

# Reusing Domains for the Construction of Reverse Engineering Tools

Felipe Gouveia de Freitas and Julio Cesar Sampaio do Prado Leite  
Departamento de Informática  
Pontifícia Universidade Católica do Rio de Janeiro  
Rua Marquês de São Vicente, 225 Rio de Janeiro, 22453-900, Brasil  
{freitas, julio}@inf.puc-rio.br

## Abstract

*One of the challenges of reverse engineering is the extraction of a specification from source code. Our work proposes a singular approach to the construction of reverse engineering tools. Using a transformation based software machine we have built two languages designed to help the construction of reverse engineering tools. Each one of these languages embodies an important domain in reverse engineering. This approach is based on the idea that reuse in a high level of abstraction is possible, if a domain is written for a class of problems. This is the premise of the Draco paradigm, a reuse based strategy for software construction. We also demonstrate the use of these languages, by writing a reverse engineering tool that was able to derive a specification from a system built by others and of which we had no previous knowledge. This paper explains how we have built the languages and how they were reused for building a reverse engineering tool.*

## 1. Introduction

Our view of software maintenance is biased towards reengineering. We believe that software organizations should use a reengineering approach to deal with maintenance. In this context, it is very important to have methods and tools that support reengineering. This paper reports on a domain oriented reuse strategy for automatic generation of reverse engineering [27] tools. The generated tools are similar to other proposals, but the novelty is the process by which these tools are generated. The tools are generated by a software machine driven by domain knowledge.

We understand domain as a class of problems and believe that this knowledge can be encapsulated in a language. This is the central idea of the Draco paradigm for software construction [19]. In this paradigm, reuse is achieved by the usage of high level languages (domain

languages) geared to a particular class of problems (domain). As such, reuse is performed at a high level of abstraction. Software, in the Draco paradigm, is specified in the language of its domain. For instance, if we need a software system for lift control we will specify it using the lift language. This special language will embody enough lift knowledge such that different lift control software could be specified using the same language. The paradigm is supported by a transformation based software machine that provides an environment for the construction and usage of these domain languages.

Reverse engineering tools construction is the class of problems we have addressed. In studying the domain, Section 3, we identified the need for two different languages to support the specification of reverse engineering tools: an extraction language and a graphical model language. These languages are used by reverse engineers in order to write a specification for a particular reverse engineering tool, which will be able to extract information from source code and to present it according to a given graphical model.

Building such languages, we will empower the reverse engineers in their ability to write specifications of different strategies to recover information from source code. As such, the reverse engineer will have the possibility of customizing the strategy for recovering and presenting information. This is only possible by the reuse of specific languages, achieving reuse at a high level of abstraction.

The Draco paradigm, Section 2, concept was instantiated by a software machine, called Draco [19], which embodies an insightful combination of programming languages and artificial intelligence in order to deal with complex integration of components. We, at PUC-Rio, have been working on a Draco clone for a while [12] [14]. We now have a version of Draco-PUC that, although similar to the original Draco, has a strong emphasis on the transformation technology.

This paper reports on the first two complex languages we have built using Draco-PUC. We have defined syntax

and semantics for both of them, Section 4. Section 5 gives an example of the application of the reverse engineering tool we have produced. We conclude stressing our contributions, as well as pointing out what was learned and what are our research plans.

## 2. The Draco Paradigm

The Draco paradigm may be seen and characterized by several viewpoints [6]. The original motivation for its development was to provide a way of building software from pre-existing components. It may also be seen as a generator generator, being able to build and maintain a class of similar systems. A third viewpoint is that of high level domain languages to be used in the systems construction process supported by a transformation mechanism capable of generating executable programs from the given specification.

What is behind these definitions is a way of organizing components for reuse that is different from the approaches based on libraries. In the Draco paradigm, the reuse elements are the formal languages named domains. These languages are built with the objective of encapsulating objects and operations of a given domain. The programs written in the domain languages need a refinement process to reach an executable language.

In order to build software from the point of view of the Draco paradigm, it is first necessary that the domain knowledge be formalized, with well-defined syntax and semantics. Once these domain languages are available, it is possible to achieve reuse in the domain level of abstraction. Clearly the paradigm has a very high initial cost, but on the other hand will lower considerably the cost of writing a particular software. The paradigm pays off when a domain will be reused several times.

The Draco-PUC machine, on the other hand, is a software system that aims to implement the paradigm. The role of the machine is to implement domain languages. For that purpose it uses a powerful general parser generator, a prettyprinter generator, and a powerful transformation system. The transformation system is able to handle vertical and horizontal transformations on an abstract syntax tree representation. Since the translation occurs between languages, it is necessary to have executable domains in languages like: C, Pascal, Basic, Lisp and others. So these domains are the target of the refinement process implemented in Draco-PUC.

The process in which a specification written in a source domain reaches a target domain goes through several other domains. A set of domains that are linked together is called a domain network. This property

distinguishes DracoPUC from the usual generator generators, which usually uses a single refinement step. Reusing previous domain languages, Draco makes the strategy of divide and conquer for building complex domains possible.

Section 4 details the languages we have built.

## 3. The Architecture of a Typical Reverse Engineering Tool

By reviewing recent work on reverse engineering [9], we believe that four of them had a major influence in our vision of an ideal reverse engineering tool:

- the project GRASP[2] that aims to produce graphical representations of program static structures from source code,
- the project RECAST[5] that aims to create a method, with tools, to deal with reverse engineering of COBOL programs,
- the work at MITRE[3] that aims to build a transformation library with transforms geared to the recognition of architectural elements in source code, and
- the work of Jarzabek and Keam[10] that proposes a tool strategy anchored on a knowledge base of source code information.

