Uma Arquitetura Transformacional de Apoio a Manutenção de Software

Ulf Bergmann[*]
Julio Cesar Sampaio do Prado Leite[*]
Marcelo Sant'Anna[+]
<bergmann , julio , santanna>@inf.puc-rio.br
PUC-RJ - Pontifícia Universidade Católica do Rio de Janeiro
Rua Marques de São Vicente 225 - Gávea
Rio de Janeiro - RJ - CEP 22453-900

Resumo: Este artigo é uma seqüência a trabalhos anteriores na área de engenharia reversa baseada na maquina transformacional Draco-PUC [1]. Em particular estamos apresentando uma arquitetura voltada para o domínio de problemas cuja solução segue o mesmo padrão do problema do ano 2000 (Problema do A2K). Para isso, apresentamos a caracterização do domínio, identificando os seus cenários de execução, as fases a serem seguidas na construção da solução e listamos alguns dos problemas específicos ao qual essa solução se aplica. A maior ênfase do artigo é a apresentação da arquitetura montada com base na maquina Draco-PUC, através do reuso do domínio EXL (extração de informações) e da construção de heurísticas de análise. Utilizamos o problema do A2K em código COBOL para demonstrarmos a validade de nossa proposta.

Palavras chave: sistema transformacional, engenharia reversa, re-engenharia, reuso, problema do ano 2000.

Abstract: This paper is a sequence of previous works in reverse engineering based in the transformation machine Draco-PUC [1]. In particular we are presenting a architecture turned to a domain of problems which solution has the same pattern of the Year 2000 Problem. As such, we present the characterization of the domain, identifying his scenarios of execution and listing some specific examples where this solution can be applied. The main emphasis of this paper is the presentation of the architecture using the Draco-Puc machine trough the reuse of the EXL (information extraction) domain and the construction of heuristics analysis. We use the Year 2000 Problem for the demonstration of our proposal.

Keywords: transformation system, reverse engineering, reengineering, reuse, year 2000 problem

1. Introdução

O uso de sistemas transformacionais na engenharia reversa é um campo de pesquisa no Departamento de Informática da PUC-Rio iniciado por Prado [6], que propôs a utilização do paradigma Draco [7] para a engenharia reversa do próprio Draco. Esses estudos tiveram prosseguimento com o trabalho de Freitas e Leite [5] para a geração de programas a serem utilizados na construção de ferramentas de engenharia reversa.

Em continuação a estes trabalhos estamos executando os estudos iniciais do Projeto A2K-PUC com parceiros que possuam um grande acervo de sistemas a serem modificados. O objetivo deste projeto é a construção de uma ferramenta de apoio a detecção e correção do Problema do A2K [2][3][4] usando o sistema transformacional implementado pela Máquina Draco-Puc. A principal contribuição deste projeto é permitir uma melhor avaliação da Máquina Draco-Puc em situações reais, tendo em vista que a utilização, por si só, de sistemas transformacionais para resolver o Problema do A2K, já foi proposta pela Reasoning [8].

[*] *Apoio parcial do CNPq*
[+] *Apoio parcial da CAPES*

Na execução destes estudos verificamos a existência de um domínio de problemas que poderiam ser tratados utilizando-se a mesma abordagem inicialmente definida para o Projeto A2K-PUC. Este artigo tem como objetivo propor uma arquitetura transformacional a ser utilizada na construção de ferramentas de suporte a evolução de sistemas. Esta arquitetura está sendo utilizada no Projeto A2K-PUC e acreditamos que possa vir a ser utilizada nos outros problemas do mesmo domínio.

Apresentamos na Seção 2 deste artigo a definição do domínio de problemas que podem ser tratados pela arquitetura proposta. Partimos da caracterização do cenário padrão seguido pelos usuários ao resolver os problemas e concluímos apresentando uma lista de problemas que acreditamos se enquadram no domínio especificado. Na Seção 3 especificamos a arquitetura proposta através do processo de solução do problema, da arquitetura propriamente dita e de um framework para uma interface que proporcionará uma forma de trabalho amigável para os usuários. A Seção 4 apresenta a arquitetura aplicada à solução do Problema do A2K. A solução deste problema foi parcialmente implementada e comprova a validade da arquitetura. Concluindo o artigo, apresentamos a situação atual dos trabalhos referentes ao Projeto A2K-PUC, os problemas encontrados e os trabalhos futuros a serem executados.

2. Definição do Domínio de Problemas Tratados

O domínio de problemas que acreditamos poderem ser tratados pela arquitetura proposta neste artigo é aquele formado pelos problemas que requerem a extração de informações de componentes, a análise destas informações segundo heurísticas definidas pelos especialistas na solução do problema e a manipulação dos componentes originais através da sua modificação ou geração de novos componentes ou relatórios.

Na definição do domínio procuramos utilizar a visão do usuário a respeito de cada problema. Utilizando esta visão, constatamos que os problemas apresentam um mesmo cenário de execução da solução. Este cenário comum é apresentado na Figura 1, onde utilizamos na sua descrição a notação proposta por Leite [9].

Título: Resolver o problema apresentado
Objetivo: Modificar os componentes do sistema de maneira a satisfazer novos requisitos
Contexto: O sistema encontra-se em produção e necessita ser modificado.
Atores:
Engenheiro de Software responsável pela aplicação das modificações.
Especialistas no problema
Recursos: Descrição textual dos diversos componentes do sistema.
Episódios:
Identificar ocorrências analisando os componentes de um sistema e identificando pontos de violação de restrições ou ocorrência de um padrão
Gerar modificações nos componentes a partir das ocorrências identificadas
Gerar novos componentes a partir das ocorrências identificadas
Gerar novos componentes a partir das modificações executadas nos componentes originais

Figura 1 - Cenário comum ao domínio de problemas

2.1 Exemplos de Problemas

Para exemplificar o domínio de problemas, apresentamos a seguir dois exemplos de problemas que acreditamos serem passíveis de solução utilizando a arquitetura proposta.

