# A Hybrid Architecture for Documentation Production

Christiano de Oliveira Braga
Arndt von Staa
Julio C. S. P. Leite

Departamento de Informática
PUC-Rio
R. Marques de S. Vicente 225
Rio de Janeiro - RJ
22453-900 Brazil
{cbraga,arndt,julio}@inf.puc-rio.br

PUC-RioInf.MCC15/97 Maio, 1997

**Abstract**

Good documentation is essential for the production and evolution of quality software. Based on a survey on program documentation, we propose a documentation architecture that aims to fulfill several requirements of an ideal solution. The architecture integrates a powerful transformation system (Draco-PUC) and a versatile CASE tool (Talisman) in order to produce documents using reverse engineering strategies. We have implemented such approach in a prototype tool called Documentu, which uses a database to store system wide information and uses a standard hypertext browser to help the access and navigation of the system documentation. The prototype was used in a large scientific system in an oil company, with very positive results.

**Keywords**

Documentation, transformation systems, CASE tools, software engineering, tool integration

**Resumo**

Este trabalho propõe uma arquitetura para uma ferramenta de documentação, que integra um poderoso sistema de transformação (Draco-PUC) com uma versátil ferramenta CASE (Talisman). Esta arquitetura foi implementada numa ferramenta chamada Documentu, que utilizando estratégias de engenharia reversa, armazena informações a nível sistêmico numa base de dados e utiliza um *browser* de documentos hipertextuais para visualizar estas informações. Este protótipo tem sido utilizado num centro de pesquisa de uma companhia petrolífera com resultados bastante positivos.

**Palavras-Chave**

Documentação, sistemas de transformação, ferramentas CASE, engenharia de software, integração de ferramentas

# 1. INTRODUCTION

One of the goals of a software process is to provide the necessary information to maintain evolving software systems. Usually, most of the information is solely embedded in the current source code. However, other sources of information should be used in order to gain a better understanding of the software and its environment [Biggerstaff94], [Selfridge93], [Leite95].

Good documentation, easily accessible and easy to read is being sought for a long time [Horowitz85] [Garg90]. It is essential to software production which employs several software engineers, which must work in a cooperative manner. We have surveyed some of the existing proposals at the code level, and based on their strong and weak points we proposed our architecture. Our proposal uses the HTML markup language and relates the information in terms of the whole system, and not only for single programs. The markup language is based on a proposal [Staa95b] that standardizes C\C++ comments with design level information.

In order to produce such documentation our approach ideally uses code already marked with tags as defined by Staa's proposal. However, we can also produce documentation for bare source code. This is possible because we generate the software documentation based on the source code grammar and on the markup grammar. We have used the transformation system of Draco-PUC [Leite94] to produce the language dependent information and to recover the system structure. Talisman [Staa93] CASE tool was used to deal with the markup information. Finally we use a standard browser to access the resulting documentation.

The transformation system plays the role of the extractor, analyzing and retrieving code information. This information is extracted using syntax analysis and, if the case, the code tags. This information is passed to the CASE tool which stores it in a repository. This repository will have database management capabilities in order to provide a series of consistency checks that can be applied for the entire system. Using the information in the repository, the CASE generates a hypertext in HTML by means of special procedures called linearizers.

Before detailing our proposal we make a brief survey of existing documentation proposals (Section A Brief Survey on Documentation Preparation). In The Proposed Architecture, we describe the tool architecture. In Documentu section, the Documentu tool is presented. Using *Documentu* in a Real System reports on the results of applying Documentu to a working scientific software. The last section is the conclusion of the paper, referring to related work and reporting on our plans for the evolution of Documentu and its use.

# 2. A BRIEF SURVEY ON DOCUMENTATION PREPARATION

There is no doubt of the importance of presenting software artifacts as structured and readable documents. Nowadays this research issue presents several new possibilities by the widespread diffusion of the World Wide Web and its lingua franca, HTML.

Our work was based on this perception and on key literature, which we classified under: standards, literate programming, using markup languages and databases and recent documentation tools.

## 2.1 Document standards

SGML [Goldfarb90] is the ISO-8879 ISO standard for structured documents. The standard describes a language for document description. This language defines tags to be embedded on the text, in order to structure the document. The tags have the form <cmd> to begin a structure and <\cmd> to end it. A simple example would be :

```
<Paragraph>
This an example of a paragraph tag in SGML
<\Paragraph>
```

A style-sheet can be applied over the document to produce a pretty-printed view of the document, following a document DTD (Document Type Definition).

A DTD defines a structure for a class of documents. The following gives an example of DTDs.

```
<! -- ELEMENT CONTENTS -- >
<!ELEMENT DOCUMENT( MANINFO, USEINFO, IMPLEMENTINFO)>
<! -- ELEMENT MANINFO -- >
<!ELEMENT MANINFO ( Project, FileType, IdNumber?,
                    Version, DateAp, Author+      )>
<!ELEMENT Project ( #PCDATA )>
<!ELEMENT FileType( #PCDATA )>
<!ELEMENT IdNumber( #PCDATA )>
<!ELEMENT Version ( #PCDATA )>
<!ELEMENT DataAp  ( #PCDATA )>
<!ELEMENT Author  ( #PCDATA )>
...
```

**Figure 1 - SGML DTD example**

A document created following the DTD defined in figure 1 would be composed of MANINFO (managerial information), USEINFO (use information) and IMPLEMENTINFO (implementation information). In the example of figure 1 we have detailed the managerial information that could be composed of Project, FileType, IdNumber, Version, DataAp and Author. Among those, IdNumber is optional and Author may occur one or more times. All documents created using this structure would have those sections.

HTML [HTML] is said to be an application of the SGML standard. The language is used to create portable hypertext documents. The HTML markups could represent hypertext news, electronic mail, documentation, hypermedia, options menu, databases query results and structured documents with graphics. HTML has been used in the World Wide Web since 1990.

The most important difference between HTML and SGML is that the former has the capacity to refer to other documents through links defined with markups.

Another relevant difference between SGML and HTML is that the programmer can not develop different DTDs in HTML. All HTML documents have to follow a fixed structure, defined by the HTML standard.

