Documentu: A Flexible Architecture for Documentation Production Based on a Reverse Engineering Strategy

CHRISTIANO de OLIVEIRA BRAGA, ARNDT von STAA AND JULIO CESAR SAMPAIO do PRADO LEITE

Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, R. Marquês de S. Vicente 225, 22453-900, Rio de Janeiro, Brazil

SUMMARY

Good documentation is essential to 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 HTML hypertext browser to help the access and navigation of system documentation. The prototype was used on a large scientific system that belongs to an oil company, with positive results.

Keywords: documentation; transformation systems; CASE tools; software engineering; tool integration; reverse engineering

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 system information is solely embedded in the current source code. However, other sources of information should be available in order to provide a better understanding of the software and its environment (Biggerstaff, Mitbarden and Webster,1994), (Selfridge,Waters and Chikofsky, 1993), (Leite and Cerqueira, 1995). Good documentation, easily accessible and easy to read has been sought for a long time (Horowitz, Kemper and Nagarasimham, 1985) (Garg and Scacchi, 1990). It is essential to software production, which employs several software engineers, working in a cooperative fashion.

CENPES is the research institute of Petrobrás, the Brazilian Government Oil Company. This center uses several software systems developed both internally and by third parties. One of its projects, DOC, was created in order to improve the quality of the software products developed at CENPES, by making the maintenance process easier and increasing the quality of the documentation assets. The project is a partnership between LES – Software Engineering Laboratory at PUC-Rio and this research center.

One of the deliverables of the DOC project was a set of guidelines for development and documentation of software code (Staa *et al*, 1995, pp. 5). However, practical experience has shown us that the guidelines would only produce the expected feedback, if a software tool could assist them.

We have surveyed some of the existing proposals for documentation tools at the code level, and based on their strong and weak points, we proposed our architecture for a tool that implements

automatic support to the guidelines. Our proposal uses the HTML markup language and relates the information in terms of the whole system, not only for individual programs. The tool generates documentation for annotated and unmarked code as well.

We have implemented our proposal in a tool called *Documentu* (Braga, Staa and Leite, 1997, p. 9-13). The main objectives of the tool are:

- help software developers in the maintenance task;
- if the software being developed is a library, aid the software users in handling the product;
- help a new developer to understand the software system that he is going to work with.

Documentu ideally uses code tag marked as defined by Staa's guidelines [1]. However, it can also produce documentation from bare source code. This is possible because we generate software documentation based on both the source code grammar and the markup grammar. We have used the transformation system of Draco-PUC (Leite, Sant'anna and Freitas, 1994) to recover system information, from code and markups. Talisman (Staa, 1993, pp. 1.1) CASE tool was used to store, validate and format the information. Finally we use a standard HTML 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, markups. This information is passed to the CASE tool that stores it in a repository. This repository has database management capabilities and is able to provide a series of consistency checks that can be applied to the entire system. Using the information in the repository, the CASE tool generates a HTML file by means of special procedures called linearizers.

Before detailing our proposal we make a brief survey of existing documentation proposals (Section 2). In Section 3, we describe the tool architecture. In Section 4, the *Documentu* tool is presented. Section 5 reports on the results of applying *Documentu* to a working scientific software. Section 6 presents what can be generated with the current version of the tool. We conclude the paper in Section 7, 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 issue is confronted with a plethora of new possibilities through 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 the following categories: standards, literate programming, using markup languages and dat abases, and recent documentation tools.

[1] The guidelines provide suggestions on the coding process, like naming variables or functions and inserting annotations in comments to document design decisions. It also has guidelines for object-oriented programming and testing. The guidelines deal with higher abstraction levels as well, making suggestions on how to structure software projects.

2.1 Document standards

SGML (Goldfarb, 1990, pp. 217-494) 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. Figure 1 shows 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 SGML 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 necessarily have those sections.

HTML (World Wide Web Consortium, 1995) is said to be an application of the SGML standard. The language is used to create portable hypertext documents. The HTML markups can 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) (Appelt and Tetteh-Lartey, 1993) (Brown, 1989) 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 of the document (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 supporting the creation of hypertext documents.

