas em Common Lisp que podem também ser utilizadas diretamente através de uma linguagem proprietária. Há uma ênfase atual da Reasoning Systems na produção de sistemas sobre o Refine que apoiem as tarefas de reengenharia de sistemas escritos nas linguagens de programação mais populares.

Também desenvolvido em Common Lisp, o sistema transformacional **Popart** [Wile 93] (Producer of Parsers and Related Tools System Builders) é um sistema desenvolvido no Information Sciences Institute da University of South California (EUA). O Popart possui linguagens para a descrição de parsers e analisadores léxicos que são mapeados para código Lisp. As transformações são elaboradas sobre a forma de regras de re-escrita chamadas de Syntax-Directed Experts e podem ser associadas a funções Lisp que usem a interface de programação provida pelo Popart, dando a ele uma potencialidade muito grande em termos de manipulação de descrições.

Outro sistema transformacional disponível e escrito em Lisp é o sistema **Tampr** [Boyle 89] (Transformation Assisted Multiple Program Realization). O Tampr foi desenvolvido por James Boyle no Argonne National Laboratory (EUA), existindo também uma versão em Fortran gerada pelo próprio Tampr. A linguagem utilizada para a descrição de parsers e transformações é chamada de linguagem Poly. O sistema Tampr tem sido muito utilizado na conversão de programas escritos em Lisp para Fortran. Os conjuntos de transformações elaborados tem mostrado ótimos resultados na produção de código eficiente. Alguns trabalhos de porte semi-automático têm sido feito também com as linguagens Pascal e C.

TXL [Cordy 93] é o nome do sistema transformacional desenvolvido pelo Software Technology Laboratory da Queen's University at Kingston (Canadá). TXL tem obtido um grande sucesso entre os interessados por sistemas transformacionais por apresentar uma elegante linguagem de descrição para transformações e também por contar com uma boa documentação, além de exemplos práticos de sucesso entre os seus usuários.

3 Projeto Draco-PUC

3.1 Visão Geral

O projeto Draco-PUC [Leite 94] [Leite 93] [Prado 92] [Leite 91] é desenvolvido pelo Departamento de Informática da PUC-Rio com o objetivo de testar, desenvolver e colocar em prática o paradigma Draco [Neighbors 91] para a construção orientada a domínios de software. Dentro deste objetivo, a equipe responsável pelo projeto tem centrado seus trabalhos na construção do ambiente de desenvolvimento Draco-PUC em que são testadas as idéias que formam a base do paradigma.

Algumas das características que tornam a máquina Draco-PUC relevante no panorama atual de sistemas transformacionais são:

- Sistema gerador de parsers baseado na tradicional sintaxe Lex/Yacc utilizando mecanismo de análise LALR(1) com backtracking.
- Núcleo transformacional totalmente aberto, possibilitando o uso de pontos de controle para o disparo de eventos e direção do fluxo de controle.
- Linguagem para descrição de transformações que permite o uso de transformações locais e globais através do uso de variáveis do tipo sequência e múltiplos localizadores.
- Mecanismo de casamento de padrões altamente expressivo, fornecendo ao usuário um grande potencial para a descrições de padrões utilizando a sintaxe de qualquer linguagem que tenha sido definida através de uma gramática no sub-sistema de parsers.
- Facilidade para o agrupamento de transformações em conjuntos segmentados por domínio, permitindo a utilização racional de escopos de busca para a aplicação de transformações.
- Implementação em C++ com arquitetura dedicada e desenvolvida segundo as experiências obtidas ao longo de vários anos de pesquisa e aprimoramento da tecnologia.

3.2 O Núcleo Transformacional

O primeiro passo na utilização do sistema transformacional Draco-PUC é a definição de uma gramática para a linguagem na qual estarão escritas as descrições a serem transformadas. Caso estas descrições utilizem mais de uma linguagem é necessário definir uma gramática para cada uma das linguagens que compõem as descrições.

Em muitos casos o sistema transformacional é utilizado para realizar o mapeamento entre linguagens diferentes, sendo as linguagens utilizadas no produto final diferente das linguagens de entrada. Nestes casos é necessário realizar também a definição da gramática de cada uma destas linguagens.

As transformações que mapeiam estruturas de uma linguagem em estruturas desta mesma linguagem são chamadas de transformações horizontais. Já as transformações que mapeiam estruturas de uma linguagens em estruturas de uma outra linguagem são chamadas de transformações verticais.

Cada uma das regras de transformação definida possui duas partes principais:

- **Lado Esquerdo (lhs** - Left Hand Side) descrevendo o padrão que deve ser encontrado nas descrições para a aplicação da regra.
- **Lado Direito (rhs** - Right Hand Side) descrevendo o padrão de re-escrita que irá substituir a parte da descrição instanciada a partir do lhs.

No mecanismo de controle primário da máquina Draco-PUC, uma vez que um trecho da descrição tenha sido selecionada e um conjunto de transformações haja sido selecionado para aplicação, este trecho é percorrido de modo bottom-up da esquerda para a direita em busca de possíveis locais de aplicação para as regras de transformação. Caso uma transformação horizontal tenha sido aplicada e o mecanismo de controle de fluxo primário haja sido mantido, o local transformado passa novamente a ser alvo de busca para novas aplicações de regras de transformação dentro do mesmo conjunto de transformações

Quando se faz a utilização da máquina Draco-PUC, colocam-se transformações horizontais em conjuntos de transformações distintos daqueles que contém transformações verticais. Deste modo, caso uma transformação vertical tenha sido aplicada e o mecanismo de controle de fluxo primário haja sido mantido, seleciona-se o próximo local na ordem bottom-up da esquerda para a direita.

