



Compiler Hacking for Source Code Analysis

G. ANTONIOL and M. DI PENTA

antoniol@ieee.org, dipenta@unisannio.it

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

G. MASONE

gianluca.masone@telsey.it

Telsey Telecommunications S.p.A., Viale Mellusi 68, 82100 Benevento, Italy

U. VILLANO

villano@unisannio.it

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

Abstract. Many activities related to software quality assessment and improvement, such as empirical model construction, data flow analysis, testing or reengineering, rely on static source code analysis as the first and fundamental step for gathering the necessary input 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, and is developed and maintained with it. A different approach builds tools on the top of external already-available components such as compilers that output the program abstract syntax tree, or that make it available via an API.

This paper discusses techniques, issues and challenges linked to compiler patching or wrapping for analysis purposes. In particular, different approaches for accessing the compiler parsing information are compared, and the techniques used to decouple the parsing front end from the analysis modules are discussed.

Moreover, the paper presents an approach and a tool, XOGastan, developed exploiting the gcc/g++ ability to save a representation of the intermediate abstract syntax tree. XOGastan translates the gcc/g++ dumped abstract syntax tree format into a Graph eXchange Language representation, which makes it possible to take advantage of currently available XML tools for any subsequent analysis step. The tool is illustrated and its design discussed, showing its architecture and the main implementation choices made.

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

1. Introduction

Source code analysis is an essential preliminary step for activities focused on assessing, monitoring, and improving software quality. Activities such as reverse engineering, reengineering, testing, or building empirical models for software quality, require a preliminary extraction of facts from the source code, or the building of Abstract Syntax Tree (AST) to allow source code transformation.

Several types of languages and toolkits for source code analysis and transformation have been developed during the last decade. Some of them are particularly suited to program comprehension and transformation, such as the *Design Maintenance Systems (DMS)* (Baxter, 1992), the *TXL programming language* (Cordy et al., 1988, 1996), *Re-fine* (Reasoning Systems, 1993), and *FerMaT* (Ward, 1989). These tools have powerful analysis capabilities, 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* (Lyle

et al., 1995), *CANTO/ART* (Antoniol et al., 1997) or *Datrix* (Lague et al., 1998), are more oriented towards static source code analysis and metrics computation than to source code transformation. Developing a tool to assess, monitor or extract quality-related features from source code requires the availability of some kind of parsing technology.

In essence, one of the two following strategies has to be adopted: to develop a parser for the language of interest (e.g., *DMS*, *Unravel*, *Datrix*, *TXL*, *Rigi*) or to rely on one available parser (e.g., *CANTO*, *ART*). Redeveloping from scratch the entire infrastructure, including the parser, may have some advantages. For example, the parser development and the parsing activities may be limited to a subset of the language features of interest, and consequently to a subset of the grammar. This can be done adopting an island parsing strategy (Moonen, 2001). Island parsing significantly eases the parsing task; however, the resulting analyzer cannot be reused for different purposes.

The second alternative requires the availability of a parsing development infrastructure (e.g., lexer, parser generators, grammar plus semantic actions), or, at least, of a grammar for the language to be analyzed. One common problem is that very often the language implementation and the grammar do not perfectly correspond to one another. Either the grammar is incomplete (e.g., missing templates in C++), or it may not be fully compatible with the language dialect to be handled (e.g., GNU or Microsoft C and C++ extensions). Often the only acceptable solution is to extend and to maintain existing grammars, or, in the worst case, to write a new language grammar from scratch. Writing a new grammar can be extremely expensive; the result can be incomplete, not very robust, or can even accept a super-set of the standard programming language. However, the continuous evolution of programming language dialects implies frequent updates to the developed grammars, and this is not a trivial task.

Fortunately, precise, robust, and up-to-date grammars and parsers are embedded in compilers and thus are available. Indeed, the parsing front ends of all compilers currently used to build applications may be customized to analyze and to extract information from the source code. However, compiler and analyzer goals conflict. The compiler aims at producing efficiently executable code, whereas the analyzer goal is essentially to recover high-level abstractions. Depending on the compiler and the compiler interfaces, two alternatives can be followed:

1. *Patching the compiler* source code, to permit the accessing of compiler internal representation via an API (e.g., IBM Montana), or
2. *Building a wrapper*: when the compiler (e.g., GNU *gcc*) stores the AST as an intermediate output format, it is possible to build the analyzer on top of such a representation, without the need for any patching intervention. If such an intermediate output is not directly available, a patching intervention is however required.

All the above mentioned approaches have pros and cons and, in general, there are contrasting opinions on the use of a compiler parsing front end for analysis tasks. If, on the one hand, it could appear appealing and effective, on the other it is worth discussing the challenge of relying on an object developed (by a third part) for different purposes.

These problems will be extensively dealt with in this paper, whose main contributions can be summarized as follows:

- It discusses challenges and approaches for patching or wrapping compilers to develop front ends for source code analysis tools;
- It describes and discusses state of the art of methods and techniques to decouple the parsing front end from analysis engines;
- It proposes an approach and a tool, *XOgastan*, which exploits the possibilities offered by *gcc/g++* interfaces to save the AST representation.

A companion paper (Antoniol et al., 2003) summarizes the *XOgastan* architecture. Here a thorough analysis and discussion of the compiler patching approaches is presented, providing examples, suggestions and insights. The approaches to decouple the front end from analyses are also detailed, and the overhead introduced by XML representations better discussed and quantified.

The paper is organized as follows: Section 2 describes approaches and challenges related to compiler patching. Then, the problem of decoupling the front end from analyses is discussed in Section 3. Section 4 describes the *XOgastan* architecture, detailing the source code analysis process, the tool internal representation, the analysis capabilities and the types of output produced. Section 5 discusses the technology, design and implementation choices made in the development of *XOgastan*, analyzing their strength and weakness. Conclusions and work-in-progress are then outlined.

2. Compiler patching

Analyzing real-world C or C++ systems to extract useful and reliable information requires industrial strength tools. Such tools need to deal with the full set of details, subtleties, variants or dialects of programming languages encountered while parsing the application source code. Given the wide range of available hardware and software platforms, not only should the chosen tools be robust, but they should also be either platform independent or retargetable. In other words, it should be possible to easily port the tool set on a large variety of software and hardware environments (e.g., Sparc—Solaris vs. i486—Windows 2000). Finally, they should support customization and extension; in other words, it should be possible to add new types of analysis.