The Office Document Architecture (ODA) [Appelt93] [Brown89] is an object oriented framework, created to be a standard to document exchange. ODA, like SGML, distinguishes between the logical structure and the layout structure (see Figure 2).

Nevertheless, SGML and ODA have different approaches to solve the same problem - documentation exchange. But both of them emphasize a very important issue in documentation; the essential difference between document structure and the layout structure. Also, both of them fail in a crucial problem related to software systems documentation; they do not support the creation of hypertext documents.

## 2.2 Literate Programming

Knuth [Knuth91] proposed the join of a document format language and a programming language. In WEB, he uses the well known languages $T_EX$ and Pascal. To develop a system using Literate Programming, the programmer must, at first, write an WEB file, which is the document that describes the system being developed. This file is composed by the program description with chunks of code. The next step is to apply a processor called WEAVING to produce a $T_EX$ file.

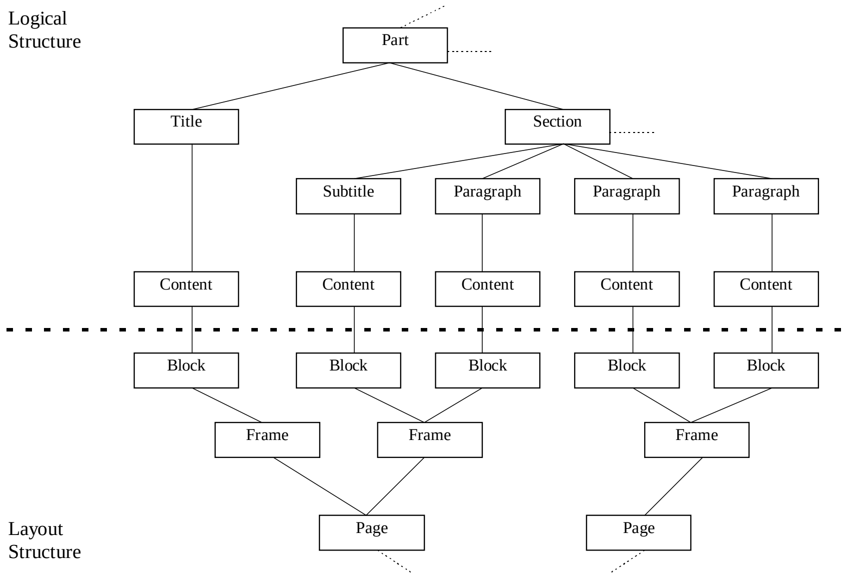



**Figure 2 - ODA document example regarding the difference between the logical and layout structure**

The use of the TANGLE processor sets up the code files to be compiled. In this way documents tend to be consistent, since they come from the same source. Figure 3 shows the WEB development process. Knuth's proposal is indeed very interesting since the main idea is to develop readable code to other human beings and not just code to be compiled. But the main problem is that Literate Programming was conceived to describe algorithms and not to produce system wide documentation. The consequence is that the final documentation keeps related to just a single module. The cross-references are bounded to the document edge, and they have to be explicitly created by the developer.

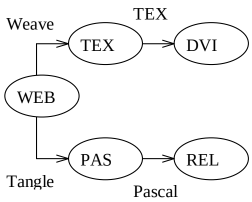



**Figure 3 - Generating WEB documents**

Another problem is that $\mathrm{T_EX}$ does not provide support to document structure definition, like SGML DTDs, in such a way that format markups are embedded in the document together with the text.

## 2.3 Markup languages and databases

[Cowan94] presents a solution that we consider richer than the one presented on WEB. In Cowan's approach, SGML markup language is used to embed specification information in the code. A database is then filled with code information and through the use of database resources, this information can be viewed in a hypertext format, through indexes and links.

In [Cowan94], all the documents resulting from the development process could be in SGML format linked to each other inside a textual database. All the information could be accessed by views in a WYSIWYG (what you see is what you get) environment, allowing database update through those views.

[Cowan94] presents some important issues concerning the preparation of software systems documentation :

- the need for a document logical structure, defined by a standard in order to achieve document portability ;
- the need for databases, to store the system wide code information, allowing validations over the gathered data;
- the use of a hypertext browser to navigate across the information.

## 2.4 Documentation tools

There are several tools available to support the software development process documentation, but most of them only produces documentation with no consistency checking between the code and the documentation.

Java [Java] is a programming language developed at Sun Microsystems to support the development of portable and reusable software components through the Internet. JavaDoc is a tool that generates HTML documents from Java source code annotated with tags.

JavaDoc formats public and protected information from classes and interfaces. shows an example of JavaDoc tags. HTML tags can also be used to format comments but cross-reference tags - @see (not shown in the example) - have to be manually inserted. Figure 4 shows the browsing of this example.

Microsoft also has a tool called AutoDuck. This tool works much like JavaDoc. It uses markup tags in the source code to generate documents. These documents are generated in RTF (Rich Text Format - a Microsoft standard available for Windows, OS/2 and MacOS platforms). RTF is the format used by the Microsoft Help Compiler to generate Windows Help files.

```
/**
 * Return the character at the specified index. An index
 * ranges from <tt>0</tt> to <tt>length() - 1</tt>.
 *
 * @param index The index of the desired character
 * @return The desired  character
 * @exception  StringIndexOutOfRange Exception When
 * the index is not in the range <tt>0</tt>> to
 * <tt>length()  - 1</tt>.
 */
public char charAt(int index) {
    ...
}
```

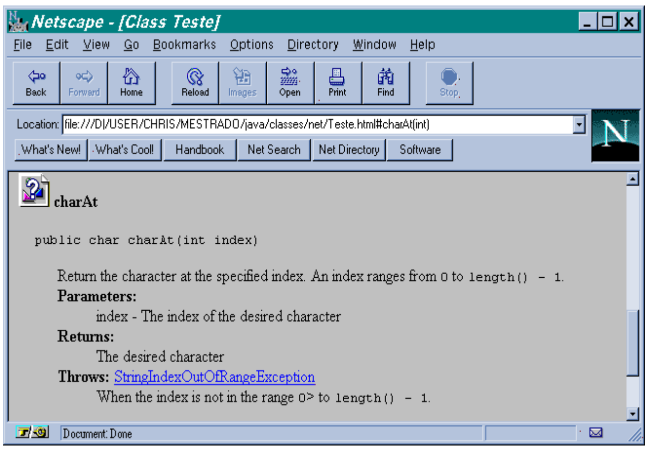



**Figure 4 - JavaDoc documentation browsing**

In Figure 5, the tag @class defines the name of a class and @cmember defines a class member. Since the class in the example is a template, @targs defines the class templates arguments. figure 5 shows the browsing, using the Windows help system, for the example presented in Figure 5.

