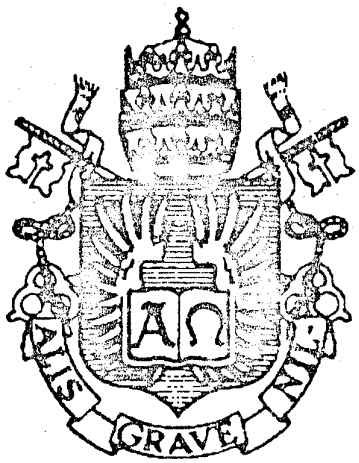

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 35/92

Draco-PUC: a Case Study on Software Re-Engineering

Julio C. S. P. Leite
Antonio F. Prado
Marcelo Sant'Anna

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 35/92
Editor: Carlos J. P. Lucena Dezembro, 1992

Draco-PUC: a Case Study on Software Re-Engineering*

Julio C. S. P. Leite
Antonio F. Prado
Marcelo Sant'Anna

* This work has been sponsored by the Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br
techrep@inf.puc-rio.br (for publications only)

Draco-PUC: A Case Study on Software Re-Engineering

Julio Cesar Sampaio do Prado Leite
Antonio F. Prado
Marcelo Sant'Anna
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil
e-mail:julio@inf.puc-rio.br

September 1992

Abstract

Software re-engineering is a new approach to software maintenance. Instead of working on the source code of systems, we work on high level abstractions and from them procceed in a forward manner reusing the available implementations, when it is the case. As such we view re-engineering as centered on design recovery. We have been working on methods for re-enginnering and applying them to real cases. This article reports on a domain oriented re-engineering method and its use in re-engineering Draco, a machine that produces software by component assembly.

1 Introduction

We understand maintenance as a broad activity embracing not only corrections on the software but also modifications needed for software evolution. Although there are authors that believe that maintenance is part of the development effort, we believe that in most cases of existing software artifacts that view can not be applied. That is, in most cases the artifacts are already dissociated from the processes that created them. As such, maintenance has to be performed independently. Re-Engineering is a well suited approach for situations where maintenance is independent of the process that created the artifact.

The idea of maintenance as a re-engineering process has been pointed by Parikh [Parikh 88] and Chikofsky [Chikofsky 90]. Several researchers have been working with a combination of reverse engineering and forward engineering to enhance the productivity of maintenance tasks. A growing community is devoting research efforts towards methods and tools that help software engineers perform maintenance [Biggerstaff 89] [Baxter 90] [Rugaber 90].

In this article we describe a re-engineering method and focus on its application to the Draco machine. Since the Draco machine is an instantiation of the Draco paradigm [Neighbors 84] [Freeman 87] [Arango 88], details of its architecture are itself of interest to the reuse community. The Draco paradigm puts forward the idea that it is possible to produce software by reusing high level abstractions implemented as component libraries. Our method uses the ideas of the paradigm and the Draco machine structures to represent the recovered design of software artifacts. The work presented here is a follow up of a previous work on design recovery [Leite 91]

The article is divided in more 4 Sections. On Section 2 we review the Draco machine architecture and how it works. Section 3 gives an overall description of our re-engineering method. Section 4 describes how Draco-PUC replaced the original LL(1) Draco parser by Yacc. We conclude listing some lessons learned and describing our future research agenda.

2 Viewing Draco as a Software Production Machine

In the Draco paradigm view, software production is a knowledge intensive task that composes two different types of knowledge, application oriented knowledge and software engineering knowledge. The key point in the Draco idea, as proposed and prototyped by Neighbors [Neighbors 80], is the structure of the reusable resources search space. Neighbors, by proposing a network of encapsulated application knowledge, called **domains**, reduces the task of searching for a suitable reusable resource. In the Draco organization, search takes place not only in a high level of abstraction, but in a domain oriented space. In order to guide the search at a more detailed level and reduce inefficiency brought by usage of several layers of application knowledge, Draco uses encoded software engineering knowledge. The software engineering knowledge in Draco is basically encoded as `tactics` for guiding the search and in `transformations` for tighting up descriptions using the domain network.

The Draco machine can be viewed as a application generator generator, that is a meta generator. As such it has two distinguished parts, one that build the generators, called domains, and other that uses the domains to build software systems. In order to build a domain, it is necessary to build:

- a parser,
- a prettyprinter,
- a component library and
- a transformation library.