Além do lhs e do rhs, uma transformação na máquina Draco-PUC pode disparar eventos e/ou alterar o fluxo de controle de aplicação de transformações através de pontos de controle, aos quais podem estar associados código para desempenhar estas tarefas. Os pontos de controle disponíveis na máquina Draco-PUC cobrem todas as possibilidades para disparo de ações presentes em qualquer sistema transformacional.

Presentemente o código associado aos pontos de controle deve estar descrito em C++ através do uso de uma interface de programação própria para o uso das variáveis instanciadas pelo mecanismo de casamento de padrões e também para a gerência dos múltiplos localizadores que definem segmentos importantes durante a manipulação dos programas.

Os pontos de controle de uma regra de transformação são (Figura 1):

- **Pre-Match**, executado toda vez que a regra for testada sobre um trecho da descrição de entrada.
- **Match-Constraint**, executado toda vez que é realizada uma amarração de variável do padrão definido no lhs com um trecho da descrição de entrada.
- **Post-Match**, executado logo após um casamento entre o lhs e um trecho da descrição de entrada ter sido finalizado com sucesso.
- **Pre-Apply**, executado imediatamente antes que o trecho da descrição de entrada selecionado seja substituido pelo rhs instanciado pelas variáveis amarradas durante o casamento de padrões com o lhs.
- **Post-Apply**, executado após a realização da substituição do trecho da descrição de entrada selecionado pelo rhs instanciado pelas variáveis amarradas durante o casamento de padrões com o lhs.

Como já citado anteriormente, as regras de transformação podem ser agrupadas em conjuntos de transformação, que por sua vez também possuem os seguintes pontos de controle (Figura 1):

- **Initialization**, executado sempre que o conjunto de transformações for selecionado para aplicação.
- **End**, executado sempre que o conjunto de transformações acabar de ser aplicado.

Cada conjunto de transformação também possui uma lista de propriedades na qual são definidos aspectos de controle sobre o tipo de disparo, navegação e controle que serão utilizados.

Por sua vez, os conjuntos de transformações podem ser agrupados em unidades chamadas transformadores (Transformers) que possuem os seguintes pontos de controle (Figura 1):

- **Declaration,** área para a declaração de variáveis que serão utilizadas nos pontos de controle dos conjuntos e regras de transformações.

- **Initialization**, executado sempre que o transformador for selecionado para aplicação.
- **End**, executado sempre que o transformador acabar de ser aplicado

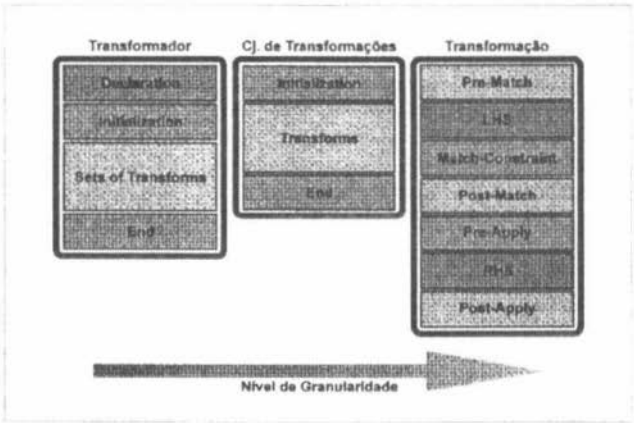

Fig. 1 - Framework Conceitual para Transformações Draco

A introdução destes vários pontos de controle confere ao sistema transformacional da máquina Draco-PUC não só o poder dos sistemas de produção como também o poder dos sistemas reativos.

4 De Cobol para C/C++

O modelo utilizado para o porte de programas Cobol para C/C++ é apresentado na Figura 2. Este modelo é composto por três processos: estruturação e modularização de programas Cobol, conversão direta de Cobol estruturado para C/C++ e estruturação C++.

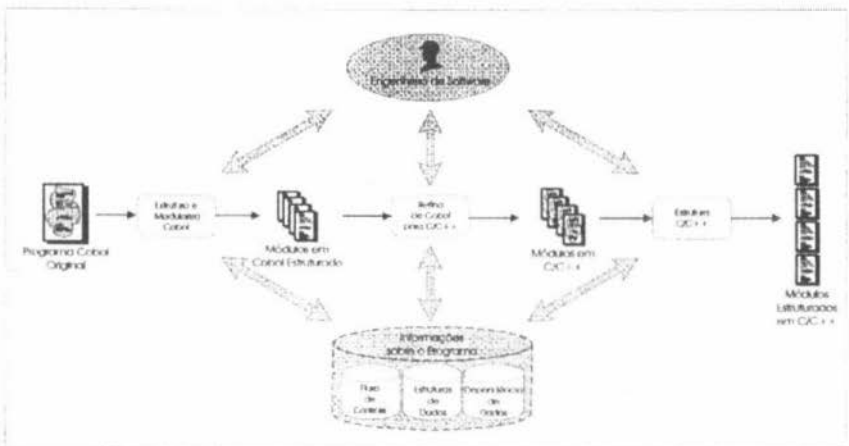

Fig. 2 - O Processo de Porte

4.1 Estruturação e Modularização Cobol

Neste estágio os programas Cobol originais são manipulados com o intuito de dar-lhes a forma de programas elaborados segundo as técnicas de programação estruturada. São realizadas análises de fluxo de dados e controle e a seguir o programa é reestruturado de modo que seja composto por parágrafos que se comportem como procedures.

No passo seguinte é levantado uma referência cruzada a partir do grafo de chamadas, procurando-se analisar o fluxo de controle entre procedures. Deste modo podem ser agrupadas procedures que possuam alto grau de coesão entre si. A partir de uma análise dos dados do programa, indicadores de acoplamento são disponibilizados de forma a possibilitar a definição de módulos que possam ser compilados em separado.