For example, as for the Java programming language, the JavaCC parser generator (JavaCC home page, 2004) and the available Java grammar permit an easy development of a platform independent, robust, extensible analyzer. Unfortunately, the same development is more challenging in other programming languages, such as COBOL, C and C++. These languages have often been extended creating *dialects*, only partially conforming to ISO or ANSI standards. Sometimes, the compiler itself has its own set of *extended* features such as types, different variable argument passing mechanisms or predefined macros.

Macros and preprocessing features are likely to be the best known C and C++ nightmares. There are basically two approaches for dealing with macros: to expand the source code via a preliminary preprocessor step, or to analyze the source code *as it is* with no expansion at all. At a first glance, the latter approach seems easier to implement. However, it may lead to the extraction of imprecise information, for example because of the lack of knowledge about types (Aversano et al., 2002), or because of the difficulties to execute symbolically preprocessor directives to identify the code to

be included in the application (Hu et al., 2000). Furthermore, this approach promotes verbose and fairly complex language grammars, in that grammar rules to handle pre-processor directives must be interleaved with language rules.

The second approach stems from the consideration that all the required components must be available to compile a system. This allows a preliminary preprocessing step to be safely carried out, and a new set of preprocessed source files to be produced. An example of this category of tools is the *Refine C* parser (Reasoning Systems, 1993); C source files are instrumented via comment-like directives retained by the C pre-processor. Preprocessed files are successively parsed, inserting hooks to map precisely such information as type and function definition points.

Industrial-strength tools can be often classified in between these two extreme categories. For example, the *DMS* tool by Semantics Designs, an environment to analyze and transform software systems, belongs mostly to the first family. However, it also provides a switch to preprocess the source. A well known and fairly robust tool, the *Datrix Bell Canada C++* analyzer (Bell Canada, 1995), adopts a different strategy, being able to recover from missing include files and missing types.

2.1. Approaches to develop a C or C++ software analyzer

Besides the two different philosophies for preprocessing, dealt with in the previous subsection, a further strategic decision linked to the development of a C or C++ software analyzer is concerned with the choice of reusing existing assets, or of developing everything from scratch.

Re-developing from scratch a new environment on the basis, for example, of freely available technologies, is likely to be a costly and risky decision, often leading to failure. There are many tools, grammars, environments no longer developed and maintained, or covering only a subset of what is really needed to be useful. For example, there are several C and C++ public-domain grammars. However, in the authors' knowledge, none of them is able to parse a sizable piece of real-world software.

In other words, if limited resources or effort are available, the first strategy is not recommendable. Indeed, there are environments and tools of industrial strength available, which permit the development on the top of a reliable and robust source code analyzer. For example, IBM made available a component named *Montana* (Karasick, 1998) for the *C++ Visual Age*TM suite. This permits the accessing of the C++ compiler information at the different compilation stages via an API. It would be therefore relatively 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. However, other reliable and robust tools are available. For example, in 2002, Sybase, Inc. released the *Watcom* compiler (Open Watcom home page, 2004) source code; this was the largest industrial project going open source. The compiler suite (C, C++ and Fortran) can be freely downloaded under an Open Source license and thus, at least in theory, an analyzer could be built on top of the *Watcom* compilers. On the other hand, starting from version 2.9, the GNU C Compiler (*gcc*) development team added a new functionality to the *gcc/g++* compilers. This is a switch that permits storing in a file the ASCII repre-

sentation of the AST for each compilation unit (i.e., for each file). This is a particularly important point, as GNU Compilers are available for almost any available computing system or architecture.

2.2. *The case of gcc*

The GNU C Compiler is actually a suite of compilers covering different programming languages (i.e., C, C++, Fortran, Java, Objective C and Ada). It is a *real* compiler, available on all widely adopted platforms and operating systems. The first release dates back to 1987; the latest stable release at the time of writing is the 3.3.2. Another freely available but less successful compiler dating back to the beginning of 90's is the *lcc* retargetable C compiler. The two compilers follow different approaches. *lcc* is a C compiler, developed with a top-down parsing approach; the GNU compiler is a suite of compilers, based on a bottom up strategy. *lcc*, even if less famous than its cousin *gcc*, is extremely appealing (Hanson and Fraser, 1995), and considerably *smaller*. There is extensive documentation and a book providing insight on *lcc*, information and details on the compiler and compiler development decision choices. From a negative point of view, *lcc* is limited to the C programming language. Clearly, the availability of the source code makes it possible to patch the compiler to develop a source code analyzer.

This perspective makes open source compilers and *gcc* a reasonable platform to rely upon. Earliest *gcc* releases were inspired by an extreme parsimonious use of valuable resources, such as CPU time and memory. A complete AST was not built while compiling a module, but only sub-trees were instantiated and immediately thrown away to keep the process footprint small. However, starting from releases dating back to 1999 (i.e., 2.95) not only the complete AST was built, but also an API to navigate and to save the tree was made available. This decision was probably the consequence of both hardware cost reduction and the complete re-design of *gcc*. At the end of 90's the *gcc* complexity reached a non-return point, and the development team decided to re-design and re-develop the compiler, thus creating *gcc* 3. In between *gcc* 2 and 3, "beasts" such as *egcc* and *kgcc* appeared. Fortunately, these *gcc* variants are now extinct.

Evolution also influenced the *gcc* AST interface. 2.95 *gcc* switches permit the correct production of a complete and precise AST, in a GNU-defined encoding, of a C++ source. This encoding was indeed peculiar: for example, the *for* statement was defined in the C grammar file, and it was impossible to obtain the AST of a C function definition via the C compiler. While the *gcc* with the switch *-dump-translation-unit* only printed out a single non-informative node, the *g++* with the same switch produced the expected result. At this stage, two main difficulties arise:

1. Clearly *gcc* is not useful for analyzing the C code; however, nor can the *g++* accomplish that task satisfactory. It is in fact well-known that not all the C code is also C++ compliant (e.g., a declaration such as `int class;` is not C++ compliant);
2. The *g++* dump is not complete for analyzing C++ code; it lacks, in fact, of information regarding the class hierarchy relationships.

A first step forward was obtained during the development of the *CPPX* (Dean et al., 2001), it was discovered that it was possible, activating a special define while bootstrapping the compiler, to obtain a complete AST of a C module via the C compiler

Table 1. *gcc/g++* AST dump switches.

