XOgastan: XML-Oriented gcc AST Analysis and Transformations

G. Antoniol*, M. Di Penta*, G. Masone**, U. Villano**
antoniol@ieee.org dipenta@unisannio.it lucas75it@yahoo.it villano@unisannio.it

* RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano 82100 Benevento, Italy

**Department of Engineering, University of Sannio
Palazzo Bosco Lucarelli, Piazza Roma 82100 Benevento, Italy

Abstract

Software maintenance, program analysis and transformation tools almost always rely on static source code analysis as the first and fundamental step to gather information. In the past, two different strategies have been adopted to develop tool suites. There are tools encompassing or implementing the source parse step, where the parser is internal to the toolkit, developed and maintained with it. A different approach builds tools on the top of external, already available, components such as compilers that output the abstract syntax tree, or make it available via an API.

This paper presents an approach and a tool, XOgastan, developed exploiting the gcc/g++ ability to save a representation of the intermediate abstract syntax tree into a file. XOgastan translates the gcc/g++ format into a Graph eXchange Language representation, thus taking advantage of the high number of currently available XML tools for the subsequent analysis phases. Here the tool is illustrated and its design is discussed, showing its architecture and the main implementation choices.

Keywords: source code analysis tools, gcc, XML, GXL

1. Introduction

The software maintenance, the program comprehension, the reverse engineering community and, more generally, people involved in software evolution have always struggled to analyze source code. Activities such as the extraction of abstract representations, the extraction of metrics or source code tranformations rely on a preliminary, indispensable step, devoted to build an Abstract Syntax Tree (AST) of the source code itself.

Several types of languages/toolkits for source code analysis and transformation have been developed during the last decade. Some of them provide a powerful language oriented to program comprehension and transformation, such as the *Design Maintenance Systems* (DMS) produced by Semantic Designs Inc. [1], the *TXL programming language* produced by TXL Software Research Inc. [2, 3], *Refine* [4] produced by Reasoning Systems Inc., and *FermaT* by Martin Ward [5]. These tools have powerful analysis capabilities, compared to standard development environments, in that they provide pattern-matching languages and a way to query and transform the AST produced by a parser. Other tools, such as *Unravel* [6], *CANTO/ART* [7] or *Datrix* [8] are more oriented towards static source code analysis and metrics computation, than to source code transformation.

Very promising technologies, such as program slicing [9] and impact analysis [10] rely on the extraction from source code of information and representations such as the control flow graph and def/use of variables. These technologies have been sometimes integrated in commercial environments, such as *GrammaTech CodeSurfer*, *Semantic Designs Source Code Browser*, *Codecrawler*, *Rigi* and many others. In other words, these use compiler-like tools, included in the environment, to extract all the required information. These tools have very powerful and well-known capabilities, though they have been often designed to solve only particular categories of problems. For example, *CodeSurfer* [11] is particularly indicated for point-to analysis and slicing, *Rigi* [12] was conceived as a visual tool to help program comprehension and reverse engineering, *Codecrawler* [13] is a language independent reverse engineering tool integrated with metrics computation and large software system visualization capabilities, *Semantic*

Designs Source Code Browser [14] helps program comprehension by allowing navigation of documentation extracted from source code and of hyperlinked Java code.

Two basic strategies are common to all the above mentioned tools, as well as to other code analysis tools: either the toolkits provide their own parsers (e.g., *DMS*, *Unravel*, *Datrix*, *TXL*, *Rigi*), or they rely upon an external suite of tools (e.g., *CANTO*, *ART*). A notable exception is *Refine*, which adopts a hybrid solution for the C language: the preprocessor is an external component, while the grammar and the parsers are internal to the *Software Refinery*, even if based on *Lex* and *Yacc*.

This second strategy aims at decoupling the analysis from the language parsing problem, and in the long run it is also likely to be more robust, as programming languages evolve and new features are added. Both strategies have pros and cons. For example, if a custom parser is used, a software system could be analyzed even if some components are missing. On the other hand, tedious work is needed to cover all the language features. An external parser speeds up the development of software analysis tools, and parser maintenance is not required as new language features are added. On the other hand, if an external parser is used, there is the need of an intermediate representation or an API to access the compiler internal structures.

