g^{4}re: Harnessing GCC to Reverse Engineer C++ Applications

Nicholas A. Kraft, Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634, USA
{nkraft,malloy}@cs.clemson.edu

James F. Power
Computer Science Department
National University of Ireland
Maynooth, Co. Kildare, Ireland
jpower@cs.nuim.ie

Abstract

In this paper, we describe g^{4}re, our tool chain that exploits GENERIC, an intermediate format incorporated into the gcc C++ compiler, to facilitate analysis of real C++ applications. The gcc GENERIC representation is available through a file generated for each translation unit (`tu`*), and g^{4}re reads each* `tu` *file and constructs a corresponding Abstract Semantic Graph (ASG). Since* `tu` *files can be prohibitively large, ranging from 10 megabytes for a "hello world" program, to 18 gigabytes for a version of Mozilla Thunderbird, we describe our approach for reducing the size of the generated ASG.*

1 Introduction

Software tools are essential for the comprehension, analysis, testing and debugging of software applications. Tools can automate repetitive tasks and, with large scale systems, enable computation that would be prohibitively time consuming if performed manually. The Java language is accompanied by an abundant selection of libraries and tools to support application development [6, 28, 29]. However, there are relatively fewer tools to support applications that use the C++ language.

One explanation for the lack of software tools for C++ is the difficulty in constructing a front-end for the language, as described in references [5, 16, 17, 18, 20, 24, 25, 26, 27]. This difficulty results from the complexity and scale of the grammar and language of C++ as well as the insufficient understanding of the engineering aspects of grammarware [16]. Despite the pervasive role of grammars in software systems and the continuing research in this area, a complete understanding of grammar-based systems remains an elusive goal [16]. Recent efforts in constructing a parser and front-end for C++ have focused on using novel techniques such as token-decorated parsing [20] or GLR parsing [24]. However, these parsers have difficulty parsing real applications and although there is an excellent compiler in *gcc*, the *gcc* parser is tightly coupled to the compiler itself.

In this paper, we describe *g^{4}re*[1], our tool chain that exploits `GENERIC`, an intermediate format incorporated into the *gcc* C++ compiler, to facilitate analysis of real C++ applications. The *gcc* `GENERIC` representation is available through a file generated for each translation unit (`tu`), and *g^{4}re* reads each `tu` file and constructs the corresponding Abstract Semantic Graph (ASG) for each translation unit. Since `tu` files can be prohibitively large, ranging from 10 megabytes for the single `tu` file generated for a "hello world" program, to 18 gigabytes for the collection of `tu` files generated for Mozilla Thunderbird (version 1.0), we describe our approach for reducing the size of a generated ASG through transformations on the ASG. Our *g^{4}re* tool chain can reverse-engineer any application that can be parsed by the *gcc* C++ compiler.

In the next section we provide background about the *gcc* ASG schema, `GENERIC`, and the XML-based exchange format GXL, and in Section 3 we provide details about the construction and usage of our *g^{4}re* tool chain. In Section 4 we provide some results about the sizes of both `tu` files - generated with and without stub headers, and re-created ASGs - before and after optimization. In addition, we compare our

[1]The name of our tool, *g^{4}re*, when expanded, is ggggre, which is a mnemonic for *gcc* and generic gxl graphs for reverse engineering

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

g^4re tool chain to the gcc2gxl subsystem of XOGastan [3] and summarize the differences in the size of the GXL graphs generated by each system as well as the time required by each system to generate the GXL graphs. Finally, in Section 6 we draw conclusions and describe future work.

2 Background

In this section we provide some background on the two major technologies we use in this paper: the *gcc* ASG schema, `GENERIC`, and the Graph eXchange Language (GXL). Section 2.1 provides a brief overview of `GENERIC`, which has recently been exploited by several research tools for program analysis and reverse engineering [2, 3, 9, 23]. Section 2.2 provides a brief overview of GXL, commonly used by reverse engineering tools.

2.1 GENERIC - The gcc ASG Schema

The Abstract Semantic Graph (ASG) is a common program representation used by compiler front ends and other grammarware tools. A UML class diagram is frequently used to describe the nodes and edges in an ASG; such a class diagram is referred to as a *schema* for the ASG. The C++ compiler from the GNU Compiler Collection, *gcc*, uses an ASG to facilitate recognition, analysis, and optimization of a program. Since version 3.0, *gcc* has begun to develop an ASG schema known as `GENERIC`.

The *gcc* ASG schema, `GENERIC`, consists of over 200 node types and is documented - almost exclusively - in the form of source code comments. Example node types include: `record_type`, `call_expr`, and `field_decl`. The `GENERIC` instance for each translation unit in a C++ program is available as a text file via the command line option `-fdump-translation-unit-all`. The format of the text files, known as `tu` files, is illustrated in Figure 1.