Our analysis of the literature on reverse engineering conclude there are four major aspects any tool must address. They are: how to extract information, how to store the information, how to visualize the information and how to selectively visualize parts of the recovered information. Based on that, we designed an architecture to implement these requirements (Figure 1). The architecture was based on three key design decisions.

The first design decision was to use a transformation system for the extraction of information [3] [10]. As such, we decided to use the Draco-PUC transformation system, previously used in Draco's reengineering [13].

The second decision was to use a database to store the information extracted from the code [10]. This decision was based on the following arguments:

- Need to work with a large volume of data, since different views of the source code need to be available.
- Freedom from limits on the size of the recovered system.
- Need to access repeated times the recovered information without the need to proceed a re-analysis of the source code.

The third decision was related to information visualization. We need to work with different types of

information and at the same time cross reference them. In order to achieve this, Freitas [8] built a generic model<sup>1</sup> viewer, reusing the graphical library wxWindows[29][24]. The library eases the inclusion of new models and has the concept of links between visualized objects.

The architecture, Figure 1, has an extractor as a set of transformations working on the DAST (Draco Abstract Syntax Tree) is the internal representation used by the our transformation system. A DAST is an annotated AST with directives for tagging the syntax tree of the programs to be recovered. The extractor needs a Draco-PUC parser and a set of Draco-PUC transformations. The extracted information would be stored in MetalBase [17], which is a public domain software that implements, without junctions, a relational database.

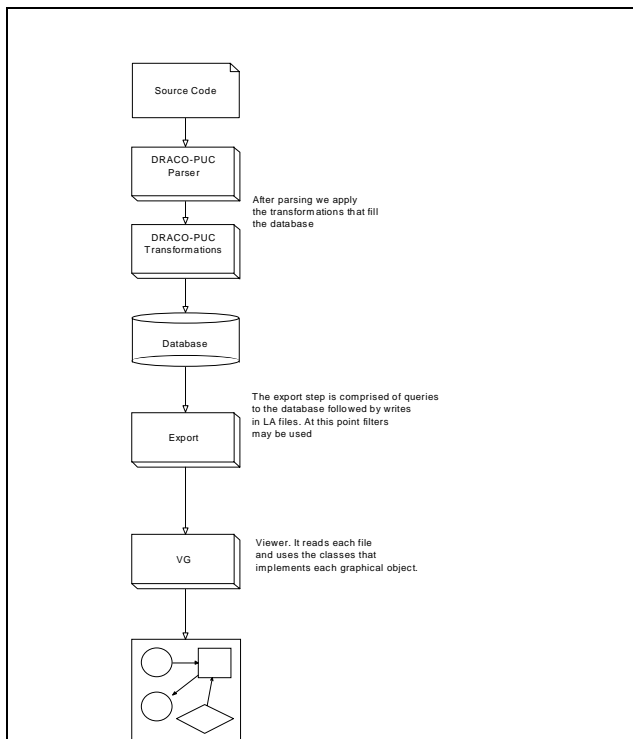


Figure 1 - The proposed Architecture

The viewer was implemented and named VG. It works reading a file written on a general attribute language LA (which makes it possible that the objects, edges or nodes, be described by their attributes) and dynamically loading classes to make it possible the visualization and navigation of recovered object models.

<sup>1</sup> A model, in general, is a graph with a given semantics. Examples of models are: Booch diagrams [1], an association diagram [23] and a entity structure diagram [16].

In order to integrate the database with the viewer it is necessary to export the recovered information in the database to the viewer using the attribute language LA. This is achieved by procedures that read the database and write LA files.

To attain our requirement of selected visualization we decide to implement a filter capability before generating the graphics. This filter is encapsulated in the EXPORT part of the architecture, since it uses the database and as such has a global view of the recovered program.

Our approach to the domain of reverse engineering was driven by the viewpoint of *programming concepts* [25]. As such, we focus our attention on extracting the information from the source code of existing systems. We do not deal with *human concepts* [11] [25] were other information sources are used. We also concentrate our attention on the tool aspect of recovering information from source code.

#### 4. The Domain Network Built to Support the Generation of Reverse Engineering Tools

The architecture (Figure 1) we just proposed reflects the state of the art, but its reusability is limited. Using the Draco paradigm we managed to provide a more abstract, hence more reusable, solution. This solution makes it possible the generation of different tools for reverse engineering, which may work with different source languages as well as with different visual languages.

Given this possibility and trying to avoid strategies like the one proposed in [3], in which the software engineer needs to write the transformations, even though eventually reusing the ones available in the transformation library, we decide to build a network of domains to support the generation of *programming concepts* oriented reverse engineering tools.

We have used the classification proposed by Neighbors [18], which divides domains as:

- application domains – which express the semantics of real world domains or problem classes,
- modeling domains – which are domains built to help the implementation of application domains, by encapsulating concepts with a broad range of reusability, and
- executable domains – which are domains to which a social accepted translator exists, as it is the case of well known programming languages.