IBM made available a component named *Montana* [15] for the *C++ Visual Age*^{TM} suite. This allows the access to the C++ compiler information at the different compilation stages via an API. It would be therefore easy to develop C++ code analysis on top of *Montana*. Unfortunately, the IBM C++/*Montana* interface has never been made available on systems other than Windows NT, even if there were rumors of a Linux release. On the other hand, starting from version 2.9, the GNU C Compiler (*gcc*) development team added a functionality to the *gcc/g++* compiler: a switch to store in a file the ASCII representation of the AST for each compilation unit (i.e., for each file). This is a particularly important point, as GNU Compiler back-ends are available for almost any available computing system.

This paper proposes an approach and presents a tool, XOgastan, to exploit the possibilities offered by the new *gcc/g++* interfaces that allow to save the AST representation.

Other projects aim to exploit *gcc* AST and to represent it in XML. Among them, it is worth mentioning the *CPPX* project [16], *gccXfront* [17], the *srcML*-based fact extractor proposed in [18] and GCCXML [19]. Unlike these tools, XOgastan design is based on the key choice of applying no patch to the *gcc/g++* compiler source code. In other words, XOgastan takes as input the *gcc/g++* output *as it is*, without any modification to the *gcc* source distribution. The decision relies on the following observation. At the time the XOgastan project was started, the latest available *gcc* was one belonging to the 2.9 releases. Meanwhile, at that time, the *gcc* suite was undergoing a major redesign and the long awaited new *gcc* 3.0 was not yet available. Since the 3.0 release was a major compiler redesign, it was also likely that the compiler organization, flags, and format could have been changed. Therefore, we decided that the decoupling between compiler and tool was fundamental: by the way, this choice gives us other advantages, such as the ability to filter out any unnecessary information. Thus we conceived and implemented a filter that maps *gcc/g++* output to the widely adopted *Graph eXchange Language* (GXL) representation [20]. GXL is an XML based graph representation common to several software engineering tools, such as *Rigi*, *Columbus/CAN* [21], *Source Navigator*, *GUPRO*, the *Nokia source code analyzers* and many others (see *http://www.gupro.de/GXL/tools/tools.html*).

XML and GXL are *per se* useful for several reasons. XML is *de facto* a new emerging standard in information representation and exchange. GXL is widely adopted in the software maintenance and evolution community. Furthermore, XML allows to exploit available parsers (e.g., *Xerces*) and transformation tools (e.g., *Xalan*) to build source code analyzers and implement pretty printing or source code transformations.

This paper is organized as follows: Section 2 describes the XOgastan architecture, detailing the source code analysis process, the tool internal representation, the analysis capabilities and the types of output produced. Section 3 discusses the technology, design and implementation choices made in XOgastan development, analyzing their strength and weakness. Our conclusions and work-in-progress are finally outlined.

2. XOgastan Architecture

This section describes the overall design architecture of XOgastan, and also gives details on some important implementation issues. The XOgastan analysis process can be described as shown in Figure 1, and it is made up of the following phases:

1. The AST is dumped by *gcc*, compiling the source file (`foo.c` in our example) with the option `-fdump-translation-unit`. It is worth noting that detailed AST information is only present in the dump produced by the C++ compiler (*g++*), even for C programs, while the *gcc* output is almost useless for AST analysis. From here onwards, by *gcc* dump we mean the dump produced by *g++*;

2. The dump produced by *gcc* (file `foo.c.tu`) is translated in the GXL format by means of a *Perl* script supplied with XOgastan, `gcc2gxl`; and

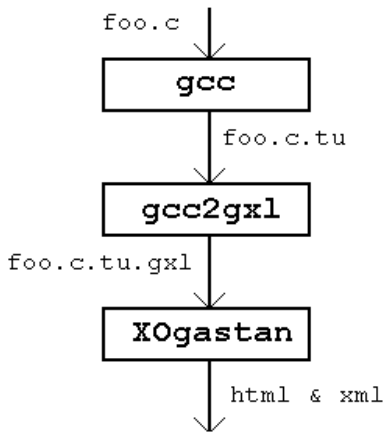

Figure 1. XOgastan analysis process.

3. Finally, the GXL output is analyzed by XOgastan, producing outputs in HTML, XML and other formats.

Overall, XOgastan has an hybrid pipe/filters and object-oriented architecture. In particular, the sequence of pipes is represented in Figure 1, while the package structure of the last filter is shown in Figure 7.

Point 1. needs some clarification. Since the *g++* AST is used, the tool is mainly to be used to analyze C++ code. However, it is also suitable for C code analysis, provided that the code is adequately pre-processed to avoid C code that is not C++ compliant. For example, C identifiers corresponding to C++ reserved words (e.g., a declaration such as `int class;`) are to be consistently replaced (the latter declaration should become `int _class;`).

