Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-2001-647

Observations on literate program structure

M.A. Guravage

Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-2001-647

Observations on literate program structure

M.A. Guravage

This report is based on a presentation held at EuroTeX 2001, Kerkrade, The Netherlands, 24 September 2001.

Division:	Information and Communication Technology
Issued:	January 2002
Classification of title:	Unclassified

Contents

(10 pages in total)

OBSERVATIONS ON LITERATE PROGRAM STRUCTURE

Michael A. Guravage

National Aerospace Laboratory (NLR)
guravage@nlr.nl, fax: +31 (0)20 511 32 10

abstract

In this paper we compare the structure of programs written in literate and traditional imperative programming styles. Considering literate programs as "webs" of interconnections, we model and analyze the patterns of connections particular to literate programs, and compare them to the call graph structures of traditional imperative programs. The application of structural graph metrics allow us to visualize the literate and call-graph structures. We then compare, contrast, and interpret the various graph metrics in light of each programming style.

Keywords: Software Engineering; Configuration Management; Process Management; Change Traceability

1 Introduction

A program's structure, whether a hierarchy of ideas as in literate programming or of function calls as in traditional programming, determines the shape and characterizes the functions and limitations of the code that is built upon it. If it is true, as Professor Knuth says, that a literate program is a "web" of interconnections, can the pattern of connections particular to literate programs be *modeled*, *analyzed*, and *interpreted*? But what is an appropriate representation for the structure of literate programs, and what measurements can we extract from our representation? How do we interpret what we measure, and can we compare the structure of literate programs to the call graph structure of traditional programs?

To begin to answer these questions, we start with a short review of the principles of literate programming; including a comparison between literate programming and traditional imperative programming styles. Next we introduce GraphXML - a graph description language in XML - which we use to model both literate and call graph structures. Next we introduce Royere - a graph visualization and graph metric package which allows us to see and measure various structural graph metrics such as tree depth and tree impurity. We end with a structure based comparison of literate and imperative programming styles.

2 Literate Programming Review

> "A complex piece of software consists of simple parts and simple relations between those parts; the programmer's task is to state those parts and those relationships, in whatever order is best for human comprehension." – D.E.K.

Literate programs are written with an emphasis on *exposition*. They are differentiated from programs written in other programming styles by three characteristics:

- A single literate source file produces both a compiled program and a nicely typeset document.
- The order of presentation is independent of the order of compilation.
- Cross-references, indices, and navigational hints are automatically generated.

The basic unit in a literate program is a *section* or *chunk*, and is analogous to a paragraph in prose. A segment has two parts: an informal specification expressed in natural language, and a formal specification expressed in code. Ideally, the description and code are balanced to convey the section's essence clearly, without distracting the reader with unnecessary detail.

Figure 1 shows a section of a literate source file and its typeset result. Notice that the result is actually typeset, that all the referenced sections have been assigned numbers, and that a cross reference appears at the bottom naming the section which referenced this section.

```
@ Now we scan the remaining arguments and try to open a file, if
possible.  The file is processed and its statistics are given.
We use a |do|~\dots~|while| loop because we should read from the
standard input if no file name is given.
@<Process...@>=
argc--;
do@+{
  @<If a file is given, try to open |*(++argv)|; |continue| if unsuccessful@>;
  @<Initialize pointers and counters@>;
  @<Scan file@>;
  @<Write statistics for file@>;
  @<Close file@>;
  @<Update grand totals@>; /* even if there is only one file */
}@+while (--argc>0);
```

ascii literate source code

§8 WC-SNIPPET CWEB OUTPUT 1

8. Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a **do** ... **while** loop because we should read from the standard input if no file name is given.

⟨ Process all the files 8 ⟩ ≡
```
argc−−;
do { ⟨ If a file is given, try to open ∗(++argv); continue if unsuccessful 10 ⟩;
  ⟨ Initialize pointers and counters 13 ⟩;
  ⟨ Scan file 15 ⟩;
  ⟨ Write statistics for file 17 ⟩;
  ⟨ Close file 11 ⟩;
  ⟨ Update grand totals 18 ⟩;      /∗ even if there is only one file ∗/
} while (−−argc > 0);
```
This code is used in section 5.

typeset literate source code

Figure 1 Literate Code Example

2.1 Compare and Contrast

The programs chosen for this study were the thirteen demonstration programs and the eighteen library modules written in cweb[1] that together comprise *The Stanford GraphBase(SGB)* – A Platform for Combinatorial Computing[2]. From each literate source file we created a typeset document from which we extracted the literate structure hierarchy, and a compiled executable from which we extracted the call graph hierarchy.