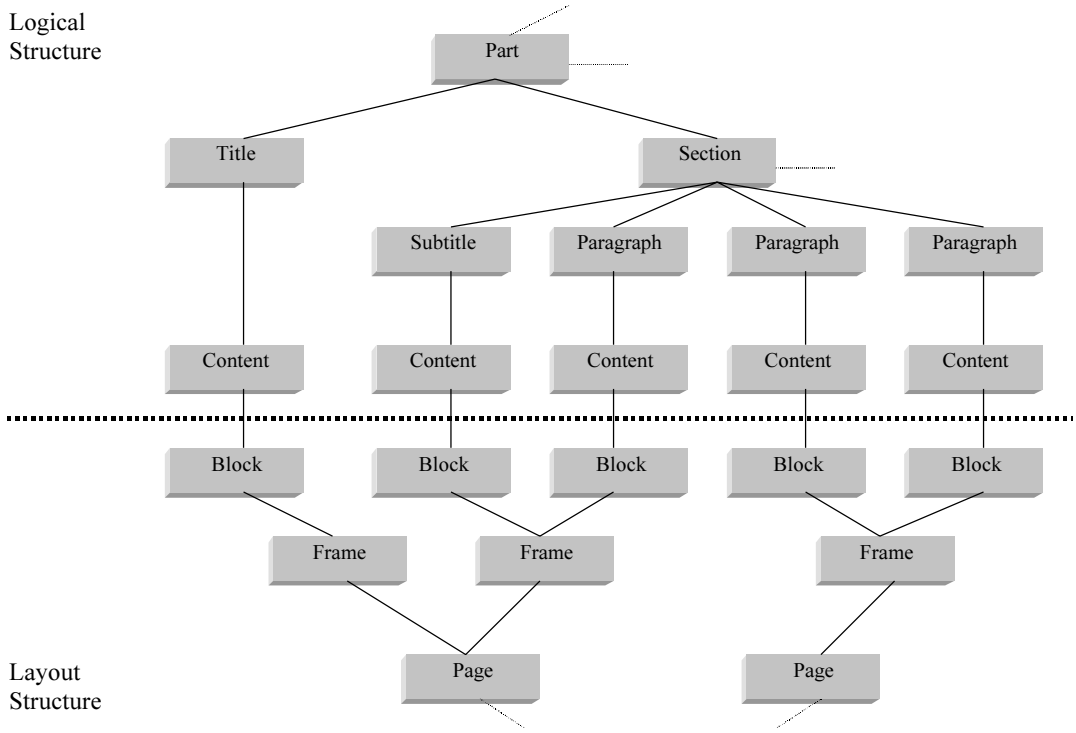

Figure 2. ODA Architecture

2.2 Literate Programming

Knuth (Knuth, 1984, pp. 97-98) proposed the joining of a document format language to a programming language in an approach called Literate Programming. In WEB— Knuth's implementation of Literate Programming proposal— he uses the well known languages TEX and Pascal. To develop a system using Literate Programming, the programmer must, at first, write a 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 WEAVE to produce a TEX file.

The use of the TANGLE processor sets up the Pascal 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.

Another problem is that TEX 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.

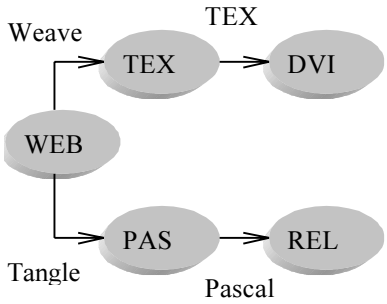

Figure 3. Generating WEB documents

2.3 Markup languages and databases

Cowan presents a more elaborated solution (Cowan, Germán, Lucena and Staa, 1994, pp. 2) than the one proposed by Knuth. 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 hypertext format.

In (Cowan, Germán, Lucena and Staa, 1994, pp. 5), all 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 through different views in a WYSIWYG (what you see is what you get) environment, allowing database update through those views.

Cowan, Germán, Lucena and Staa (1994, pp. 8-10) bring to light 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 generation of software development process documentation, but most of them only produce documentation automatically without checking for consistency with the code.

Java (Gosling, Joy and Steele, 96, pp. 1) is a programming language developed at Sun Microsystems to support the development of object-oriented software. Figure 4 shows the browsing of a document generated by JavaDoc (Flanagan, 1996, pp. 220-221), a tool that generates HTML documents from tag annotated Java source code. The document in Figure 4 was generated from the annotated code presented in Figure 5.

JavaDoc formats public and protected information from classes, interfaces, constructors, methods, and fields. Figure 5 shows an example of JavaDoc tags. HTML tags can also be used to format comments. User defined cross-reference tags, such as @see (not shown in the example), may be manually inserted. JavaDoc creates documentation (an index, a class hierarchy and description files) for Java source files or for packages.