AutoDuck has an interesting capability of parsing user defined tags. JavaDoc parses only its own defined tags. Both of them are easily integrated to the build process through its invocation from the makefile, improving documentation consistency with the source code.

Neither JavaDoc nor AutoDuck supports system wide documentation. They are only concerned about local modules. Cross-references have to be created manually by the developer, generating possible fonts of inconsistency. Also, no consistency checking is performed.

### 2.5 Some important issues about documentation preparation

The main concept to keep in mind, is that software programs have to be written for human beings, and not only to be processed by computers, as Knuth states. The existence of a document logical structure, defined using a standard in order to achieve **document portability** is a major aspect. It is also important to store the **system wide** code information, allowing validations among them, so the need for a database. Another important issue is the use of a **hypertext** browser to navigate through the information. The use of a hypertext provides an easy way to access on-line information, as well as allowing information layering, such that information can be structured.

## 3. THE PROPOSED ARCHITECTURE

This proposal is geared to marked code, but it also treats unmarked code, making possible the generation of the whole documentation hypertext based solely on a system source code. It is important to emphasize that the tags only do complement the information captured by the tool.. **Documentu** is a tool that instantiates such proposal. The marks treated by our proposal are based on coding standards [Staa95b] for the creation of code comments. This coding standards will be described in Documentu section. If there is no documentation available in the code, the information shown in the generated documentation is the one that can be retrieved from the source code language syntax through parsing and transformation.

```
/* @doc
 * @module TEMPLATE.CPP - Some Template Examples
 * This module shows how to code Autoduck tags for C++
 * template classes and member functions.
 * Autoduck example by Eric Artzt (erica@microsoft.com)
*/
//@class Template class
//@tcarg class | T | A class to store in stack
//@tcarg int | i | Initial s of stack
template<class T, int i> class MyStack
{
    //@cmember The top
    T*  pStack;
    //@cmember The stack
    T StackBuffer[i];
    //@cmember The count
    int cItems = i * sizeof(T);
  public:
    //@cmember Constructor
    MyStack( void );
    //@cmember Add one
    void push( const T item );
    //@cmember Remove one
    T& pop( void );
};
```

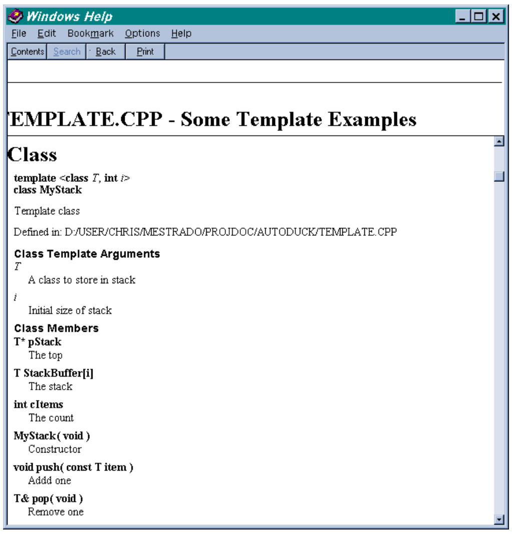



**Figure 5 - AutoDuck tag browsing**

The information extraction system (see Figure 6) parses the source code and produces information that is going to be used by the information formatting system. The formatting system then produces a hypertext system in HTML. The first part of the process, the extraction system, uses Draco-PUC [Leite94] transformation capabilities. The second part of the architecture, the formatting system, is based on a repository based CASE tool called Talisman [Staa93], presented in Talisman CASE tool. Both Draco-PUC and Talisman are the result of several years of research at PUC-Rio in the areas of domain-oriented reuse and software engineering environments. Draco-PUC is a prototype running on Solaris, SunOS, Linux, Windows NT and Windows 95. Talisman is already a commercial product, running on DOS environments, including Windows DOS shells.

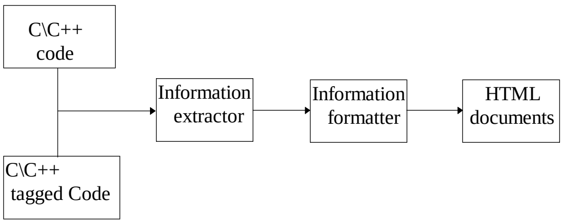



**Figure 6 - Outline of the proposed architecture**

### *3.1.1 Draco-PUC*

Draco-PUC [Leite94] is a transformation system that implements the Draco paradigm [Neighbors84]. This paradigm proposes software development by creating and reusing domains, in a high level of abstraction. An essential part of a domain description in Draco-PUC is a grammar definition written in a BNF-like style. Using this grammar as input, the **pargen** subsystem (**par**ser **gen**erator), produces the domain parser. The unparser or pretty-printer is also created when the grammar is annotated with lay-out directives.

A special abstract syntax tree (DAST - Draco Abstract Syntax Tree) is created to store the information retrieved by the parser. Once this structure is created, it is possible to apply the unparser to view the information in the DAST as described in the lay-out directives. It is also possible to apply transformations to map the information in the DAST to a new structure over the same domain (intra-domain transformations) or to a new domain using inter-domain transformations.