Though both the literate and call graph hierarchies were extracted from the same body of code, an initial comparison of revealed fundamental differences between them: The unit of organization in a literate program is a *section* – analogous to a paragraph in prose; The unit of organization

in a traditional program is a *subroutine*. The organizing principle in a literate program is clarity of *exposition*, The organizing principle in a traditional program is *execution*, i.e. a hierarchy of subroutine calls. The scope of a literate program written in **cweb** is a *single file*; the scope of a traditional program is an *executable*. The main reason for these differences is that literate programming encourages developing a hierarchy of ideas, whereas traditional programs construct hierarchies of subroutines.

3 GraphXML

The structure of literate programs and the call graphs of traditional programs are represented as *directed acyclic graphs* (DAGs). To model these graphs we chose *GraphXML* – a graph description language in XML – whose goal is to provide a general interchange format for graph drawing and visualization systems. Figure 2 shows a *GraphXML* example graph composed of two nodes connected by a single edge.

```
<?xml version="1.0"?>
<!DOCTYPE GraphXML SYSTEM "file:GraphXML.dtd">
<GraphXML>
  <graph>
    <node name="first"/>
    <node name="second"/>
    <edge source="first"  target="second"/>
 </graph>
</GraphXML>
```

Figure 2 graphXML example

The literate program structure graphs were extracted from auxiliary files created by **cweave** in the process of typesetting literate source code. As **cweave** reads the literate source code and writes the corresponding typeset document in TEX, it compiles a list of sections where each section is followed by a list of sections that include it. We wrote a small program in *ICON*[3] to extract the necessary information from an auxilary file and rewrite the result in graphXML.

The call structure graphs were extracted from profiling information generated by *gprof*. After extracting and rearranging the the C code from the literate source with *ctangle*, the C code was compiled with GCC with profiling enabled. Running the resulting executables generated profiling information - including the call graphs. We wrote another small *ICON* program to extract the call graph and timing information from each profile, and rewrite the result in graphXML. Figure 3 shows the various processing steps necessary to create the literate and call graph structure graphs.

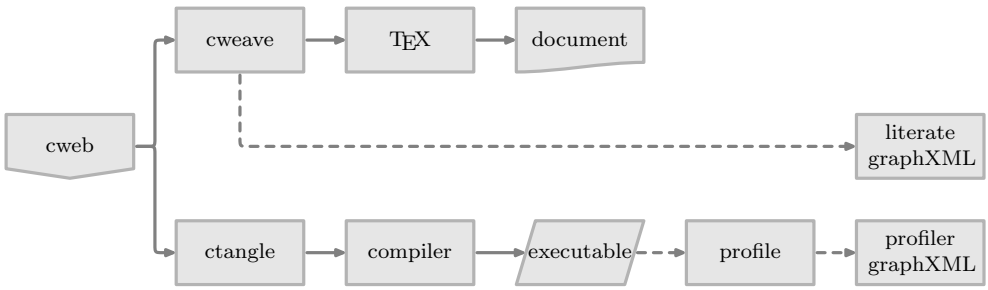


Figure 3 Literate Tool Chain

4 Structure Metrics

A *metric* is a measure of some aspect of the graph that is associated its nodes or edges. A metric can consider the graph's structure, include domain-specific information, or a combination of both. A metric is *structure-based* if it only uses information about the structure of the graph[4–5].

Royere is a graph visualization and metric program developed at CWI in Amsterdam. We choose it both for its built-in metrics and its ability to read several graph description languages – including graphXML. From *Royere's* repertoire of metrics, we choose the following four.

- Tree Depth
- Tree Impurity
- Recursive Citation
- Spreading Activation

Tree Depth cite[Fenton96] is a metric that counts how many nodes lie along the longest path from the root to a leaf. We expected that call graphs would be deeper than their literate counterparts. The average literate depth is 3.23, whereas the average call graph depth is 4.85 – see Figure 4.