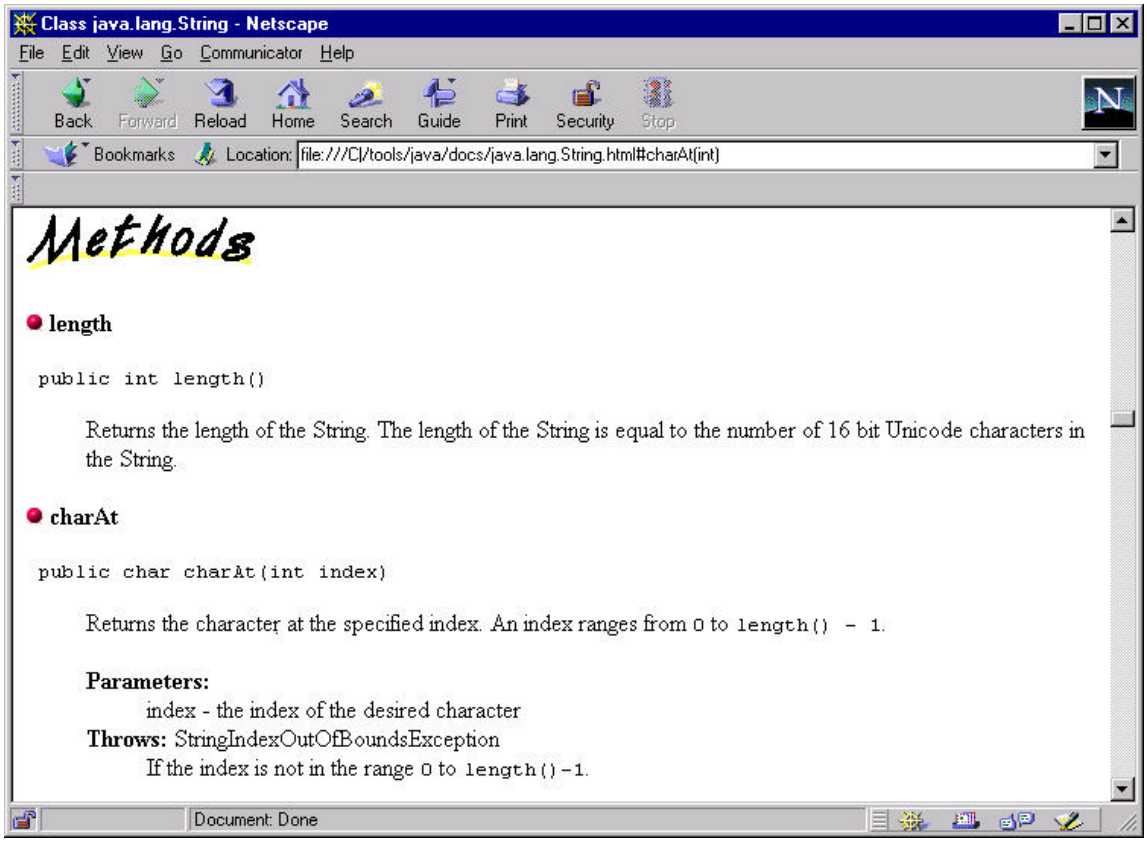

Figure 4. JavaDoc document browsing

```
/**
* Returns the length of the String.
* The length of the String is equal to the number of 16 bit
* Unicode characters in the String.
*/

public int length()
{
    return count;
}

/**
* Returns 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
* @exception StringIndexOutOfBoundsException If the index is not
*   in the range <tt>0</tt> to <tt>length()-1</tt>.
*/
public char charAt(int index)
{
    if ((index < 0) || (index >= count))
    {
     throw new StringIndexOutOfBoundsException(index);
    }
    return value[index + offset];
}
```

Figure 5. Annotated Java code

Microsoft provides an unsupported tool called AutoDuck. This tool works much like JavaDoc. It uses markup tags in the source code to generate documentation. This documentation is 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. The AutoDuck tag @class (Figure 6) 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 7 shows the browsing, using the Windows help system, for the example presented.

AutoDuck provides the capability of parsing user defined tags. JavaDoc parses only its predefined tags. Both of them are easily integrated into the building process through makefile invocation, improving documentation consistency with the source code. If the markups were consistently updated and the documentation generated whenever the code changed, then the documentation would always be consistent with its associated source code.