Switch	Language	Description
-fdump-translation-unit	C/C++	Dumps the entire translation unit
-fdump-class-hierarchy	C++	Dumps the class hierarchy and virtual function table
-fdump-tree-original	C++	Dumps before any tree based optimization
-fdump-tree-optimized	C++	Dumps after any tree based optimization
-fdump-tree-inlined	C++	Dumps after function inlining

front end. Noticeably, the authors of the above-mentioned paper identified attributes that such kind of tool should have, for instance to be open-source, to minimize maintenance, to adopt a standard software interchange format, to extract complete information, to have good performance and to be able to support large-scale analyses. The previously mentioned paper also presented an approach to convert the *gcc* schema to other schemas (e.g., *Datrix*) using union schemas. To properly dump information, *CPPX* requires the *gcc* compiler to be patched.

The *gccXfront* (Hennessy et al., 2003) authors followed a similar approach, and equipped their tool with a parse tree browser. Differently from other tools, *srcML* (Collard et al., 2003) is not devoted to producing a complete XML dump of the *gcc* model. Instead, the target is the development of a robust, lightweight C++ fact extractor. Also *GCCXML* (GCCXML home page, 2004) works on a patched version of the *gcc*, often provided with the tool. At the time of writing, it only supports C analysis.

Meanwhile *gcc* underwent a major evolution including the AST production switches between releases 2.9x and 3.2. It is worth noting that in the 2.9x releases some information, such as the C++ class hierarchy structure, was difficult or even impossible to be obtained.

From the source code analysis point of view, this evolution positively affected *gcc*. The latest stable *gcc* release has a completely new set of switches, and accurate AST representation of C and C++ modules are made available. First and foremost, the *gcc* 3.2 dump, made using the switch *-fdump-translation-unit*, is basically consistent when parsing with both *g++* and *gcc*. The difference is that, of course, *g++* also considers all the C++ syntax (e.g., class declarations, etc.). This permitted the use of the *gcc* to parse C files (with the past versions, the *g++* was used, causing problems with all the C code that was not compliant for a C++ parser, e.g., code containing a variable named *new* or *class*, etc.). Secondly, for C++ files, useful information that was previously missing can now be dumped. For instance, the *-fdump-class-hierarchy* switch allows accessing inheritance relationships between classes. Other switches (see Table 1) permit the dumping of the tree before and after optimization and function inlining.

2.3. Pros and cons

At the end of 90's, at the University of Sannio a project was started, whose goal was to develop an analyzer to assess C and C++ quality metrics, reverse engineering *as is* design and, in general, to obtain accurate analyses. Industrial tools were extremely appealing, but either they were limited in the supported platforms (e.g., *Refine C*) or

the source code was not available (e.g., *Datrix*). Freely available front ends such as the Brown University *CPPP* (Reiss, 2004) only covered a subset of the language. At that time, the Zephir project had not yet released its infrastructure. Compiler construction kits, such as the Siemens Cocktail compiler development tools, the PCCTS or the more traditional *Lex* and *Yacc* were available. Having a very limited amount of available resources, it was decided the start from something existing, open source, robust, reliable and retargetable.

In 1998, one of the authors was involved in a project whose goal was to patch the *lcc* compiler, to extract C software metrics. *gcc* was far too complex and not sufficiently documented to attempt a similar project. The *lcc* documentation was accurate enough to allow University graduated students to modify the compiler. The project was carried out at the University of Verona by four students of a software engineering course. The *lcc* 4.0 source code was modified; the patched compiler was able to extract a suite of software metrics at the file and function level, and to store metrics into files while compiling the code. Compiler modification was organized into a patch, so that it could be easily applied and disseminated. However, soon after the end of the project, a new *lcc* version was released. Changes between the newest release and the previous 4.0 were not dramatic, but enough to cause the patch to fail and to require a restructuring of the extra code that computed and saved metrics.

The lesson learned was that if you do not have the control of the evolution process of an application, and if the goal of the application and your product are different, there is no guarantee that your product will be forward compatible with future application releases. This consideration, plus the complexity of the *gcc* source code, led to the decision to avoid patching the compiler. Besides, the above experience showed clearly that *gcc* was undergoing an evolution, and that it would have been too risky to attempt for another *lcc*-like project.

A second key decision was related to the AST representation. The aim was to obtain software quality, design and, in general, reverse engineering information from the source code. This is a fairly different task from the production of object code. Such details, as the maximum integer size that can fit in a variable, or the number of bits of a mask, are likely to be irrelevant for the majority of quality evaluation or reverse engineering tools. A compiler must expand and handle all the code; an analyzer should be focused only on the limited fraction of code developed or modified by programmers. System include files, for example, are of no interest. *gcc* AST information accounts for all the details needed to produce an executable. Preliminary evaluation of the *gcc* information, stored in ASCII files corresponding to compilation units (i.e., source code files), led to the conclusion that a substantial pruning was necessary to reduce the size (considered excessive), by removing information useful for a compiler and not likely to be of any use for an analyzer.

3. Decoupling the parser from analyses

A fundamental task when developing a source code analysis tool is the choice (or the design) of a good representation for the extracted information. This has multiple effects:

Table 2. Some AST representation schemas.

Schema	Description
Datrix (Bell Canada, 1995)	Two implementations: Bell Canada and CPPX
Columbus (Ferenc et al., 2001)	C/C++ schema
Bauhaus (Koschke et al., 1998)	Models C and a subset of Ada
Stanford University Intermediate Format (SUIF) (Aigner et al., 1999)	Works on C and Fortran
CIA (Chen et al., 1998)	Stores facts according to the ACACIA database
Semantic Designs (Baxter, 1992)	Different schemas for different languages
Visual Age C++ (Karasick, 1998)	C/C++ schema
cppML (Mamas and Kontogiannis, 2000)	C/C++ schema

1. Decoupling the parsing front end from analysis/transformation features, thus making it possible to change/add them transparently;
2. Permitting the export of the front end output in a format that can be understood by other tools; and
3. Filtering out, especially when exploiting a compiler's front end, any information that is useless for the prefixed analysis purposes.

A first possible approach is to have an object-based representation of the AST, and then use API and above all, visitors (Gamma et al., 1995), for accessing it and performing the required analyses. Examples of this approach have been followed by JavaCC and, recently, by the Eclipse project (Shavor et al., 1995) (the Eclipse Java Development Toolkit is provided with documented API for accessing, via a visitor object, the source code AST). This approach does not rely on a temporary representation, and thus it is very efficient in terms of performance and space required to store the AST. However, this approach prevents accessibility from external tools. The only possibility is to make the API public. This requires that other tools should rely/understand a custom AST representation. This implies that, in order to allow different people exchanging ASTs or, in general, facts extracted parsing source code, it is necessary to agree on a common representation, (i.e., a *schema*) to represent such information. The *schema* defines the form, in terms of entities with attributes and relationships, which the data will have.