2.1 From gcc AST dump to XML

XOgastan is an XML-oriented application, in that it performs analyses/transformations of an XML input, producing an XML output. On the other hand, the *gcc* AST dump (i.e., the XOgastan input) is not represented in XML. Hence, the first step is to obtain its XML representation.

A widely adopted XML-based representation for ASTs (and, more in general, for graphs) is the already mentioned GXL format. Basically, the purpose of GXL is to allow the exchange of graphs between different tools, such as those performing program analyses. As said before, the transformation from *gcc* dump to GXL is carried out by a *Perl* script. This script relies on a transformation map, supplied in a file named `g2x.map`. The AST of *gcc* can contain several different node types. Every node is characterized by a code (descriptive of its purpose), a list of attributes, and a list of possible linked nodes. The file `g2x.map` describes all the nodes that an AST may contain (also C++ and Pascal ones); for each type of node, a set of translation rules is specified. A translation rule transforms information contained in the `foo.c.tu` file in a GXL element. The process can be readily understood by examining the following example:

1. To represent a function declaration, *gcc* uses a special node, whose code is *function_decl*. This node contains information about the status of the function declaration: static or extern memory class, the name of the source file where it is declared, the line number in the source file where the declaration is located. Moreover, this node is linked to the node containing the function name (an *identifier_node* node), to the first node of the body (*compound_stmt* node), and to the next declaration in the same scope (this may be any type of declaration node).

2. The following lines are an example of the information dumped by *gcc* for a *function_decl* node:

```
@15     function_decl
name: @29     mngl: @30
type: @31     srcp: div.c:101
chan: @32     args: @33
static        body: @34
```

   In the above example, 15 is the index (unique in the dumped unit file) of the *function_decl*, 29 is the index of the node containing the function name, etc.

3. Some of the translation rules for *function_decl* nodes are shown in Figure 2. Rules may have two different formats. The interested reader is referred for details to the on-line documentation of the tool [22].

4. At the end of the translation process, the AST (in this case, the *function_decl* node) will be represented in XML as shown in Figure 3.

The transformation rules contained in the file `g2x.map` were written after gaining insight on *gcc* AST by studying the *gcc* functions devoted to produce/handle the AST. After a thorough examination of the source code and of the available documentation, a comprehensive set of translation rules was produced. In particular, the *gcc* AST structure (and therefore the translation rules) was deduced by analyzing the *gcc* source files `tree.def`, `tree.h`, `c-common.def`, `cptree.def`, `cptree.h`, `dump.h`, `dump.c` and `cpdump.c`.

2.2 XOgastan AST internal representation

XOgastan is written in C++. The AST is represented using the hierarchy of classes NAST (New Abstract Syntax Tree). The NAST hierarchy is similar to the one proposed

```
case FUNCTION_DECL:
name:*%<edge from="index" to="*"><type xlink:href="gccast.xml#name"/></edge>
type:*%<edge from="index" to="*"><type xlink:href="gccast.xml#type"/></edge>
scpe:*%<edge from="index" to="*"><type xlink:href="gccast.xml#scope"/></edge>
srcf:*%<attr name="source_file"><string>*</string></attr>
srcl:*%<attr name="source_line"><int>*</int></attr>
artificial %<attr name="flag"><string>artificial</string></attr>
chan:*%<edge from="index" to="*"><type xlink:href="gccast.xml#next_decl"/></edge>
args:*%<edge from="index" to="*"><type xlink:href="gccast.xml#arguments"/></edge>
undefined  %<attr name="flag"><string>undefined</string></attr>
extern %<attr name="flag"><string>extern</string></attr>
static %<attr name="flag"><string>static</string></attr>
body:* %<edge from="index" to="*"><type xlink:href="gccast.xml#body"/></edge>
fn:*%<edge from="index" to="*"><type xlink:href="gccast.xml#body"/></edge>
```

Figure 2. `function_decl` **translation rules.**

```
<node id="15">
   <type xlink:href="gccast.xml#function_decl"/>
   <attr name="source_file"><string>div.c</string></attr>
   <attr name="source_line"><int>101</int></attr>
   <attr name="flag"><string>static</string></attr>
</node>

<edge from="15" to="29"><type xlink:href="gccast.xml#name"/></edge>
<edge from="15" to="31"><type xlink:href="gccast.xml#type"/></edge>
<edge from="15" to="32"><type xlink:href="gccast.xml#next_decl"/></edge>
<edge from="15" to="33"><type xlink:href="gccast.xml#arguments"/></edge>
<edge from="15" to="34"><type xlink:href="gccast.xml#body"/></edge>
```