AutoDuck does not support system wide documentation. It is only concerned about local modules and cross-references have to be created manually by the developer, generating possible sources of inconsistency. Neither AutoDuck nor JavaDoc perform consistency checking in the retrieved information.

```
/* @doc
 * @module TEMPLATE.CPP - Some Template Expls.
 * This module shows how to code Autoduck tags
 * template classes and member functions.*/
//@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 6. AutoDuck tags

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 (Knuth, 1984, pp. 97). The existence of a document logical structure, defined using a standard in order to achieve document portability is a major issue (Goldfarb, 1990, pp. 402-433). It is also important to store system wide code information, allowing checking among information fragments, so the need for a database (Cowan, Germán, Lucena and Staa, 1994, pp. 8). Another important issue is the use of a hypertext browser to navigate through the information (Bigelow, 1988), (Rajlich, Damaskinos, Khorshid and Linos, 1990). The use of hypertext provides an easy way to access on-line information, as well as allowing information layering, such that information can be structured.

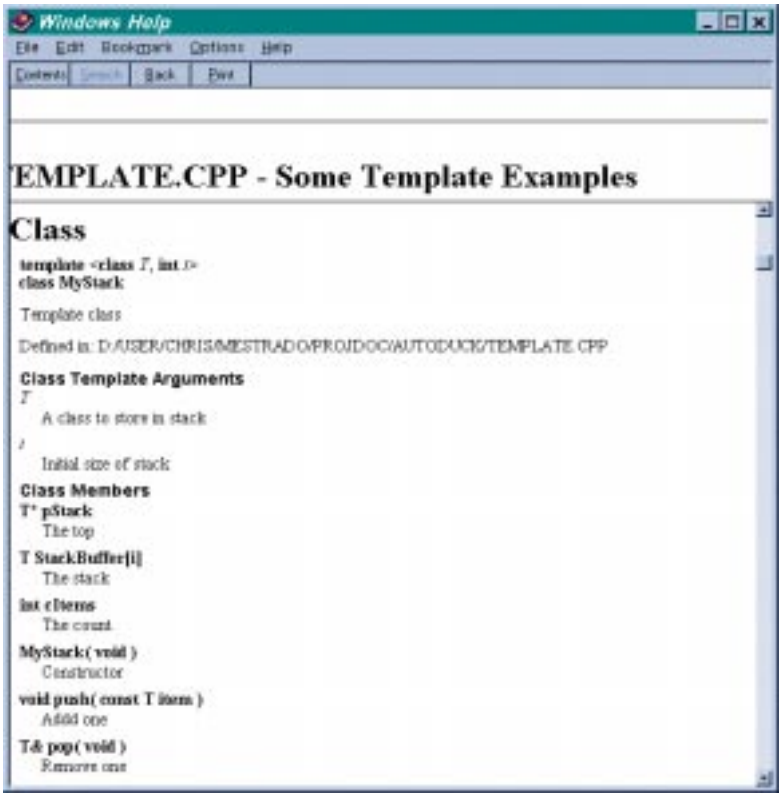

Figure 7. AutoDuck browsing

3. THE PROPOSED ARCHITECTURE

Our proposal is geared to marked code, but it also treats unmarked code. This makes possible the generation of the documentation based solely on a system source code (currently C\C++ and Fortran 77). It is important to emphasize that the tags only complement the information already captured by the tool. *Documentu* is a tool that instantiates such proposal. The marks treated by our proposal are based on coding guidelines.

These coding standards will be described in section 4. 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.

The information extractor (Figure 8) parses the source code and produces information that is going to be used by the information formatter system. The formatter system then produces a hypertext system in HTML. The first part of the process, the information extractor system, uses Draco-PUC (Leite, Santˆanna and Freitas, 1994) transformation capabilities. The second part of the architecture, the information formatter system, is based on a repository based CASE tool called Talisman (Staa, 1993, pp. 1.7).

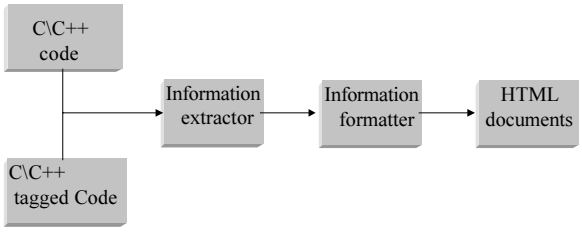

Figure 8. Outline of the proposed architecture

3.1.1 Draco-PUC

Draco-PUC (Leite, Sant'anna and Freitas, 1994) is a transformation system that implements the Draco paradigm (Neighbors, 1984). 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 layout 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 layout 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 kinds of features are responsible for driving the transformation engine in a 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 bound 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 also have control points (Initialization and End) and properties. Sets of transformations are encapsulated on Transformers. Transformers have three control points: Declaration (global object declaration), Initialization and End.

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 allow the developer to instantiate the RHS in target workspaces.

The use of workspaces can be viewed as an asynchronous replacement, where the RHS is written in a 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 (Staa, 1993, pp. 1.7) is a repository based CASE tool. The environment has predefined database schemas, called *Talisman Definition Languages* (Staa, 1993, pp. 1.11) that give the 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 (Staa, 1993, pp. 1.7) is a database composed of textual fragments and references to textual fragments inside the database. A definition language (Staa, 1993, pp. 14.1) is characterized by the relations among its elements (modules, classes and functions, for example) and also *form programs* (validators and linearizers).

Those *form programs* could be loaded at run-time to customize the environment. The validators check for consistency of the data inside the software base, whereas 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 an 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 (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, indicating its robustness. However, it was necessary to extend the domain in order to make possible the analysis of special tags.

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

The process starts with the analysis of the source code. Once the code of a given module is analyzed and the Draco-PUC abstract syntax tree is created, a set of transformations, created specially to implement this approach, are applied over that internal format in order to create the Talisman import file for that module. This operation 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 9 shows this process. In the first phase, 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 restricted to Talisman. Each import file outputted by Draco-PUC machine 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-reference 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.

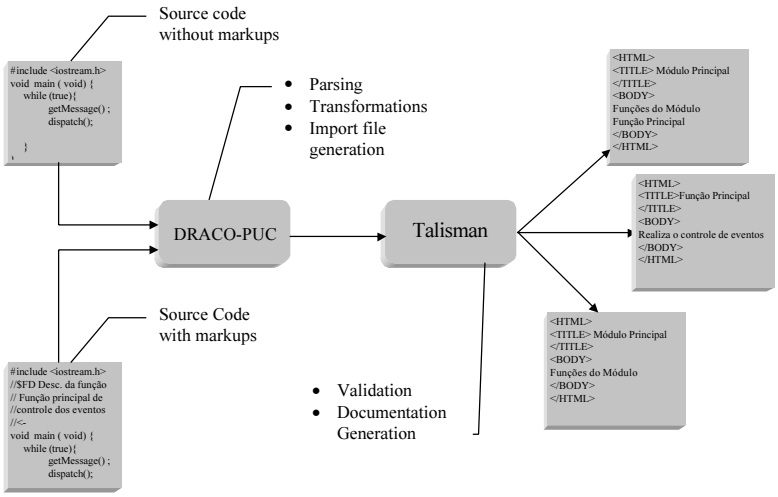

Figure 9. Execution process

4. DOCUMENTU

We describe *Documentu* by focusing on details of its architecture, its execution and the results produced. The tool is based on three Draco-PUC domains, C++, Fortran and the Talisman domain, and on the its repository defined in Talisman. Below we give details on how these 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. A domain language grammar description is written on an extended-BNF style. An example of such grammar descriptions is shown in Figure 10, which presents the CPP grammar format— for C++ language— fragment responsible for the description of class heads like:

```
class Cwindow {…}
//Or …
class CMyWindow : CWindow {…}
//Or …
class CAnotherWindow : CWindow, CBaseWindow {…}
```

Just like with C++, the Talisman import file grammar description was also encapsulated in Draco-PUC. An excerpt of this grammar (Talis), which describes textual import entry for the repository is also 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.

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 could 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 (Staa *et al*,1995, pp. 11-32) 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.

```
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'
           ;

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. Fragments of the CPP and Talis grammars

```
////////////////////////////
//$CNAME Class long name
//VMS - Segment
//$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.
//////////////////////////////
class VM_Segment
{
     ...
};
```

Figure 11. Example of embedded tags

In *Documentu*, there are two types of transformations: tag matching and structure matching (concerning language structures). The tag matching transformations retrieve the design information embedded in the code as comments (Staa *et al*, 1995, pp. 11-32). The structure matching transformations intend to retrieve information about the program structure using the code itself as the main source of information. The documentation structure is created using the information retrieved by these transformations.

An example of a tag matching transformation is shown in Figure 12. The transformation looks for a class name tag specification (`$CNAME`) and then stores the proper information in a workspace (WSClassLongName). Following Staa's commenting standard, each class should have a long name in order to provide a better understanding of the abstraction. Later in the transformation process, a transformation that looks for classes—Figure 13— will be executed and will use the name stored in that workspace to set the long name attribute for the associated class. The class that will have the attribute set is the next class following the comment describing the CNAME attribute.

```
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

```
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 Class transform is an example of structure matching transformation. This transformation is responsible for generating the text that will create a class element in the Talisman repository (this is like creating a record in a relation database). This transformation also gathers other information from several workspaces, concerning the class being analyzed, and generates the text for that. This text will be imported by Talisman and populate the repository. An example of the generated text would be:

```
DA "Class" "VM_Segment" Name
DA "Class" "VM_Segment" Text ClassName
VMS - Segment
#@#
```

The first line creates a Talisman class element named VM_Segment. The second line sets the ClassLongName attribute of VM_Segment to VMS – Segment.