The `tu` file format, illustrated in Figure 1, can be parsed easily by a tool wishing to use the *gcc* ASG for program analysis, comprehension, testing, and transformation. A node in a `tu` file is represented by:

- a unique identifier consisting of '@' concatenated with a unique integer,
- a node type from the `GENERIC` ASG schema,

```
@8  field_decl  name: @15  type: @16  scpe: @5
                srcp: test.cpp:5      chan: @17
                public     size: @18  algn: 32
                bpos: @19  addr: 4065e000
```

Figure 1. *Example*: **This figure illustrates** `tu` **file node representation.**

- edge tuples consisting of the edge name and the unique identifier of the destination node,
- field tuples consisting of the field name and the field value,
- single word attributes.

For example, in Figure 1, node '`@8`' has type `field_decl`, an edge `name` with destination '`@15`', a field `srcp` with value `test.cpp:5`, and a single word attribute `public`.

2.2 GXL - Graph eXchange Language

Selecting an exchange format is an important aspect in the design of a reverse engineering tool such as a parser, analyzer, or visualizer. Currently, GXL (Graph eXchange Language) is the de facto standard exchange format for use with reverse engineering tools [12]. GXL is an XML sublanguage defined by an XML DTD (Document Type Definition) and conceptualized as a typed, attibuted, directed graph. GXL is used to describe both instance data and its schema; schemas in GXL can be represented by UML class diagrams [12].

3 The g^4re Tool Chain

Our g^4re system reads each `tu` file and re-creates an in-memory representation of the corresponding ASG. Since `tu` files can be prohibitively large, we have developed two optimizations and incorporated them into the g^4re tool chain: (1) removing extraneous library code, and (2) pruning the ASG.

To remove extraneous library code, we use stub headers in place of the C++ standard library files, reducing the size of the generated ASG without losing information needed for static analysis of C++

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

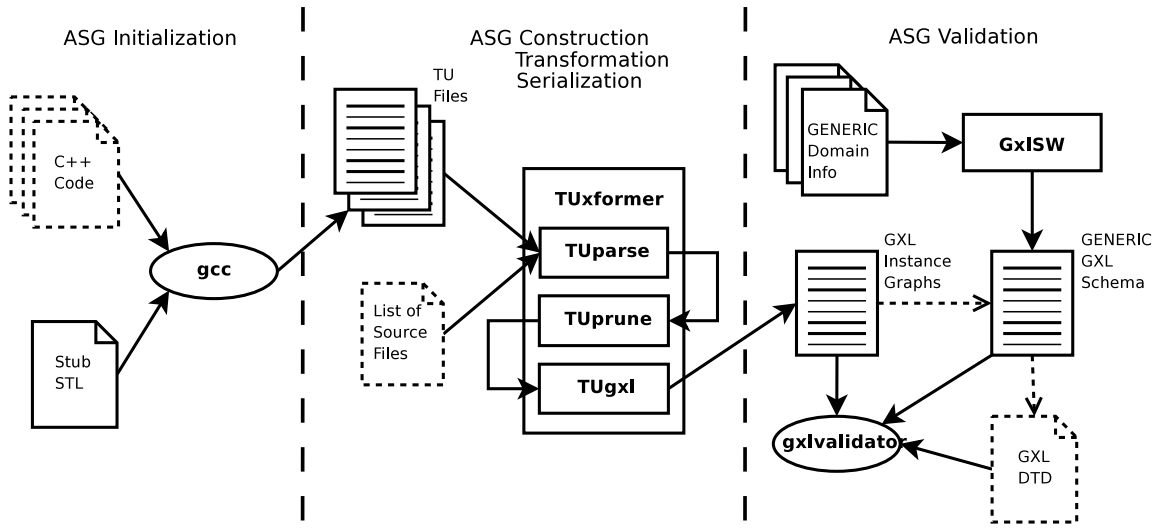

Figure 2. *System Architecture*. **This figure provides an overview of the phases performed by the** g^4re **tool chain to create a GXL instance graph for each translation unit in a C++ program. User inputs to the system, e.g. the C++ Code, are shown as tabbed, dashed rectangles, external programs, e.g.** *gcc*, **are ellipses, generated files, e.g. the GXL instance graphs, are lined rectangles, and our inputs and programs, e.g. the stub STL and** TUxformer, **are tabbed, solid rectangles and non-tabbed rectangles, respectively. I/O in the system is shown as solid edges with solid arrows, while conformance is shown as dashed edges with open arrows.**

user code. Stub header files contain class declarations, function signatures, type definitions, and variable declarations, but do not contain class definitions or function bodies. We can elide details of the library implementation because the details are not required for static analysis of user code. In fact, only details of the interactions between user code and library code are required for static analysis of user code. The stub header files that we have constructed to replace C++ standard header files are based on the *gcc* implementation, version 3.3.4.