Once a domain (generator) has the parts listed above built, it is possible to construct software systems in that domain. Four main subsystems are responsible for the software construction process. Following we describe them.

- Parse: the subsystem responsible for domain programs analysis and the creation of the Draco abstract syntax tree (DAST). DAST is the basic Draco representation.

- Prettyprinter: the subsystem that displays the contents of the DAST using the original syntax of a given domain.
- Transform: this subsystem applies the transformation rules performing manipulations on the DAST. Those transformations are horizontal, that is they are intra domain.
- Refinement: with this subsystem it is possible to perform vertical transformations of the DAST. These transformations are inter domains. Guiding these refinements we have a set of tactics that help automate the process of translating one domain description into other domains.

Although Draco has been discussed and evaluated in different occasions [Neighbors 84], [Arango 86], [Freeman 87] [Arango 88] [Neighbors 91], the Draco prototype itself has been basically the same as the one built in 1980. Most of the work around Draco has been on the ideas surrounding it, and not on empirical work of trying to use it. Our research strategy on reuse is centered on the hypothesis that the Draco idea is sound. As such, our agenda is built around the Draco machine. First trying to make it as usable as possible and then trying to effectively use it to build software.

Pursuing our goal of having a usable Draco, we have re-engineered Draco into what we call Draco-PUC. Draco-PUC version 0.1 was the result of a design recovery process [Leite 91]. It is basically a port from the original UCI-Lisp code, except for a new interface. Version 0.1 was written in Scheme. Draco-PUC version 1.0 has several re-designed parts. The major re-design was in terms of the structure of the parsing mechanism. Draco-PUC v.1.0 uses an off-the-shelf parser generator, Yacc, and has an Draco-Yacc editor that helps domain construction. The production of this new version is a result of a Draco oriented re-engineering method proposed by Prado [Prado 92]. In what follows we will briefly describe such method.

3 A Draco Oriented Re-Engineering Method

Re-Engineering is in our view composed of two process:

- a design recovery process and
- a construction process based on software reuse.

Design recovery as seen by Biggerstaff [Biggerstaff 89] rebuilds design abstractions using a combination of: source code, existing documentation, personal experience and general knowledge of the problem. Design recovery is similar to an archeology process, where the structure of the original design, its architecture, is captured from the existing source code. This process is a bottom-up oriented process where modeling concepts of aggregation, generalization and association are involved. Usually the main source of information is the code, whereas the availability of other sources are dependent of each particular case.

Reuse happens in the process of re-building the artifact according to new needs. This reuse does not implies in an overall reuse of the recovered architecture neither of its implementation. Reusing at the re-design level gives more freedom in the choice of implementations, that is, the existing code can be reused in total, modified or just replaced.