The relatively deeper call graph structure of traditional programs reflects their fine-grain decomposition of functionality into subroutines. In contrast, literate programming tends to deemphasize low level functionality in favor of espousing what a piece of code is doing and why.

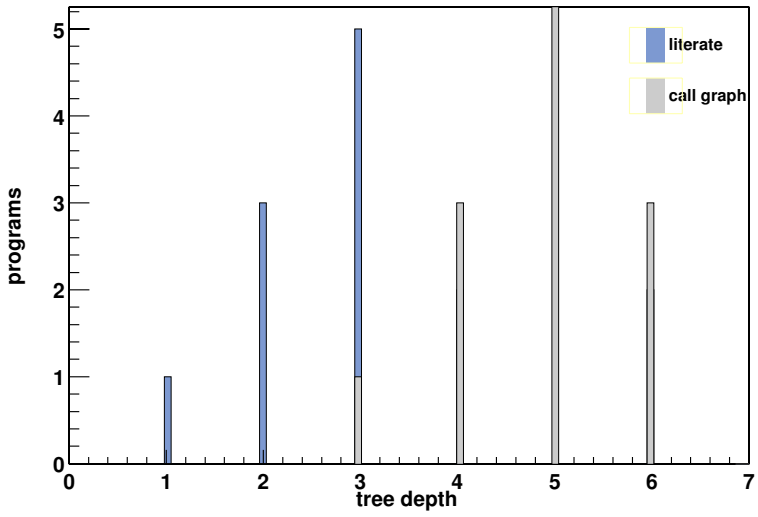


Figure 4 Tree Depth

Tree Impurity measures the degree to which a graph's structure deviates from being a pure tree, i.e. no cycles. Its value ranges between zero and one, and is considered inversely proportional to the quality of the structure's design.

A graph's *Tree Impurity* is a function of its nodes and edges as expressed in following equation:

$$m(G) = \frac{number\ of\ edges\ more\ than\ the\ spanning\ tree}{maximal\ number\ of\ edges\ more\ than\ the\ spanning\ tree} = \frac{2(e-n+1)}{(n-1)(n-2)}$$

It appears that literate organizations are very clean compared to the call graphs of their corresponding imperative implementations. The average literate impurity is 0.00246, whereas the average call graph impurity is 0.06292 – see Figure 5.

Traditional imperative programming make heavy reuse of low level functions. The SGB ladders program, for example, repeatedly calls low level allocation and input/output routines – all of which raises the call graph's tree impurity. The exposition and refinement of ideas in a literate programming style induces fewer graph cycles.

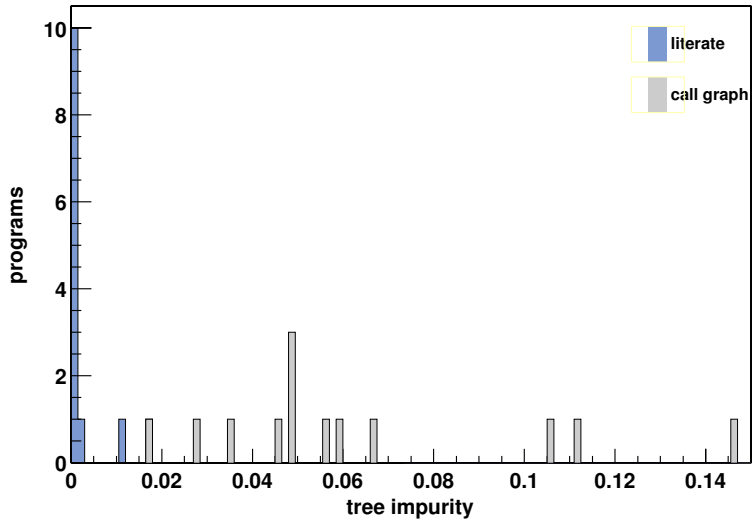


Figure 5 Tree Impurity

5 Word Ladders

The next two metrics, *Recursive Citation* and *Spreading Activation*, differ from the previous metrics in that they are calculated for each node in a graph. We can see how these metrics are distributed over a graph by mapping their values to its nodes and edges. Assigning a visual attribute, e.g. color, brightness, color saturation, or line width, that reflects a metric value consists of two steps[5]:

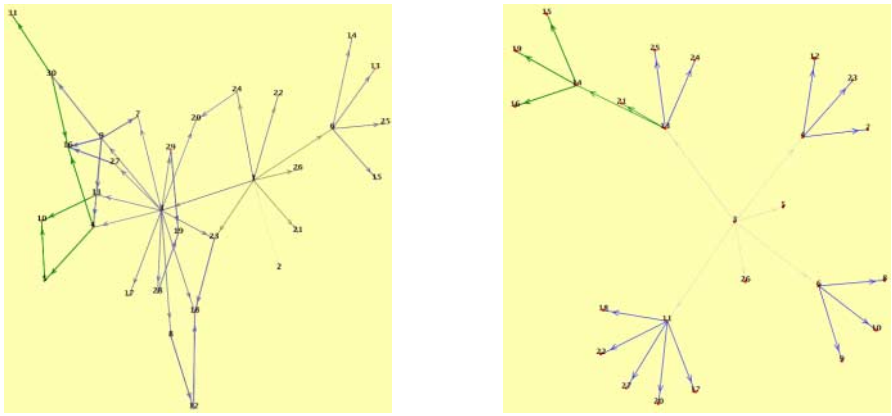
1. Assign an abstract value, usually between zero and one, to each displayable element based on the element's metric value. We refer to this abstract value as the emphasis of the element, and we refer to this mapping as the emphasis mapping.

2. Map the emphasis to a visual attribute. We refer to this mapping as the attribute mapping.

The metrics are extracted from the *Stanford GraphBase* example program named *ladders* which first produces a graph of five-letter words, and then uses Dijkstra's algorithm to find the shortest path between a pair of words solicited from the user.

5.1 Recursive Citation

Recursive Citation is a measure of how many times a node is referenced from above. This measure reveals the most heavily referenced nodes.



imperative literate

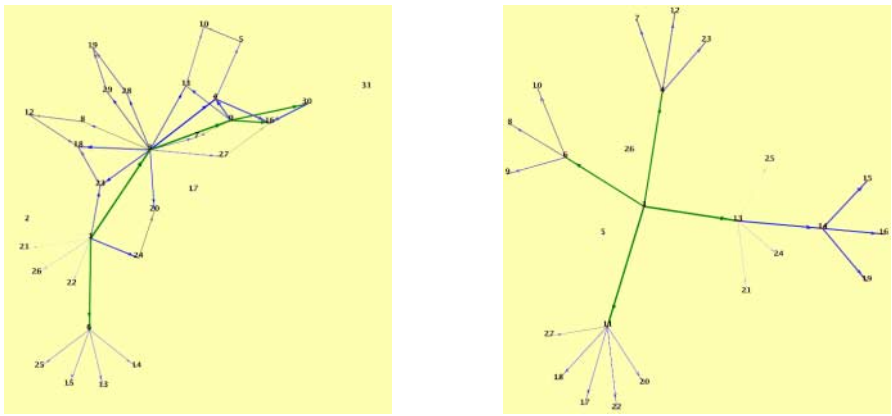
Figure 6 Recursive Citation

The most heavily referenced nodes in the literate graph are those that take the start and goal words and apply Dijkstra's algorithm to find the shortest path between the two. The main abstract divisions of input, process, and output are clearly identified – see Figure 6.

The most heavily referenced nodes in the call graph are those concerned with low level routines data manipulation and validation. While the metric correctly reveals the most heavily trodden path through the code, the emphasis on details obscures the program's purpose.