Duas são as formas principais como podem ser feitas a integração e comunicação destes módulos: através da *linkage section*, quando o acoplamento se der através de variáveis em memória, ou através de uma linguagem de script quando o acoplamento se der através de arquivos e/ou bases de dados em disco.

4.2 Conversão de Cobol para C/C++

No processo responsável pelo porte direto de Cobol para C/C++ são realizadas quatro tarefas básicas:

- Encapsulamento das `RECORD DESCRIPTIONS` em tipos abstratos de dados C/C++.
- Declaração de variáveis globais C/C++ a partir das variáveis Cobol, realizando-se a conversão de tipos Cobol em tipos C/C++.
- Mapeamento das estruturas de controle Cobol para estruturas de controle C/C++.
- Mapeamento dos blocos de parágrafos em procedimentos C/C++ sem parâmetros.

Ao término deste processo é gerado para cada módulo Cobol um arquivo de definição (.h) e um arquivo contendo o código C++ que implementa a funcionalidade do módulo (.cc).

Também neste processo estão previstos a existência e uso de bibliotecas que emulem alguns tipos básicos Cobol. Esta biblioteca pode ser linkada estática ou dinamicamente aos módulos gerados quando na construção final do código executável. É importante observar que o grupo de reengenharia deve analisar com cautela a relação custo-benefício da incorporação de novas funcionalidades a estas bibliotecas de apoio. As principais variáveis para esta análise são: eficiência do código gerado, legibilidade do código, esforço de geração do código e eficiência dos algoritmos presentes nas bibliotecas de apoio.

4.3 Estruturação C/C++

O código C/C++ produzido pelo porte direto descrito na seção anterior possui uma forma que não o caracteriza como um programa C/C++ tradicional. O processo de estruturação C/C++ tem por objetivo manipular os diversos módulos produzidos a fim de que este programa seja sobretudo legível e de fácil entendimento segundo as técnicas de programação em C/C++.

Nesta fase a análise de dependência de dados representa papel preponderante pois é a partir dos resultados gerados por ela que as variáveis globais são distribuídas por entre os módulos e

procedimentos. Em paralelo são também definidos os parâmetros de entrada e saída de cada um dos procedimentos (funções).

As técnicas para estruturação realizadas na fase de Estruturação e Modularização Cobol estavam limitadas pelas possibilidades da linguagem Cobol, assim esta fase de Estruturação C/C++ complementa o trabalho realizado anteriormente, permitindo o uso de variáveis locais, funções e tipos abstratos de dados.

Em síntese, esta etapa procura tornar o código produzido semelhante a um código desenvolvido manualmente, tornando-o mais fácil de ser entendido por programadores C/C++.

4.4 Observações Sobre a Automatização

A estratégia delineada não faz qualquer suposição sobre automatização, podendo ser utilizada tanto manualmente como automaticamente. É de se esperar que a sua utilização seja efetuada através do apoio de ferramentas. No presente trabalho oferecemos a possibilidade da instanciação da estratégia através do paradigma transformacional, mais especificamente utilizando o ambiente Draco-PUC.

Ainda sobre a questão da automatização é digno de nota a especial atenção que deve ser dada ao banco de informações sobre o programa que está sendo portado. Este banco de informações captura o fluxo de controle, a dependência de dados do programa e informações sobre as estruturas de dados utilizadas.

O resultado da análise de fluxo de controle do programa Cobol original é utilizado na fase de estruturação Cobol para que seja aplicada a heurística de "clustering". Portanto, em se automatizando a estratégia de porte, torna-se necessário carregar o banco de informações com os resultados da análise de fluxo de controle do programa Cobol original antes que seja realizada a fase de estruturação Cobol.

Como dito anteriormente, o banco de informações possui também informações sobre as estruturas dos dados utilizados no programa Cobol que está sendo portado. O resultado de uma análise de dados que captura a estrutura dos tipos de dados utilizados precisa ser armazenado no banco de informações antes que a fase de porte direto seja realizada. O porte direto utiliza estas informações para que possam ser gerados tipos de dados em C/C++, equivalentes aos tipos utilizados no programa Cobol.

As descrições acima sobre o banco de informações são pertinentes ao segmento deste banco relacionado com o programa ainda em Cobol. Para que o processo possa ser completado é necessário que se armazene também informações sobre a dependência de dados no programa C/C++. Esta análise de dependência de dados torna possível a fase de estruturação C/C++ em que variáveis globais são associadas a grupos de funções e em que algumas delas são transformadas em parâmetros para procedimentos que foram transformados em funções. Estes dados sobre dependência são produzidos antes do início da fase de estruturação C/C++ e caso necessário, são reatualizados, por demanda, durante o transcorrer desta fase.

Portanto o conteúdo do banco de informações possue informações sobre fluxo de controle, estrutura de dados e dependência de dados. A partir do exposto acima, é possível estabelecer a sequência temporal em que as informações devem ser disponibilizadas no banco durante um processo automatizado de aplicação da estratégia de porte.

5 O Uso do Draco-PUC na Automatização

Apresentamos, nesta seção, o processo de encapsulamento parcial da estratégia de porte segundo o paradigma transformacional adotado pelo ambiente Draco-PUC. Um exemplo da automatização é apresentado com base em um programa real.

5.1 Implementando a Estratégia na Máquina Draco-PUC

Para que pudéssemos encapsular a estratégia proposta na máquina Draco-PUC decidimos por instanciar cada uma das três etapas do processo de porte através de um transformador. Para que isto fosse possível foi necessária a descrição dos domínios Cobol e C/C++.

Os elementos indispensáveis de um domínio executável são:

- a **gramática** que permite a geração de um parser para a linguagem e,
- o **prettyprinter** que permite a geração de um unparser responsável por mapear representações em sintaxe abstrata para a sintaxe concreta da linguagem.