Figure 3. Result produced by the translation.

in [23], to the *JavaCC* AST [24] and to the AST object model proposed in [25]. The NAST is fairly different from the *gcc* AST[1], in that some parts of the AST are not present at all into the NAST. In particular, the NAST is a three-level hierarchy of nodes:

1. The first level of the NAST is the *root node*;
2. The second level is composed of the set of different language constructs: declarations, constants, expressions, statements, etc., and
3. The third level (i.e., the NAST leaves), contains all the elements making up a C program, e.g., identifier, variable declaration, predefined type, literal, call expression, for statement, etc.

When XOgastan reads a node from the GXL input file, it parses the code of the node and creates an object of a leaf class. For example:

- For a *var_decl* node, it creates an object *var_decl_n*;
- For an *integer_type* node, it creates an object i*nteger_type_n*;
- For a *real_cst* node, it creates an object *real_cst_n*;
- For a *call_expr* node, it creates an object *othr_expr_n*;
- For a *for_stmt* node, it creates an object *for_stmt_n*;
- For an *identifier_node* node, it creates an object *identifier_n*; and
- For a *start_catch_stmt* node, it creates an object *othr_stmt_n*.

2.3 XOgastan analysis capabilities

The analysis performed by XOgastan is function-oriented, in that XOgastan searches the NAST for function declarations. For each function declaration, it performs further analysis, as follows:

[1] In the following, the acronym AST is used to refer to the "original" *gcc* AST, and NAST to refer to the XML-based XOgastan AST.

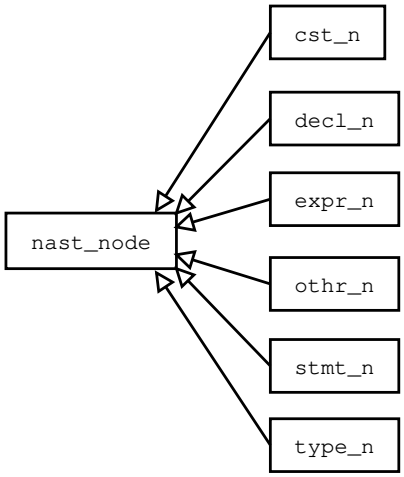

Figure 4. The first two levels of the NAST.

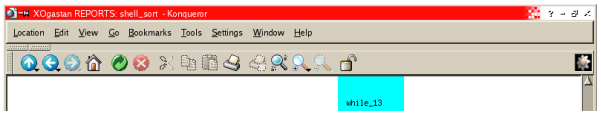

Figure 5. XOgastan HTML Output: Control Flow Graph.

- Gets the name, the type returned, the parameter list of the function;
- Gets some information about the statements in the function body: produces statistics of the statements used, builds a control flow graph, etc.
- Builds the list of declarations in the function body: variable declarations, typedef declarations, etc.
- Produces statistics about number of expressions and operators used;
- Builds a list of the variables used inside expressions: variables with local scope, variables with global scope, parameters; and
- Builds a call-graph of the function called.

XOgastan also produces statistical information about the given NAST (total number of nodes, frequency of a node, etc.). Analyses are performed using a *visitor* design pattern [26] (see Section 3.4 for details).

The analyses that can be performed using XOgastan output are out of the scope of this paper. In fact, it is worth pointing out that XOgastan has not been designed to obtain a strong source analyzer, but simply to interface a strong parser (the *gcc* one) with other analysis tools/plugins relying on XML representation. Exploiting directly compiler analysis capabilities is theoretically feasible, but in practice turns out to be very difficult (see [27]).

2.4 XOgastan output

The output produced by XOgastan is available in several formats:

- HTML pages, containing browseable Control Flow Graphs (CFGs) and statistics produced analyzing the NAST (examples are shown in Figures 5 and 6);
- XML files, that can be read by other XML-oriented applications (see Section 3.5) or browsed using an XML browser;
- DOT representations of body graphs, CFGs and call graphs, to be visualized using the *Dotty* tool [28].
- ASCII representations of the CFG; and
- Graphic Interchange Format (GIF) plots of the CFG.