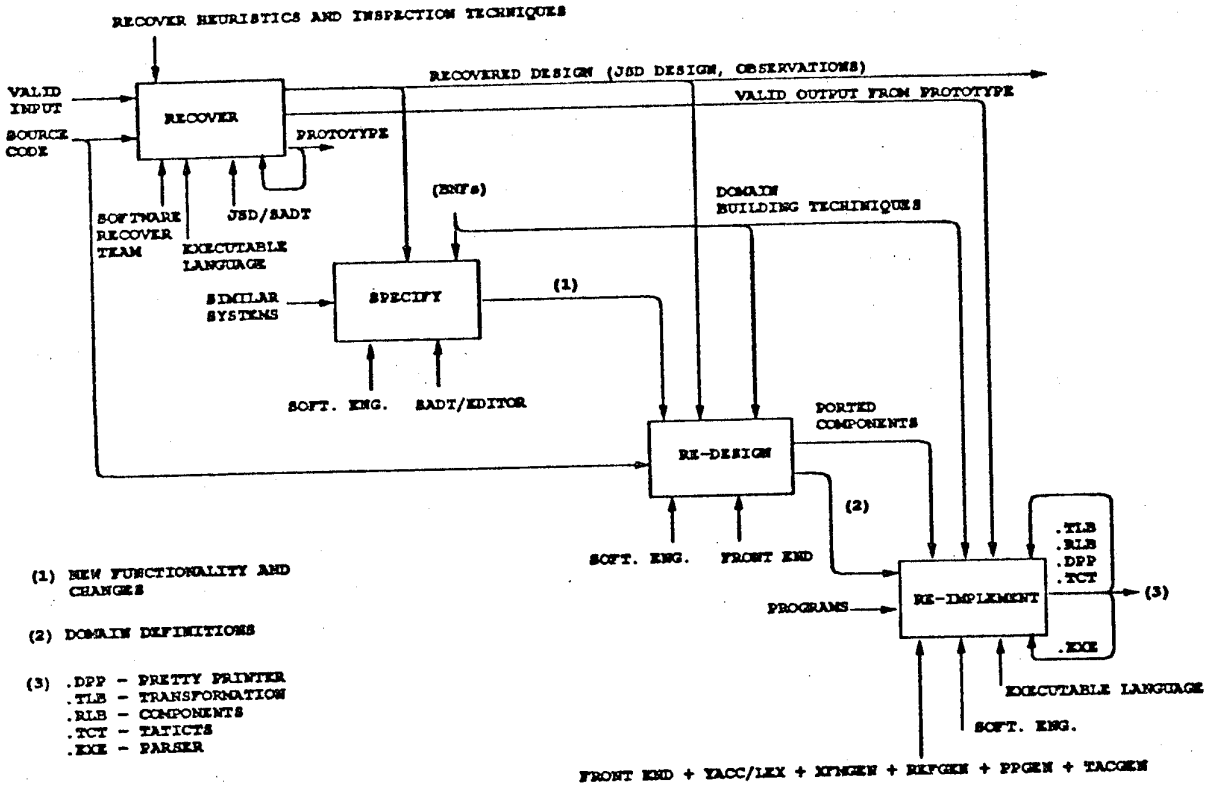

Figure 1: Re-Engineering Method

The re-engineering method used in the Draco-PUC construction is described in a SADT [Ross 77] actigram, Figure 1. This method, proposed by Prado [Prado 92], is composed of four activities: recover, specify, re-design, re-implement. Departing from the source code, the method combines inspections [Fagan 76] and modeling technique, like JSD [Jackson 83] and SADT to recover the design. Using the recovered design, the observations made to it and performing a more detailed study of the problem area, one can specify the desired changes. In the next step those changes are used to re-design the artifact, representing this new design as a Draco domain. Finally the design is re-implemented in a executable language, that can be different from the original implementation language.

3.1 Recovering the Design

The process starts by a first cut division of the artifact in subsystems. This division is necessary in order to better understand the code, from where the design information will be drawn. The identification of subsystems is possible by applying a cross-analyzer and a simple clustering heuristic, driven by the level of cohesion of possible clusters. Although the scope of each subsystem is not clear upfront, the clustering heuristic together with analysis of system's inputs and outputs makes the division good enough to start the process.

In order to recover each subsystem, a inspection process is used. If a running system is not available [Leite 91] a prototype is created. After inspection, the Jackson's structured diagrams are composed in a specification diagram. The inspection process is executed in 4 steps.

- Preparation: the moderator describes the overall area and the main and intermediate goals to be achieved. The participants read the code, plus any other extra material, and the designer is responsible for representing the recovered design using JSD structure diagrams (Figure 2). The level of abstraction chosen for casting the design depends not only on the implementation language, but on the problem itself.

- Inspection: the designer describes each recovered structure diagrams and the implementor and the moderator ask questions. These questions are based on existing checklists and in some recover heuristics. The questioning process tries to discover any existing mistakes, errors or problems.

- Correction: the problems are fixed and a new version of the diagrams are produced.

- Validation: the moderator makes sure the new version is correct. If there is no validation, then the whole process is repeated.

Once the design of each of the subparts are recovered, it is time to integrate each of these subparts in a JSD System Specification Diagram, the network model used in JSD. In this representation, the processes (subparts) interface between each other by means of data stream or state vector. Arguments are usually seen as data stream and global variables are transformed in a fictitious process, globalvar, from where access can be made by state vector or data stream.