A traditional domain network has as a starting point an application domain, and may involve several application domains and modeling domains in order to be grounded in an executable domain. We have proposed the following domain network:

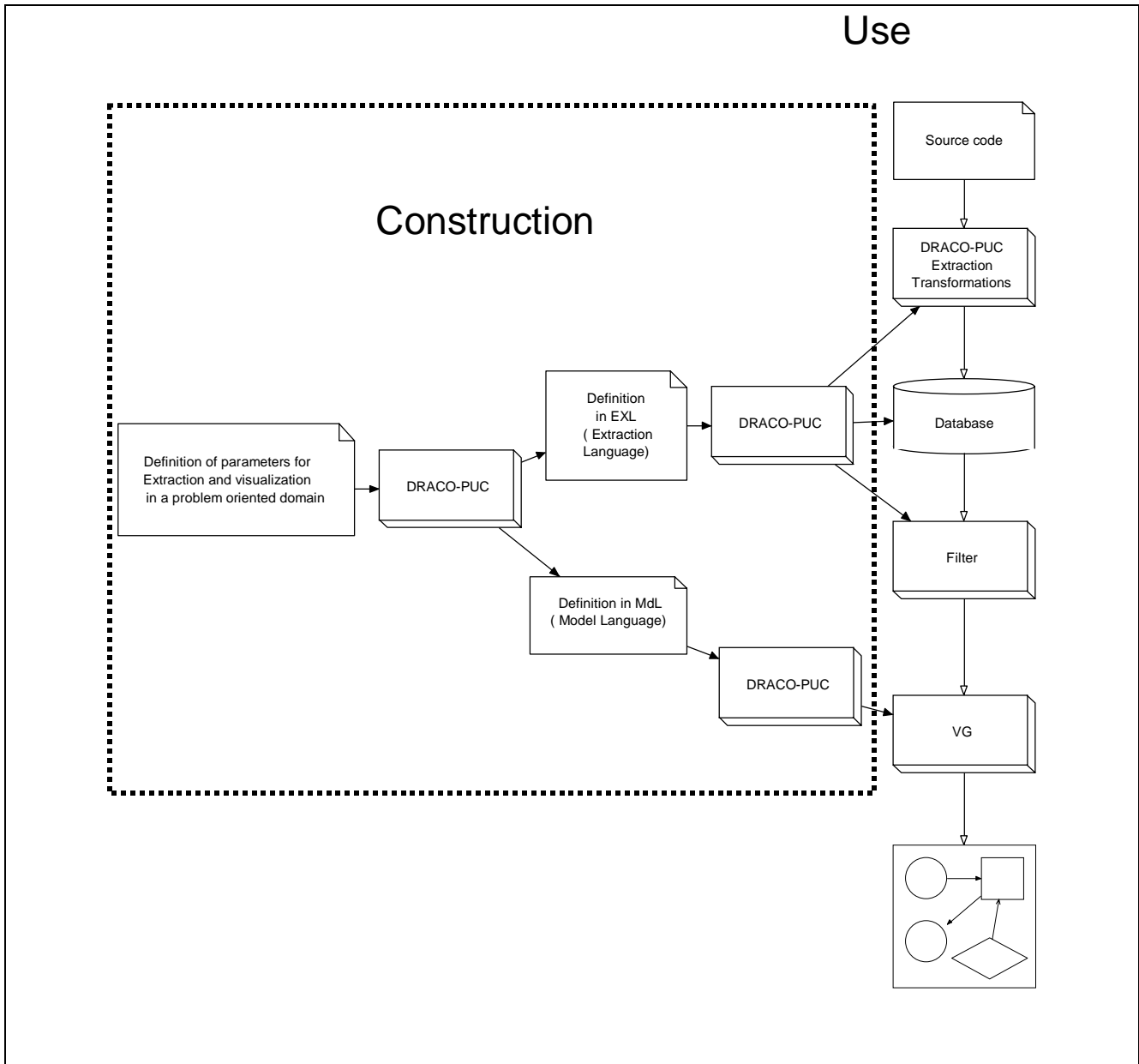


Figure 2 - Domain Network

In this domain network (Figure 2) we may observe that we have two modeling domains: the extraction domain and the model domain. The extraction domain, EXL (*Extraction Language*), is a high level language that embodies: data definition, information extraction, database insertion and database exportation. The model domain, MdL (*Model Language*), is a language that makes it possible the description of different drawing conventions for a software model.

These domains make it easier the implementation of reverse engineering application domains, since the engineer will not need to be worried with database, transformation languages, viewers and other details. So by providing this middleware, the possibility of reuse at a very high level of abstraction is enhanced. A comparison between Figure 1 and Figure 2 shows that the domain network provides a more abstract model, which emphasizes the reuse of knowledge already encapsulated (extraction language and model language). It is also important to notice that, based on these modeling domains, we may build an application domain, like for instance a Y2K domain in which the focus of attention

will be the problems of data manipulation that involves dates.

Considering the proposed domain network, the process of building and using of tools would have the following steps for a hypothetical domain D:

1. The encapsulation of the reverse engineering domain knowledge D, which means building a parser for the D domain as well as the transformations that implements its semantics. These transformations would have as target the EXL and the MdL domains.
2. The writing of a specification in the D language that addresses a given application in the D domain.
3. The use of the resulting programs (extractor and viewer) to analyze and visualize a given program in the executable language X. Note that a parser for X has to be available.

As of today [9], we have focused our efforts in using, without describing a particular application domain, like the D above, the modeling domains available. So, for each of the modeling domains, EXL and MdL, we have built a parser, a set of transforms from the language to C++ to describe the language semantics, and a prettyprinter.

## EXL

The EXL domain language has three sections: a data definition section, an extraction method definition section and an exportation section. The separation has the main purpose of making it simple the task of the specifier. The sections are interconnected. For instance, if we include a table record, automatically the semantics for inclusion will be updated in the extraction section and the semantics for reading will also change in the exportation section.

The design of this language had a strong influence of fourth generation database management languages. For this reason it does contain high level commands such as FOR\_EACH and EXIST. On the other hand, in the extraction section, EXL aims to simplify the work for extraction by means of more appropriated commands for these tasks, see for instance the IF\_MATCH command,

Follows an example of the extraction section, that discovers which class belongs to which modules (mdef). The result is stored in the MetalBase database.