Ferenc et al. (2001) compared existing schemas for C/C++, discussing the issues for developing a common schema. As discussed in the paper, a schema can be automatically generated from the AST, or manually created (i.e., deciding what is worth storing). Some of the most well-known schemas are reported in Table 2.

As stated in Section 2.2, Dean et al. (2001) proposed to use *union schemas* as an approach for identifying a common schema. Once the schema for data representation has been identified, there is still a detail to be handled, i.e. how to store information according to that schema and to ensure interoperability. XML is *de facto* a new emerging standard for information representation and exchange. It permits the exploitation of available parsers (e.g., *Xerces*) and transformation tools (e.g., *Xalan*) to build source code analyzers and to implement pretty printing or source code transformations.

Thus, it is necessary to encode schemas (i.e., graphs) in XML. To this end, Holt et al. have proposed the *Graph eXchange Language* (GXL). GXL is an XML based graph representation, widely adopted in the software maintenance and evolution community.

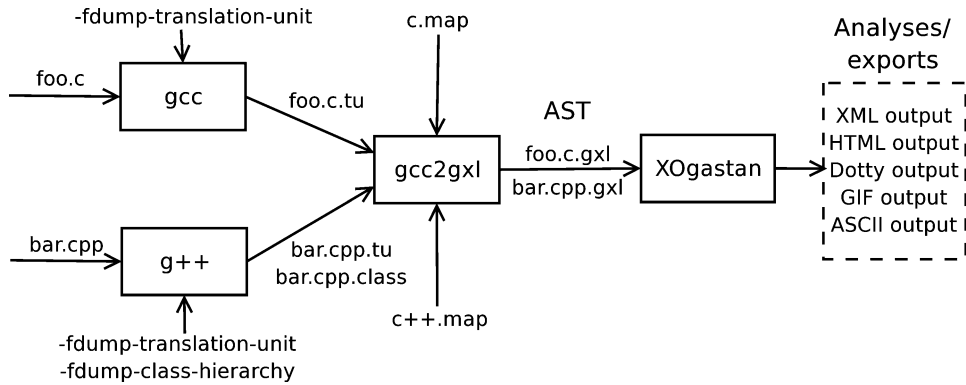


Figure 1. XOgastan analysis process.

At the time of writing, it has been adopted by several software engineering tools, such as *Rigi*, *Columbus/CAN*, *Source Navigator*, *GUPRO*, the *Nokia source code analyzers* and many others (GXL tools, 2004).

When choosing to rely on an XML representation, however, some considerations should be made regarding performance. Performing analyses from ASTs stored in XML files requires an additional parsing task. This is undoubtedly more time-consuming than, for example, to access directly the AST data structure via an API. Secondly, it is worth considering the (sometimes enormous) overhead introduced by XML encoding: even a simple “hello world” C program could cause the dumping of hundreds of XML lines. In Section 5.5 we will try to quantify these aspects.

4. XOgastan

This section describes *XOgastan*, the *gcc* wrapping tool developed at the University of Sannio. Different from many other tools, *XOgastan* does not require the application of any patch to the compiler, but it only relies on the information dumped by means of the command line switches described in Table 1. At the time of development, this approach had suffered several weaknesses: C++ dumped information which lacked of hierarchical relationships among classes, and it was necessary to use the *g++* for analyzing C programs. However, now this has turned out to be a winning choice, since the switches available with the current C++ compiler permit the dumping of all the required information, and the C compiler produces a consistent dump, usable for analysis purposes.

This section describes the overall design architecture of *XOgastan*, and also gives details regarding/relative to some important implementation issues. The *XOgastan* analysis process can be described as shown in Figure 1, and it is composed of the following phases:

1. The AST is dumped by *gcc*, compiling the source file (*foo.c* for C, *bar.cpp* for C++ in our example) using the option *-fdump-translation-unit*, and also the *-fdump-class-hierarchy* switch is enabled to recover class hierarchy information. For C++,

the `g++` compiler is used. From this point onwards, for simplicity sake, both compilers will be referred to as `gcc`. `g++` will be used only to refer explicitly to the C++ compiler.

2. The dump produced by `gcc` (file `foo.c.tu` for C, `bar.cpp.tu` and `bar.cpp.class` for C++) is translated into GXL format by means of a *Perl* script supplied with *XOgastan*, `gcc2gxl`. As will be described in detail later, the translation relies on translation maps;
3. Finally, the GXL output is analyzed by *XOgastan*, producing outputs in HTML, XML and other formats.

Overall, *XOgastan* has a hybrid pipe/filter 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 9.

4.1. 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 of ASTs (and, more in general, of graphs) is the already mentioned GXL format. Basically, the purpose of GXL is to permit the exchange of graphs between different tools, such as those performing program analysis. As previously stated, the transformation from `gcc` dump to GXL is carried out by a *Perl* script. This script relies on transformation maps, supplied in files named respectively `c.map` and `c++.map` (from here onwards, both files will be referred to as `c.map`). The AST of `gcc` can contain several different node types. Every node is characterized by a code (describing its purpose), a list of attributes, and a list of possible linked nodes. The file `c.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 the 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
```

```

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>
scope:*%<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.

```

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.

- Some of the translation rules for *function_decl* nodes are shown in Figure 2. Rules may have two different formats (the interested reader can refer, for further details, to the on-line documentation of the tool (XOgastan home page, 2004)):
 - field: * % <gxl element>: in this case any occurrence of the sequence field is translated with the corresponding <gxl element>. Then, any occurrence of the '*' symbol inside the <gxl element> is replaced by any string appearing on the right of field. An example if this type of rule is the name rule in Figure 2;
 - field % <gxl element>: this is similar to the previous case, however no substitution is made in the <gxl element>. An example is the artificial rule in Figure 2.
- 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 *c.map* were written after gaining insight on *gcc* AST by studying the *gcc* functions devoted to producing/handling 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*, *cp-tree.def*, *cp-tree.h*, *dump.h*, *dump.c* and *cpdump.c*.

The *.def* files contain code definitions related to *gcc* AST nodes, accompanied with detailed comments describing each of them. The analysis of these files gave a great help in understanding the *gcc* AST, as well as the format of the *gcc* dump. Figure 4 reports an excerpt of the *tree.def* file.