2.5 XOgastan Design

XOgastan is composed of the following packages:

- The *NAST Factory*, which loads the NAST from the GXL generated by the `gcc2gxl` utility, and creates the NAST internal representation described below;
- The *NAST internal representation*, composed of classes for representing the data structure of the NAST (see Sections 2.2 and 3.3);
- The *visitor package*, composed of an abstract visitor class and some *concrete visitor* classes for implementing the different analyses performed on the NAST. The analysis features currently implemented include statistics about:
  1. Declared functions;
  2. Call graph;

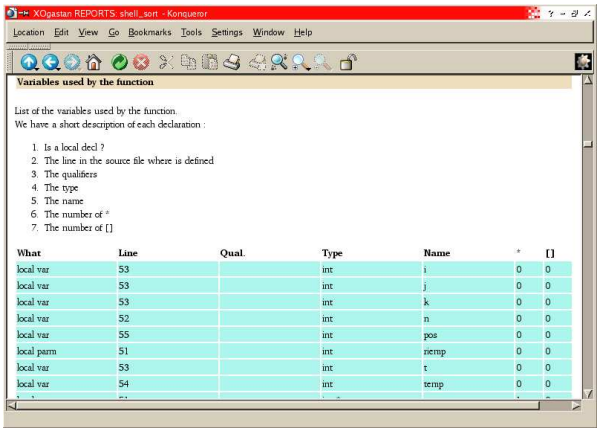

Figure 6. XOgastan HTML Output: Uses of Variables.

3. Used variables;
4. Expressions contained inside functions;
5. Statements contained inside functions; and
6. General NAST statistics.

- The *data package*, containing the data structure representing the results obtained from the different analyses performed by the visitors;
- The *HTMLWriter package*, composed of a hierarchy of classes for producing the HTML output. In particular, the hierarchy is composed of a base class for generating the main elements of the output pages, classes for generating the statistics pages of the analyzed functions and to perform queries on the AST; and
- The *XML Manager package*, which generates different XOgastan XML outputs. This package is composed of an XMLBuilder, i.e. a *builder* design pattern [26] that isolates the internal NAST representation from the output generation, delegated to the *concrete builder* classes.

The UML package diagram of XOgastan is shown in Figure 7.

3. Lessons Learned

This section discusses the design and technology choices and the features of the main XOgastan components, analyzing their strength and weakness and considering possible improvements and evolutions.

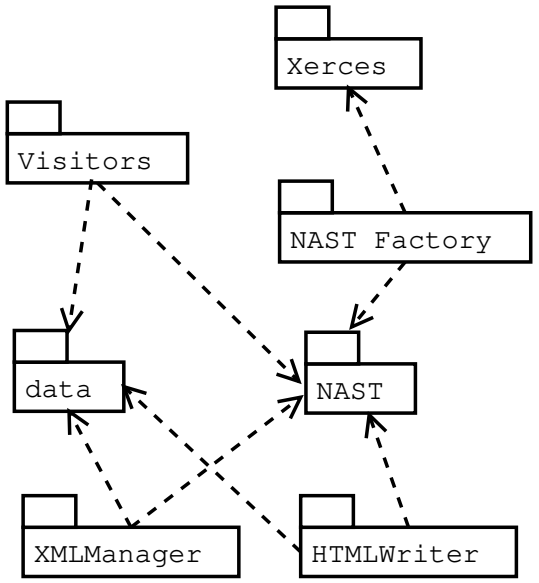

Figure 7. XOgastan UML Package Diagram.

3.1 XOgastan parsing capabilities

As explained before, XOgastan has no parsing capability. Instead, it relies on the AST produced by the *gcc* compiler. In the Authors' opinion, this is a strong point of the tool. Instead of implementing a parser, source code analysis and transformation tools should rely on what compiler technology makes readily available. In fact, it is widely recognized that compiler technology is by far more mature than technology of source code analysis tools. Moreover, compilers evolve as programming languages evolve: C compilers incorporate all the language features proposed by new standards (e.g., ANSI 99) or by dialects (e.g., GNU, Microsoft, Borland, etc.).

What developers of analysis tools should do is simply to exploit analysis capabilities of compilers, relying, where possible (as in the case of *gcc*) on the possibility of directly dumping the AST, or to modify the compiler source code where available (again *gcc*, or the *javac* Java compiler, for example).

However, relying on the analysis performed by compilers has also some disadvantages:

- It is widely recognized that the ASTs produced by compilers are not the most suitable for code analysis [29]. The compiler goal is to translate a high-level language into assembly code: the analyses performed are usually limited to what is strictly necessary to perform this task (even if most current compilers perform lot of static analyses on source code). On the other hand, the compiler AST may contain lot of details that

are indeed useful for compiling, but not for analysis purposes. As is done in XOgastan, these details should be pruned to avoid input ASTs full of useless details; and

- Exploiting the compiler AST is not trivial. Often compilers do not allow the AST dump, and hacking their source code (when available) may be a difficult task.