### EXTRACTION SECTION

```
PUBLIC BOTTOM_UP_METHOD Analysis
```

```
IF_MATCH {{ dast cpp.class_specifier
  class [[CLASS_NAME CN]] { [[member_declaration *mdef]]
}
}}
DO
```

```
GET_VAR CN AT class_name;
GET_VAR CN AT classe.nome;
INSERT_REG classe;
strcpy(modper.nomeclasse,class_name);
strcpy(modper.nomemodulo,module_name);
INSERT_REG modper;
CALL_METHOD MetInline AT mdef;
END_IF
```

This example shows an EXL code with a search method (PUBLIC BOTTOM\_UP\_METHOD Analysis) and a pattern matching command (IF\_MATCH...DO...END\_IF). The command recognizes the pattern in the C++ DAST (dast cpp.class\_specifier) and copies the value of CN to a global variable "class\_name" and to the field "nome" of the table "classe." After that, the current record is inserted in the table "classe" asserting the existence of the class. The next step is to fill the record of the "modper" table with the values of the actual class and module (the global variable filled previously) making the insertion. The command ends by invoking the method "MetInline" over the variable "mdef" that will search in this variable, declarations for inline methods. We also observe the use of the "GET\_VAR", "INSERT\_REG" commands to capture and store information about the definition of a class and the "CALL\_METHOD" command that calls another extraction method.

The IF\_MATCH...DO...END\_IF command will generate around 30 lines of Draco-PUC transformation code, which will be responsible for changing the DAST. Comparing to SCRUPLE [20], Draco-PUC transformations works directly on an annotated syntax tree and has a similar pattern language, but instead of having specialized wildcards like SCRUPLE, we have a general wildcard (\*), which is used together with the grammar rule descriptor. As such we believe our pattern language to be simpler as well as not dependent of any particular language. We also have a choice of search methods that we use according to the kind of transformation we apply. Note that the transformation used here does not change the DAST, but retrieves information and stores it a database.

Follows an example of the exportation section:

### EXPORTATION SECTION

```
FUNCTION LoadModules(file file1) AS void;
integer curr;
PRINTF file1,
"[nTitle='Modulos'\nobjfile='mod.of'\nBC=0\nArranged=0\n]n";
FOR_EACH modulo:ind1 modulo DO
  sprintf(strtmp,"%s.vg",modulo.nome);
  IF modulo.nome[strlen(modulo.nome)-1]='h'
  THEN
    PRINT file1,"Node %s
[nTitle='%s'\nX=100\nY=100\nW=80\nH=80\nLink(1)=Todas
Classes@classes.vg'\nLink(2)='Classes@%s'\nNodeType='NHeader'\n",
modulo.nome,module.nome,strtmp;
```

```

ELSE
  PRINTF file1,"Node %s
[\nTitle=%s\nX=100\nY=100\nW=80\nH=80\nLink(1)='All
Classes@classes.vg\nLink(2)='Classes@%s\nNodeType='NBody\n]",
modulo.nome,module.nome,strttmp;
END_IF
END_FOR
DO curr=0;
FOR_EACH moddep DO
  IF EXIST modulo:ind1modulo(moddep.nomedependente)
  THEN
    IF EXIST modulo:ind1modulo(moddep.nome)
    THEN
      PRINTF file1,"Edge e%d
[\nSource=%s\nDest=%s\nEdgeType='MDep\n]",
curr,module.nome,module.nomedependente;
      DO curr=curr+1;
    END_IF
  END_IF
END_FOR
END_FUNCTION

```

This shows an exportation procedure that exports to a file named “file1” all the modules and their dependencies.

This function does not have a return value and uses the local variable “curr” of the type integer. The function does write in the file using the syntax of the LA language (attribute language), needed by VG, the viewer, (see Figure 2). The writing is done by the command PRINTF. First the function scans the module table generating the graph nodes. After that, the dependency table ( “moddep” ) is generated, that is the edges are generated. The local variable “curr” is used to produce a sequential numbering of nodes and edges.

In this example we can also observe database commands like FOR\_EACH and the access syntax to the tables, as “modulo:ind1modulo(moddep.nome) “, which means that we are accessing the records of the “modulo” table by means of the “ind1modulo” index and which follows the “moddep.nome” criteria.

EXL is not a code analyzer language [4] [26], the programs written in EXL will be able to transverse a DAST (representing the system to be recovered), find specific patterns and group them in a repository. The transverse strategy and the grouping will be specified by the reverse engineering according to EXL syntax. Note that we are reusing transformation knowledge as well as database knowledge in the EXL language.

## MdL

The MdL purpose is to help us in the construction of the recovered models. These models are composed of objects that can be: nodes, edges and pages. The page contains the edges and nodes, and the edge does have a source node and a destination node.

The models as we could see are easily modeled by using objects, so that was the paradigm we choose to write the MdL domain. The MdL domain was written

using a very useful characteristic of the Draco paradigm, that is to allow programs to be written mixing several languages. The MdL domain works together with the C++ domain, in which we write the methods of the MdL classes. In the MdL domain we describe the behaviour of the three main objects, so the MdL domain was designed as class definition language.

In the following example we show the node “Sequential” of the JSD entity structure diagram [16].

```

NODE Sequential {
METHOD:
{{method_bodyCPP.function_definition.RC("vg.ctx")
  virtual void Draw(VGDC *dc)
{
  float strw,strh;
  int strx,stry;
  dc->DrawRectangle(x, y,w,h);
  // Does not allow that the node external region be invalid
  dc->SetClippingRegion(x,y,w,h);
  // Calculates the title position
  dc->GetTextExtent(title,&strw,&strh);
  strx=x+(w-strw)/2;
  stry=y+(h-strh)/2;
  DrawText(dc,title, strx,stry);
  // Draws the special node character
  DrawText(dc,"*", x+w-13,y+1);
  dc->DestroyClippingRegion();
}
}}
};

```

In this example we describe the “Sequential” class that embodies the “Draw” method. The syntax for domain change appears after the word “METHOD:”, where there is an identification that there was a change from the domain MdL by the rule “method\_body” to the CPP (C++) domain by the rule “function\_definition” using the context file “vg.ctx”[7].