Tratar o Problema do Ano 2000

O problema do ano 2000 (A2K) refere-se à existência de grandes quantidades de

sistemas que se encontram em produção, onde o armazenamento de datas é feito apenas com dois dígitos. Um cenário de solução para este problema é o apresentado na Figura 2.

Título: Resolver o problema A2K
Objetivo: Modificar os componentes do sistema de maneira a armazenar as datas de maneira completa
Contexto: O sistema encontra-se em produção e continuará sendo utilizado na virada do milênio
Atores:
Engenheiro de Software responsável pela aplicação das modificações.
Especialistas na linguagem de programação utilizada no sistema
Especialistas no sistema a ser corrigido
Recursos:
Código fonte dos programas
Arquivos de Dados
Episódios:
Identificar pontos de erro
Modificar o sistema original de maneira a corrigir os erros identificados
Gerar programas necessários à importação da massa de dados para o sistema modificado
Gerar procedimentos de testes

Figura 2 - Cenário para correção do Problema do A2K

Verificar o cumprimento de padrões de desenho e codificação de sistemas

Uma das maneiras de conseguirmos incorporar qualidade nos sistemas em desenvolvimento é seguir um conjunto de normas ou padrões no seu desenho e codificação. A verificação do cumprimento destas regras e geração de relatórios e modificações pode ser enquadrada neste domínio. Na Figura 3 apresentamos um cenário para este problema.

Título: Verificar o cumprimento de regras de desenho e codificação
Objetivo: Verificar se o sistema em desenvolvimento cumpre as diversas regras existentes quanto ao desenho e codificação.
Contexto: Deseja-se verificar se o sistema está sendo desenvolvido de acordo com as regras estipuladas
Atores:
Engenheiro de Software responsável pela aplicação das modificações.
Especialistas nas regras e padrões de desenho e codificação
Recursos:
Código fonte dos programas
Regras e padrões de desenho e codificação
Episódios:
Analisar os programas fonte identificando violação de regras
Analisar os programas fonte identificando a aplicação de padrões de desenho refletidos no código
Emitir um grau classificando cada desenvolvedor quanto ao cumprimento das regras
Emitir um grau classificando o sistema como um todo quanto ao cumprimento das regras
Gerar modificações nos programas fonte de maneira a atender as regras especificadas

Figura 3 - Cenário para a verificação do cumprimento de regras de desenho e codificação

3. Especificação da Arquitetura

Nesta seção detalhamos a arquitetura básica proposta, o processo a ser utilizado, o uso da Máquina Draco-Puc e mostramos um framework que pode ser utilizado para a construção

da interface com o usuário.

3.1 A Arquitetura

A arquitetura proposta é a apresentada na Figura 4 . A descrição dos componentes da arquitetura é apresentada a seguir:

Heurísticas: armazenam o conhecimento necessário à solução dos problemas específicos. As heurísticas são aplicadas através do acionamento de transformações na Máquina Draco-Puc.

Sistema Existente: conjunto de sistemas de software inerentes ao problema e que serão tratados durante o processo de solução do problema.

Sistema Modificado: sistema obtido à partir do sistema original incorporando as modificações criadas durante o processo.

Relatórios: apresentam informações de interesse dos usuários obtidas durante a execução do processo.

Máquina Draco-Puc: responsável pelo parser dos componentes e pela aplicação das transformações. Na subseção 3.3 apresentamos uma descrição dos recursos implementados e que servem de base para a construção de soluções para problemas específicos.

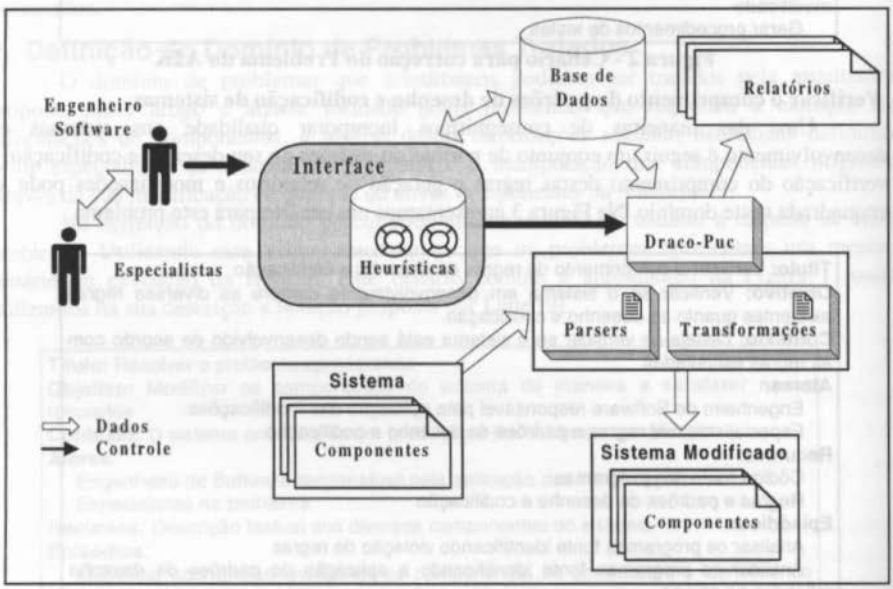

Figura 4 - Arquitetura Proposta

Interface: acessa as informações contidas na Base de Dados apresentado-as ao usuário. Armazena as heurísticas e aciona a Máquina Draco. A subseção 3.4 apresenta uma proposta de framework que pode ser instanciado para cada problema específico.

Base de Dados: conjunto de tabelas que armazenam os resultados obtidos durante a execução do Processo. Estes dados não são de interesse direto dos usuários na solução de um determinado problema mas podem ser utilizados

3.2 O Processo

A Figura 5 apresenta o processo geral a ser seguido no uso da arquitetura proposta na solução dos problemas pertencentes ao domínio.