```

<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.

```

DEFTREECODE (IDENTIFIER_NODE, "identifier_node", 'x', -1)
DEFTREECODE (OP_IDENTIFIER, "op_identifier", 'x', 2)
DEFTREECODE (BLOCK, "block", 'b', 0)
DEFTREECODE (INTEGER_TYPE, "integer_type", 't', 0)
DEFTREECODE (FUNCTION_DECL, "function_decl", 'd', 0)
DEFTREECODE (LABEL_DECL, "label_decl", 'd', 0)
DEFTREECODE (CONST_DECL, "const_decl", 'd', 0)
DEFTREECODE (COND_EXPR, "cond_expr", 'e', 3)

```

Figure 4. Excerpt of the tree.def file.

4.2. *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 in the Appel's book (Appel, 1998), to the *JavaCC* AST and to the AST object model proposed in (Antoniol et al., 2003). The NAST is quite different from the *gcc* AST, in that some parts of the AST are not present at all in the NAST. In particular, the NAST is composed of three types of nodes:

1. The NAST *root node*;
2. The nodes representing the set of different language constructs: declarations, constants, expressions, statements, etc.; and
3. The NAST leaves i.e., identifiers, predefined types, literals, etc.

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 6(a) reports the NAST representation corresponding to the simple C function shown in Figure 5. The next subsection will explain *XOgastan* features by showing outputs obtained analyzing the same function.

4.3. *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 analyses, as follows:

```

void leggiNumero(struct nodo **n)
{
    char c;
    while ((c = getchar()) != '\n')
        inserisciCifra(n,c-'0');
    return;
}
    
```

Figure 5. C function to be analyzed.

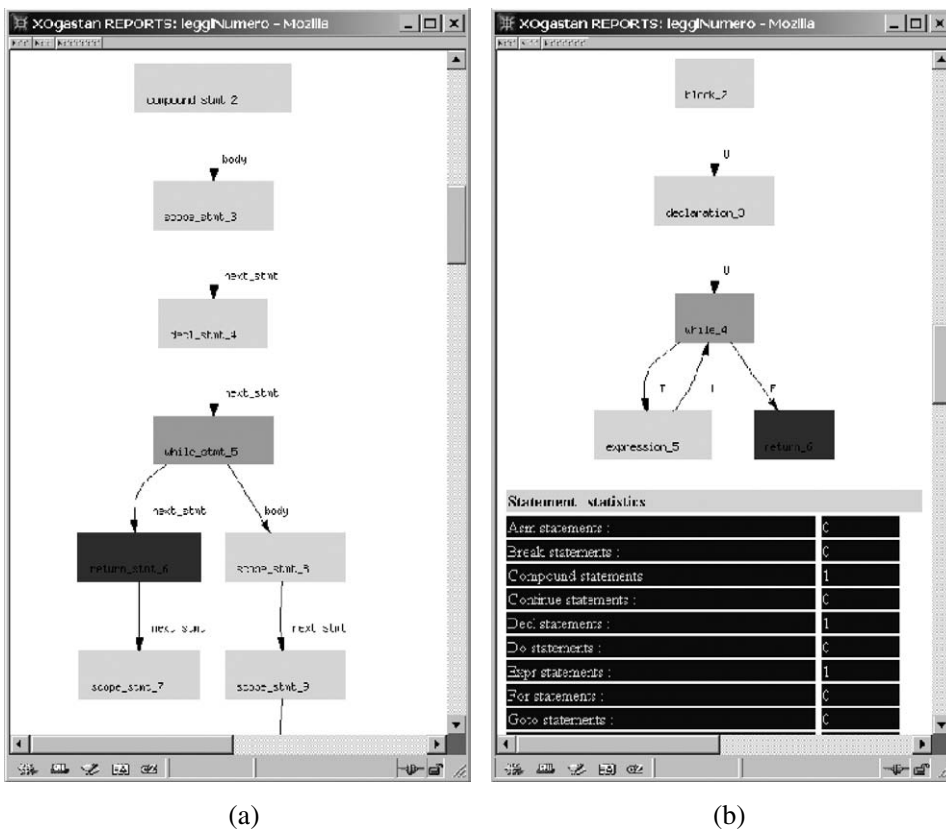



Figure 6. (a) Function body NAST and (b) control flow graph and statement statistics.

- It gets the name, the type returned, the parameter list of the function. Figure 7(a) reports an hyperlinked list of all the functions contained in the source code analyzed, while Figure 7(b) reports detailed information for a particular function.
- It gets some information regarding the statements in the function body: produces statistics of the statements used, builds a Control Flow Graph (CFG), etc. The table shown in the bottom part of Figure 6(b) counts, for each category, the number of statements a function body includes. The function CFG is represented at the top of the figure.



Function name	index
cancella	13
insensciCifra	148
insensciCifraCoda	108
leggiNumero	194
man	3
prodotto	48
scriviNumero	27
sommaNumeri	75

(a)



This is the page of the function leggiNumero.

leggiNumero

Nast data :

- address of the function_decl : 0x8178570
- index of the function_decl : 194
- address of the function's body : 0x818a120

Type returned :

- void

List of the parameters :

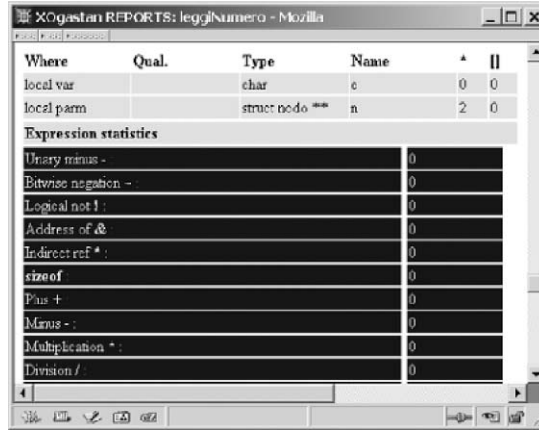
Qual.	Type	Name	^	
	struct modo **	n	2	0

(b)

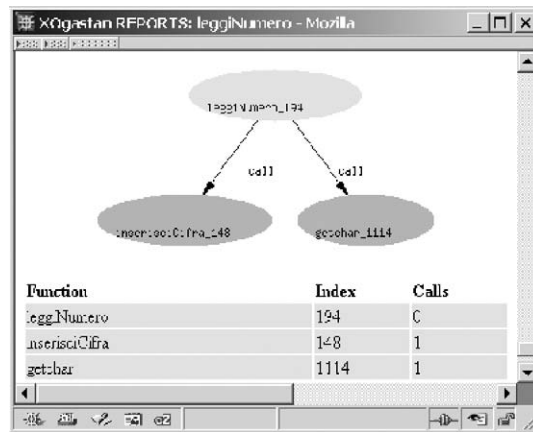
Figure 7. (a) List of analyzed functions and (b) function parameters.