A Draco-PUC transformation has a left-hand-side (LHS), which defines a searching pattern, and a right-hand-side (RHS), which defines a replacement pattern. Control points and transformation properties are also available. These kind of features are responsible for driving the transformation engine in a non-default bottom-up search-and-replace way. The control points of a transformation are:

- **Pre-Match** - executed when the engine tries to execute the transformation.
- **Match Constraint** - executed every time a pattern variable is bind against a program fragment when the transformation matching is being attempted.
- **Match-Failure** - executed when the transformation matching fails.
- **Post-Match** - executed right after the successful matching of the transformation.
- **Pre-Apply** - executed before the replacement of the LHS by the RHS.
- **Post-Apply** - executed after a pattern is substituted.

Transformation rules are grouped in transformation sets, which do also have control points (Initialization and End) and properties. Sets of transformations are encapsulated on Transformers. Transformers have three control points: Declaration (global objects declaration), Initialization and End. figure 7 depicts the structural framework of transformations in Draco-PUC.

Draco-PUC transformations can be in-place transformations, when the substitution pattern is replaced in the source, or they can work with workspaces. Transformation that uses workspaces are non-destructive, and the replacement pattern is not instantiated in analyzed source, but rather in an external workspace. When using workspaces, it is also necessary to define Templates, which are Draco-PUC transformation language structures that allows the developer to instantiate the RHS in target workspaces.

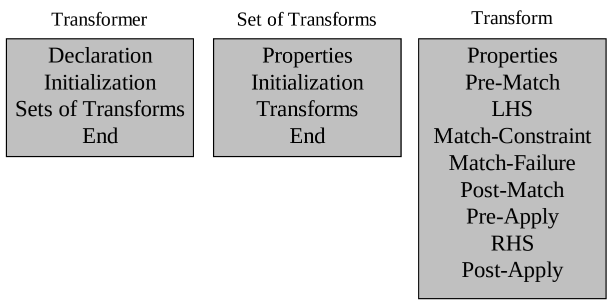



**Figure 7 - Draco-PUC structural transformation framework**

The use of workspace can be viewed as an asynchronous replacement, where the RHS is written in an temporary space for further use. As an example of such use consider the creation of a relation between one element that was already found by the analyzer and another that is to be found. A workspace should be used to create this late association. An example will be presented in section 4.1.

### *3.1.2 Talisman CASE tool*

Talisman [Staa93] is a repository based CASE tool. Under its repository lays the definition of the *Talisman definition languages*. Those definition languages give semantics to the existing information inside a software base. Talisman supports languages for structured analysis, like Data Flow Diagram, for data modeling, like Entity Relationship Diagrams and for information systems modeling. Talisman also allows the definition of such languages, thus making customization an easy task.

A software base is a database composed of textual fragments and references to textual fragments inside the database. A definition language is characterized by the relations among its elements (modules, classes and functions, for example) and also forms (validators and linearizers). Those forms could be loaded at run-time to customize the environment. The validators check for consistency of the data inside the software base, and the linearizers produce reports with the information contained inside the software base. When a software system is developed, editor forms are used to populate the software base with specification text fragments. Each text fragment can have different roles depending on which definition language is being used. For instance, a data repository in a DFD can be represented as a table in a Entity-Relationship diagram.

Once the software base is populated, the validators have to be applied to perform consistency checking with the information inside the base. With the information validated, the linearizers are applied to produce documentation or code.

### *3.1.3 Integration of the tools*

The information extractor system (see Figure 8) is set up on Draco-PUC by means of two domains: the C\C++ domain, and the Talisman domain.

The C\C++ domain was already available as a Draco-PUC executable domain. This domain has been used by several people in different projects, implying some robustness to that domain, although it was necessary to add some little modifications to make possible the analysis of the special tags.

Talisman has an importing feature that allows the CASE environment to populate an existing base with information provided by import files. The rules that guide the creation of this file can be written as a grammar. Using this grammar, the Talisman domain was created.

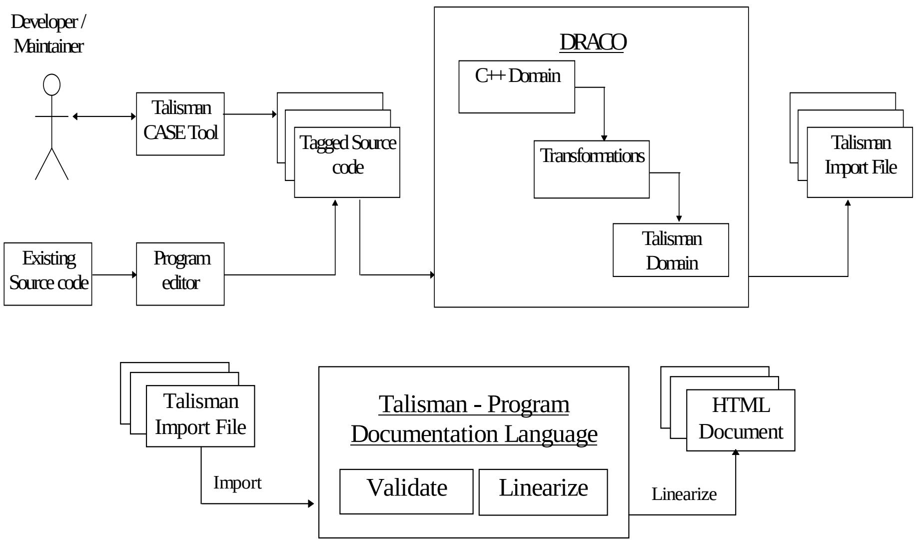



**Figure 8 - Execution process**

The process starts with the analysis of the source code. Once the code of a given module is analyzed and the Draco-PUC internal form (DAST) is created, a set of transformations, created specially for our approach are applied over that internal form in order to create the Talisman import file for that module. It is repeated for each module described in a Draco-PUC script file (**.dsf** file). A Draco script file is written for the whole system, capturing all the modules it comprises. Figure 8 shows the first phase of the proposed process, where the tagged (or not tagged) code is analyzed by the transformation system, generating the import files to the Talisman CASE tool.

The second part of the process is concerned to Talisman. Each import file created by Draco-PUC is then imported to the Talisman environment.Once all the files are imported, the **validators** are executed checking the data consistency inside the base. They look for text fragment completion and cross-referencing validation. A log is created with the inconsistencies that have been found. When the base is validated, the linearizers are executed to create the HTML documents.