In the current version of the tool, the set of transformations can not be expanded by a regular user. Once the transformation is written and compiled, it is only necessary to add it to the Draco-PUC script that applies the transformations. The problem is to write the transformations. The user needs to understand the C++ grammar— written in BNF-like syntax— learn Draco-PUC transformation language and the structure of the already implemented transformations, in order to use the retrieved context information. Also, the compiler used to generate Draco-PUC has to be the same for all transformations.

4.2 The Role of Talisman in the proposed Architecture

A new definition language has been implemented in Talisman to support the *Documentu* project. It is called Program Documentation language. Figure 14 shows a diagram with the language schema. This new language is based on Modular Programming (Staa, 1993, pp. 22.1), but the information it manages is related to source code documentation. When a file is imported, the information inside the file populates the software base.

Once the base is filled, the validators should be applied followed by the linearizers. In *Documentu* a linearizer is responsible for the creation of the hypertext system in HTML and 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 linearizer navigates inside the software base looking for specific text fragments and relations and generates a file with the text fragment information in HTML format. An example of a Talisman form is shown in Figure 15.

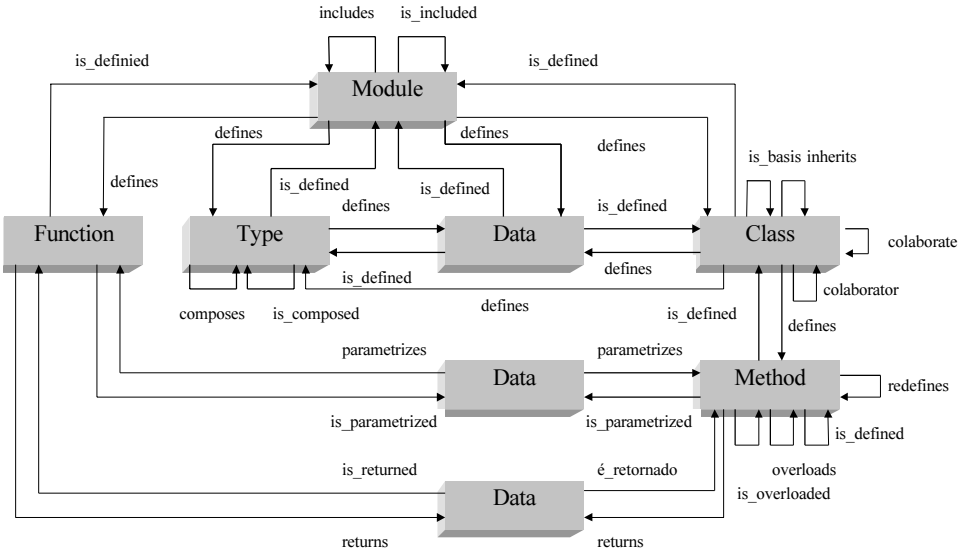

Figure 14. Model for Program Documentation definition language in Talisman

Figure 15 is an excerpt of the class linearizer. The first two lines of the code, after the forall statement, just call two other forms (like procedures, in this case). The first form program creates annotations that will be used by the file splitter at the post-processing phase. The second form program produces the markups that all HTML file should have (<HTML>). A line with the class name is created next, using the HTML markup <H1>, since 1 was passed as a parameter to "Create Title" form. A reference to the file class.htm is also created using the form "Create Reference" with "System Classes" as the text reference. The if clause checks weather a class has a relation with some module or not. If it does, a reference to the related module will be created writing the module name in the reference.

```
Forall Classes do
    Frm "Create BeginSplittingSeq"(Convert Sequence [Name]);
    Frm "Create BeginHTMLFile";
    seqTitle = ConcatSeq ( "Class ",Convert Sequence [Name]);
    Frm "Create Title" (seqTitle,1) ;
    Frm "Create Reference" ("classes.htm","System Classes") ;
    if Exists ([Relation Module]) then
     Forall Module do
       seqClasseHead = ConcatSeq (Convert Sequence [Name],".htm");
        Frm "Create Reference"(seqClasseCabec,   ConcatSeq("Module",
                                  Converte Sequence [ Name ]));
     end ; /* Forall */
  else
  end ; /* if */
end ; /* forall */
```

Figure 15. Talisman form excerpt

First the linearizer creates the documentation structure and the index and only then the documentation for each element of an entity is created. Viewing the documentation as a graph, first the raw nodes with the links are created, then the nodes attributes are filled.

One may be thinking why Talisman was used, since Draco-PUC could generate all HTML files

itself. Talisman was used for the same reasons a database is used in software development. We wanted to be able to generate the documentation using a high 4GL like language construction. Talisman *form language* possesses these constructions, as shown in Figure 15. The language provides constructions like forall, exists and also the capability to define new procedures. If Talisman were not used, we would have to manipulate text in files generated by Draco-PUC. Also, Talisman is freeware— a pre-requisite since we want to distribute *Documentu* tool— well documented and with the people who built it available to implement the features we wanted — like creating a command line version of it.

Lua[2] (Ierusalimschy, Figueiredo and Celes, 1996) scripts files drive all the execution process. The user only needs to call the main script file with module name he or she wants to document. Once the main script file is called, the first thing it does is to call a preprocessing script file, responsible for calling the C++ pre-processor, that expands the source code with the information inside the included header files. The pre-processed code is then analyzed and further transformed into another file.