- It builds the list of declarations in the function body: variable declarations, type declarations, etc.
- It produces statistics regarding the number of expressions and operators used.
- It builds a list of the variables used inside expressions: variables with local scope, variables with global scope, and parameters. Figure 8(a) describes function parameters and variables, and lists statistics on the different expressions used in the function.
- It builds a call-graph of the analyzed function. Figure 8(b) depicts the call-graph for the function shown in Figure 5, as well as the number of call sites for each called function.

XOgastan also produces statistical information regarding the given NAST (total number of nodes, frequency of a node, etc.). Analyses are performed using a visitor design pattern (see Section 5.3 for details).



(a)



(b)

Figure 8. (a) Variable and expression statistics and (b) call graph.

The detailed description of the analyses that can be performed using the *XOgastan* output is out of the scope of this paper. This is also due to the fact that *XOgastan* was not designed to obtain a strong source analyzer, but simply to interface a strong parser (the *gcc* one) with other analysis tools/plugins relying on *XOgastan* XML fact representation. Exploiting directly compiler analysis capabilities is theoretically feasible, but in practice turns out to be very difficult (Hendren et al., 1992).

4.4. *XOgastan* output

The output produced by *XOgastan* is available in several formats:

- HTML pages, containing browseable CFGs and statistics produced analyzing the NAST;
- XML files, that can be read by other XML-oriented applications (see Section 5.4) or browsed using an XML browser;
- DOT representations of body graphs, CFGs and call graphs, to be visualized using the *Dotty* tool (Koutsofios, 1994);
- ASCII representations of the CFG; and
- Graphic Interchange Format (GIF) plots of the CFG.

4.5. *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 4.2 and 5.2).
- 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 regarding:
 1. Declared functions;
 2. Call graph;
 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 for performing queries on the AST.
- The *XML Manager package*, which generates different *XOgastan* XML outputs. This package is composed of an XMLBuilder, i.e., a *builder* design pattern (Gamma et al., 1995) 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 9.

5. Discussion

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

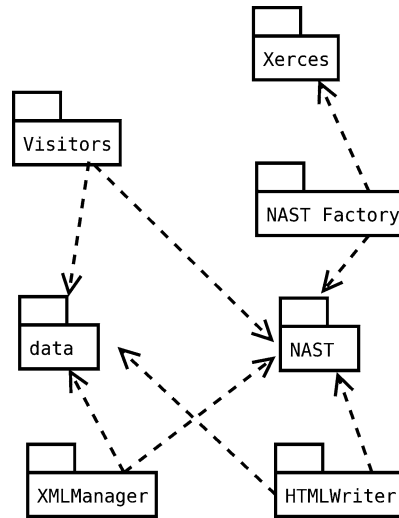


Figure 9. XOGastan UML package diagram.

5.1. 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 actually 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 the possibility 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 `c.map` and the `gcc2GXL` script. Compared to tools where the AST is dumped by patching the compiler, this approach permits 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 straight-forward, 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 permits tool interoperability. In the authors' opinion, this should be an important target for the entire community, allowing every research team to concentrate its effort on some specific tasks, relying on analysis produced by others, and permitting at the same time other teams to take advantage of their "services."

5.2. *XOgastan AST representation*

As shown in Section 4.2, *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. This permits the separation of the operations performed from the data structure on which these operations are carried out (Gamma et al., 1995).
- An object-oriented AST model, properly extended, may permit the use of the Object Constraint Language (OCL) (Object Management Group, 2001) 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 query it by 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 (refer to the work of Antoniol et al. (2003) for additional details).

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 the browsing/navigation of useless details.

5.3. *XOgastan analysis package*

As previously mentioned, the analyses are separated from the AST internal representation using the well-known *visitor* design pattern. This solution is almost the same as the one 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, most of the methods of a visitor are often 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 permits the separation of the analysis from output generation.

5.4. *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 separate 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 data separate 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 understand, and permits 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 the construction of easily graphical tools to interact with the graph itself.

Finally, one of the most important features of *XOgastan* is the possibility to generate an 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 the one proposed by Antoniol et al. (2003) 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 is currently in progress 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 apply. One possible application of XSL transformations (XSLT) is pretty printing such as, for example, the production of browseable representations of the source code to improve program comprehension and maintenance. Another application is source code instrumentation: once the XSLT for generating the 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 permits more complex transformations. However, in some cases, the STX (Streaming Transformations for XML) (Cimprich, 2002) can constitute a valid alternative to XSLT.
- Finally, as is widely recognized (Holt et al., 2000; Winter, 2001; Ferenc et al., 2001), XML outputs constitute a fundamental step for tool interoperability and data exchange.

5.5. Dealing with XML overhead

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

Table 3. File sizes and compression rates for different compressors.

File type	Total size [Kbytes]	% of the GXL file
GXL	6,201,268	—
Zip	457,757	7.38
Gzip	457,725	7.38
Bzip2	270,342	4.36
XMLppm	302,970	4.89
Xmill	250,268	4.04
Xbmill	218,929	3.53
C sources	7,905	0.13

representation is not generally a problem, due to the abundance 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), along with 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). This representation may be even 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 into the 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.

When XML-encoded ASTs have to simply be stored on disk to be analyzed afterwards, a possible solution to that overhead is the compression. This can be performed using traditional compressors (e.g., *zip*, *gzip*, *bzip2*). However, specific compressors have also been developed for XML files; these compressors do a very good job on tree representations. Examples are *XMLPPM* (Cheney, 2004) or *XMill* (Liefke and Suci, 2000).

To give some figures about the compression capabilities of the above mentioned tools, and of the size of the GXL files, we extracted the AST from *Samba 3.0.1* (an open source file sharing system) `.c` files. *Samba* sources consist of 713 `.c` files, 140 `.h` files, for a total of 320 KLOC. GXL, C and compressed file sizes (in Kbytes and as a ratio of the GXL file size) are reported in Table 3. Figure 10 reports boxplots of the compression rates obtained, for each source file, by the different compressors. Clearly, GXL representations are expensive (8 MBytes of C source lead to 6 GBytes of GXL). However, even common compressors are effective to reduce its size (in particular, *bzip2*). The *Xmill* compressor proved to be even more efficient, especially when combined with *bzip* (*Xbmill*).