Para representarmos a gramática Cobol utilizamos parte da especificação ANSI 85 [Stern 93] e utilizamos a especificação de Bjarne Stroustup para o C/C++ [Stroustrup 91]. Tivemos problemas em adequar estas especificações ao formato LALR(1) do antigo gerador de parsers do Draco-PUC e acabamos por decidir por sua reformulação [Freitas 95]. Estas modifcações foram realizadas sobretudo através da implementação de um mecanismo de backtracking sobre o antigo gerador, agora trabalhando com autômatos não-determinísticos, englobando toda a classe de gramáticas livres de contexto.

Uma vez construídos as gramáticas e prettyprinters do Cobol e C/C++ elaboramos uma pequena biblioteca em C++ que serve de run-time library para os programas gerados. Esta biblioteca encapsula os tipos decimal e string do Cobol bem como algumas características de entrada e saída como tipos de acesso a arquivos e apresentação e recuperação de informações em tela. Esta biblioteca recebeu o nome de *CobLib*.

O processo de porte direto de Cobol para C/C++ foi o primeiro a ser mapeado parcialmente para um transformador, gerando código C/C++ sobre a biblioteca *CobLib*. Este transformador recebeu o nome de *CobC*. Posteriormente atacamos, também parcialmente, os processos Estrutura e Modulariza Cobol e Estrutura C/C++. Os transformadores implementados receberam os nomes de *CobStructurer* e *CStructurer*.

5.2 Estruturando Programas Cobol

A principal função do transformador *CobStructurer* é realizar uma análise de fluxo de controle sobre o programa Cobol e segmentá-lo em blocos que dependam entre si somente através de chamadas do tipo PERFORM.

Para que esta funcionalidade fosse provida, utilizamos a facilidade que o sistema transformacional Draco-PUC nos fornece para anotar trechos de programas com estruturas de dados auxiliares. Utilizando este mecanismo foi elaborado um primeiro conjunto de transformações que analisa o programa de entrada e anota todos os comandos com uma estrutura de dados que define as dependências em termos de fluxo de controle.

Agindo desta forma, ao término do processo de análise temos na estrutura auxiliar um grafo de dependência para o fluxo de controle do programa. Estes dados compõem a parte de fluxo de controle prevista no banco de informações.

O reconhecimento das estruturas de controle é realizado através de transformações que possuem somente lado esquerdo (lhs), executando no ponto de controle Post-Match ações que carregam a estrutura de dados com as informações relevantes.

Após o passo de análise é realizado o passo de estruturação em si. Neste passo são utilizados conjuntos de transformações de suporte que são deflagrados por um conjunto de transformações diretor que detecta fragmentos a serem transformados. Esta detecção dos pontos de aplicação é apoiada diretamente sobre os dados que estão armazenados na estrutura de dados auxiliar. A detecção envolve principalmente o reconhecimento de parágrafos que formem blocos coesos em que os pontos de entrada e saída em termos de fluxo de controle sejam únicos (heurística do tipo "clustering").

5.3 Mapeando Cobol Estruturado para C/C++

Para o refinamento (mapeamento semântico) de programas em Cobol para C/C++ foi desenvolvido o transformador CobC. Este transformador recebe como entrada programas em Cobol estruturado em que grupos de parágrafos formam blocos coesos em termos de fluxo de controle. A dependência de fluxo entre estes blocos se dá apenas por chamadas do tipo `PERFORM`. Os programas refinados para C/C++ devem ser linkados com a biblioteca *CobLib* e não apresentam estruturação C/C++ completa.

O transformador *CobC* é composto por dois conjuntos de transformações principais. O primeiro conjunto de transformações se comporta como um analisador das estruturas de dados definidas na `DATA-DIVISION`. As informações aí colhidas correspondem ao segmento estrutura de dados do banco de informações sobre o programa.

O segundo conjunto de transformações e seus conjuntos de suporte implementam o porte em si. Nestes conjuntos são definidos os mapeamentos semânticos entre o Cobol estruturado e o C/C++. A forma geral do programa é mantida, sendo gerado um programa C/C++ com muito da forma do programa em Cobol estruturado. Algumas das transformações verticais[1] presentes nos conjuntos geradores são apresentadas a seguir:

```
Transform If1
 Lhs: {{dast cobol_statement
      IF [[expr COND]] THEN [[statement *STMTS]]
  }}

 Rhs: {{dast cpp_dstatement
     if ( [[expression COND]] ) { [[dstatement *STMTS]] }
 }}

Transform Accept
 Lhs: {{dast cobol_statement
   ACCEPT ([[INT X]],[[INT Y]]) [[simple_term V]]
 }}

 Rhs: {{dast cpp_dstatement
  [[postfix_expression V]].Accept([[INTEGER_CONSTANT X]],
                                  [[INTEGER_CONSTANT Y]]);
 }}
```

Fig. 3 - Exemplos de Regras de Transformações (*CobC*)

[1]Convém observar que na versão atual de Draco-PUC as diferentes possibilidades de refinamento de um componente são expressas por um conjunto de transformações.

5.4 Estruturando o Código C++

Uma vez que o código gerado em C/C++ pelo transformador *CobC* ainda carrega a forma de programa Cobol, o transformador *CStructurer* tem por objetivo principal dar forma C/C++ às estruturas que não são identificadas como características de programas C/C++. Este passo torna o programa gerado mais legível para um usuário C/C++.

Como exemplo representativo deste transformador temos o tratamento das estruturas de repetição. Em Cobol estas estruturas são implementadas através de combinações **if-goto-label**. Para a manipulação destas estruturas foram criados 5 conjuntos de transformacões.