The file built by the transformation process is imported by Talisman, and it is to populate a software base with the information contained in the file. The HTML generator script file then calls Talisman to linearize the software base. The post-processing batch file calls the file splitter executable and the character converter to generate the multiple HTML files and to generate the HTML entities for the accentuated characters respectively. When this all done, the user may invoke his or hers HTML 3.0 (since the produced documentation is frame based) compliant browsers to navigate through the documentation.

4.3 The Hypertext

All the links are created using the extractor facilities of Draco-PUC (parser and transformations). The detailed information about each element (class and methods, for example) comes from the tags text embedded in the code. The information retrieved from the code analysis is used to create the hyper document structure: the index and hyperlinks cross-referencing system entities like modules, functions and classes. This structure is what is created when no annotations are available in the code comments. The current implementation of *Documentu* generates a hypertext interface organized in two frames as shown in Figure 16.

The frame in the left side is an index to system elements. The index is organized in tabs, like Windows selection tabs. The first one is the index to the system modules. Each bullet inside that frame represents a link to a module in the system. When a module is selected in the left frame, the big pane in the right changes, showing that module information, like its associated classes, its functions, types and data. The same thing can be done with the other three tabs in the left pane, 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 `Shell` module (Figure 16). When selected, in the pane on the left side, the big pane changes. The user can browse through information such as: classes in a module, functions, types and all the information defined in the model in Figure 14 and also, the information in the markups— as defined in (Staa *et al*,1995, pp. 11-32). Figure 16 presents a snapshot of the documentation browsing of the module (Shell.h) that has its classes annotated with markups describing their functionality.

---

[2]Lua is a portable script language developed at Department of Informatics of PUC-Rio.

Figure 16[3]. The hypertext browsing

5. USING *DOCUMENTU* IN A REAL SYSTEM

The development of *Documentu* tool started as part of the DOC project. This project— between LES (Software Engineering Laboratory at PUC-Rio) and CENPES (Brazilian Petroleum Company Research Center)— has the objective of increasing the quality of the software developed in that center.

In order to validate the tool, a large system being developed at CENPES was chosen to be the test case. The chosen system is called Petrox, a simulator for chemical processes. It is composed of an editor where the user may drop objects from a chemical process and simulate the execution of a particular instance of a process. The editor calls functions implemented in Fortran, which implement the algorithms needed to run the simulation.

[3] The generated text is written in Portuguese. "Módulo Shell" means "Shell Module" and " Descrição" means "Description". There are two links bellow the big pane title: " Módulos do sistema", that means "System Modules" and "Descrição detalhada do módulo" that means "Detailed module description".

Module name	Size (Kbytes)	#Tag Tfs.	#Lang. Tfs.	#Tfs p/module
ABSTBOX.H	82,057	82	89	171
ABSTSLCT.H	126,587	160	167	327
APPLIC.H	65,025	0	0	0
BINDER.H	14,126	57	56	113
BINDMENU.H	103,286	56	65	121
BSELECT.H	87,889	180	181	361
BUTRADIO.H	82,431	93	99	192
BUTTON.H	78,854	229	226	455
CANVAS.H	78,801	372	354	726
COLLBIND.H	50,192	92	99	191
COMPOSER.H	46,441	461	538	999
CONTROL.H	71,931	0	708	708
DATUM.H	12,322	91	94	185
DFLTBIND.H	130,874	114	117	231
DIALOG.H	93,217	309	299	608
DIMFLOAT.H	63,637	196	194	390
DIMTEXT.H	251,038	221	230	451
DIMTYPES.H	4,573	0	0	0
ENTRBIND.H	16,555	85	94	161
ENUMERAT.H	3,328	73	88	161
FILL.H	73,488	47	54	101
FLAGITEM.H	75,797	69	75	144
FLTVLD.H	51,331	193	187	380
FRAME.H	81,415	85	89	174
GERAL.H	1,362	8	8	16
HBOX.H	83,906	59	68	127
HEAPVIEW.H	101,035	62	70	132
INDEXDLG.H	110,243	60	69	129
INTVLD.H	50,298	152	146	298
LABEL.H	99,784	184	183	367
MENU.H	98,398	89	91	180
MENUITEM.H	73,950	88	93	181
MFLOAT.H	14,446	91	94	185
MINTEGER.H	14,216	91	94	185
MULTLINE.H	104,036	181	182	363
MULTSLCT.H	129,102	109	114	223
NUMTYPES.H	121,888	0	23	23

Module name	Size (Kbytes)	#Tag Tfs.	#Lang. Tfs.	#Tfs p/module
OBJBIND.H	94,502	75	79	154
OBJSTRM.H	65,726	0	1067	1067
OPTBIND.H	18,322	126	143	269
OPTBUT.H	83,856	165	168	333
OPTMENU.H	103,870	57	66	123
PTRBIND.H	71,040	110	125	235
RADIO.H	84,653	96	104	200
SAFEREF.H	23,185	46	51	97
SELECTOR.H	132,599	218	225	443
SEQBIND.H	72,085	122	137	259
SHELL.H	79,648	218	207	425
SLCTDLG.H	122,726	66	73	139
SLCTMENU.H	103,913	57	67	124
SRCBIND.H	31,041	127	141	268
STATUSLN.H	101,138	65	72	137
STREAMBL.H	5,249	43	53	96
STRNG.H	37,297	246	255	501
STRVLD.H	70,635	94	101	195
SUBMENU.H	99,907	60	67	127
SYSDEPEN.H	8,062	0	0	0
TABLE.H	156,298	0	560	28
TASK.H	3,049	14	14	28
TEXT.H	107,168	161	166	327
TOGGLE.H	79,375	208	204	412
TOOLBOX.H	85,174	59	69	128
UNIT.H	53,308	345	378	723
UNITBIND.H	129,936	90	93	183
UNIT_ID.H	10,520	4	0	4
VALIDAT.H	40,816	208	180	388
VBOX.H	83,966	59	68	127
XGETCH.H	3,473	2	2	4
ZBOX.H	86,102	151	155	306
#Modules	**69**			
Size of Modules	5,057,595			
#Applied transformations	**17609**			