Na fase de Planejamento são definidas as prioridades, alocação de recursos, técnicas e ferramentas a serem utilizadas. Nesta fase é feita, ainda, a divisão do trabalho em incrementos. Tratando a solução de forma incremental, podemos reusar o conhecimento obtido em incrementos anteriores criando novas heurísticas que podem ser usadas nos incrementos seguintes.

Para cada incremento, inicialmente é executada a fase de Extração de Informações. Nesta fase são retiradas informações dos diversos componentes do sistema, tais como os códigos fonte, arquivos de dados e arquivos de configuração.

De posse das informações do sistema entramos na fase de Manipulação, onde são aplicadas as heurísticas necessárias a modificar ou gerar novos componentes ou relatórios.

Cada incremento deve ser primeiramente testado isoladamente na fase de Testes de Unidades e, em seguida, ser testado em conjunto eom os incrementos já manipulados, caracterizando a fase de Testes de Integração.

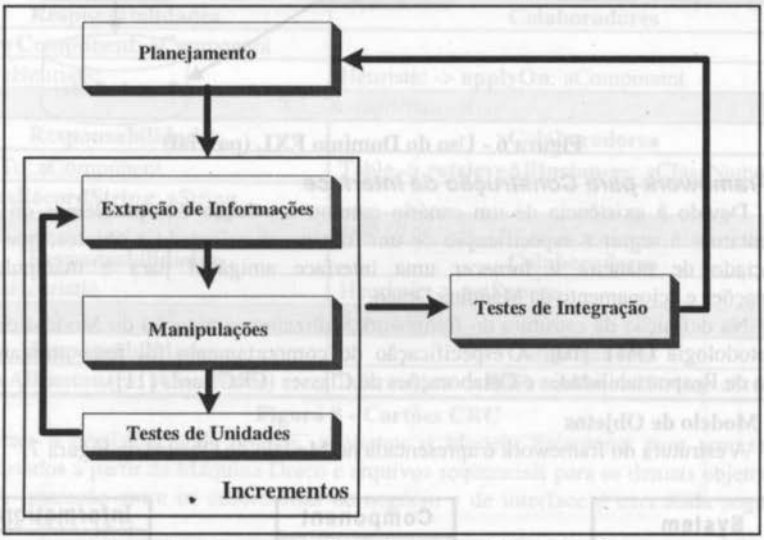

Figura 5 - Processo de Solução do Domínio de Problemas

3.3 A Máquina Draco-Puc

A Máquina Draco é responsável pela identificação de padrões e pela geração de modificações ou novos componentes no sistema original. Este trabalho é executado aplicando-se transformações. Uma transformação é, a grosso modo, a busca de algum padrão no componente de origem e a geração de modificações associadas ao padrão identificado.

O uso da Máquina Draco é feito através da construção de domínios. Um domínio é composto pelo parser da linguagem que constrói uma árvore de sintaxe abstrata usada internamente pela Máquina Draco, pelo pretty-printer que funciona como um unparser traduzindo a árvore de sintaxe utilizando a linguagem do domínio e pelas regras de transformação que permitem manipular as árvores de sintaxe. A Máquina Draco possui diversos domínios já implementados, dentre eles podemos citar os domínios C++, COBOL, Pascal, HTML e VisualBasic.

Na definição das transformações é utilizado o domínio **EXL**[5] (Extraction Language).

Na Figura 6 mostramos o uso do domínio EXL. Este domínio provê uma linguagem mais abstrata para definirmos transformações de extração de informações, armazenamento das mesmas em tabelas de dados e a manipulação destas informações. A linguagem EXL foi estendida de maneira a permitir a geração de bibliotecas dinâmicas necessárias à utilização das transformações diretamente pela ferramenta. Exemplos de uso da linguagem e da Máquina Draco podem ser vistos na subseção 4.2.

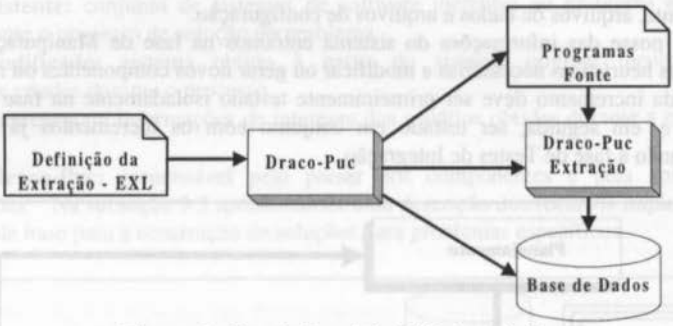

Figura 6 - Uso do Domínio EXL (parcial)

3.4 Framework para Construção da Interface

Devido à existência de um cenário comum à solução dos problemas do domínio, apresentamos a seguir a especificação de um framework orientado a objetos, que pode ser instanciado de maneira a fornecer uma interface amigável para a manipulação das informações e acionamento da Máquina Draco.

Na definição da estrutura do framework, utilizamos a notação do Modelo de Objetos da metodologia OMT [10]. A especificação do comportamento foi feita utilizando-se os cartões de Responsabilidades e Colaborações de Classes (CRC cards [11]).

3.4.1 Modelo de Objetos

A estrutura do framework é apresentada no Modelo de Objetos da Figura 7.