## 4. DOCUMENTU

We describe **Documentu** by focusing on details of its architecture, its execution and the results produced. Documentu is based on two Draco-PUC domains, C++ and the Talisman domain, and on the Documentu repository defined in Talisman. Below we give details on how this parts are organized.

### 4.1 The Role of Draco-PUC in the Architecture

As cited before, the basis for a Draco-PUC domain description is its language grammar description. Domain languages grammar descriptions are written on an extended-BNF style. An example of such grammar descriptions is shown in Figure 9, which presents the C\C++ grammar fragment responsible for the description of class heads like:

- class Cwindow
- class CMyWindow : CWindow
- class CAnotherWindow : CWindow, CBaseWindow

```
class_head     : class_key  class_name? base_spec?
               ;
class_key      : ‘class’
               | ‘struct’
               | ‘enum’
               ;
base_spec      : ':' base_specifier+’,’
               ;
base_specifier : class_name
               | ‘virtual’ access_specifier? class_name
               | access_specifier ‘virtual’? class_name
               ;
class_name     : CLASS_NAME
               ;
access_specifier     : ‘private’
                     | ‘protected’
                     | ‘public’
                     ;
```

**Figure 9 - Fragment of the C++ grammar**

Just like with C++, the Talisman import file grammar description was also encapsulated in Draco-PUC. An excerpt of this grammar, which describes textual import entry for the repository is shown in Figure 10. An example of a textual import entry is the one Documentu uses when defining descriptive long names for classes in Talisman repository.

```
import_file    : import_entry*
               ;
import_entry   : command
               | comment
               | exc_obj
               | exc_attrib
               ;

command  : ‘DA’ category name attribute
         ;
category       : STRING
               ;
name           : STRING
               ;
attribute      : ‘Name’
               | text_field
               | alias
               | relation
               ;
```

```
text_field     : 'Text' ( ID | NUMBER ) lines_of_text
               ;
```

**Figure 10 - Fragment of the Talisman grammar**

This entry stores in the repository a textual attribute labeled `30` for object `VM_Segment` (of category `Classes`), having value `"VMS - Segment"`. In particular this means that the C++ class named `VM_Segment` will be associated with the long name `"VMS - Segment"` and so all references to it, when document reports are produced, should use this format.

After the grammars were defined, transformations were created to retrieve the information from the C/C++ code into the Talisman domain, in such a way that it can be used in the CASE tool. The retrieved information comes from the language (C/C++) analysis, and also from the embedded tags following the specification information, defined in [Staa95b] as part of a set of C/C++ coding standards.

The embedded tags appear as comments, so they do not change the compilation process. Figure 11 presents an example of such tags embedded on C++ code.

```
////////////////////////////
//$CNAME Class long name
//VMS - Segment
//$CD Class description
//Segments are storage units. Each Segment is related to
//a file. Segments are composed of pages of fixed size.
//Segments can be created, deleted, opened or closed
//just like files.
//Pages can be allocated, freed or read from a segment.
//When a page is read, a virtual memory system
//associates an object of the Port class to this
//page.
//The virtual memory has a fixed number of ports.
//When a page is read and there is not enougth room
//for it a LRU mechanism is used to decide which port
//should be cleaned in order to receive a new page.
//$CMU How to use this class
//(To be defined)
//<-
///////////////////////////////
class VM_Segment
{
     ...
};
```

**Figure 11 - Example of embedded tags**

In **Documentu**, there are two types of transformations: tag matching and structure matching (concerning C++ language structures). An example of a tag matching transformation is shown in Figure 12.

The transformation presented in figure 12 looks for a class name tag specification (`$CNAME` ) and then stores the proper information in a workspace. Following Staa's commenting standard, each class should have a long name in order to provide a better understanding of the abstraction. This long name is referred through the tag label `CNAME`.

Patterns and control points in Draco-PUC can be defined either using the DAST notation or using a domain language syntax. In the later case the description is surrounded by double braces and the used mapping between the DAST notation and the domain syntax is annotated right after the first pair of left braces. For example, in the `GetClassLongName` LHS, instead of writing down the pattern directly in DAST notation we found more properly to use the C++ syntax for tags. Tags, in our C++ domain, do belong to the `cpp` declaration syntax type and that is the reason why the mapping dast->cpp.declaration is annotated right after the first pair of braces.

```
Transform GetClassLongName
      Lhs: {{dast cpp.tag
      //$CNAME [[IDENTIFIER* tag_specifiers]]
      [[TAG_LINE long_name]]
      }}

      Post-Match: {{dast cpp.statement_list
      TEMPLATE("StoreCurrentClassLongName");
         TRANSPORT_VALUE(“long_name”);
         PLACE_AT(“WSClassLongName”);
      END_TEMPLATE;
      }}

      Template StoreCurrentClassLongName
      Rhs: {{dast talisman.lines_of_text
      [[LINE long_name]]
      }}
```

**Figure 12 - A tag matching transformation**

The label CNAME is followed by some tag specifiers that are not relevant for the presented transformation, followed by a line encompassing a class long name. In the Draco-PUC transformation language, pattern variables are defined by surrounding double brackets and must always belong to a domain syntax type. The star is a type constructor meaning sequence. So, the pattern variable tag_specifiers, when the matching occurs, will be bound to a sequence of cpp IDENTIFIERs (C++ domain syntax type defining identifiers). When this transformation matches against the example of Figure 11, the variable tag_specifiers is bound to the sequence of IDENTIFIERs with value <”Class”, “long”, “name”> and the variable long_name is bound to a TAG_LINE with the value “VMS - Segment”. This last value is also bound to the variable long_name on the template StoreCurrentClassLongName, through the TRANSPORT_VALUE command, and then the template is instantiated at the WSClassLongName workspace. This value will then be further referenced and used during the matching and application of other transformations.

```
Transform Class
      Lhs: {{dast cpp.class_head
       class [[CLASS_NAME short_name]]
      }}

      Post-Match: {{dast cpp.statement_list
      ...
      TEMPLATE("ClassLongNameDeclaration");
         TRANSPORT_VALUE(“short_name”);
         MOVE(“WSClassLongName”,
               "class_long_name") ;
         PLACE_AT(“WSTalismanImportFile”);
      END_TEMPLATE;
      ...
      }}

      Template ClassLongnameDeclaration
      Rhs: {{dast talisman.import_entry
       DA “Classes” [[ID short_name]] Text 30
       [LINE class_long_name]]
      }}
```

