Universal Report: A Universal Reverse Engineering Tool

Claude Tadonki
University of Geneva

Informatics Center, 24, avenue du Général Dufour, Geneva, Switzerland
claude.tadonki@unige.ch

July 29, 2003

Abstract

In this paper, we present a software for automatic source code documentation. The tool, so-called Universal Report, is written using C++ Builder environment. The main difference with other existing solutions is its universality in regard of the programming language of the input program. Moreover, one can document a set of routines written with different programming languages at a time. The Universal Report engine is based on heuristics that perform powerful pattern analysis with acceptable execution times. Each step is validated interactively by the user, but a completely automatic execution is possible. Output report is built and formatted at the user convenience, and can be in text, html, or latex based type. The tool provides several attractive features like multiple interface languages, automatic items extraction, spell checking, automatic code formatting, automatic glossary, multiple output languages, and more. The intuitive interface makes Universal Report the perfect tool for both new and experienced users.

Keywords: programming language, code documentation, html, latex, graph, routine.

1 Introduction

While or after writing an important set of codes, providing and maintaining the corresponding documentation is crucial and quiet helpful. Such a document helps to understand the program both in its global structure and its internal details. This is therefore useful for end-users to find their way in the program usage, and for the programming staff to easily maintain the source code. Without a documentation, one needs to be in touch with the authors of the program, and/or is forced to go into the code itself in order to get information from local comments [1]. This situation is seriously uncomfortable, and requires a permanent and repetitive effort.

One can write an independent documentation, means the content is related to the program at a given state. In this case, there is no dynamic link between items in the program and items in the documentation. Such a static documentation is difficult to maintain and is not updated as frequently as necessary.

An efficient way to easily get a detailed and up-to-date documentation is to build a high level document structure and let its items being instantiated automatically. This approach clearly requires the use of source code documentation tools. Assuming a well documented program, such a tool should be able to extract meaningful source code comments and global structure to create a complete and understanding external documentation. At this point, we also advice the reader to see [2] for a so-called literate programming. In our opinion, a source code documentation application should get as best as possible some of the following characteristics:

- easy to use with some steps being accomplished automatically
- it should be able to manage common information categories
• the user should be free of knowing things about the programming language behind the input program
• the output documentation should be parameterizable, both in the items to be displayed and also in the global layout preferences.
• a documentation session should result in a file recording all provided and extracted information.

A number of tools are available on the market and on the Internet. Table (Table 1) gives a list of some operational solutions (there are more) and the web address where they can be found.

<table>
<thead>
<tr>
<th>Product Name</th>
<th>Web URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal Report</td>
<td><a href="http://www.omegacomputer.com">www.omegacomputer.com</a></td>
</tr>
<tr>
<td>Doc-O-Matic</td>
<td><a href="http://www.doc-o-matic.com">www.doc-o-matic.com</a></td>
</tr>
<tr>
<td>TwinText</td>
<td><a href="http://www.ptlogica.com">www.ptlogica.com</a></td>
</tr>
<tr>
<td>Time2Help</td>
<td><a href="http://www.time2help.com">www.time2help.com</a></td>
</tr>
<tr>
<td>Doxygen</td>
<td><a href="http://www.stack.nl/dimitri/doxygen">www.stack.nl/dimitri/doxygen</a></td>
</tr>
<tr>
<td>DocJet</td>
<td><a href="http://www.talltree.com/docjet">www.talltree.com/docjet</a></td>
</tr>
<tr>
<td>Javadoc</td>
<td>java.sun.com/j2se/javadoc</td>
</tr>
<tr>
<td>AutoDOC</td>
<td><a href="http://www.oche.de/akupries/soft/autodoc">www.oche.de/akupries/soft/autodoc</a></td>
</tr>
<tr>
<td>CcDoc</td>
<td><a href="http://www.joelinoff.com/ccdoc">www.joelinoff.com/ccdoc</a></td>
</tr>
<tr>
<td>Cxref</td>
<td><a href="http://www.gedanken.demon.co.uk/cxref">www.gedanken.demon.co.uk/cxref</a></td>
</tr>
<tr>
<td>Doc++</td>
<td>docpp.sourceforge.net</td>
</tr>
<tr>
<td>Perceps</td>
<td>starship.python.net/tbryan/PERCEPS</td>
</tr>
<tr>
<td>ReThree-C++</td>
<td>students.cs.byu.edu/pbiggs/re3-cpp.html</td>
</tr>
<tr>
<td>RoboDoc</td>
<td><a href="http://www.xs4all.nl/rfsber/Robo/robodoc.html">www.xs4all.nl/rfsber/Robo/robodoc.html</a></td>
</tr>
<tr>
<td>ScanDoc</td>
<td><a href="http://www.sylvantech.com/falin/projects/scandoc/scandoc.html">www.sylvantech.com/falin/projects/scandoc/scandoc.html</a></td>
</tr>
<tr>
<td>DocBuilder</td>
<td><a href="http://www.gfai.de/produkte/docbuilder/e_index.htm">www.gfai.de/produkte/docbuilder/e_index.htm</a></td>
</tr>
<tr>
<td>VB Project Analyser</td>
<td><a href="http://www.codework.com/ProjectAnalyze/product.html">www.codework.com/ProjectAnalyze/product.html</a></td>
</tr>
</tbody>
</table>