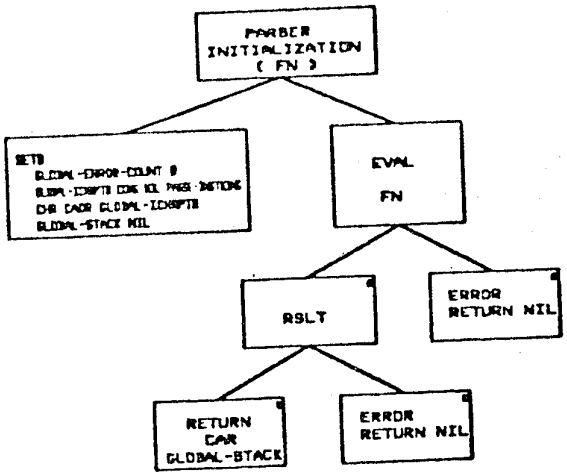

Figure 2: JSD Structure Diagram

3.2 Specifying the Changes

In the activity specify (see Figure 1) changes and improvements are represented in SADT and in text descriptions. Analysis of the recovered design together with the study of similar systems, or the requirements for changes, are the main sources to the specification process. Controlling this activity we have the Draco domain representations.

Several heuristics guide this process [Prado 92]. Following we show a few of them.

- Identify parts that have performance problems. Traces and monitors can help in the identification.
- Use common sense in the choice of new specifications, not proven techniques could be risky to use.
- Keep in mind the basic concepts of software design, coupling and cohesion.
- Use and study similar systems, describe them with your own words
- Validate with the user your understanding of the requirements.

The task of studying similar systems is primarily concerned in obtaining ideas or concepts that could improve the target system of the re-engineering process. There is no effort in customization, that is, there is no direct concern in generalizing concepts from the systems studied. Although our strategy is based on the Draco idea, we are not stressing the idea of domains. We believe that the process of re-engineering applied several times on the same class of systems could, eventually, produce a domain.

3.3 Re-Designing the Recovered Software

In order to represent the new design, we use the Draco domain's four parts: the parser, the set of transformations, the components and the prettyprinter. As pictured in Figure 1, the specification of changes, the recovered design and the techniques for Draco domain construction are determining the way the re-design is performed. We should note, however, that it would be possible to have parts of the code being re-designed. This special re-design, called porting, happens when a part of the semantic, not represented as Draco components, needs to be ported to a different target language.

There are 5 big steps in the process of re-design:

1. grammar definition
2. components and tactics definition
3. porting
4. transformation definition
5. prettyprinter definition

Besides defining a domain for the software being re-engineered, it is necessary to have defined and implemented the target domain, here understood as an executable domain. In order to have a target domain, for instance C or Pascal, we need a parser for that domain, and a prettyprinter. A transformation library, although recommended, is not mandatory.

It is worth to note that to perform re to the target language and have both defined as Draco domains, it is possible to rewrite any system -engineering by porting [Arango 86] it is not necessary to recover the design of the artifact being re-engineered. That is, if we write the refinements from the source language from the source language into the target language, without understanding the system being rewritten.

3.4 Re-Implementation of the Recovered Software

In this phase of the re-engineering process we build the Draco domain. As such we have to make operational, the parser, the prettyprinter, the component library and the transformation library. The Draco-PUC machine has an editor that helps the construction of the domain parts.

Once the domain is available the software engineer can use it as a generator, and as such improves the possibility of different specifications for that artifact, but reusing the same implementations of the original artifact. One needs to specify a set of programs to represent the new artifact (see Figure 1). These programs would be read by the machine and will produce the needed software in the target language of the implementor's choice.

4 Integrating the LALR Strategy with Draco

Of the changes made in the original design, the one that was harder and with high impact was the replacement of the original LL(1) parser by a LALR(1) parser. This change used, with