Dois destes conjuntos são conjuntos de suporte que não são aplicados diretamente, sendo utilizados indiretamente por outros conjuntos. Estes dois conjuntos são parametrizados por um nome de label e recebem como entrada uma lista de comandos C/C++. A função do primeiro conjunto é verificar se há alguma referência ao label na lista de comandos. A função do outro conjunto é substituir referências ao label por comandos de break, assumindo que a lista de comandos de entrada estará envolvida por uma estrutura de repetição.

Outros dois conjuntos são aplicados automaticamente e convertem estruturas **if-goto-label** em estruturas do tipo **do-while** quando se puder verificar que é possível a eliminação dos labels envolvidos na transformação. Estes conjuntos fazem a verificação e aplicação destas regras utilizando os conjuntos de suporte descritos anteriormente.

Por fim um último conjunto de transformações elimina definições de labels que já eram ou se tornaram expúrios durante o processo de estruturação.

A listagem de uma das transformações do transformador *CStructurer* é apresentada a seguir:

```
Transform: Label-If-Goto
Lhs: {{dast cpp compound_statement
  {
    [[dstatement *ST1]]
    [[IDENTIFIER_1]]: [[dstatement ST]]
    [[dstatement *ST2]]
    if ( [[expression E]] ) goto [[IDENTIFIER_1]];
    [[dstatement *ST3]]
  }
}}

Post-Match: {{dast cpp statement_list
  if (apply("FIND_LABEL_CALL", "ST1"))
    return(0);
  if (apply("FIND_LABEL_CALL", "ST2"))
    return(0);
  if (apply("FIND_LABEL_CALL", "ST3"))
    return(0);
  return(1);
}}

Rhs: {{dast cpp compound_statement
  {
    [[dstatement *ST1]]
    do {
      [[dstatement ST]]
      [[dstatement *ST2]]
    } while ( [[expression E]] );
    [[dstatement *ST3]]
  }
}}
```

Fig. 4 -Exemplo de Regra de Transformação (*CStructurer*)

5.5 Exemplo

Para que o entendimento de alguns passos de transformação fiquem claros são apresentados alguns trechos de conversão realizados sobre um programa para catálogo e busca de radares chamado `radar.cbl`. Este programa possui aproximadamente 500 linhas e é representativo por realizar acesso a arquivos, possuir características de programação não-estruturada e implementar uma interface com usuário que não é mapeada diretamente para a linguagem C/C++. No Apêndice, são apresentadas cinco listagens com trechos do programa exemplo. Estas listagens fornecem uma idéia geral da conversão realizada, sendo que as alterações são enfatizadas por sombreamento nos trechos correspondentes.

Na primeira listagem são apresentados trechos do programa `radar.cbl` que expressam estruturas importantes do programa. Nesta listagem são apresentados a descrição do arquivo de dados, algumas declarações de variáveis dos mais diversos tipos e alguns parágrafos que possuem ligação entre si a nível de fluxo de controle.

Na listagem seguinte são apresentados os mesmos trechos de programa após o uso do transformador *CobStructurer*. Não há qualquer alteração na área de dados. Já na área de código, percebemos a incorporação do parágrafo `LISTAR-ERRO-ARQ` e a adição do parágrafo `LISTAR-EXIT`. O objetivo da introdução destes parágrafos é a criação de um bloco coeso constituído pelos parágrafos `LISTAR`, `LISTAR-LOOP`, `LISTAR-END`, `LISTAR-ERRO-ARQ` e `LISTAR-EXIT`. É também alterada a referência a `ERRO-ARQ` que passa a ser uma referência ao parágrafo `LISTAR-ERRO-ARQ` local e é introduzido um desvio incondicional antes do início do parágrafo `LISTAR-ERRO-ARQ` para o parágrafo `LISTAR-EXIT`.

A terceira listagem contém os trechos de programa C++ correspondentes ao trechos de programa Cobol apresentados na segunda listagem. Estes trechos são resultantes da aplicação do transformador *CobC*. É interessante notar a declaração de classe criada para encapsular o arquivo de dados e as diversas declarações de dados que correspondem as declarações de dados Cobol. Também importante é notar como os parágrafos que formavam um bloco coeso foram agrupados no corpo de uma única função chamada `Func_listar`.

Como pode ser percebido o programa gerado (Listagem c) guarda ainda a mesma estrutura de labels e desvios do programa Cobol. Nesta listagem é importante prestar atenção à área em destaque, já que apresentamos na quarta listagem (Listagem d) o passo intermediário da aplicação direta da transformação *Label-If-Goto* sobre este trecho destacado de código.

A quinta listagem apresenta-nos a versão final dos trechos de programa em C++, após a aplicação de todo o transformador *CStructurer*. Vale observar nesta listagem a eliminação de labels e e as conversões de estruturas de repetição do tipo **if-goto-label** em sequências de comandos do tipo **do-while**.

5.6 Definição de Transformações

A linguagem de definição de transformações em Draco-PUC permite que estas sejam definidas tanto declarativamente (padrões) como operacionalmente (pontos de controle). A parte declarativa facilita a expressão da semântica em termos de pré- e pós-condições, mas a validação é feita pelo autor daquela transformação. Portanto cabe ao autor a certificação da qualidade da transformaão proposta. A prática corrente em nosso projeto tem sido

experimental, isto é as transformações propostas são validadas por teste, mas procuramos usar padrões disponíveis na literatura.

A máquina Draco-PUC em sua versão atual fornece pouca ajuda na validação da aplicação das transformações, basicamente tem-se acesso ao trace da aplicação e à nova Dast após a transformação. No entanto, dada a flexibilidade do próprio sistema de transformação, através dos pontos de controle, é possível a construção de uma interface mais amigável para auxiliar a validação de transformações.