Table 1 - Statistics obtained running *Documentu* using Magoo class library as input

Magoo GUI (Graphical User Interface Library) class library is a significant part of Petrox, currently with 72 header files in 5,057,595 bytes, in almost 208,552 lines of expanded code. We have been using Magoo as one of our test cases[1], since its code uses advanced features of the C++ language.

Documentu took 30 minutes transforming the source code on a Pentium 100 running Windows NT 4.0 to produce the documentation, applying 17,609 transformations. The documentation was spread along 1803 files using 1,128,953 bytes of disk space.

These results sound pretty much acceptable, considering the amount of information retrieved and that the documentation process is a batch process and may run in off-duty hours. summarizes the results of running *Documentu* with Magoo as input.

Talisman linearizers have 32 functions and 2012 lines of code. *Documentu* has 64 transformations implemented, distributed as follows:

- 43 concerning C++ classes;
- 30 concerning modules;
- 40 concerning functions
- 61 concerning methods;
- 23 concerning structures, unions and enumerations.

Several features have been implemented in the tool since the delivery of the first prototype and many difficulties were encountered. The major problem found concerns the C++ parser. One of the first project decisions was that the symbol table generated by the currently available C++ parser— that ships with Draco-PUC— should be kept. The reason was because we wanted to evolve the project towards a reverse engineering tool. This decision made a serious impact in the parsing phase, because the pre-processing of the source code became an obligation, since every type identifier was supposed to be in the symbol table.

Most of the parsing problems encountered were in system files included in the source code by the pre-processor. The majority of our efforts were in making the parser as reliable as possible to support parsing included system files.

It was indeed a very difficult task, which was not fully completed, since the C++ language is not yet standardized. What should have been done— and will be in the near future— is to remove the symbol table and thus handle every type identifier simply as identifiers and do not save context information. Any necessary context information will be handled by the transformation system. With this strategy it will not be the necessary to handle compiler specific features of the language, sometimes implemented in system header files.

Another modification that could be done in the parser is to flatten some parts of the grammar to sequences of identifiers, making the C++ grammar more flexible in relation to its formal description. When using the whole grammar we have to be aware of language constructions that will not be handled by the transformation system, but that must be present in order to have the code parsed. This is another inheritance from the project decision of implementing a parser for a reverse engineering tool.

Concerning the generated documentation, our experience has shown that using the language grammar to produce the structure of the documentation was pretty successful. Generating documentation from code that did not have any markup induced the user to annotate his or her

[1] Classlib— another relevant part of the Petrox project with 89 files and 369 Kbytes— and Vix— a class library for visual objects handling, with 43 files and 189 Kbytes— were also used to test the tool.

code in order to produce a better documentation.

The system is presently installed in three different host systems at CENPES and it is still undergoing tests. The actual version is almost stable but still needs some technical support. We expect that it will reach desired stability in the next version provided the modifications discussed above are included.

6. RESULTS ACHIEVED

We realize that system documentation is a widely studied field with significant proposals and commercial products providing very powerful and complex solutions. We believe that our architecture contributes with the following:

- A flexible architecture that can be easily maintained and updated. Without changing the documentation generators, new programming languages can be supported by adding a new parser and transformers to the process. Also, the markup structure could be changed to follow different documentation guidelines, with different tag structures in the same way a new language is added. By loading new Talisman linearizers the documentation structure can be changed or instead of HTML files, a postscript file could be generated.
- Compiler independence. Most of the currently available IDEs (Integrated Development Environment) provide features for code exploration. Our tool supports most of those features (like attribute, method, class, cross-referencing browsing) and it is compiler independent.
- System wide documentation. The documented project can be as wide as the user wants, with references to as many modules and syntactic elements as the existing code dependencies. Raw documentation, if no documentation tag is used at all, showing the system basic structure can be produced.
- Documentation portability as a consequence of the usage of a standard language.

It is also important to stress that using the CASE tool Talisman (Staa, 1993, pp. 14.1) it was possible to write form programs that could validate the integrity of the data relations (validators). 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. We understand that one of the strong points of our proposal is the low coupling among its components.

Also, there are some points we realize that must be reviewed in the next versions of the tool:

- Draco-PUC is still a prototype with some problems concerning usability and documentation. Creating the parser is not a very easy task. Talisman has an obsolete user interface. Creating a command line version, which also fits a lot better with our batch documentation process, has solved this. Also, the tool is only available in Windows platform. We are working in a Unix version.
- The generated documentation is static. The user can not "ask questions" different from those defined in the form programs. We intend to add a feature in the next version of the tool allowing the user to generate his own "questions", generating HTML documents that satisfy those questions.

7. CONCLUSIONS

We have proposed an architecture to produce high quality system wide information based on source code information. 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 he information retrieval and the other is a CASE tool that deals with the storage and the manipulation of the retrieved information.