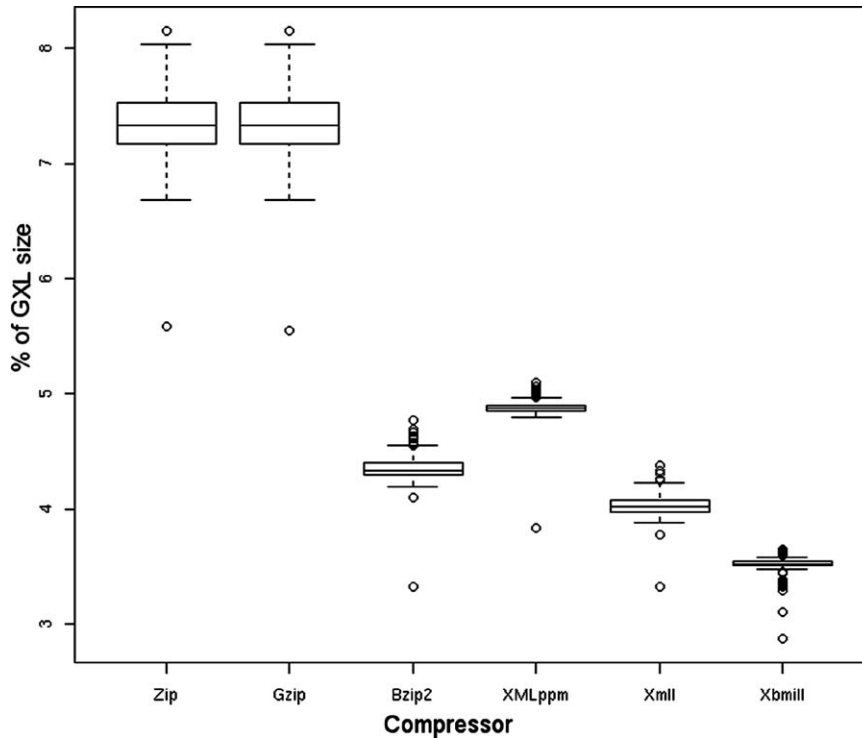


Figure 10. XML compression rate boxplots.

6. Conclusions and work-in-progress

The diffusion of many different programming languages and the significant increment of dialects continuously require the development of new parsers, and the updating of existing ones. The alternative is to rely on compiler parsing capabilities. To this end, the strategies that can be followed are multiple, as for example patching the compiler's source code, or relying on dumps produced by the activation of suitable compiler switches. The former requires a significant effort and suffers from a lack of portability to future compiler releases; the latter can be easier, provided that the compiler permits the dumping of all required information. In both cases, it is necessary to deal with a tool, the compiler, that was built for fairly different purposes than source code analysis. This, even in the most benign cases, requires pruning lots of useless information.

This paper has proposed a tool, *XOgastan*, which 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, permits a separation of the different activities of the analysis process: parsing, AST analysis and transformation, and output generation. Moreover, XML outputs can be easily analyzed and transformed using consolidated languages (such as XQL and XSL) or tools (*Xerces*, *Xalan*, etc.).

Approaches similar to the one followed for *XOgastan* can be, in the authors' opinion, adopted for other programming languages, if compilers made the AST dump available. This could be a possible way to handle the 500-Language problem (Lämmel and Verhoef, 2001). 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.

Finally, the GXL AST representation allows interoperability with a wide variety of tools. However, this produces a significant overhead both in terms of disk space and of time required to perform the analyses. For the former, XML compression techniques can be therefore adopted.

References

- Aigner, G., Diwan, A., Heine, D., Lam, M., Moore, D., Murphy, B.R., and Sapuntzakis, C. 1999. The basic SUIF programming guide, November.
- Antoniol, G., Di Penta, M., Masone, G., and Villano, U. 2003. XOgastan: XML-oriented gcc AST analysis and transformations, *International Workshop on Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, October, pp. 173–182. IEEE Computer Society Press.
- Antoniol, G., Di Penta, M., and Merlo, E. 2003. YAAB (Yet Another AST Browser): Using OCL to navigate ASTs, *Proceedings of the IEEE International Workshop on Program Comprehension*, Portland, OR, May, pp. 13–22. IEEE Computer Society Press.
- Antoniol, G., Fiutem, R., Lutteri, G., Tonella, P., and Zanfei, S. 1997. Program understanding and maintenance with the CANTO environment, *Proceedings of IEEE International Conference on Software Maintenance*, Bari, Italy, October, pp. 72–81.
- Appel, A.W. 1998. *Modern Compiler Implementation in Java*. Cambridge University Press.
- Aversano, L., Di Penta, M., and Baxter, I. 2002. Handling preprocessor-conditioned declarations, *International Workshop on Source Code Analysis and Manipulation*, Montréal, QC, Canada, October, pp. 83–92. IEEE Computer Society Press.
- Baxter, I.D. 1992. Design maintenance systems, *Communications of the Association for Computing Machinery* 35: April.
- Bell Canada. 1995. *Datrix: A Tool for Software Evaluation*, Reference Guide.
- Chen, Y., Gansner, E., and Eleftherios, K. 1998. A C++ data model to support reachability analysis and dead code detection, *IEEE Transactions on Software Engineering* 24(2): 682–693.
- Cheney, J. XMLppm compressor, <http://sourceforge.net/projects/xmlppm/>.
- Cimprich, P. 2002. Streaming transformations for XML (STX) version 1.0 working draft, <http://stx.sourceforge.net/documents/spec-stx-20021101.html>.
- Collard, M.L., Kagdi, H.H., and Maletic, J.I. 2003. An XML-based lightweight C++ fact extractor, *Proceedings of the International Workshop in Program Comprehension*, Portland, OR, May, pp. 134–143. IEEE Press.
- Cordy, J.R., Dean, T.R., Malton, A.J., and Schneider, K.A. 1996. Source transformation in software engineering using the TXL transformation system, *Information and Software Technology* 44(October): 827–837.
- Cordy, J.R., Halpern, C., and Promislow, E. 1988. TXL: A rapid prototyping system for programming language dialects, *International Conference on Computer Languages*, pp. 280–285. IEEE Press.
- Dean, T.R., Malton, A.J., and Holt, R. 2001. Union schemas as a basis for a C++ extractor, *Proceedings of IEEE Working Conference on Reverse Engineering*, Germany, October. IEEE Press.
- Ferenc, R., Elliott Sim, S., Holt, R., Koschke, R., and Gyimóthy, T. 2001. Towards a standard schema for C/C++, *Proceedings of IEEE Working Conference on Reverse Engineering*, Stuttgart, Germany, October. IEEE Computer Society Press.
- Ferenc, R., Magyar, F., Bezédés, A., Kiss, A., and Tarkiaainen, M. 2001. Columbus—tool for reverse engineering large object oriented software systems, *Symposium on Programming Languages and Software Tools*, Szeged, Hungary, June, pp. 16–27.
- Ferenc, R., Sim, S.E., Holt, R.C., Koschke, R., and Gyimóthy, T. 2001. Towards a standard schema for C/C++, *Working Conference on Reverse Engineering*, pp. 49–58.