The drawing method makes the rectangle, followed by the title and then by the indication of interaction or selection or sequence. The methods “SetClippingRegion” and “DestroyClippingRegion” take care of not drawing out of the established region.

The great advantage of MdL is that the defined classes encapsulate completely the aspects of dynamic loading of models, characteristic not available in C++ and very important for our architecture.

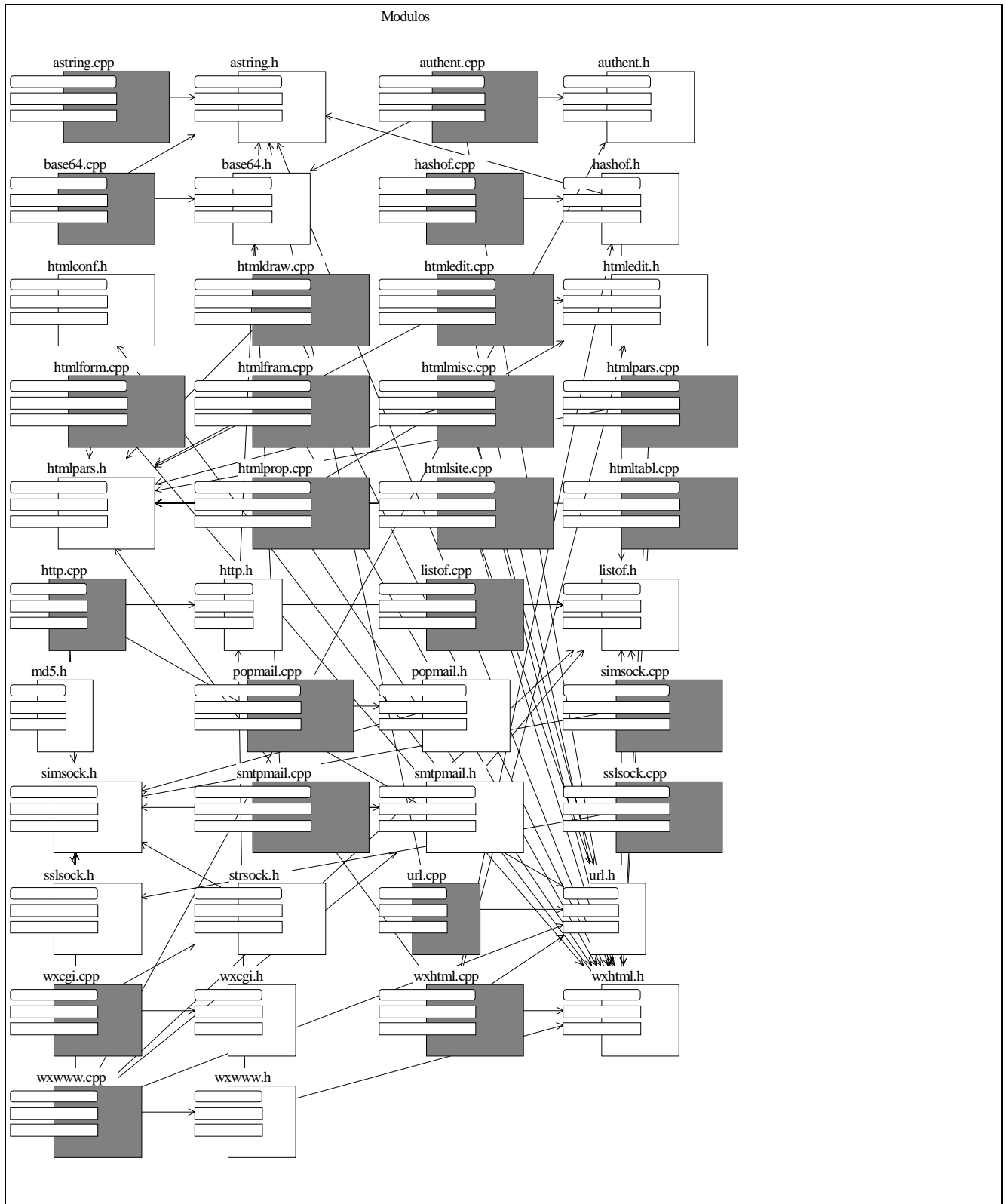


Figure 3 Booch Diagram for the whole system

## 5. Results

We have used the domain network (Figure 2) to recover information of a medium size real system, instead of a toy example [21]. The system we used is called wxWeb[28] and implements a HTML browser. This public domain browser has the major functionality of the well-known Netscape and Internet Explorer, having features like mail and authentication. It is important to stress that this system was not known to us before building our domain network. In order to test the strategy we browsed the Internet trying to find a medium sized public domain system with source code available. The system wxWeb is written in C++ and uses internet technology.