Table 1: Some Existing Source Documentation Solutions

Most of existing solutions are valid for a predetermined class of programming languages, especially C and C++. This limitation, even if it helps to provide a more robust system, can quickly turn to a practical inconvenience.

In this paper, we present a solution, Universal Report, which does not suffer from programming language restriction, and provides several attractive and powerful features for a complete documenting task. The Universal Report engine is based on efficient and accurate heuristics, that perform the whole task in a syntactical way, without any connection with the native compiler of the input program. The output documentation is automatically generated from comments inside the codes and the structure of the program. The tool is parameterizable both in the behavior point of view and the type/quality of the outputs. Technically, we have build the software under C++ Builder environment[3], and Table (Table 1) displays quantitative information:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of files</td>
<td>66</td>
</tr>
<tr>
<td>Total number of lines</td>
<td>26820</td>
</tr>
<tr>
<td>Total Size (KB)</td>
<td>829</td>
</tr>
<tr>
<td>Total number of routines</td>
<td>624</td>
</tr>
<tr>
<td>Total number of classes</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 2: Quantitative information about Universal Report

The next section gives an overview of the application. The rest of the paper presents the tool in each of its fundamental aspects.
2 Overview and general steps

A standard session with Universal Report is composed with ten steps, that should be accomplished sequentially as shown in figure (Fig. 1).

The only independent and optional step is step 4, where general information about the program to be documented are required. The other steps are mandatory, and should be done in the order displayed in figure (Fig. 1), the result of a given step being the input of the next one. In order to guarantee the correctness of the final result (regarding the content validity), each step should be done carefully under the control of the user. However, the tool provides a feature called direct report, which allows the user to perform all steps in one time. This is particularly useful for new users who want to get familiar with what the tool can do, and also for any user who wants to generate a documentation with a minimum effort and time overhead. Note in that case, since the engine will make some assumptions (based on heuristics), the final output may contain some blanks or incorrect items, but with a very low probability.
We now give some technical explanations about the idea behind each step and related features.

### 3 Collecting files

There are typically two categories of files involved in a given program:

- **program files**: containing the source codes
- **data and header files**: containing data structure declarations

The files in each category are referenced by their names ending with a generic extension, which depends on the programming language. Since several directories may be concerned, the files collecting task requires the following inputs:

- the list of independent directories to be considered, with an indication whether we should consider recurse in subdirectories or not
- a list of file name extensions (e.g.: .c, .f, .bas, .pas, .m, .java, etc.)

Next, each directory is scanned (with its subdirectories or not) while looking for files whose the name matches one of the specified extension. This operation yields a list of files, that should be refined to remove undesired selections.
Figure 3: Directories selection

Figure 4: Files selection
4 Extracting routines

In order to produce a detailed report, it is important to get the program as a set of routines. Indeed, routine is the generic entity of a program. A documentation tool should be able to localize and extract routines from the input files. When concerned with a specific programming language, it is better to consider the underlying grammar and then derive an exact algorithm. Otherwise, heuristics should be considered for routines localization. The (syntactical) task is quiet simple when the programming language provides special keywords for routines declaration (Basic: sub, Fortran: subroutine, Matlab: function, Pascal: procedure, etc...). When such a keyword doesn’t exist (like in C, java, etc...), the pattern recognition task is slightly complicated. One reason of this is that routine declaration is sometimes syntactically close to some instruction statements. Another reason is functions are of various types, including user defined types.