Finally, if compilers for other programming languages should AST dump available, the same approach can be extended to them. This could be a possible way to handle the 500-Language problem described in [30]. However, at the time of writing, we do not know of compilers, other than *gcc*, making the AST available by means of a command line switch or something similar.

3.2 From gcc to GXL

The translation of the *gcc* AST dump in a GXL format is a fundamental point in the XOgastan architecture, for the following reasons:

- GXL is XML, and therefore XOgastan can rely on a *Simple API for XML* (SAX) parser such as *Xerces* for parsing its input;
- As the compiler evolves, its AST representation may change; the use of an intermediate representation avoids that this could eventually affect the whole tool structure. If the information dumped by the compiler changes, one simply needs to update the translation map `g2x.map` and the `gcc2GXL` script. If compared to tools where the AST is dumped by patching the compiler, our approach allows an easier and cheaper upgrade. In fact, re-patching a new compiler version could be difficult, especially if its internal structure has been radically changed/refactored. On the contrary, changes on the translation map are usually straightforward, taking also into account that the syntactic structure of the source language (C/C++) is fairly stable;
- Performing analyses from ASTs produced by other compilers, or even extending the tool to further programming languages, turns out to be relatively simple;
- GXL tends to be widely adopted in the source code analysis, program comprehension and reverse engineering community: producing intermediate results and outputs in this format allows tool interoperability. This, in the Authors' opinion, should constitute an important target for the entire community, allowing each research team to concentrate its effort on some specific task, relying on analysis produced by others and letting at the same time other teams taking advantage of their "services". However, before making this possible, tool developers have to agree on the format of data exchanged. Existing schemas, and the challenge to devise a standard schema for C/C++, have been discussed in [31].

3.3 XOgastan AST representation

As shown in Section 2.2, XOgastan relies on an object-oriented representation of the AST. This gives several advantages, in particular:

- The possibility of using visitor design patterns to visit the AST for performing any kind of analysis, transformation or pretty-printing operation. As detailed in [26], this allows to separate the operations from the data structure on which these operations are performed;
- An object-oriented AST model, properly extended, may allow the use of the Object Constraint Language (OCL) [32] to perform analysis on the AST. OCL is a *de-facto* standard language for defining constraints on UML object models, and its power to navigate object models and to manipulate collection types makes the query composition very simple. Given an AST, one can "send" a query on it as sending a SQL query to a relational DBMS, obtaining, as a result set, a piece of the original AST, a scalar value, a collection of values or, in general, any kind of object. Additional details can be found in [25].

As regards the AST structure, it is a three-level tree, where the main families of nodes are represented as subtypes of the root node, while further distinctions are made by means of attributes. This may appear as a counterintuitive choice, but it significantly simplifies the AST navigation. Where needed, the model can be easily detailed using a *Decorator* design pattern, thus avoiding to force the browsing/navigation of useless details.

3.4 XOgastan analysis package

In this Section we will not discuss in detail the XOgastan analysis capabilities. It computes several customary metrics and statistics on the AST, produces CFGs, call graphs, etc. Instead, it is worth illustrating its main design choices, dealing separately with:

- The AST representation;
- The analysis performed on the AST; and
- The generation of the outputs.

As mentioned before, the analyses are separated from the AST internal representation using the well-known *visitor* design pattern. This solution is almost the same implemented in the *JavaCC* tool, which automatically generates the *visitor* abstract class to be implemented by the *concrete visitors* that perform different tasks, such as pretty printing, computing metrics, instrumenting code, performing transformations, etc.

The main problem of the visitor structure is that a visitor has a method for visiting each class of the data structure hierarchy (i.e., each class of the AST). This means that the AST hierarchy should be as stable as possible, otherwise the entire visitor hierarchy needs to be frequently updated. However, often most of the methods of a visitor are very similar to each other (e.g., a pretty-printing visitor, or an instrumenting visitor, etc.), and therefore even the skeleton of a concrete visitor may be automatically generated (and updated).

The results of the analysis performed by visitors (metrics, statistics, etc.) are stored in a suitable data structure: this allows to separate the analysis from output generation.

3.5 XOgastan output capabilities

The XOgastan output is handled by two different packages, the *HTMLWriter package* and the *XML Manager package*. The former produces a browseable HTML output of XOgastan analysis. It is kept separated from the *XML Manager package*, which produces any other kind of output. Since the HTML output constitutes the user interface of the tool, more interactive features are needed.