6 Conclusão

Portar sistemas de uma linguagem para outra é uma demanda constante dada a rapidez no aparecimento de novas plataformas computacionais. Nosso trabalho procurou atacar esse problema centrado-se no processo de re-implementação dentro de uma estratégia de re-engenharia de software. Descrevemos o sistema transformacional da máquina Draco-PUC e mostramos os resultados obtidos na construção de regras e sua aplicação para portamos programas Cobol para programas C/C++.

É importante observar de que não dispomos de um ambiente pronto para portar programas Cobol para programas C/C++, mas conseguimos mostrar a praticabilidade e a pontencialidade da máquina Draco-PUC, bem como testar um conjunto razoável de regras de transformação. É também importante ressaltar a importância do sistema gerador de parsers com backtracking que facilitou enormemente a construção dos parsers para C/C++ e para Cobol, linguagens com gramáticas muito complexas. Além disso vale observar o uso de estruturas com dados auxiliares para guardar informações sobre o programa transformado, além daquelas informações sintáticas que estão contidas na DAST (Draco abstract syntax tree).

Conforme ressaltamos, nosso trabalho procurou concentrar-se no porte, de forma que a parte de recuperação e re-desenho não foi explorada. Portanto se compararmos nosso trabalho com Guedes e von Staa [Guedes 93], veremos que a estratégia utilizada é calcada na recuperação automática e na re-implementação baseada nas capacidades de linearização do Talisman. Para isso Guedes e von Staa embutiram no recuperador, um parser R*S com ações semanticas, regras para abstrair estruturas de programas C, e montar estruturas organizacionais por módulos. Nossa vantagem é dispormos de um ferramental mais flexível e não acoplado ao parser. Nossa intenção é aprimorarmos os transformadores de estrutura de modo a disponibilizar também transformações de recuperação. Dentro do projeto Draco-PUC, continuaremos a explorar e aprimorar o apóio ao processo de re-engenharia.

Referências

[Arango 86] Arango G, Baxter I.et al., *TMM: Software Maintenance by Transformations*, IEEE Software, 3(3), pp. 27-39, May 1986.

[Baxter 94] Baxter I., *Design (Not Code!) Maintenance*, Palestras Convidadas do VIII Simpósio Brasileiro de Engenharia de Software, pp. 1-7, Out. 1994.

[Boyle 89] Boyle J., *Abstract Programming and Program Transformations - An Approach to Reusing Programs*, in Software Reusability, Vol 1, pp. 361-413, Ed. Ted Biggerstaff, ACM Press, 1989.

[Cordy 93] Cordy J., Carmichael I, *The TXL Programming Language Syntax and Informal Semantics (V. 7)*, Technical Report, Queen's University at Kingston - Canada., June 1993. (TXL pode ser obtido a partir do endereço http://www.qucis.queensu.ca/STI.ab/TXL)

[Edwards 93] Edwards H., Munro M., *RECAST: Reverse Engineering from COBOL to SSADM Specification*, Proceedings of the IEEE 1993 Conference on Reverse Engineering, pp. 44-53, 1993.

[Freitas 95] Freitas F., *A Evolução do Draco-Pargen*, Relatório Técnico, Projeto Draco-PUC, Pontifícia Universidade Católica do Rio de Janeiro: Jan., 1995.

[Guedes 93] Guedes L., Staa A., *Um processo de Re-engenharia Econômico e Eficaz*, Anais do VII Simpósio Brasileiro de Engenharia de Software, pp. 77-91, Out. 1993

[Leite 91] Leite J., Prado A., *Design Recovery - A Multi-Paradigm Approach*, Proceedings of the First International Workshop on Software Reusability, pp. 161-169, Jul., 1991.

[Leite 92] Leite J., Prado A., Sant'Anna M., *Draco-PUC, experiências e resultados de re-engenharia de software*, Anais do VI Simpósio Brasileiro de Engenharia de Software, pp. 115-128, Out. 92

[Leite 93] Leite J., Prado A., Sant'Anna M., *Draco-PUC: A Case Study on Software Re-*Engineering, Proceedings of the Second International Workshop on Software Reusability, pp. 121-124, Mar. 1993.

[Leite 94] Leite J., Sant'Anna M., Freitas F., *Draco-PUC: A Technology Assembly for Domain Oriented Software Development*, Proceedings of the IEEE 1994 International Conference on Software Reuse 94, pp. 94-100, Nov. 1994

[Neighbors 91] Neighbors J., *The Evolution from Software Components to Domain Analysis*. Anais do V Simpósio Brasileiro de Engenharia de Software; pp. 1-14, Out. 1991.

[Newcomb 93] Newcomb P., Markosian L., *Automating the Modularization of Large COBOL Programs: Application of an Enabling Technology for Reengineering*, Proceedings of the 1993 IEEE Conference on Reverse Engineering;, pp.222-230, 1993

[Partsch 83] Partsch, A. and Steinbruggen, R., *Program Transformation Systems*, Computing Surveys, Vol. 15, No.3, Sep. 1983, pp. 199-236.

[Prado 92] Prado, Prado, A. F., *Estratégia de Reengenharia de Software Orientada a Domínios*, DI/PUC-RJ, 1992.

[Reasoning 92] *REFINE User's Guide*, Reasoning Systems Incorporated, Palo Alto, 1992.

[Sant'Anna 93] Sant'Anna M., *Lavoisier: Uma Abordagem Prática do Paradigma Transformacional*. Monografia de Graduação, DI/PUC-Rio; Feb. 1993.

[Smith 85] Smith D., Kotik G., Westfold S., *KIDS: Research on Knowledge-Based Software Environments at Kestrel Institute*, pp. 1278-1295, IEEE Transactions on Software Engineering SE-11, Nov. 1985

[Stern 93] Stern N., Stern R., *The Wiley COBOL Syntax Reference Guide*.