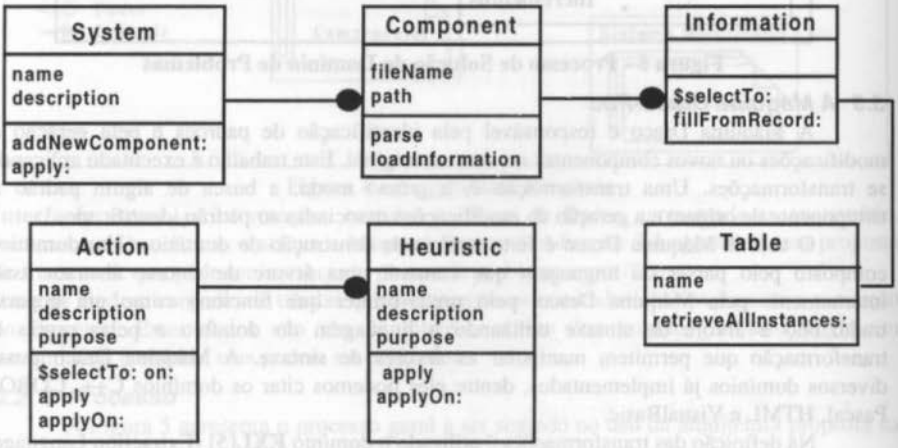

Figura 7 - Modelo de Objetos

3.4.2 Comportamento

A definição do comportamento do framework é mostrada nos *CRC cards* da Figura 8 .

Classe *Component*	
Responsabilidades	**Colaboradores**
parse	$Heuristic -> **selectTo**: #parse **on**: aComponent Heuristic -> **applyOn**: aComponent
loadInformation	$Information -> **selectTo**: aComponent
Classe *Heuristic*	
Responsabilidades	**Colaboradores**
$selectTo: aSymbol **on**: aComponent	
applyOn: aComponent	Draco -> **apply**: aHeuristic
getSource	
Classe *System*	
Responsabilidades	**Colaboradores**
addNewComponent: aComponent	
apply: aHeuristic	Heuristic -> **applyOn**: aComponent
Classe *Information*	
Responsabilidades	**Colaboradores**
$selectTo: aComponent	Table -> **retrieveAllInstances**: aClassName
fillFromRecordString: aString	
Classe *Draco*	
Responsabilidades	**Colaboradores**
apply: aHeuristic	Heuristic -> **getSource**
Classe *Table*	
Responsabilidades	**Colaboradores**
retrieveAllInstances: aClassName	Information -> **fillFromRecordString**: aString

Figura 8 - Cartões CRC

Para a persistência de objetos, adotamos o Modelo Relacional para armazenar os objetos criados a partir da Máquina Draco e arquivos seqüenciais para os demais objetos.

A interação entre os subsistemas de negócio e de interface é executada seguindo o framework MVC [12].

4. Uso da Arquitetura no Problema do Ano 2000

Nesta seção apresentamos o uso da arquitetura proposta na construção de uma ferramenta de apoio à solução do Problema A2K para componentes COBOL. Detalhamos o processo que estamos propondo, uma instanciação do apresentado na subseção 3.2; a arquitetura, com ênfase nas transformações utilizadas e a instanciação da interface, reutilizando o framework apresentado na subseção 3.4.

4.1 O Processo

O processo a ser seguido na solução do Problema A2K é o apresentado na Figura 9 e detalhado a seguir.

Levantamento Preliminar

Levantar as informações necessárias ao planejamento dos trabalhos de correção do Problema do Ano 2000.

Planejamento / Levantamento de Requisitos

Planejar a estratégia de correção do Problema do Ano 2000 definindo os componentes a serem corrigidos, requisitos a serem alcançados, restrições, prazos e custos.

Análise / Localização de Problemas

Localizar pontos de ocorrência de problemas referentes ao Ano 2000.

Transformações / Correção de Problemas

Executar semi-automaticamente as transformações necessárias à correção dos problemas identificados e criar os procedimentos necessários aos testes e reimplantação do sistema na plataforma original.

Testes de Unidades

Testar as modificações realizadas no conjunto de componentes modificados.

Testes de Integração

Integrar o conjunto de componentes modificados à porção do sistema original que já foi corrigida e testada em incrementos anteriores procedendo os testes necessários.

Reimplantação / Entrada em Produção

Colocar o sistema modificado em produção na plataforma original.

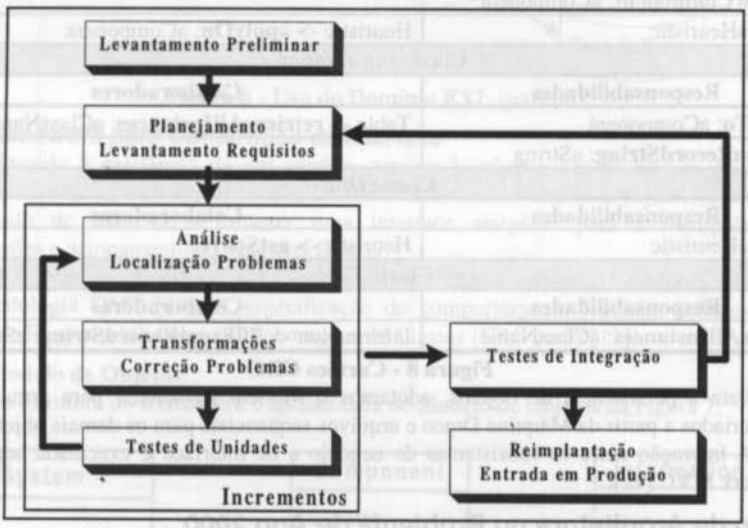

Figura 9 - Processo de Desenvolvimento do Projeto A2K

4.2 A Arquitetura Transformacional

Para definirmos as transformações utilizamos o domínio EXL. O parser COBOL [13] foi utilizado para criar a árvore de sintaxe abstrata dos programas fonte. A seguir, detalharemos os diferentes tipos de transformações utilizadas de acordo com o processo visto anteriormente.

4.2.1 Extração de Informações dos Programas Fonte COBOL

A extração de informações de programas COBOL é executada de acordo com o esquema da Figura 10. O detalhamento de cada transformador é apresentado a seguir.