**Figure 13 - A structure matching transformation**

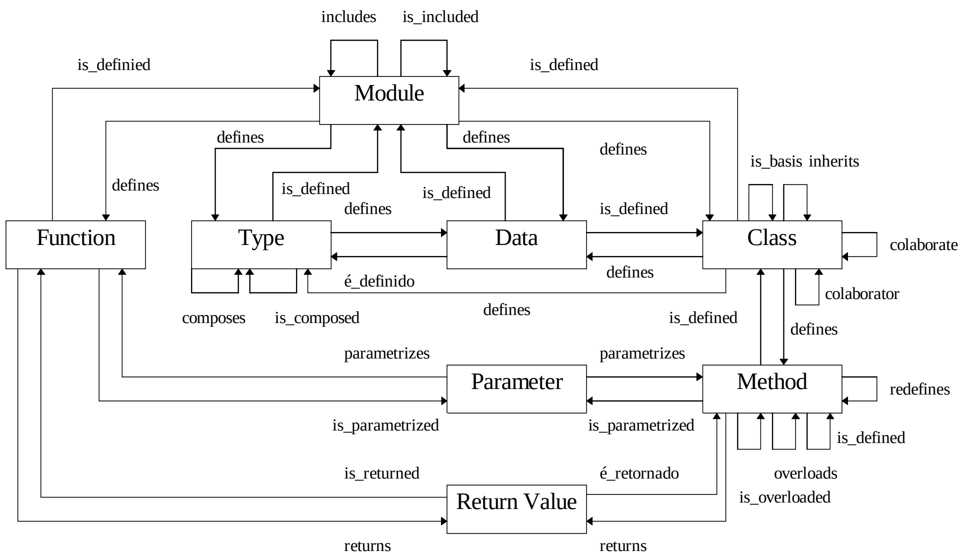
The placement of an instantiated template pattern in a workspace is issued through the PLACE_AT command. The commands TEMPLATE and END_TEMPLATE define the scope for commands such as TRANSPORT_VALUE and PLACE_AT. Figure 13 provides an example of structure matching. In this case, the information in `WSClassLongName` is associated with the recently found C++ class name (referred to as `short_name`) and all this information is then stored in another workspace called `WSTalismanImportFile`. This last workspace accumulates all import entries for the repository. This workspace is saved into a disk file at the end of the application of all transformations and will be used by Talisman as the import file for the processed module. The `MOVE` command is responsible for binding a pattern variable to the contents of a workspace and then clearing the workspace.

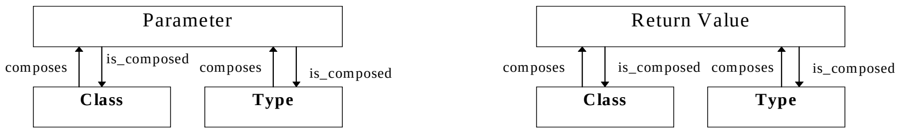
## 4.2 The Role of Talisman in the Architecture

To the **Documentu** project, a new definition language has been defined in Talisman. It is called Program Documentation.This new language is based on Modular Programming, but the information it manages is related to source code documentation. The relationships defined for this new language accomplish the model in

Figure 14. When a file is imported, the information inside the file populates the software base. Once the base is filled, the validators should be applied and then the linearizers. In **Documentu** a linearizer is responsible for the creation of the hypertext system in HTML.

The linearizer, just like the validator, is a form program. A form program is written using the language available in Talisman to create programs that manipulates the software base text fragments. The validators check the information inside the software base in order to verify its consistency. The linearizers navigate inside the software base looking for specific text fragments and relations, then it generates a file with the text fragment information, creating a file with all the system information in HTML format, since Talisman is capable of generating only one file at a time. After that, a postprocessor is applied, splitting the large file in many different HTML files.





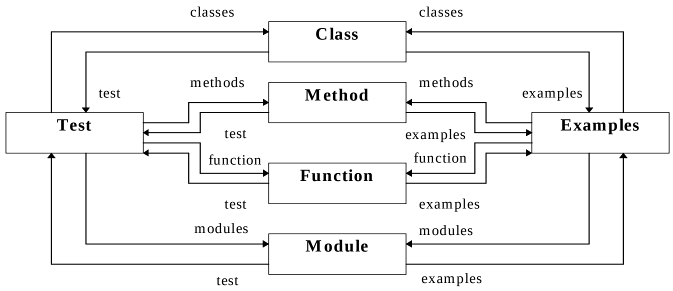




**Figure 14 - Model for Program Documentation Talisman definition language (part II)**

**Figure 15**

## 4.3 System execution

The current implementation of **Documentu** still requires some interaction with the tool user. Draco-PUC script file and the Talisman import process have to be manually written. **Documentu**, running as a batch file, will be invoked inside the makefile using as an input, a file generated by GMAKE with the modified files. Using this file, a program called initializer is called by Documentu batch file to produce two other files: system.talis.macros and system.dsf. The former will be used by Talisman. The later is the input to Draco-PUC machine. The **Documentu** batch file would then invoke Draco-PUC with system.dsf file as parameter. The last step is to execute Talisman (modified to execute in command line mode) with system.talis.macros file as an argument, generating the HTML documents. Finally, the user can browse the HTML files using a tool like Netscape.

## 4.4 The Hypertext

All the links are created using the extractor facilities of Draco-PUC (parser and transformations). The detailed information about each element (class, methods, for example) comes from the tags text embedded in the code. The current implementation of **Documentu** generates a hypertext interface organized in five frames as shown in Figure 14.