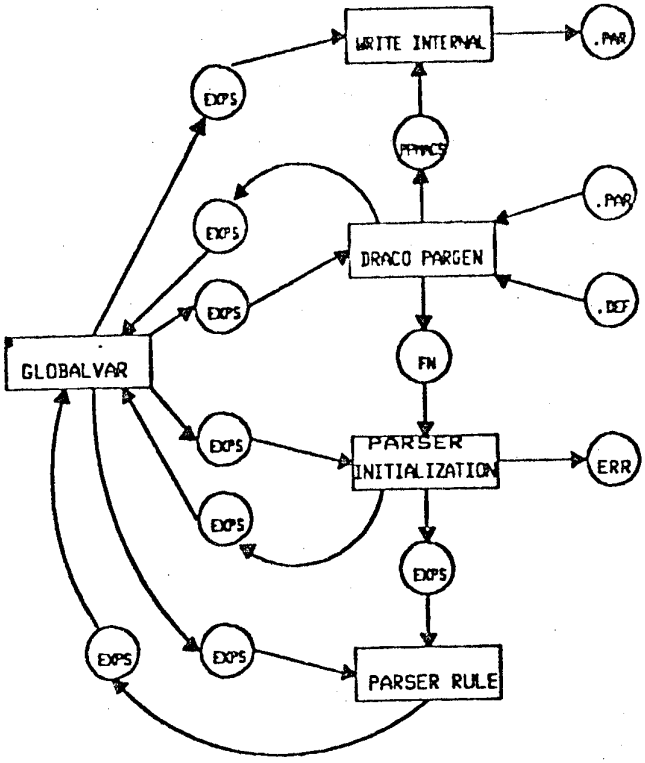

Figure 3: JSD PARGEN System Specification Diagram

some modifications, the described method. The PARGEN and PARSE designs were recovered and represented in JSD. In the specification the main input was the study of similar systems, as well the observations made about Draco-PUC version 0.1. The re-design main focus was on the integration of Yacc with the rest of the Draco system. Another aspect new to the design was a grammar oriented editor, also integrated to the Draco system. The implementation used C as the target language.

4.1 Recovering PARGEN – PARSE

A cross-analyzer, the Draco-PUC prototype (version 0.1) and readings of the literature on Draco, were the information sources used to map all the Draco subsystems to JSD diagrams. Figure 3 shows one of these diagrams for the PARGEN subsystem. As such it was possible to completely separate all the Lisp functions implementing the PARGEN and PARSE subsystems. In the exercise of recovering we were helped by the weak coupling of Draco modules, which shows that the original Draco had a good structure at this level of abstraction.

The recovered design made it possible to separate the differents parts of the PARGEN – PARSE subsystems. The original subsystems were divided in four main parts:

- the construction of the LL(1) parser,
- program parsing.

- the construction of the DAST,
- the mechanisms for multiple domains DAST and

4.2 Specification of Changes

The study of parsers generators performed by one of us [Prado 90] together with observations made on the use of Draco and literature on Draco evaluation [Arango 88] were the main sources for the specification of changes.

The study performed in recovering the design signaled the feasibility of changing the original LL(1) system with an off-the-shelf LALR parser generator. Two main advantages were the more widespread use of LALR systems and the availability of off-the shelf systems to build parsers. One disadvantage was the problem of multiple domains, since bottom-up parsers does not allow the flexibility of multiple entry points. Our choice of parser generator was the Yacc package, and as such we now have a clear division of lexical analysis (Lex) and syntactic analysis.

Another new feature added to Draco was the specification of a new interface. The interface follows the standard used in several Borland products, where the main menu is posted on the top of the screen and each of the selections opens its own menu. An important feature of the interface specification was the introduction of a grammar oriented editor to help the editing and consistency of Yacc grammars.

4.3 Re-Designing PARGEN – PARSE

In this phase of the method (Figure 1) we did represent the design by the Yacc grammar, Figure 4, and by a library of Lisp functions written in C. Since we were using a closed package, there was no need to detail the semantics, and as such we did not use the component library, neither the transformation library.

The library of Lisp functions was used to help the porting of original modules of Draco to C. In Figure 5 we show one of those modules, AgendaAdd, rewritten in C. Looking in detail at the Figure we can observe several Lisp functions (car, assoc, set) used in the C function AgendaAdd. Here we have had several levels of reuse. First, in building a library of C functions to implement Lisp we were using Lisp design. Second, coding in C the original modules, we were reusing the original design and re-using the Lisp/C library. The original modules ported to C were those modules dealing with the the construction of the DAST, the mechanisms for multiple domains DAST and program parsing. The identification of these parts in the recovering phase made it possible the replacement of these modules by C functions easing the integration of the new PARGEN-PARSE with the rest of Draco.

The design of the interface system was not expressed as a Draco domain. Its design was represented in JSD.