Transformador AnaED

Analisa a *Enviroment Division* dos programas COBOL. Extrai as informações referentes ao relacionamento do programa com o mundo exterior. Identifica arquivos de dados

e sistemas externos relacionados ao programa. A Figura 11 apresenta o código EXL para a transformação de identificação de arquivos externos.

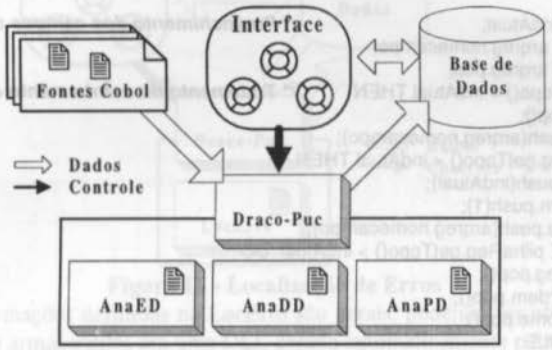

Figura 10 - Extração de Informações

```
IF_MATCH {{dast cobol.select_statement  /* seleciona a regra que deve ser pesquisada */
SELECT [[optional O]] [[ID I]] ASSIGN [[to X]] [[anything *an]] [[file_attrib *FAt]] .  /* padrão */
}} DO
    GET_VAR I AT arqint.nomeint;                    /* Preenchimento dos campos do registro */
    GET_POS I AT arqint.pos;
    strcpy(arqint.nomemodulo,module_name);
    strcpy(arqint.id , CONT.getID());
    CALL_METHOD AnaliseAssign AT an;                /* chamada de outras transformações */
    CALL_METHOD AnaliseAtribs AT FAt;
    INSERT_REG arqint;                              /* Inserção do registro na tabela */
END_IF
```

Figura 11 - Transformação de identificação de arquivos externos

Transformador AnaDD

Analisa a *Data Division* dos programas COBOL. Extrai as informações referentes aos registros dos arquivos de dados e das variáveis de trabalho. A Figura 12 apresenta a transformação que identifica um arquivo de dados. Esta transformação aplica então a transformação *AnaliseReg* mostrada na Figura 13.

```
IF_MATCH {{dast cobol.file_description      /* seleciona a regra que deve ser pesquisada*/
   FD [[ID N]] [[fd_part *fd]] . [[data_description *dd]]      /* padrão de procura*/
}} DO
    GET_VAR N AT ArqAtual;                          /* Gerenciamento da pilha de registros */
    pilhaOrdem.push(1);
    pilhaNome.push("");
    pilhaReg.push(0);
    CALL_METHOD AnaliseReg AT dd;                   /* chamada da transformação para identificar
END_IF                                              o formato registro*/
```

Figura 12 - Código EXL da transformação de identificação de arquivo

Transformador AnaPD

Analisa a *Procedure Division* dos programas COBOL. Extrai as informações referentes à árvore de chamada de sub-rotinas e as variáveis acessadas dentro de cada parágrafo. A Figura 14 mostra a transformação que encontra a chamada de sub-rotinas dentro de um parágrafo COBOL.

```
IF_MATCH {{dast cobol.data_d              /* seleciona a regra do cobol que deve ser pesquisada*/
   [[INT I]] [[ID N]] [[data_part *dp ]] .   /* padrão de procura*/
}} DO
   integer indAtual;
   GET_VAR I AT indAtual;                 /* Preenchimento dos campos do registro */
   GET_VAR N AT arqreg.nomecampo;
   GET_POS N AT arqreg.pos;
   IF pilhaReg.getTopo() = indAtual THEN  /* Tratamento do aninhamento dos registros */
      pilhaNome.pop();
      pilhaNome.push(arqreg.nomecampo);
   ELSE IF pilhaReg.getTopo() < indAtual THEN
         pilhaReg.push(indAtual);
         pilhaOrdem.push(1);
         pilhaNome.push(arqreg.nomecampo);
      ELSE WHILE pilhaReg.getTopo() > indAtual DO
            pilhaReg.pop();
            pilhaOrdem.pop();
            pilhaNome.pop();
         END_WHILE
         IF pilhaReg.getTopo() < indAtual THEN
            pilhaReg.push(indAtual);
            pilhaOrdem.push(1);
            pilhaNome.push(arqreg.nomecampo);
      ELSE
            pilhaNome.pop();
            pilhaNome.push(arqreg.nomecampo);
         END_IF
      END_IF  END_IF
   strcpy(arqreg.nomemodulo,module_name);  /* Preenchimento dos campos do registro */
   strcpy(arqreg.nomearq,ArqAtual);
   strcpy(arqreg.nomereg ,pilhaNome.getTopoAnterior());
   sprintf(arqreg.ordem,"%d", pilhaOrdem.getTopo());
   CALL_METHOD AnaliseTipo AT dp;  /*Transformação para identificar o tipo do campo */
   strcpy(arqreg.tipo , str);
   strcpy(arqreg.id , CONT.getID() );
   INSERT_REG arqreg;                     /* Inserção do registro na tabela */
   pilhaOrdem.incTopo();
END_IF
```

Figura 13 - Código EXL da transformação de análise de registros

```
IF_MATCH {{dast cobol.perform_statement /* seleciona a regra do cobol a ser pesquisada*/
   PERFORM [[ID I]] [[times_until_or_varying_part tu]]      /* padrão de procura*/
}} DO
   GET_VAR I AT arqchama.inicio;          /* Preenchimento dos campos do registro */
   GET_POS I AT arqchama.pos;
   strcpy(arqchama.idparag , ParagAtual);
   sprintf(arqchama.ordem , "%d" , NrOrdem);
   strcpy(arqchama.id , CONT.getID());
   INSERT_REG arqchama;                   /* Inserção do registro na tabela */
   DO NrOrdem = NrOrdem + 1;
END_IF
```

Figura 14 - Código EXL da transformação de identificação de chamadas de sub-rotinas

4.2.2 Localização de Erros

A localização de erros foi implementada no transformador *LocErro* e obedece ao esquema mostrado na Figura 15. A ativação deste transformador é feita pela interface através de heurísticas.