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA, Addison-Wesley.
- GCCXML home page. 2004. <http://www.gccxml.org>.
- GXL tools. 2004. <http://www.gupro.de/GXL/tools/tools.html>.
- Hanson, D.R. and Fraser, C.W. 1995. *A Retargetable C Compiler*. Addison-Wesley.
- Hendren, L.J., Donawa, C., Emami, M., Gao, G.R., Jugstiani, and Sridharan, B. 1992. Designing the mccat compiler based on a family of structured intermediate representations, *Lecture Notes in Computer Science*, pp. 406–420.
- Hennessy, M., Malloy, B.A., and Power, J.F. 2003. gccXfront: exploiting gcc as a front end for program comprehension tools via XML/XSLT, *Proceedings of the International Workshop in Program Comprehension*, Portland, OR, May, pp. 298–299. IEEE Press.
- Holt, R., Winter, A., and Sim, S. 2000. GXL: Towards a standard exchange format, *Proceedings of IEEE Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, November. IEEE Press.
- Hu, Y., Merlo, E., Dagenais, M., and Lague, B. 2000. C/C++ conditional compilation analysis using symbolic execution, *Proceedings of IEEE International Conference on Software Maintenance*, San Jose, CA, October, pp. 196–206.
- JavaCC home page, <https://javacc.dev.java.net/>.
- Karasick, M. 1998. The architecture of Montana: an open and extensible programming environment with an incremental c++ compiler, *ACM SIGSOFT Software Engineering Notes* 23(November): 131–142.
- Koschke, R., Girard, J., and Würthner, M., An intermediate representation for integrating reverse engineering analyses, *Proceedings of IEEE Working Conference on Reverse Engineering*, Honolulu, HI, October. IEEE Computer Society Press.
- Koutsoufios, E. 1994. Editing graphs with Doty, Technical Report, AT&T Bell Laboratories, Murray Hill, NJ, July 1994.
- Lague, B., Leduc, C., Bon, A.L., Merlo, E., and Dagenais, M. 1998. An analysis framework for understanding layered software architecture, *Proceedings of the 6th International Workshop on Program Comprehension*, Ischia, Italy, June 24–26, pp. 37–44.
- Lämmel, R. and Verhoef, C. 2001. Cracking the 500-language problem, *IEEE Software*, November–December: 78–88.
- Liefke, H. and Suciu, D. 2000. XMill: an efficient compressor for XML data, Technical Report, AT&T.
- Lyle, J., Wallace, D., Graham, J., Gallagher, K., Poole, J., and Binkley, D. 1995. Unravel: A case tool to assist evaluation of high integrity software, Technical Report NISTIR 5691, U.S. Department of Commerce, August.
- Mamas, E. and Kontogiannis, K. 2000. Towards portable source code representations using XML, *Proceedings of IEEE Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, November, pp. 172–182. IEEE Computer Society Press.
- Moonen, L. 2001. Generating robust parsers using island grammars, *Proceedings of IEEE Working Conference on Reverse Engineering*, October, pp. 13–22.
- Object Management Group. 2001. Object Constraint Language Specification, February.
- Open Watcom home page. 2004. <http://www.openwatcom.org/>.
- Reasoning Systems. 1993. *Refine User's Guide*.
- Reiss, S.P. 2004. CPPP project, <ftp://ftp.cs.brown.edu/pub/cppp.tar.Z>.
- Shavor, S., D'Anjou, J., Kehn, D., Kellerman, J., and McCarthy, P. 1995. *The Java Developer's Guide to Eclipse*. Reading, MA, Addison-Wesley.
- Ward, M. 1989. *Proving Program Refinements and Transformations*, Ph.D. Thesis, Oxford University.
- Winter, A. 2001. Exchanging graphs with GXL, Technical Report 9-2001, Universität Koblenz-Landau, Institut für Informatik, Koblenz.
- XOgastan home page. 2004. <http://web.ing.unisannio.it/villano/students/masone/>.



Giuliano Antoniol received his doctoral degree in electronic engineering from the University of Padua and the Ph.D. from the Ecole Polytechnique de Montreal.

He worked at ITC/Irst Research Institute for ten year were he leads the the Irst Program Understanding and Reverse Engineering (PURE) Project team.

Giuliano Antoniol published more than 90 papers in journals and international conferences. He served as a member of the Program Committee of international conferences and workshops such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, the International Symposium on Software Metrics.

He is presently member of the Editorial Board of the Journal Software Testing Verification & Reliability, the Journal Information and Software Technology, the Empirical Software Engineering Journal and the Journal of Software Quality.

He is currently Associate Professor the University of Sannio, Faculty of Engineering, where he works in the area of software for bioinformatics application, software engineering software evolution and maintenance.



Massimiliano Di Penta received his laurea degree in computer engineering in 1999 and his Ph.D. in computer science engineering in 2003 at the University of Sannio in Benevento, Italy. Currently he is with RCOST—Research Centre On Software Technology in the same University. His main research interests include software maintenance, software quality, reverse engineering, program comprehension and search-based software engineering. He is author of more than 30 papers appeared in international journals, conferences and workshops. He serves the program and organizing committees of workshops and conferences in the software maintenance field, such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, the Workshop on Source Code Analysis and Manipulation.



Gianluca Masone is software engineer at Telsey Telecommunications, working in the multimedia department. He is graduate in software engineering at University of Sannio in Benevento; his thesis is about measuring performance of computer's cache systems. He is developer and maintener for video streaming applications on set-top-boxes.



Umberto Villano is a Professor at the University of Sannio at Benevento, Italy, where he is President of the Didactical Committee of the Degree in Computer Science Engineering. His major research interests concern performance prediction and analysis of parallel and distributed computer architectures, tools and environments for parallel programming and distributed algorithms. He received the Laurea degree in electronic engineering *cum laude* from the University of Naples in 1983.