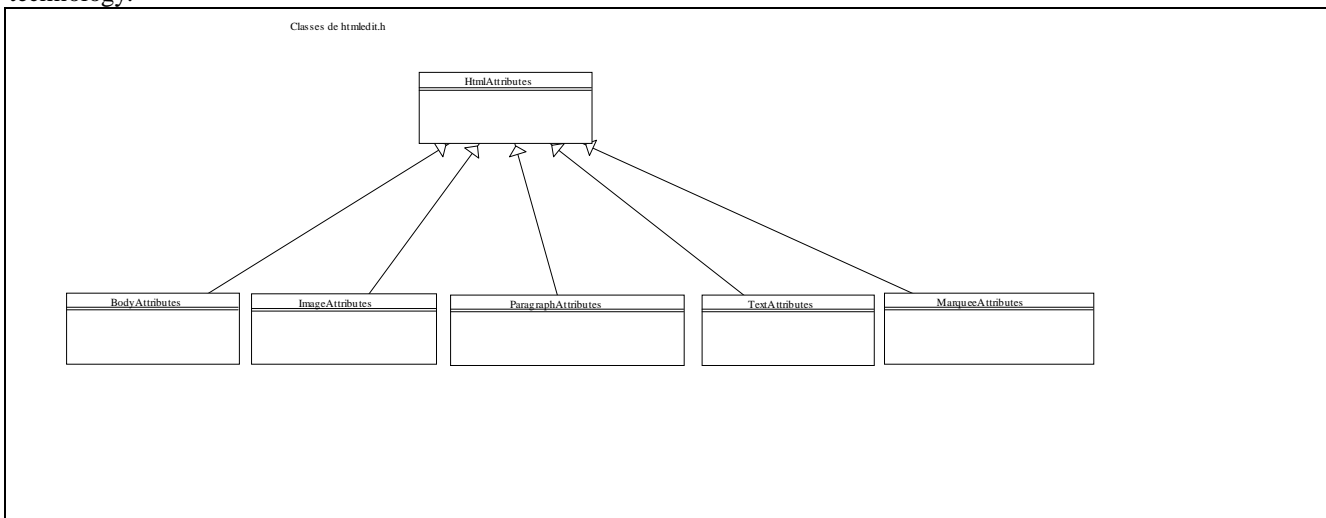


Figure 4 - Association diagram for the classes of Htmledit

We decide to recover three types of information from the code: the system structure, the relationships between objects and the internal structure of a class. For that we have used three well known software engineering diagrams: the module diagram [1], the association diagram [23] and the entity structure diagram [16].

To recover the structure of wxWeb we wrote two specifications in **EXL** and three specifications in **MdL**. Each specification in EXL focuses on a different type of information, the first specification focus the recovery of structure and the second focuses on the interrelationship between the different classes of the system.

The EXL specifications have 850 lines each and the ML specifications are in the order of 120 lines. The size of the EXL specifications is basically due to the complexity of the C++ grammar, but the size would be much larger if we were not reusing the transformation knowledge embodied in EXL. On an average an EXL command will generate 20 lines of Draco-PUC

transformation code. From the EXL specifications Draco-PUC generates the transformations, which in turn are finally transformed in C programs, that perform the recover process.

We assert that, by reusing those modeling languages, there is a considerable productive gain if we compare the effort of writing the specifications mentioned above with the effort of writing a system in a conventional programming language with the same functionality. Below we show the results obtained in recovering the structure of wxWeb as well as classes relationships. For each one of the cases below, once we have the database (see Figure 2), we use the three MdL programs to make it possible to display the results.

### Structure Recovery

The first EXL specification generates the module and class diagrams for the whole system, making it possible the navigation to the methods and among the methods, making it possible to return to the classes or modules. Figure 3 shows the Booch diagram for the whole system.

From the diagram shown in Figure 3 we can examine each of the classes defined for a given module. We did this for the "Htmledit.h" module and we produced an OMT association diagram for the module, Figure 4.

Given the class model as seen above it is also possible to navigate to the methods of a given class. Figure 5 shows the JSD diagram of one of the methods of the class ImageAttributes.

### Class Recovery

From the previous examples we can notice that working with models that take in consideration all the modules and all the classes give a global vision, but do not allow a more detailed view. For this reason we developed a second EXL specification in which we have

a more focused filter. The filter aims a chosen class. As such we can view the module in which the class is defined, the dependency graph of this module and the class association diagram. Figures 6 and 7 shows the result of applying the filter of the second EXL specification to the class “SimSock” .

information presentation. These domains were implemented in the Draco-PUC machine and applied to recover the structure of a public domain software. The use of both languages suggests that not only the domain approach works, but most of all has the flexibility necessary to the development of tailored reverse