Universal Report consider the following generic syntax:

\{(keyword, data_type) routine_name(routine_arguments)\}

In addition, there is a special symbol that the user can put in a comment line just before the routine, in order to ensure it will be captured. This issue is not mandatory, but it is the ultimate solution to force the heuristic succeed in resisting cases. The Universal Report engine consider each file of the collection and proceed as follows:

**Step 1:** The input file is scanned line by line, and each time a routine is detected, the corresponding pair (routine_name, first_line_number) is recorded.

**Step 2:** The user is invited to remove wrong selections. For this purpose, the original line is displayed.

**Step 3:** Assuming that the input file contains only the set of its routines, the routines extraction task is performed with the previous entry lines. Note that, when consider a very large set of codes, loading all routine bodies into the memory may result in some memory troubles (if the free memory space is insufficient). In that case, we can only keep on the entry line and the file name, and perform the extraction task only at the appropriate time.

![Figure 5: Routines extraction](image-url)
5 Extracting comments and general information

Comments in a program are the most important static items for its documentation. They explain how the program runs and behaves. They also provide indications about the structure of the program, and also non technical information (owner, date, institution, references, related links, etc...). In a given programming language, comments are edited with a special symbol at the beginning, or covered by special brackets (/* */ or { }). In order to be able to correctly extract comments in a given program, one needs the following information which depend on the programming language:

- the special symbol which introduces a comment line
- the specific position (if any) of the commenting symbol

For a given routine, the resulting block comment is built in one of the following way:

- all comment lines
- the first group of contiguous comment lines

We just analyze the input routine line by line, scanning each line to detect if it is a comment line or not, until the requirement of the previous selected option is fulfilled.

We also manage some information like

- routines author (extracted or given)
- routines date (extracted or given)
- routines input/output (extracted)
- routines size (calculated)

5.1 Extracting author information

The user can directly provide the authors name for each routine. However, this task can be done automatically under the following conditions:

- a complete list of authors has been yet provided (in the general information session)
- for a given routine, each author has at least on of its name written with the correct spelling

If the above conditions are satisfied, Universal Report succeeds in the task of authoring routines. The case of abbreviated name is also handled, but this often lead to confusions.

Another way of performing an automatic extraction is to consider keywords (like author, contributor, written by, etc...), which introduces the authors name. This option is useful when the list of authors is unknown in advance.

5.2 Extracting date information

By "date", we means the date of creation or the date of the lastest modification of the routine. As in the case of routine authors name, information on routines date are either provided directly by the user or automatically extracted with a pattern recognition algorithm. The algorithm tries to located any pattern that can be considered as a date in a standard format (mm:dd:yyyy, mm:dd:yy, mm/dd/yyyy, mm/dd/yy) and build the correct output information. Another option consider special keywords as previous.

5.3 Extracting input-output information

The set of input/output variables is more often available from the routine header. But, they are put together in the same list (except some case like matlab where output variables are separated from input variables). Output variables are often indicated by an additional syntax (like var in Pascal, and * for languages using pointers). However, these symbols only indicates potential output variables. In order to build the set of variables that are really used for outputs, we select
those who are really modified by the routine. For this purpose, we scan the routine and collect those variables that have been found at the left hand side of an assignment instruction.

Figure 6: Comments and input/output extraction

6 Routines calling graph

A program is composed of a set of routines that calls each other at running time. The underlying graph of this interaction provides a substantial information on the global structure of the program and how it works. In a formal point of view, we have a graph $G = (P, C)$, where $P$ is the set of routines, and $C$ is the set of arcs. There is an arc from $P_i$ to $P_j$ iff the routine $P_i$ calls the routine $P_j$. Note that this graph is built from static information, therefore an arc $(P_i, P_j)$ indicates a potential call. However, the trace of any execution of the program is a subgraph of $G$. In an interaction graph built by Universal Report, every arc is valuated with the line number of the corresponding call. This yields a weighted graph $G = (P, C, L)$, where $L_{ij}$ is the weight of the arc $(P_i, P_j)$. Next, we need to know the root routine, which can be directly specified by the user (generally the main routine), or automatically selected as the node with a null indegree and a
maximum outdegree. From this graph together with the root routine, we compute:

- a spanning tree (possible execution trace of the program),
- for each routine, the list of child (resp. parent) routines,
- a list of useless routines (those who are not connected to the root).