It is worth pointing out the design structure of the *XML Manager package*, whose objective is to produce any kind of output: the *builder* design pattern, as mentioned above, keeps the internal representation of the data separated from all possible outputs, making it very easy to add new export features for different formats.

The possibility to generate a *DOT* representation of the outputs should be welcome in any kind of tool generating graph outputs. In fact, *DOT* is widely used, it is simple to be understood, and allows the generation of graphs with a large variety of shapes, labels, options, etc. Moreover, visualization and layouting are straightforward through both interactive tools (e.g., *Dotty*) or libraries (e.g., *Grappa*) that allow to build easily graphical tools to interact with the graph itself.

Finally, one of the most important features of XOgastan is the possibility to generate XML output:

- The *XML Query Language (XQL)* can be used to query the AST, in order to collect nodes or subtrees having a given property, to compute metrics, etc. The idea is quite similar to what was proposed in [25] for OCL. In this case, the representation is standard (XML) and, given a DTD, even a simple *Perl* script (relying on the `XML::XQL` package) is sufficient to perform complex analyses. Further work will be devoted to investigate the scalability and performance issues, as well as to compare this approach with others (e.g., OCL, Refine, etc.);

- For simple transformation operations, an XML Stylesheet Language (XSL) processor is sufficient and very easy to be applied. One possible application of XSL transformations (XSLT) is pretty printing, e.g., the production of browseable representations of source code to improve program comprehension and maintenance. Another application is source code instrumentation: once the XSLT for generating source code from the XML representation is available, it can be easily extended, adding a few rules, to instrument the source code. However, XSLTs are not as powerful as a source code transformation engine could be in general. For example, handling complex data structures while performing transformations is not possible;

- When XSLT is not sufficient, a combination of *Document Object Model* (DOM) or SAX parsers and XSLT processors will allow to perform more complex transformations; and

- Finally, as widely recognized [20, 33, 21], XML outputs constitute a fundamental step for tool interoperability and data exchange.

The advantage of XML is essentially the possibility to perform rapid development relying on already available tools. On the other hand, XML-based representations turn out to be fairly verbose if compared to other custom representations. Storing the representation is not generally a problem, due to the plenty of disk space available in modern computers. However, it is worth noting that the compiler-level information (i.e., all the included files and included structures), coupled to the corresponding XML representation, may lead to a remarkably large file even for a small compilation unit (i.e., for a small input source file). Just to give some figures, the `.tu` file generated by *g++* is, on average, 1000 times longer than the source file, and the GXL file is five times longer than the latter (even if its size is about one-half, since useless information is discarded). The representation may be even be too large to be processed as a whole in main memory. This problem can be tackled in two different ways. A first solution is to adopt an event-based parser (instead of the SAX parser adopted by XOgastan), in such a way that the AST is not built in memory. The second solution (the one adopted for XOgastan) is to filter the gathered information to get rid of information not actually essential to the task to be carried out.

4. Conclusions and Work-in-Progress

XOGastan generates HTML, XML and other format representations of information gathered from the AST dumped by the *gcc* compiler. This approach, as well as the design structure of the tool itself, allows to keep separated different activities of the analysis process: the parsing, the AST analysis and transformation and the output generation. Moreover, XML outputs can be easily analyzed and transformed using consolidated languages (such as XQL and XSL) or tools (*Xerces*, *Xalan*, etc.). Finally, the GXL AST representation allows interoperability with a wide variety of tools.

Work in progress is devoted to experiment the scalability and the ease of use of XQL to perform analysis on the AST, also comparing it to other languages such as OCL.

References

[1] I. D. Baxter, "Design Maintenance Systems," *Communications of the Association for Computing Machinery*, vol. 35, April 1992.

[2] J. R. Cordy, C. Halpern, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," in *International Conference on Computer Languages*, pp. 280–285, IEEE Press, October 1988.

[3] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider, "Source transformation in software engineering using the TXL transformation system," *Information and Software Technology*, vol. 44, pp. 827–837, October 1996.

[4] Reasoning Systems, *Refine User's Guide*.

[5] M. Ward, *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1989.

[6] J. Lyle, D. Wallace, J. Graham, K. Gallagher, J. Poole, and D. Binkley, "Unravel: A case tool to assist evaluation of high integrity software," Tech. Rep. NISTIR 5691, U.S. DEPARTMENT OF COMMERCE, Aug 1995.