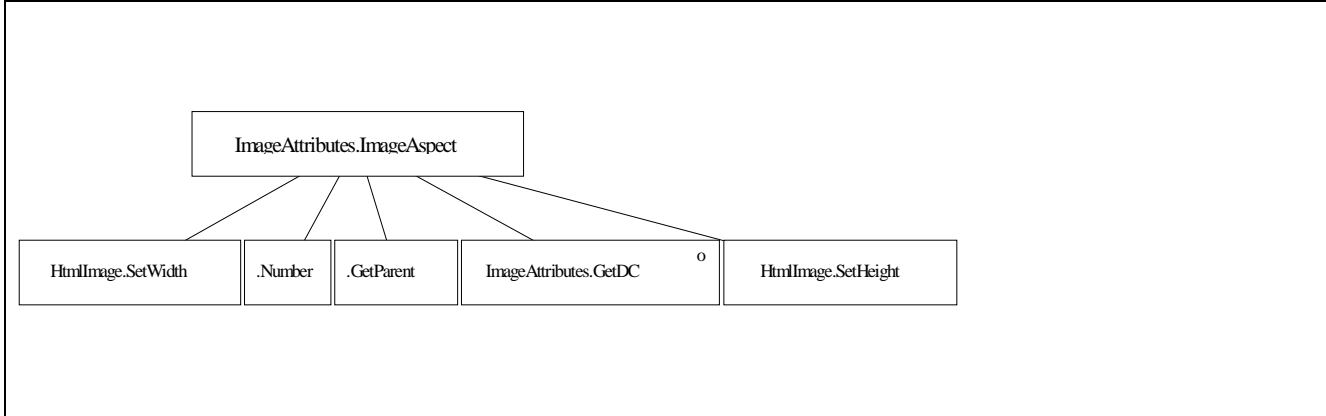


Figure 5 - Method for the Image Attributes class

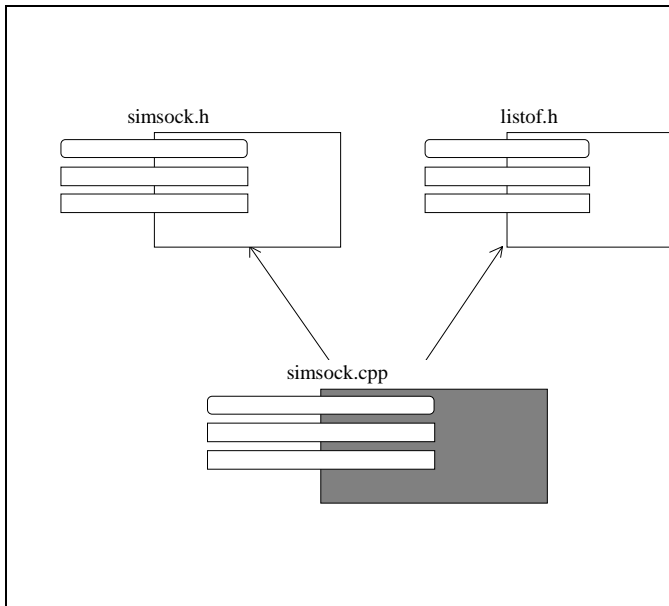


Figure 6 - Modules related to SimSock

## 6. Conclusion

We believe that our proposal of a domain network for reverse engineering tools brings important contribution to software maintenance research. We have shown that an architecture of a typical reverse engineering can be generalized by applying the domain concept [19].

In the study of the reverse engineering domain we understood that two different domains were necessary: one for dealing with recovery and the other for

engineering tools (see for instance the difference in the two EXL specifications we presented above). Since the strategy is independent of the source code language, it is flexible for a given language, in our case study C++, but also for other languages that have a Draco-PUC parser.

Since EXL and MdL were written to work on the Draco Abstract Syntax Tree it will work on different parsers. We are now working on a COBOL parser in order to apply our architecture to the Y2K problem. The initial results confirm the independence of EXL and MdL, but we still have to work on the parser to obtain the expected results.

We not only produced a prototype reverse engineering tool that contributes to the state of the art, but we assembled a domain network capable of generating other instances of the tool. We also understand that our work contributes to the research on reverse engineering since we showed that instead of using a library of transformations [3], we organized the information in domains. We also understand that, by using a relational database [17], we have more flexibility than the Jarzabek and Keam [10] proposal, maintaining, as they do, an independence with respect to the language being used.

The EXL and the MdL domains are the first non trivial domains, despite the executable domains (C, C++, COBOL), that we managed to build and use [14] [15]. We keep learning on our long term project of developing Draco-PUC and testing the ideas of domain development. We still have several aspects, which we did not deal with, that are on our agenda for future work, such as dealing with multiple refinement and having a more clean way of using the transformations. Usage of Draco-PUC transformations still requires a deep understanding of

the DAST and still requires non trivial escape mechanisms to a general purpose language.

Our main contribution is the flexibility given by a transformation based language. Tools like the suite provide by Reasoning [22] that extract information from source code are available, but the proposal presented brings a degree of flexibility not found in market tools. Our research, as compared to others, has the advantage of being domain oriented.

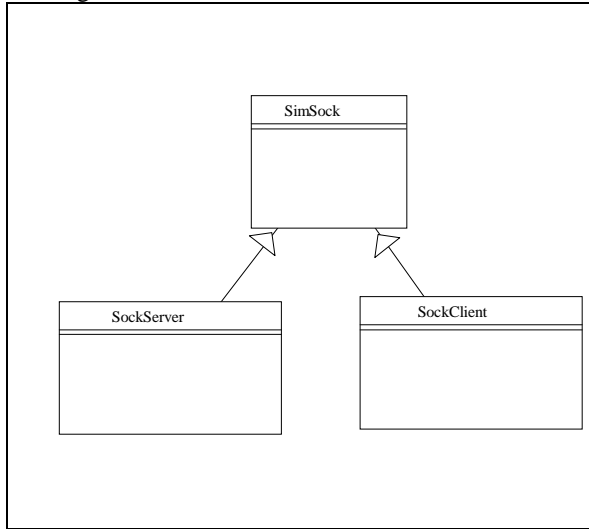


Figure 7 - Association diagram for SimSock1

Although EXL and MdL are not the only domains necessary to generate software maintenance tools, they stress the main problems faced in acquiring programming concepts from legacy code. Our work is based on research prototypes, and the results should be understood in this context. As such, we did not have package our results as to be easily used by others without assistance. There is a lot to be done, mainly in simplifying the interface of the languages, as well as providing means to help the users build parsers. Our parsing mechanism still needs that the user have a very good knowledge of compiler techniques to solve grammar conflicts as well as other errors that are not easy to locate. It is also true that our transformation mechanism still needs to use C constructs to achieve better results, and that is shown in the syntax of EXL and MdL. We also do not provide facilities for control flow analysis, nor for quality metric information as Reasoning does but believe that EXL will help constructing such tools.