The spanning tree is built by the following recursive algorithm:

```python
Routine display_tree(RootNode, NodeSet)
NodeSet = NodeSet - {RootNode};
list = sort(neighborhood(RootNode));
n = length(list);
for i=1:n
    ThisNode = NodeSet(list(i));
    draw_arc(RootNode, ThisNode);
    NodeSet = NodeSet - {ThisNode};
endfor
for i=1:n
    ThisNode = NodeSet(list(i));
    routine display(ThisNode, NodeSet)
endfor
```

Figure (Figure 7) shows how the resulting spanning tree looks like.

![Figure 7: An example of Routines calling diagram in latex](image-url)

When the report type is latex, the output for the routines calling diagram is a script that will generate the figure through the latex compiler. Moreover, this diagram gives a static trace of the program execution. In case of complex program, the importance of this diagram is quiet perceptible. Universal Report also gives the possibility to generate a true graphic which can be save in standard format (.jpeg, .bmp, etc...). In case of html report, the graphic file is automatically generated and inserted in the corresponding frame. Figure (Figure 8) shows the interface for routines interaction analysis.
7 Selected features

7.1 Data structures reporting

Another important issue in a program documentation task concerns the user defined structures (type, structure, class) documentation. Related information can be found in the header files and also in some of the program files (local definitions). The task of extracting data type declarations is technically affordable, since these declarations are made with a specific syntax including precise keywords. The case of object-oriented language is more important, since such programs are strongly based on user defined structures (specialized tools like Javadoc or Doxygen do it correctly for Java and C++ respectively). Again, Universal Report uses a heuristic to extract structures and classes, and next generates the documentation.

7.2 Spell checking

Spell checking is a well known feature in text editor. In the case of Universal Report, this is done on user code comments. Our algorithm is quite simple and can be described as follows. Given a misspelled word $w$ (not available in the dictionary), $|w| = n$, a dictionary word $u$ is a suggestion for word $w$ if one of the following exclusive statements is true:

- $u \in \text{Out\_Two\_Char}(w)$
- $u \in \text{Out\_One\_Char}(w)$
- $\text{dif}(u, v) = 1$
- $u \in \text{Put\_One\_Joker}(w)$
- $u \in \text{Put\_Two\_Joker}(w)$
The function \texttt{Put\_One\_Joker}(w) (resp. \texttt{Put\_Two\_Joker}(w)) generates a set of word derived from \(w\) by inserting one (resp. two) joker character at all possible positions.

The function \texttt{Out\_One\_Char}(w) (resp. \texttt{Out\_Two\_Char}(w)) generates a set of word derived from \(w\) by removing one (resp. two) character at all possible positions.

The function \texttt{diff}(u, v) returns the number of positions with different characters in \(u\) and \(v\).

On a given word, we generate each of the previous pattern (taking care of symmetry) and then look at them in the dictionary using a dichotomic search. The algorithm can stop earlier if the number suggestions is reached. In practice, the algorithm provides a real time processing and the result is satisfactory.

### 7.3 Code formatting

Code formatting tries to improve the visual aspect of a given source code. The motivation is not only esthetic, but a well formatted code is more easier to read and understand, which is one of the main purpose of a documentation task. \textit{Universal Report} performs the following actions automatically:

- block statements indentation
- keywords coloring
- comments coloring
- instruction aeration by inserting spaces
- long lines wrapping
- lines numbering

In the html version, it also insert hyperlink in each routine call so that the corresponding routine can be visit from this point.

### 7.4 Code visualization

Another feature related to visual aspect of code is a tree like code display. This means the code is displayed like a tree, where each block statement (\texttt{if}, \texttt{else}, \texttt{while}, \texttt{do}, \texttt{repeat}, etc.) is viewed as a node. The layout of the routine is not affected by extra drawing items, but a tree organization is handle behind. The advantage is that the user can collapse and expand block statement (node) as desired in order to get a certain level of abstraction of the code. In particular, by collapsing all leaves, one obtains a prototype of the code, where only the structure and comments are displayed.