[7] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, and S. Zanfei, "Program understanding and maintenance with the CANTO environment," in *Proceedings of IEEE International Conference on Software Maintenance*, (Bari Italy), pp. 72–81, Oct 1997.

[8] B. Lague, C. Leduc, A. L. Bon, E. Merlo, and M. Dagenais, "An analysis framework for understanding layered software architecture," in *In Proceedings of the 6 th International Workshop on Program Comprehension*, (Ischia Italy), pp. 37–44, June 24-26 1998.

[9] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[10] R. S. Arnold and S. A. Bohner, "Impact analysis - towards a framework for comparison," in *Proceedings of IEEE International Conference on Software Maintenance*, (Montreal Quebec Canada), pp. 292–301, 1993.

[11] "GrammaTech Codesurfer." `http://www.grammatech.com/products/codesurfer`.

[12] S. Tilley, K. Wong, M. Storey, and H. A. Müller, "Programmable reverse engineering," pp. 501–520, December 1994.

[13] M. Lanza, "CodeCrawler - lessons learned in building a software visualization tool," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, (Benevento, Italy), pp. 409–418, IEEE Press, Mar 2003.

[14] "Semantic Designs Inc. Source Code Browser." `http://www.semdesigns.com`.

[15] M. Karasick, "The architecture of montana: an open and extensible programming environment with an incremental c++ compiler," *ACM SIGSOFT Software Engineering Notes*, vol. 23, pp. 131–142, Nov 1998.

[16] T. R. Dean, A. J. Malton, and R. Holt, "Union schemas as a basis for a c++ extractor," in *Proceedings of IEEE Working Conference on Reverse Engineering*, (Germany), IEEE Press, Oct 2001.

[17] M. Hennessy, B. A. Malloy, and J. F. Power, "gccXfront: exploiting gcc as a front end for program comprehension tools via XML/XSLT," in *Proceedings of the International Workshop in Program Comprehension*, (Portland, Oregon, USA), pp. 298–299, IEEE Press, May 2003.

[18] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-based lightweight C++ fact extractor," in *Proceedings of the International Workshop in Program Comprehension*, (Portland, Oregon, USA), pp. 134–143, IEEE Press, May 2003.

[19] "GCCXML Home Page." `http://www.gccxml.org`.

[20] R. Holt, A. Winter, and S. Sim, "GXL: Towards a standard exchange format," in *Proceedings of IEEE Working Conference on Reverse Engineering*, (Brisbane, Queensland, Australia), IEEE Press, November 2000.

[21] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, and T. Gyimothy, "Towards a standard schema for c/c++," in *Working Conference on Reverse Engineering*, pp. 49–58, 2001.

[22] "XOgastan Home Page." `http://web.ing.unisannio.it/villano/students/masone/`.

[23] A. W. Appel, *Modern Compiler Implementation in Java*. Cambrige University Press, 1998.

[24] "webGAIN javaCC." `http://www.experimentalstuff.com/Technologies/JavaCC/`.

[25] G. Antoniol, M. Di Penta, and E. Merlo, "YAAB (Yet Another AST Browser): Using OCL to navigate ASTs," in *Proceedings of the IEEE International Workshop on Program Comprehension*, (Portland, Oregon, USA), IEEE Computer Society Press, May 2003 (to appear).

[26] E. Gamma, R. Helm, R.Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley Publishing Company, 1995.

[27] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Jugstiani, and B. Sridharan, "Designing the mccat compiler based on a family of structured intermediate representations," *Lecture Notes in Computer Science*, pp. 406–420, Aug 1992.

[28] E. Koutsofios, "Editing graphs with Dotty," technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994.

[29] M. van den Brand, A. Sellink, and C. Verhoef, "Current parsing techniques in software renovation considered harmful," in *Proceedings of the IEEE International Workshop on Program Comprehension*, (Ischia, Italy), pp. 108–117, June 1998.

[30] Lämmel and C. Verhoef, "Cracking the 500-language problem," *IEEE Software*, pp. 78–88, Nov-Dec 2001.

[31] R. Ferenc, S. Sim Elliot, R. C. Holt, R. Koschke, and T. Gyimóthy, "Towards a standard schema for C/C++," in *Proceedings of the IEEE International Workshop on Program Comprehension*, (Ischia, Italy), pp. 108–117, June 1998.

[32] *Object Management Group - Object Constraint Language Specification*. February 2001.

[33] A. Winter, "Exchanging Graphs with GXL," Tech. Rep. 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.