**Acknowledgements:** We would like to thank WCRE reviewers for their comments, they helped us in improving the paper. CNPq has provided partial support for this work.

## 7. Bibliography

- [1] G. Booch. *Object Oriented Analysis and Design with Applications*, second edition. Benjamin/Cummings 1994
- [2] James H. Cross, T. Dean Hendrix. *Using Generalized Markup and SGML for Reverse Engineering Graphical Representation of Software*. In Proceedings of the WCRE'95, IEEE Computer Society Press, 2-6, 1995.
- [3] D.R.Harris, A.S.Yeh, H.B.Reubenstein. *Extracting Architectural Features from Source Code*. Automated Software Engineering, 3(1/2), Jul. 1996, 109-138.
- [4] Devanbu, Prem, T. GENOA - *A Customizable Language and Front-End independent Code Analyzer*, Proc. 14<sup>th</sup> International Conference on Software Engineering, IEEE Computer Society Press, pp. 307-319, 1992.
- [5] H.E. Edwards; M. Munro. *RECAST - Reverse Engineering from COBOL to SSADM*. In Proceedings of the WCRE'93, IEEE Computer Society Press, 44-53, 1993.
- [6] Peter Freeman. *A Conceptual Analysis of the Draco Approach to Constructing Software Systems*. IEEE Transactions on Software Engineering, SE-13(7):830-844, July 1987.
- [7] F.G.Freitas . *A Geração de Parsers da Máquina DRACO-PUC*. Trabalho Final de Curso - Engenharia de Computação; Departamento de Informática; Puc-Rio; 1994 (in Portuguese)
- [8] F.G.Freitas *VG Um visualizador gráfico genérico*. Projeto Final de Programação; Departamento de Informática; Puc-Rio; 1996 (in Portuguese)
- [9] F.G.Freitas and Leite, J.C.S.P. *Aplicando reuso de software na construção de ferramentas de engenharia reversa*. Anais do XI Simposio Brasileiro de Engenharia de Software., 265-280, 1997 (in Portuguese)
- [10] Stan Jarzabek, Tan Poh Keam. *Design of a Generic Reverse Engineering Assistant Tool*. In Proceedings of the WCRE'95, IEEE Computer Society Press, 61-70, 1995.
- [11] Leite, J.C.S.P., Cerqueira, P.M. *Recovering business rules from structured analysis specifications*, In Proceedings of the WCRE'95, IEEE Computer Society Press, 13-21, 1995.

- [12] Leite, J.C.S.P. and Prado, A.F. *Design Recovery - A Multi-Paradigm Approach*. First International Workshop on Software Reusability, Dortmund, 161-169, Jul. 1991.
- [13] Leite, J.C.S.P., Prado, A.F., Sant'Anna, M. *Draco-PUC Experiencias e Resultados de Engenharia de Software*, VI Simposio Brasileiro de Engenharia de Software, SBC, 115-128, 1992.(in Portuguese)
- [14] Leite, J.C.S.P., M.Sant'Anna, F.G.Freitas. *Draco-PUC: a Technology Assembly for Domain Oriented Software Development*. Proceedings of the Third International Conference on Software Reuse, IEEE Computer Society Press, 94-100, 1994.
- [15] Leite, J.C.S.P, Sant'Anna, M. and Prado, A.F. *Porting Cobol Programs Using a Transformational Approach*, Journal of Software Maintenance: Research and Practice, Vol. 9, John Wiley Sons Ltd., Vol. 9, pp. 3-31, 1997.
- [16] Michael Jackson, *System Development*, Prentice-Hall, 1983
- [17] <http://cui.unige.ch/~scg/FreeDB/FreeDB.6.html>
- [18] James M. Neighbors. *Draco: A Method for Engineering Reusable Software Systems*. Software Reusability, T. Biggerstaff, A. Perlis (editors), Addison-Wesley, 1989.
- [19] James M. Neighbors. *The Draco Approach to Constructing Software from Reusable Components*. IEEE Transactions on Software Engineering, SE-10(5):564-574, Sep. 1984.
- [20] S.Paul and A. Prakash, *A Framework for Source Code Search Using Program Patterns*, IEEE Transactions on Software Engineering, SE-20 (6): 463-475, Jun. 1994.
- [21] P.G.Selfridge, R.C.Walters, E.J.Chikofsky. *Challenges to the Field of Reverse Engineering*. In Proceedings of the WCRE'93, IEEE Computer Society Press, 144-150, 1993.
- [22] Reasoning, Inc. *Products and Services Overview*, [www.reasoning.com/product/overview.html](http://www.reasoning.com/product/overview.html)
- [23] James Rumbaugh, Michael Blaha, William Premerlani, Fredrick Eddy and William Lorensen. *Object Oriented Modeling and Design*. Prentice Hall ISBN 0-13-629841-9, 1991
- [24] Julian Smart. *wxWindows User Manual*. University of Edinburgh. Artificial Intelligence Applications. Institute. 80 South Bridge, Edinburgh. 1995
- [25] T.J.Biggerstaff, B.G.Mitbander, D.E.Webster. *Program Understanding and the Concept Assignment Problem*. Communications of The ACM, 37(5), 72-82, 1994.
- [26] Christopher A. Welty, *Augmenting Abstract Syntax Trees for Program Understanding*, In Proceedings of the Automated Software Engineering, IEEE Computer Society Press, 1997.
- [27] Linda M. Wills, James H. Cross II. *Recent Trends and Open Issues in Reverse Engineering*. Automated Software Engineering, 3(1/2), 165-172, June 1996.
- [28] <http://www.ozemail.com.au/~adavison/wxweb.html>
- [29] <http://www.aiai.ed.ac.uk/~jacs/wxwin.html>