4.4 Re-Implementing PARGEN-PARSE

The integration of the Yacc package, the ported modules and the new interface was performed using C as the implementation language. The interface with the part of the system not re-

```
espec : definicoes MARCA regras final ;

final : /* vazio */
      | MARCA ;

definicoes : /* vazio */
           | definicoes definicao ;

definicao : LCHAVE /* ações semânticas */ RCHAVE
          | UNION '(' /* definições de tipos */ ')'
          | START ID
          | EXPECT NUMBER
          | PURE
          | reservada tagdef lista ;

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

tagdef : /* vazio */
       | TTIPO ;

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

seqlista : ID
         | ID NUMBER ;

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

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

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

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

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

Figure 4: Yacc Grammar

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

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

Figure 5: A Ported Module using the Lisp/C library

```
tfmfile : steq ;

steq : st
     | steq st ;

st : '(' TRANS atomo atomo sexpn sexpn ')'
   | '(' CLASS atomo listaid
   | '(' PVARS listaid ')'
   | '(' ERASEPVARS ')'
   | PVAR ;

sexpn : '(' lista1 ')'
      | atomo
      | '(' ')' ;

lista1 : sexpn
       | lista1 sexpn ;

listaid : atomo
        | listaid atomo ;

atomo : TOKID
      | TOKINT ;
```

Figure 6: XFMGEN grammar

designed, still in Scheme, was performed using the creation of a new process (spawnl command in DOS) and files.

The Draco-PUC version 1.0 is working using this hybrid system. The consequence of this implementation is the use of a standard for parser generation and an interface that helps the software engineer in the construction of the domain parts.

5 Conclusion

Draco re-engineering followed the method described here. Several modifications were performed, altering the design of Draco. The subsystems for building the the transformations and the tactics were written as domains and as such their coupling with the whole system was made weaker. The subsystem for building the component library was split in two: one for creating the library and another for filling the library. In Figure 6 we can see the grammar for the XFMGEN, the subsystem responsible for building the transformation library. The re-design of the parsing mechanism used in Draco was the central change. We have described how we have done it.

In the process of applying the method, we made several observations regarding the Draco machine and the re-engineering process itself. Following we list some of these observations.

- Although we were not using a domain analysis strategy, the recover and specify tasks managed to get the needed information, but are still very dependent on the person doing the task and in the problem addressed.

Subsystem	Language	Lines of Code
INTERFACE	C Assembler	8.200 600
DOMAIN BUILDING	C	4.900
PARSE	C	3.200
TRANSFORM AND REFINE	Scheme	3.000
	Total	19.900

Figure 7: Draco Statistics

- Although we do have a grammar oriented editor, forming the DAST is still not trivial. We still need to put by hand the necessary semantic actions for the DAST annotation.
- The DAST has a right leaning tree shape that makes trees more deep than necessary.
- We need to have a mechanism to help the engineer in writing grammars with multiple entry points.
- Writing refinements for one programming language to another is not trivial when dealing with input output statements. We should learn more about this type of conversion, and try to use standards to define executable domains.
- Although the method was used in a non trivial re-engineering exercise, it is necessary to have more examples of its use, in order to continue its validation.

Draco-PUC is a hybrid system, that runs on the DOS platform. It has a combination of three different programming languages (Figure 7). Currently, we are at the final phase of re-engineering Draco-PUC version 1.0 to Unix, where the interface will use the X standard. In this re-engineering, the interface will be designed as a Draco domain. Our goal is to have the necessary infrastructure to allow us to explore the Draco paradigm. In paralel with the effort put on the machine, we are also building domains to start the domain network, a fundamental resource to test the paradigm. We have a member of our group working on a subset of the Data Base domain, and another re-engineering a hypertext system. There is a lot of work to be done not only at the basic machine, but at the supporting domains. It is essential to build this infrastructure in order to start empirical studies on the use of Draco, as well as having grounds for studying supporting theories for the paradigm.

References

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

[Arango 88] Arango G., *Evaluation of a Reuse-based Software Construction Technology*. Proc. Second IEE/BCS Conference on Software Engineering 88. The British Computer Society, July 1988.

[Baxter 90] Baxter, I. *Transformational Maintenance by Reuse of Design Histories*. PhD. Dissertation, University of California, Irvine, USA; Nov., 1990.

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

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

[Fagan 76] Fagan, M., *Design and Code Inspections to Reduce Errors in Program Development* , IEEE Software, pp. 222-240, Jan. 1990.

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

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

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

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

[Neighbors 80] Neighbors J., *Software Construction Using Components*, PhD. Dissertation, Dept. Of Information and Computer Science, University of California, Irvine, 1980.

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

[Neighbors 91] Neighbors J., *The Evolution from Software Components to Domain Analysis*, V Simpósio Brasileiro de Engenharia de Software, Ouro Preto, MG, Out. 1991.

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

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

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

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