The four frames in the left side are indexes to system elements. For instance, the first one is the index to the system modules. Each bullet inside that frame represent a link to a module in the system. When a system module is selected in the first frame, the fifth big pane in the right changes, browsing that module information, like its associated classes, its functions, types and data. The same thing can be done with the other three panes on the left, which represent the indexes to classes, functions and types of the system.

Using the links that appear inside the big frame, the user can navigate to the detailed information of a given system element. For example, consider the GL_Lista class (see Figure 14), when it is selected, in the second pane on the left side, the big pane changes so the user can browse through information such as implemented methods, long name, class description, constraints, hypothesis, bibliography, authors and the date when the class became available [Staa95b].

# 5. USING *DOCUMENTU* IN A REAL SYSTEM

Petrox is a large system developed at CENPES, the research center of Petrobrás (Brazilian Petroleum

Company). One of its parts is the Magoo GUI (Graphical User Interface Library), that has 42 implementation (.CPP) files using 282,162 bytes and 47 definition (.H) files in 294,933 bytes, which amounts to almost 40.000 lines of code.

We have used Magoo as our test case, since its code uses advanced features of the C++ language. Documentu took 14 minutes on a 486DX50 running Windows 95 to produce the documentation. 4645 transformations were applied. This documentation is spread along 520 files using 693KBytes in the hard disk.

Some other statistics that might be important about the tool itself. Documentu has 29 transformations, distributed as follows:

- 18 concerning C++ classes;
- 7 concerning modules;
- 3 concerning functions and methods;
- 1 concerning C structures (structs).
- The Talisman linearizers have 32 functions and 2012 lines of code.

Table 1 summarizes the results of running Documentu with Magoo as input.

## 6. CONCLUSIONS

We have presented an architecture to produce high quality system wide information based on information available at source code. Such documentation will be more effective if code standards are used properly. Our approach uses two different software artifacts to parse and organize the documentation information, one is a transformation system, which deals with code grammar and the other is a CASE tool, that deals with the markup grammar.

It is important to stress that using the CASE tool Talisman [Staa93] it was possible to write checkers (validators) to analyze the produced documentation. After the analysis of the documentation, Talisman linearizers produce a hypertext document in HTML. The produced information will be extremely valuable to software developers, mainly during maintenance.

The use of transformation systems to help solving this kind of problems is being used by several others [Wells95] [Zoufaly95] [Newcomb95]. Our work innovates by using a transformation system and a repository based system. For instance we believe that our proposal, regarding the repository aspect, is related to [Edwards93] and [Jarzabek95]. Jarzabek provides a powerful parser to retrieve information creating a PKB (Program Knowledge Base) and using a wizard generates a domain knowledge base. Although we did not explore the domain oriented knowledge, having the information on Talisman repository is a first step to link with other Talisman languages, thus making possible to handle more high level representations.

An approach similar to ours was presented by Cross [Cross95]. In that case a source code grammar was not used, being the generation of the documentation based only on the markup grammar. A similar architecture is being used by Johnson and Erdem [Johnson95], where they are focusing their attention on the interactive and inquiry aspect of program understanding.

Future work will not only explore its use, but as mentioned above explore the full possible links with other representations dealing with high levels of abstractions. In particular we are planning to explore the integration with the strategy described in [Leite95], which also could be based on the Talisman environment. A wider effect of this collaboration (Draco-PUC and Talisman) is the integration of the Draco-PUC transformation system with the Totem environment, the evolution of the Talisman environment [Staa95a].