5.2 Spreading Activation

Spreading Activation is a measure of global connectivity. The value for a node is obtained by summing the contributions over the set of all its neighbors $n_1, \ldots, n_q (q \geq 1)$. This measure reveals a graph's *spine* or *center of mass*.



imperative literate

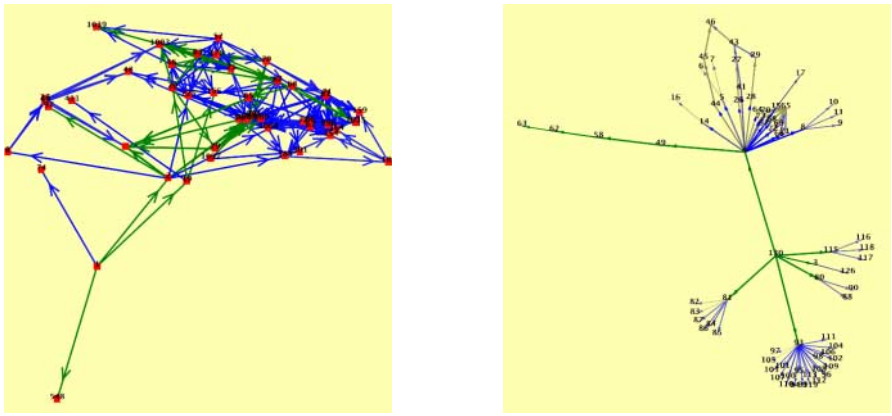
Figure 7 Spreading Activation

It is readily apparent that the spine of the literate program follows the line of: scan the command line options, setup the graph of words, prompt for start and goal words, and find a minimal ladder from start to goal – see Figure 7.

The spine of the call graph structure runs along the low level GraphBase data manipulation and checksumming routines.

6 A Real Live Example

This is a visualization of the Taxiway Collision Monitor (TCM) employing the *Spreading Activation* metric. The literate graph has 124 nodes and 134 edges. The call graph has 851 nodes and 1493 edges.



imperative literate

Figure 8 Taxiway Colision Monitor

The literate TCM program follows a line through the various C++ class constructors, and then along a path to the conflict rules – see Figure 8. The TCM is a class library. The metric correctly identified the most important conceptual parts of the TCM.

The call graph structure immediately dives into the depths of the Standard Template Library (STL). This is a consequence of profiling the entire TCM executable – including several object oriented libraries. The results would be more useful if it were possible to easily filter out unwanted functions calls.

7 Conclusions

Comparing the structure graphs of representative examples of literate and traditional imperative programming styles did not yield any new revelations. What the comparison did achieve is to quantify what we already suspected:

1. The fundamental difference between literate programs and their traditional imperative counterparts is that the former are hierarchies of ideas *(exposition)*, while the latter are hierarchies of subroutine calls *(execution)*.

2. The literate hierarchies appear routinely to be cleaner and less deep then the corresponding call graph hierarchies.

3. The metrics for traditional imperative programs would be enhanced by the addition of domain specific information such as: execution time, execution counts, etc.

4. Lots of empirical evidence is needed to make sense of these metrics in practice.

Instead of highlighting their differences, it is more profitable to view the structures of literate programs and their traditional imperative counterparts as complimentary; both offer very different views of the very same body of code. One would peruse the literate structure to discover the purpose and rationale behind a piece of code. In contrast, a call graph structure is a better source of information when the goal is making performance enhancements based on profiling data.

8 Future Work

The current study concentrated on structure metrics only – using the Royere metrics without modifications. The next logical extension of the current work is to augment the structure metrics with domain specific profiling information, and to apply the metrics to a broader collection of code to get a better idea of how they can be applied in practice. Several specific proposals are:

1. Metrics like *tree impurity* should be calculated for each subtree in the graphs.
2. The **gprof** profiler generates more data than we currently use. Information about execution time and execution counts could become edge attributes used to reveal the most often used or most expensive portions of a program.
3. Empirical evidence needs to be gathered to know how to interpret structure metrics together with other software metrics.
4. The **noweb** literate programming system can process multiple literate source files. It would be interesting to investigate larger literate systems comprised of many individual literate modules, e.g. libraries.

References

1 Knuth, D. E. and Levy, S. (1994). *The CWEB system of structured documentation*. Addison-Wesley.

2 Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press.

3 Griswold, R. E. and Griswold, M. T. (1990). *The Icon Programming Language*. Prentice-Hall.

4 Herman, I. and Marshall, M. (2000). GraphXML – an XML based graph interchange format. Technical Report INS-R0009, CWI.

5 Marshall, M. (2001). *Methods and Tools for the Visualization and Navigation of Graphs*. PhD thesis, University of Bordeaux.