[Stroustrup 91] Stroustrup B., *The C++ Programming Language*, Addison-Wesley; 1991.

[Waters 94] Waters R., Chifosky E., *Reverse Engineering - Progress Along Many Dimensions*, CACM, pp. 22-25, May 1994

[Wile 93] Wile D., *POPART: Producer of Parsers and Related Tools System Builders' Manual*, Technical Report, USC/Information Sciences Institute; Nov. 1993.

```
*
* Trechos de código
*
FILE-CONTROL.
   SELECT ARQ-RADAR
          ASSIGN TO DISK
          FILE STATUS IS FSTAT
          ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD  ARQ-RADAR
     LABEL RECORDS STANDARD
     VALUE OF FILE-ID IS "RADAR.DAT".
  01  ARQ-RADAR-RECORD.
     05 NOME-RADAR PIC X(30).
*
* Trechos de código
*
WORKING-STORAGE SECTION.
  01 TMP-NUM PIC 99999.
  01 FSTAT PIC X(02).
  01 ERR-MSG PIC X(40)VALUE
        "ERRO NO ACESSO AO ARQ RADAR.DAT".
  01 TELA-1 .
     04 LINE0 PIC X(30)VALUE  " MENU PRINCIPAL ".
     04 LINE1 PIC X(30)VALUE  "D- Detecao/Analise".
*
* Trechos de código
*
LISTAR.
    PERFORM LIMPA-TELA THRU LIMPA-TELA-IF.
    PERFORM CABC-PGM.
    DISPLAY (2, 24) HEAD2 OF TELA-2.
    PERFORM DISP-TELA2.
    OPEN INPUT ARQ-RADAR.
    IF FSTAT EQUAL 30 GO TO ERRO-ARQ.
LISTAR-LOOP.
    READ ARQ-RADAR
       AT END,  GO TO LISTAR-END.
    DISPLAY (5, 50) NOME-RADAR.
    DISPLAY (7, 50) FREQ-MAX.
    DISPLAY (9, 50) FREQ-MIN.
    DISPLAY (11, 50) FRP-MAX.
    DISPLAY (13, 50) FRP-MIN.
    DISPLAY (15, 50) LP-MAX.
    DISPLAY (17, 50) LP-MIN.
    DISPLAY (19, 50) SRP-MAX.
    DISPLAY (21, 50) SRP-MIN.
    DISPLAY (23, 20) CONTINUA OF TELA-2.
    ACCEPT (23, 43) VAL.
    IF VAL NOT EQUAL "n" AND VAL NOT EQUAL "N"
       GO TO LISTAR-LOOP.
LISTAR-END.
    DISPLAY (23, 20) FIM-ARQ OF TELA-2.
    ACCEPT (23, 43) VAL.
    CLOSE ARQ-RADAR.
*
* Trechos de código
*
ERRO-ARQ.
    PERFORM LIMPA-TELA THRU LIMPA-TELA-IF.
    DISPLAY ERR-MSG.
    STOP RUN.
```

a) Trechos do Programa Cobol Original

```
*
* Trechos de código
*
FILE-CONTROL.
   SELECT ARQ-RADAR
          ASSIGN TO DISK
          FILE STATUS IS FSTAT
          ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
  FILE SECTION.
  FD  ARQ-RADAR
     LABEL RECORDS STANDARD
     VALUE OF FILE-ID IS "RADAR.DAT".
  01  ARQ-RADAR-RECORD.
     05 NOME-RADAR PIC X(30).
*
* Trechos de código
*
WORKING-STORAGE SECTION.
  01 TMP-NUM PIC 99999.
  01 FSTAT PIC X(02).
  01 ERR-MSG PIC X(40)VALUE
        "ERRO NO ACESSO AO ARQ RADAR.DAT".
  01 TELA-1 .
     04 LINE0 PIC X(30)VALUE  " MENU PRINCIPAL ".
     04 LINE1 PIC X(30)VALUE  "D- Detecao/Analise".
*
* Trechos de código
*
LISTAR.
    PERFORM LIMPA-TELA THRU LIMPA-TELA-IF.
    PERFORM CABC-PGM.
    DISPLAY (2, 24) HEAD2 OF TELA-2.
    PERFORM DISP-TELA2.
    OPEN INPUT ARQ-RADAR.
    IF FSTAT EQUAL 30 GO TO [illegible].
LISTAR-LOOP.
    READ ARQ-RADAR   AT END GO TO LISTAR-END.
    DISPLAY (5, 50) NOME-RADAR.
    DISPLAY (7, 50) FREQ-MAX.
    DISPLAY (9, 50) FREQ-MIN.
    DISPLAY (11, 50) FRP-MAX.
    DISPLAY (13, 50) FRP-MIN.
    DISPLAY (15, 50) LP-MAX.
    DISPLAY (17, 50) LP-MIN.
    DISPLAY (19, 50) SRP-MAX.
    DISPLAY (21, 50) SRP-MIN.
    DISPLAY (23, 20) CONTINUA OF TELA-2.
    ACCEPT (23, 43) VAL.
    IF VAL NOT EQUAL "n" AND VAL NOT EQUAL "N"
       GO TO LISTAR-LOOP.
LISTAR-END.
    DISPLAY (23, 20) FIM-ARQ OF TELA-2.
    ACCEPT (23, 43) VAL.
    CLOSE ARQ-RADAR.
    GO TO LISTAR-EXIT.
ERRO-ARQ.
    PERFORM LIMPA-TELA THRU LIMPA-TELA-IF.
    DISPLAY ERR-MSG.
    STOP RUN.
LISTAR-EXIT.
    EXIT.
```