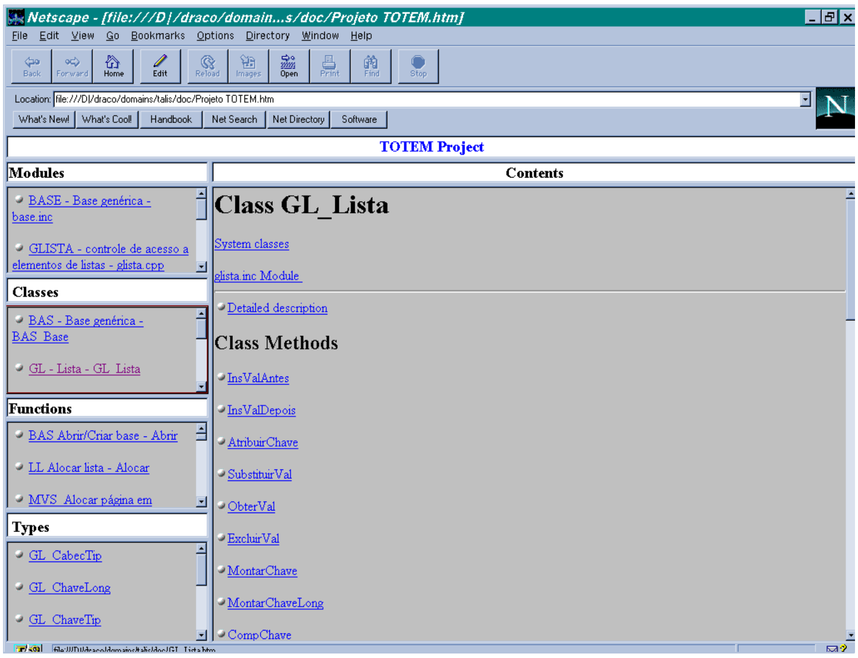



**Figure 16 - The hypertext class browsing**

| Modules | #Transformations | Size (Bytes) H Files | Size (Bytes) CPP Files |
|---|---|---|---|
| ABSTbox | 78 | 2,463 | 1,178 |
| abstslct | 100 | 4,313 | 8,123 |
| actor | 90 | 25,795 | 6,770 |
| applic | 189 | 15,302 | 34,265 |
| bdimflt | 162 | 7,299 | 5,423 |
| butradio | 85 | 2,082 | 3,849 |
| button | 105 | 5,505 | 10,498 |
| canvas | 96 | 6,361 | 5,729 |
| composer | 59 | 17,413 | 278 |
| control | 147 | 24,045 | 23,406 |
| datum | 49 | 4,707 | 1,784 |
| dialog | 136 | 12,916 | 10,702 |
| dimfloat | 154 | 7,819 | 2,731 |
| dimtext | 172 | 2,101 | 5,576 |
| erro | 114 | 2,297 | 3,903 |
| fill | 70 | 1,273 | 308 |
| frame | 85 | 1,204 | 947 |
| hbox | 79 | 1,681 | 1,593 |
| heapview | 92 | 972 | 1,468 |
| label | 86 | 953 | 830 |
| menu | 94 | 3,852 | 2,029 |
| menuitem | 92 | 1,489 | 2,698 |
| multline | 122 | 3,048 | 1,621 |
| multslct | 110 | 2,161 | 3,955 |
| numtypes | 117 | 13,814 | 912 |
| objstrm | 168 | 42,151 | 16,959 |
| optbut | 106 | 4,336 | 5,251 |
| radio | 77 | 2,307 | 1,615 |
| selector | 106 | 3,263 | 5,682 |
| shell | 98 | 4,069 | 3,826 |
| slctmenu | 91 | 2,859 | 2,623 |
| statusln | 71 | 1,073 | 810 |
| strng | 99 | 4,315 | 9,304 |
| submenu | 76 | 4,056 | 310 |
| table | 212 | 1,262 | 63,330 |
| text | 118 | 18,903 | 4,553 |
| toggle | 114 | 2,445 | 6,331 |
| toolbox | 94 | 5,677 | 3,792 |
| unit | 194 | 1,329 | 6,833 |
| vbox | 77 | 9,561 | 1,575 |
| ve | 180 | 1,619 | 2,273 |
| zbox | 81 | 2,805 | 3,804 |
| **Total Modules** | 42 | | |
| **Total Transformations** | 4645 | | |
| **Total Size H Files** | 282,895 Bytes | | |
| **Total Size CPP Files** | 279,447 Bytes | | |

**Table 1 - Statistics generated running Documentu over Magoo**

## 7. ACKNOWLEDGMENTS

We would like to thank Marcelo Sant'Anna for everything he has done for the success of this project, since its very beggining.

## 8. REFERENCES

[Biggerstaff94] Biggerstaff, T. ; Mitbarden, B.G; Webster, D.E.; "Program Understanding and The Concept Assignment Problem"; *Communications of ACM*; Vol. 37; n. 5; May 1994; pp. 72-83

[Cowan94] Cowan, D.D; Germán D.M.; Lucena, C.J.P; Staa, A.v; "Enhancing Code for Readability and Comprehension Using SGML"; Computer Science Department; Uiniversity of Waterloo, ONT, CA, March 1994

[Cross95] Cross, J.H.; Hendrix, T.D.; "Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software"; *Proceedings of Second Working Conference on Reverse Engineering*; July 1995; Toronto,ON, CA

[Edwards93] Edwards, H.E.; Munro, M.;"RECAST - Reverse Engineering from COBOL to SSADM"; *Proceedings of The Working Conference on Reverse Engineering*; May 1993, Baltimore, Maryland

[Jarzabek95] Jarzabek,S.;Keam; T.P.; "Design of a Reverse Engineering Assistant Tool"; Proceedings of Second Working Conference on Reverse Engineering; July 1995; Toronto,ON, CA

[Garg90] Garg, P.K.; Scacchi, W.; "A Hypertext to Manage Software Life-Cycle Documents"; *IEEE Software*; May 1990; pp. 90-98

[Goldfarb90] Goldfarb, C.; *SGML Handbook*; Oxford University Press, 1990

[Horowitz85] Horowitz, E; Kemper, A.; Nagarasimham; "A Survey of Application Generation"; *IEEE Computer*; Vol. 18; n. 2 (Jan 1985); pp. 40-54

[HTML] **http://www.w3.org**

[Hubscher95] Hübscher, P.J.E; "O Processo de desenvolvimento de software utilizando Talisman"; Dissertação de Mestrado; Departamento de Informática; PUC-Rio; 1995

[Java] **http://java.sun.com**

[Johnson95] Johnson, W. Lewis; Erdem, Ali; "Interactive Explanation of Software Systems"; *Proceedings of the 1995 IEEE Knowledge Based Software Engineering Conference*, Boston, MA; pp. 155-164; 1995

[Leite94] Leite, J. C. S.P.; Sant'anna M. ; Freitas, F. G.; "Draco-PUC : a Technology Assembly for Domain Oriented Software Development"; *Proceedings of the 3rd* IEEE International Conference of Software Reuse; 1994.

[Leite95] Leite, J.C.S.P.; Cerqueira, P.M.; "Recovering Business Rules from Structure Analysis Specifications"; *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press;1995

[Neighbors84] Neighbors, J.; "The Draco approach to constructing software from reusable components"; *IEEE Transactions on Software Engineering*, SE-10(05):564-574, Sep 1984

[Newcomb95] Newcomb, P; Kotik, G.; “Reegineering Procedural into Object-Oriented Systems”; *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press;1995; pp. 237-252

[Refine92] Reasoning Systems; Refine User’s Guide; Palo Alto, CA, 1992

[Selfridge93] Selfridge, P.G ; Waters, R.C. ; Chikofsky, E.J.; “Chalanges to the Field of Reverse Engineering”; *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press;1993

[Staa93] Staa, A.v.; *Ambiente de Engenharia de Software Talisman, Manual do Usuário*; Rio de Janeiro; Staa Informática; 1993 - in portuguese

[Staa95a] Staa, A.v; “Totem - Um Meta-ambiente de Engenharia de Software”; *Monografias em Computação*, MCC35, Departamento de Informática, PUC-Rio; 1995 - in portuguese

[Staa95b] Staa, A.v. et al “Regras e recomendações para a inclusão de especificações no código de programas C ou C++”; *Monografias em Computação*, Departamento de Informática, PUC-Rio (under development) - in portuguese

[Wells95] Wells; C.H.; Brand, R.; Markosian; L.;“Customized Tools for Software Quality Assurance and Reengineering”; Second Working Conference on Reverse Engineering; July 1995; Toronto,ON, CA

[Zoufaly95] Zoufaly, F.; Araya, C.; Sanabria, I.; Bendek, F. ; “RESCUE : Legacy System Translator”; *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press;1995; pp. 39-52