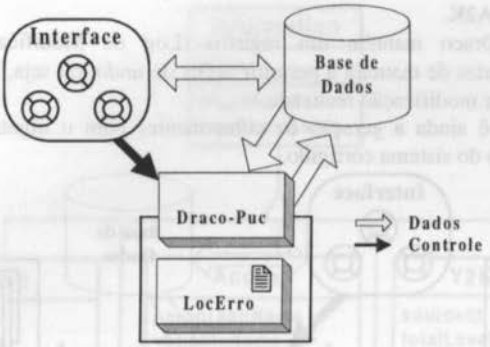

Figura 15 - Localização de Erros

As transformações definidas no LocErro são gerais, podendo ser utilizadas por várias heurísticas, e estão armazenadas em uma DLL gerada automaticamente pela Máquina Draco à partir de sua especificação na linguagem EXL. Um exemplo de transformação de localização é o mostrado na Figura 16 . Esta transformação recebe como parâmetros o nome do componente a ser analisado, o padrão a ser procurado no nome de registros ou campos de arquivos e o nível de erro correspondente. Os erros identificados são armazenados na Base de Dados. O nível de erro corresponde a um nível de certeza quanto a que, naquele ponto, exista um erro do A2K. Por exemplo, para campos que tenham em seu nome o padrão "DATA" podemos especificar um nível de erro de 3. Para campos que tenham seu tipo descrito por PIC 9(6) podemos especificar um nível de erro de 1. Como os níveis de erro são cumulativos, um campo conforme o apresentado a seguir:

10 DATA-NASCIMENTO PIC 9(6)

teria uma grande possibilidade de retratar um erro pois possui um nível de erro igual a 4.

```
EXPORT FUNCTION LocalizaCampoFS( string mod , string str , integer certeza) AS void;
  integer i;
  FOR_EACH arqreg:ind3arqreg(mod) DO      /* para os registros deste módulo */
    IF strstr(arqreg.nomecampo , str) <> NULL THEN  /* o nome do registro tem o padrão */
      strcpy(descerro.arquivo , "arqreg.rel");  /* cria uma nova descrição de erro */
      strcpy(descerro.idlocal , arqreg.id);
      strcpy(descerro.pos , arqreg.pos);
      strcpy(descerro.tipo, NomeCampo);
      sprintf(descerro.certeza , "%d" , certeza);
      getID(descerro.id);
      sprintf(descerro.descricao,"Nome do campotem o padrão l%sl - FC%d",str,certeza);
      strcpy(descerro.nomemodulo,mod);
      INSERT_REG descerro;
      TotalizaErro( atoi(descerro.idlocal) , certeza, atoi(descerro.pos));      /* Totaliza o erro */
    END_IF
  END_FOR
END_FUNCTION
```

Figura 16 - Transformação de localização de erros em nomes de campos

4.2.3 Correção de Erros

A correção dos erros identificados será executada através da aplicação de transformações nos componentes do sistema original. A transformação a ser aplicada é definida pelo tipo do problema identificado na fase anterior e pelas técnicas armazenadas como heurísticas. O Engenheiro de Software pode interferir na seleção da técnica a ser utilizada na correção de cada problema. A Figura 17 apresenta o uso da arquitetura na

correção dos erros do A2K.

A máquina Draco mantém um registro (Log de Modificações) de todas as transformações realizadas de maneira a permitir ações de *undo*, ou seja, o usuário do sistema pode desfazer qualquer modificação realizada.

Esta fase prevê ainda a geração de componentes com o objetivo de simplificar a reentrada em produção do sistema corrigido.

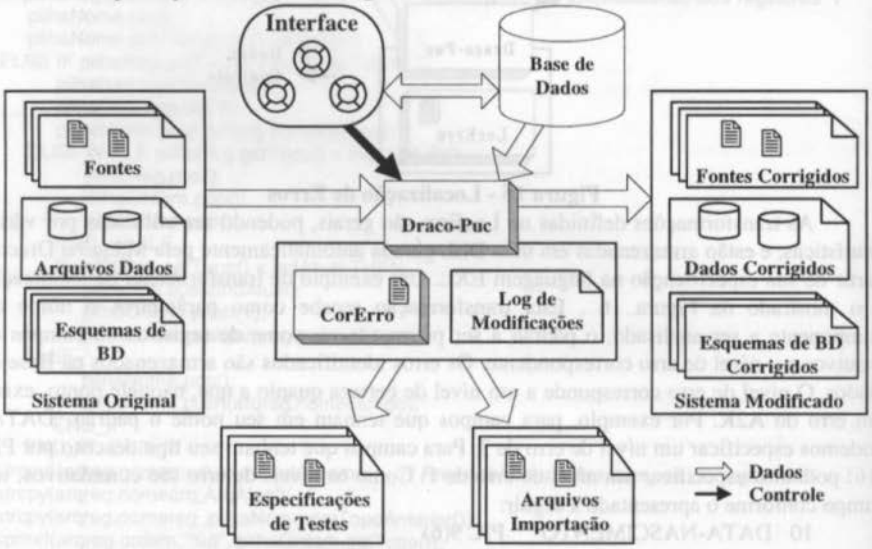

Figura 17 - Correção de Erros

4.3 Instanciação do Framework da Interface

Para mostrarmos a interface construída para a ferramenta utilizaremos o framework apresentado na subseção 3.4. Utilizamos na implementação a linguagem Smalltalk e o ambiente de desenvolvimento IBMVisualAge [14]. As tabelas foram construídas usando-se o Mbase 5.1 [15] por já ter sido utilizado em trabalhos anteriores [5].