b) Trechos do Programa após uso do CobStructurer

```
#include "CobLib.h"

class Class_Arq_radar :public CblFile{
  public:
    Class_Arq_radar() : CblFile("RADAR.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
      fseek(file,last_pos,0);
      Write();
     }
};

CblString Arq_radar_record_nome_radar (30);
/*
  Trechos de código
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERRO NO ACESSO AO ARQ
RADAR.DAT");
CblString Tela_1_line0 (30," MENU PRINCIPAL ");
CblString Tela_1_line1 (30,"D- Detecao/Analise");
/*
  Trechos de código
*/
void Func_listar() {

Listar:
  Func_limpa_tela();
  Func_cabc_pgm();
  Tela_2_head2.Display(2,24);
  Func_disp_tela2();
  Arq_radar.Open("rb");
  if (Arq_radar.Status==30)
    goto Listar_erro_arq;

Listar_loop:
  if (Arq_radar.Read()) {
    goto Listar_end;
  }
  Arq_radar_record_nome_radar.Display(5,50);
  Arq_radar_record_freq_num.Display(9,30);
  Arq_radar_record_freq_min.Display(9,50);
  Arq_radar_record_ftp_max.Display(11,30);
  Arq_radar_record_ftp_min.Display(11,50);
  Arq_radar_record_fp_max.Display(13,30);
  Arq_radar_record_fp_min.Display(13,50);
  Arq_radar_record_prp_max.Display(15,30);
  Arq_radar_record_prp_min.Display(15,50);
  Tela_2_continua.Display(21,60);
  Val.Accept(23,43);
  if (Val!="n"&&Val!="N")
    goto Listar_loop;

Listar_end:
  Tela_2_fim_arq.Display(23,20);
  Val.Accept(23,43);
  Arq_radar.Close();
  goto Listar_exit;

Listar_erro_arq:
  Func_limpa_tela();
  Err_msg.Display();
  exit(0);

Listar_exit:
  return;
}
```

c) Trechos do Programa após o uso do CobC

```
#include "CobLib.h"

class Class_Arq_radar :public CblFile{
  public:
    Class_Arq_radar() : CblFile("RADAR.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
      fseek(file,last_pos,0);
      Write();
     }
};

CblString Arq_radar_record_nome_radar (30);
/*
  Trechos de código
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERRO NO ACESSO AO ARQ
RADAR.DAT");
CblString Tela_1_line0 (30," MENU PRINCIPAL ");
CblString Tela_1_line1 (30,"D- Detecao/Analise");
/*
  Trechos de código
*/
void Func_listar() {

Listar:
  Func_limpa_tela();
  Func_cabc_pgm();
  Tela_2_head2.Display(2,24);
  Func_disp_tela2();
  Arq_radar.Open("rb");
  if (Arq_radar.Status==30)
    goto Listar_erro_arq;

do {
  if (Arq_radar.Read()) {
    goto Listar_end;
  }
  Arq_radar_record_nome_radar.Display(5,50);
  Arq_radar_record_freq_num.Display(9,30);
  Arq_radar_record_freq_min.Display(9,50);
  Arq_radar_record_ftp_max.Display(11,30);
  Arq_radar_record_ftp_min.Display(11,50);
  Arq_radar_record_fp_max.Display(13,30);
  Arq_radar_record_fp_min.Display(13,50);
  Arq_radar_record_prp_max.Display(15,30);
  Arq_radar_record_prp_min.Display(15,50);
  Tela_2_continua.Display(21,60);
  Val.Accept(23,43);
} while (Val!="n"&&Val!="N");

Listar_end:
  Tela_2_fim_arq.Display(23,20);
  Val.Accept(23,43);
  Arq_radar.Close();
  goto Listar_exit;
Listar_erro_arq:
  Func_limpa_tela();
  Err_msg.Display();
  exit(0);

Listar_exit:
  return;
}
```

d) Passo de Aplicação da Regra Label-IF-Goto

```
#include "CobLib.h"

class Class_Arq_radar :public CblFile{
  public:
    Class_Arq_radar() : CblFile("RADAR.DAT") {}
     int Read();
     void Write();
     void ReWrite() {
       fseek(file,last_pos,0);
       Write();
     }
};

CblString Arq_radar_record_nome_radar (30);
/*
  Trechos de código
*/
CblDecimal Tmp_num (5,0);
CblString Err_msg (40,"ERRO NO ACESSO AO ARQ
RADAR.DAT");
CblString Tela_1_line0 (30," MENU PRINCIPAL ");
CblString Tela_1_line1 (30,"D- Detecao/Analise");
/*
  Trechos de código
*/
void Func_listar()  {

  Func_limpa_tela();
  Func_cabc_pgm();
  Tela_2_head2.Display(2,24);
  Func_disp_tela2();
  Arq_radar.Open("rb");
  if (!(Arq_radar.Status==30)) {
     do {
       if (!Arq_radar.Read()) {
          break;
       }
       Arq_radar_record_nome_radar.Display(5,50);
       Arq_radar_record_freq_max.Display(7,50);
       Arq_radar_record_freq_min.Display(9,50);
       Arq_radar_record_frp_max.Display(11,50);
       Arq_radar_record_frp_min.Display(13,50);
       Arq_radar_record_lp_max.Display(15,50);
       Arq_radar_record_lp_min.Display(17,50);
       Arq_radar_record_srp_max.Display(19,50);
       Arq_radar_record_srp_min.Display(21,50);
       Tela_2_continua.Display(23,20);
       Val.Accept(23,43);
     } while (Val!="n" && Val!="N")
     Tela_2_fim_arq.Display(23,20);
     Val.Accept(23,43);
     Arq_radar.Close();
     return;
  }
  Func_limpa_tela();
  Err_msg.Display();
  exit(0);

}
```

e) Trechos do Programa após o uso do CStructurer