Other authors (Wells, Brand and Markosian; 1995) (Zoufaly, Araya, Sanabria and Bendek, 1995) (Newcomb and Kotik, 1995) propose the use of transformation systems to help the documentation problem. Our work innovates by using a transformation system together with a repository based system. For instance we believe that our proposal, regarding the repository aspect, is related to the work of Edwards and Munro (1993) and Jarzabek and Keam (1995). 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 domain oriented knowledge, having the information on Talisman repository is a first step to link with other Talisman languages, thus making possible to handle higher level representations.

An approach similar to ours was presented by Cross and Hendrix (1995). In their case a source code grammar was not used, the generation of the documentation was based only on the markup gra mmar alone. Johnson *et al* (1995) are focusing their attention on the interactive and inquiry aspect of program understanding in an architecture similar to ours.

Future work will not only explore its use, but as mentioned above, explore the full possible links with other representations dealing with higher levels of abstraction. In particular we are planning to explore the integration with the strategy described by Leite and Cerqueira (1995), which could also be based on the Talisman environment.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

Appelt, W. and Tetteh-Lartey, N. (1993) 'The Formal Specification of the ISO Open Document Architecture (ODA) Standard', *The Computer Journal*; **36**(3), pp. 269-279.

Bigelow, J. (1988) 'Hypertext and CASE', *IEEE Software*, **5**(2), pp. 23-27.

Braga, C.O., Staa, A. von and Leite, J.C.S. (1997) 'A Hybrid Architecture for Documentation Production', Technical Report, MCC15/97, Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, 17 pp.

Brown, H. (1989) 'Standards for Structured Documents', *The Computer Journal*, **32**(6), pp. 505-514.

Biggerstaff, T., Mitbarden B. and Webster, D. (1994) 'Program Understanding and The Concept Assignment Problem', *Communications of ACM*, **37**(5), pp. 72-83.

Cowan, D., Germán D., Lucena, C. and Staa, A. (1994) 'Enhancing Code for Readability and Comprehension Using SGML', Proceedings of the 2nd International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, pp. 181-190.

Cross, J. and Hendrix, T. (1995) 'Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software', *Proceedings of Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-6.

Edwards, H. and Munro, M. (1993) 'RECAST - Reverse Engineering from COBOL to SSADM', *Proceedings of The Working Conference on Reverse Engineering*; IEEE Computer Society Press, Los Alamitos, CA, pp. 44-53.

Flanagan, D. (1996) *Java in a Nutshell*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 438 pp.

Jarzabek, S. and Keam, T. (1995) 'Design of a Reverse Engineering Assistant Tool', *Proceedings of Working Conference on Reverse Engineering*; IEEE Computer Society Press, Los Alamitos, CA, pp. 61-70.

Garg, P. and Scacchi, W. (1990) 'A Hypertext to Manage Software Life-Cycle Documents', *IEEE Software*, **7**(3), pp. 90-98.

Goldfarb, C. (1990) *SGML Handbook*, Oxford University Press Inc., New York, US, 664 pp.

Gosling, J., Joy B. and Steele G. (1996) *The Java Language Specification*, Addison Wesley Publishing Co., Reading, MA, 825 pp.

Horowitz, E., Kemper, A. and Narasimhan, B. (1985) 'A Survey of Application Generation', *IEEE Software*, **2**(1), pp. 40-54

Ierusalimschy R., Figueiredo L. and Celes W. (1996) 'Lua---an extensible extension language', *Software: Practice & Experience*, **26**(6), pp. 635-652.

Johnson, W. L. and Erdem, A. (1995) 'Interactive Explanation of Software Systems', *Proceedings of the 1995 IEEE Knowledge Based Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 155-164.

Knuth, D. E. (1984) 'Literate Programming', *The Computer Journal*, **27**(2), pp. 97-111.

Leite, J., Santanna M. and Freitas, F. (1994) 'Draco-PUC : a Technology Assembly for Domain Oriented Software Development', *Proceedings of the 3rd IEEE International Conference of Software Reuse*, IEEE Computer Society Press, Los Alamitos, CA, pp. 94-101.

Leite, J. and Cerqueira, P. (1995) 'Recovering Business Rules from Structure Analysis Specifications', *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 13-21.

Neighbors, J. (1984) 'The Draco approach to constructing software from reusable components', *IEEE Transactions on Software Engineering*, **SE-10**(05), pp.564-574.

Newcomb, P and Kotik, G. (1995) 'Reegineering Procedural into Object-Oriented Systems', *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 237-252.

Rajlich V., Damaskinos N., Khorshid W. and Linos P. (1990) 'VIFOR: A Tool for Software Maintenance', *Software: Practice and Experience*, **20**(1), pp. 67-77.

Selfridge, P., Waters, R. and Chikofsky, E. (1993) 'Challenges to the Field of Reverse Engineering', *Proceedings of The Working Conference in Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp.144-150.

Staa, A. (1993) *Ambiente de Engenharia de Software Talisman, Manual do Usuário*; Rio de Janeiro; Staa Informática, 296 pp.

Staa, A., Derraik, A., Braga, C., Costa, G., Kanamori, L., Jaccoud, M., Giovani, P., Hübscher, P. , Baptista, R., Correia, R. (1995) 'Regras e recomendações para a inclusão de especificações no código de programas C ou C++', Technical Report, Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro, 25 pp.

Wells, C., Brand, R. and Markosian; L. (1995) 'Customized Tools for Software Quality Assurance and Reengineering', *Proceedings of the Second Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 71-77.

World Wide Web Consortium (1995), 'Hypertext Markup Language Specification 3.0', Internet Draft, Laboratory of Computer Science, MIT, Boston, Massachusetts, 170 pp. (World Wide Web Consortium web site is http://www.w3.org).

Zoufaly, F., Araya, C., Sanabria, I. and Bendek, F. (1995) 'RESCUE : Legacy System Translator', *Proceedings of the Working Conference in Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 39-52.