4.3.1 Modelo de Objetos

A estrutura da ferramenta criada é apresentada nas Figuras 18 e 19.

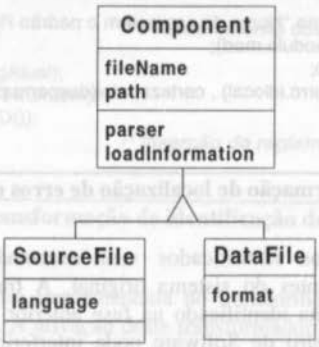

Figura 18 - Modelo de Objetos Instanciado - Componentes

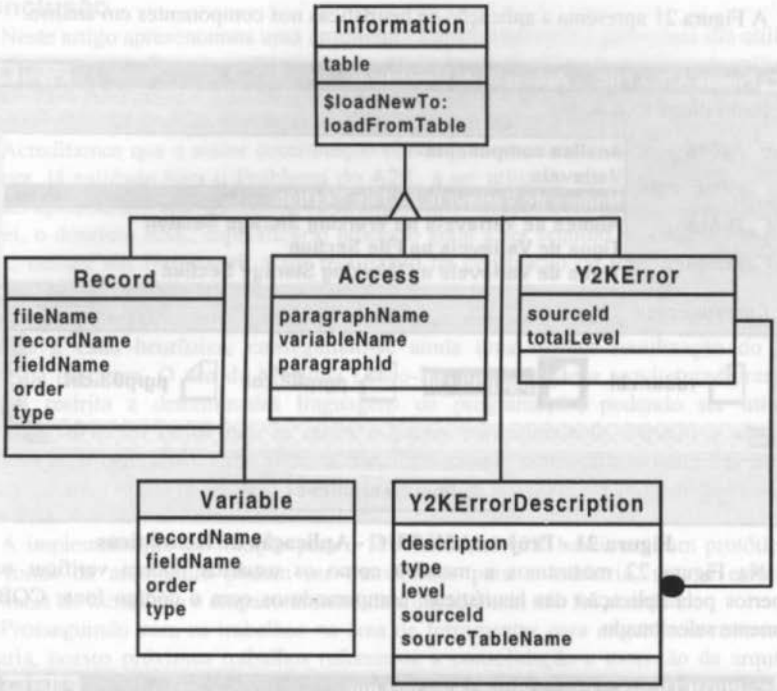

Figura 19 - Modelo de Objetos Instanciado - Informações

4.3.2 Comportamento

A funcionalidade necessária foi implementada através da definição dos métodos virtuais existentes nas classes abstratas do framework. Na Figura 20 mostramos a telas de edição de heurísticas. Estas heurísticas podem ser modificadas durante a solução dos problemas permitindo, desta maneira, a incorporação do conhecimento adquirido durante os trabalhos em cada incremento, conforme visto na subseção 3.2.

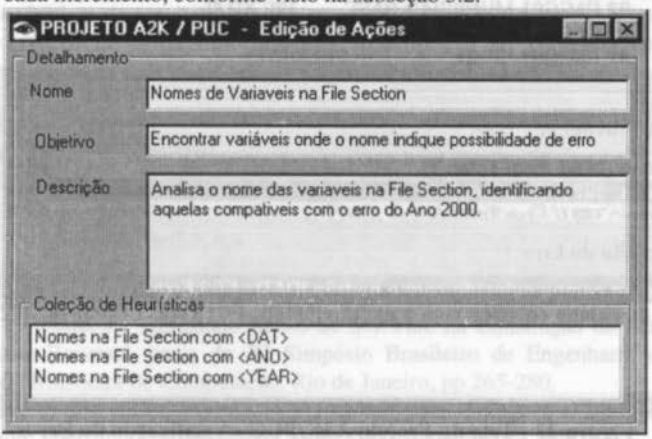

Figura 20 - Projeto A2K/PUC - Heurísticas de Localização de Erros

A Figura 21 apresenta a aplicação de heurísticas nos componentes em análise.

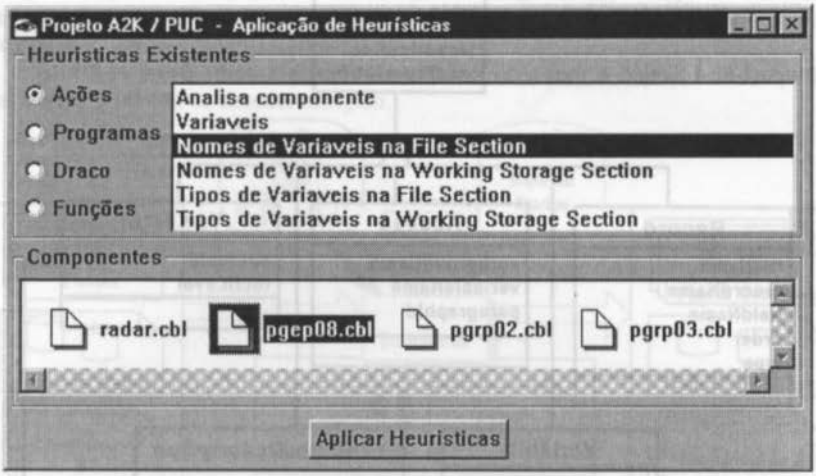

Figura 21 - Projeto A2K/PUC - Aplicação de Heurísticas

Na Figura 22 mostramos a maneira como os usuários podem verificar os erros descobertos pela aplicação das heurísticas, comparando-os com o código fonte COBOL do componente selecionado.