### 7.5 Text manipulation

\textit{Universal Report} provides a number of functionalities for text manipulation. These functionalities can act on a single or on all files involve in the documentation (those previously selected). With \textit{Universal Report}, the user can look for a set of word \(W\) in a set of file \(F\), where \(|W| \geq 1\) and \(|F| \geq 1\). In this case, \(F\) is either a specific file or the entire files of the program. The result of this query is a list of items of the form \((w, p, q)\), where \(w\) is a word in \(W\), \(p\) the number of occurrences found, and \(q\) the number of files where the word \(w\) was detected. After that, when the user click on the line of the item \((w, p, q)\), details are automatically displayed for this items. An analogue feature exist for find/replace query. In this case, the user also provide the corresponding set \(B\) such that \(|B| = |A|\). Once the query is executed, each occurrence of word in \(A\) is replaced by the corresponding word in \(B\), in all files in \(F\).
7.6 Glossary generator

Universal Report can generate a glossary for the set of user code comments. This task is done in three steps. First, all available words (with a user supplied range for word length) are extracted, this yields a starting list. Next, the list is edited by the user who adds and removes words at it convenience. Third, this final reference list is considered as the basis of the glossary. For a given entry word, the glossary establishes a link to all routines in which it appears in the main comment.

Figure 9: Glossary generator

8 Variables extractions and dependencies analysis

8.1 Variables extraction

Here, we are concerned with the task of extracting variables from routines. Our algorithm assumes that every variable of the routine appears at least one time in a left hand side of an assignment instruction. Therefore, the routine is explored instruction by instruction (mainly arithmetic ones and function calls), and each pattern that appears on the left hand side of an assignment is recorded as a variable (except common keywords). From the resulting list, structured variables are extracted by grouping items with the same prefix, which is a regular pattern ending with special symbol like . or ->. The prefix research is done from right to left in order to handled the case of depth structures. The user can choose to extract all variables, or only structured ones. Some statistics can be performed in order to count variables occurrences in different routines. This make sense only for global variables, or structured variables that are used with the same name in all routines of the program. Figure (Figure 10) displays the variables extraction interface.
8.2 Data dependence

This part is slightly marginal regarding the scope of Universal Report, since the semantic aspect of the input program is involved. However, this can also be considered as a part of a global documentation task, since it provides static (also syntaxic) information about the "atomic" structure of the program. The current task is to build the data dependence graph of a given routine. This data dependence graph is an oriented graph we denoted by $G = (V, D)$, where $V$ stands for the set of variables, and $D$ is the set of arcs expressing the data dependence relation. Two variables $v_i$ and $v_j$ are connected from $v_i$ to $v_j$ iff there is an instruction of the form $v_j = F(\cdots, v_i, \cdots)$, where $F$ is an arithmetic expression or a routine call. In case $F$ is a routine call, the analysis is done recursively. By this way, the dependence graph is applied in the whole program. For the current purpose, Universal Report extracts first the list of all variables in the selected routine. Next, the user selects a given variable and requests either a forward analysis or a backward analysis. In a forward analysis we use the graph $G$, whereas the reverse graph $G^{-1}$ is considered in a backward case. Whatever the case, the result is a spanning tree with the root being the selected variable.
9 Structure of the Report

The documentation generated by Universal Report can be in one of the following type

- html (*.htm)
- normal text (*.txt)
- formatted text (*.rtf)
- latex (*.tex)

The content of the report is built from information collected at previous steps. Depending on the selected output type, appropriate routines are called in order to produce a structured and easy-to-read report. The following items related to the input program are available in the report:

- The total number of files, routines, lines, and size.
- The list of files (name, location, number of lines)
- The list of routines (name, location, number of lines, author, date).
- The routines interaction graph
- The list of structured variables and their description
- The glossary of the set of all comments in the input program
- A complete description of each routine
  - name, location, size, arguments, author, date, comment, parent routines, child routines
  - routine body (formatted)
Figure 12: Contents and layout preferences

FORTRAN BLAS
FORTRAN BASIC LINEAR ALGEBRA SUBROUTINES
Reference Manual

By
Jack Dongarra al.

Document Generated by Universal Report www.omegacomputer.com

Figure 13: Latex main page
10 Concluding remarks and perspectives

We have presented a source code documentation tool called Universal report. The main advantage the tool is its universality. Indeed, Universal report is designed for any programming language that supports comments inside the code. The tool actually runs under Windows systems, but the corresponding version for other operating system like Linux is currently in progress.

References