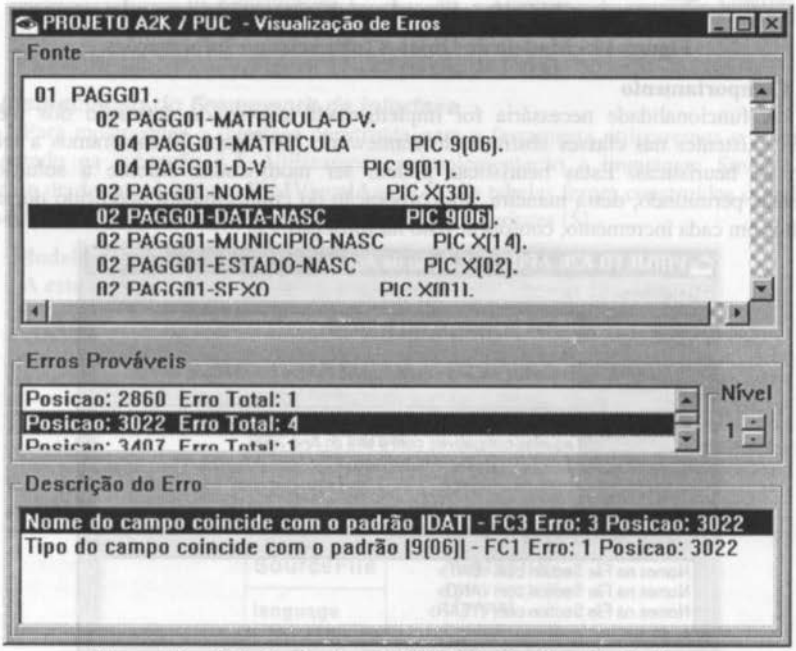

Figura 22 - Tela do Projeto A2K/PUC - Verificação de Erros

5. Conclusão

Neste artigo apresentamos uma arquitetura transformacional e propomos sua utilização em um domínio específico de problemas. Na validação da arquitetura apresentada utilizamos o Problema do Ano 2000 em programas COBOL. As heurísticas utilizadas nesta validação podem ser melhoradas por especialistas no Problema do A2K.

Acreditamos que a maior contribuição obtida por este trabalho foi a obtenção de uma arquitetura, já validada para o Problema do A2K, a ser utilizada na solução do domínio de problemas apresentado. Na montagem desta arquitetura conseguimos reutilizar um domínio de alto nível, o domínio EXL; especificar o processo geral a ser seguido na implementação das soluções; definir um framework a ser reutilizado na construção de interfaces e executar a integração com um sistema transformacional.

O domínio EXL demonstrou ser adequado para implementar as transformações necessárias a cada heurística, conseguindo-se ainda uma melhor reutilização do que a existente na literatura. O uso da Máquina Draco-Puc permite que a arquitetura apresentada não fique restrita a determinadas linguagens de programação, podendo ser utilizados componentes diversos desde que se tenha o parser correspondente. Devido a arquitetura apresentada estar centrada em um sistema transformacional, conseguimos obter um alto grau de reuso na definição das heurísticas a serem utilizadas, como pôde ser comprovado no uso do domínio EXL.

A implementação da solução para o Problema do A2K baseia-se num protótipo. Os pontos fortes da arquitetura podem ser transferidos para a indústria, mas o esforço de transferência de tecnologia e empacotamento da solução ainda não foi abordado.

Prosseguindo com os trabalhos na área de ferramentas para engenharia reversa e re-engenharia, nossos próximos trabalhos referem-se a consolidação e extensão da arquitetura apresentada através da construção do domínio correspondente na Máquina Draco-Puc; da definição e implementação de heurísticas para tratar a correção do Problema A2K e do uso de transformações em outras classes de componentes, tais como arquivos de dados. O tratamento de outros problemas do mesmo domínio é outro trabalho futuro e atualmente estamos em contato com uma empresa que deseja utilizar uma ferramenta semi-automática para certificar sistemas desenvolvidos por terceiros.

Gostaríamos de ressaltar que, até o presente momento, possuímos um protótipo funcional construído a partir da arquitetura especificada neste artigo. Este protótipo possui as funcionalidades básicas da arquitetura e implementa a localização de erros. A implementação da fase de correção de erros está sujeita à formalização de contratos com parceiros interessados no uso da ferramenta.

Bibliografia

[1] Leite, J. et. Al., (1994) 'Draco-PUC: A Tecnology Assembly for Domain Oriented Software Development', in Proceedings of the Third International Conference on Software Reuse, IEEE Computer Society Press, Los Alamitos, CA, pp 94-101.

[2] http://www.inf.puc-rio.br/~draco

[3] http://www.y2k.com

[4] http://www.y2kservices.com

[5] Freitas, F., Leite, J., 'Aplicando Reuso de Software na Construção de Ferramentas de Engenharia Reversa', anais do XI Simpósio Brasileiro de Engenharia de Software, Sociedade Brasileira de Computação, Rio de Janeiro, pp 265-280.

[6] Prado, A., 'Estratégia de Re-Engenharia de Software Orientada a Domínios', Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro, 1992.

[7] James M. Neighbors. 'The Draco Approach to Constructing Software from Reusable Components', IEEE Transactions on Software Enginnering, SE-10(5):564-574, Sep. 1984.
[8] http://www.reasoning.com/y2k.html
[9] Leite, J., et. Al., 'Enhancing a Requirements Baseline with Scenarios', Requirements Engineering, Volume 2 Number 4, pp 184-198, 1997.
[10] Rumbaught, J., et. Al., 'Object Oriented Modeling and Design', Prentice-Hall Int. 1991.
[11] Wirfs-Brock, W., Wierner, L., 'Designing Object Oriented Software', Prentice-Hall, 1990.
[12] Gamma, E., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley, 1994.
[13] Leite, J., Sant'Anna, M. and Prado , A. (1997) 'Porting COBOL Programs Using a Transformational Approach', 'Software Maintenance: Research and Practice', Vol 9, 3-31, 1997
[14] IBM VisualAge Smalltalk Reference Guide, IBM, 1997.
[15] http://cui.unige.ch/~scg/FreeDB/FreeDB.6.html