We use the stub headers to generate each `tu` file, and then re-create the corresponding ASG using the information in the `tu` file. However, this ASG contains extraneous nodes and fields that hold internal information that the *gcc* compiler uses to facilitate recognition or optimization of the input C++ program. For the second optimization, we remove these extraneous nodes and fields, since they are not needed for analysis of user code. It would be easier to simply "filter" these non-essential nodes when traversing the ASG. However, non-essential nodes can be inserted between essential nodes in the `GENERIC` ASG, and therefore appear as such in the generated `tu` file. Thus, the non-essential nodes must be "unlinked" from the ASG, with essential nodes relinked so that the optimized ASG remains a single connected graph.

In Section 3.1 we provide an overview of the g^4re tool chain, including a description of where the two optimizations fit into the chain. In Section 3.2 we provide further details about the optimizations and describe generation of GXL to provide a familiar point of access to the tool chain. Finally, in Section 3.3 we describe our approach to validation of the generated GXL.

3.1 Overview of the g^4re system

Figure 2 provides an overview of the g^4re tool chain, partitioned into three phases: (1) ASG Initialization, (2) ASG Construction, Transformation and Serialization, and (3) ASG Validation. Each partition in Figure 2 is separated by a vertical dashed line. The first phase, ASG Initialization, is illustrated on the left side of the figure where the C++ source code representation of the application under study, together with the stub headers,

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

are used as input to the *gcc* compiler. Using the `-fdump-translation-unit-all` option, the output for this first phase is a text file for each translation unit (`tu`), leading into the second stage of the g^4re system.

The second stage of our tool chain is illustrated in the middle partition of Figure 2, where the `tu` files, shown as rectangles in the upper left of the middle partition, are used as input to the g^4re transformation system, the TUxformer subsystem. The `tu` files, together with a listing of the names of the source files for the application, shown as a single dashed rectangle, are used as input to the TUxformer, shown as a solid rectangle on the right side of the middle partition. The TUxformer performs three actions referred to in the figure as TUparse, TUprune and TUgxl. These three actions include parsing the `tu` file and re-creation of an ASG, optimization of the ASG through transformation, and generation of a GXL representation of the optimized ASG. Additional detail about the TUxformer subsystem is provided in Section 3.2.

The third and final phase of our current g^4re tool chain, ASG Validation, is shown on the right side of Figure 2. In this phase, we use two tools, our GxlSW and the publically available *GXL Validator* [1], to validate the GXL instance graphs output by our TUxformer system. Input to our GxlSW, represented by the three tabbed rectangles in the upper left of the rightmost partition, is `GENERIC` domain information. The GXL schema output by GxlSW, along with the GXL instance graphs and the GXL DTD [11], constitutes the input to the *GXL Validator*. Additional detail about ASG Validation is provided in Section 3.2.

3.2 The TUxformer subsystem

The TUxformer subsystem of the g^4re tool chain reads a *gcc* generated `tu` file and performs the three actions reviewed in the previous section, including parsing the `tu` file and re-creation of an ASG, optimization of the ASG through transformation, and generation of a GXL representation of the optimized ASG. Our prototype of the TUxformer subsystem is written in Python, which is ideal for the kind of text-processing that we require [32]. Python's rapid prototyping facility and ease of interoperability with other tools were also important factors. In this section, we provide detail about the TUxformer subsystem.

3.2.1 ASG Transformation: TUparse

The TUparse module of the TUxformer subsystem provides functionality to parse an input `tu` file and re-create a corresponding ASG. The TUparse module also performs the first stage of our second size reduction optimization, pruning the ASG. In this first stage of pruning the ASG, *removing extraneous fields*, we remove from each node fields that contain internal information used by the *gcc* compiler. To explicate our actions and to enable other researchers to reproduce our results, we describe the details of extraneous field removal in Stage 1.

Stage 1 Remove Extraneous Fields

Input: n a node under construction by TUparse

```
1: procedure REMOVE-FIELDS(n)
2:     F_A ← {'addr','algn','lngt','prec','size'}
3:     F_E ← {'max ','min ','purp'}
4:     foreach field f ∈ F[n] do
5:         if f ∈ F_A ∪ F_E then
6:             F[n] ← F[n] - {f}
```

Stage 1 captures the important actions in removing extraneous fields from an ASG re-created from a `tu` file. In line 1 of Stage 1 we begin REMOVE-FIELDS, a **procedure** that takes one input, n, a node under construction by TUparse. In lines 2 and 3 we create two sets to describe the kinds of extraneous fields encountered in re-creating an ASG: F_A and F_E. The set F_A contains attribute fields and the set F_E contains edge fields; collectively, these are the kinds of fields that we delete from the nodes of an ASG. In line 4 of Stage 1 we consider each field f of node n. In line 5 of Stage 1 we consider if the kind of f is in either of the two sets, F_A or F_E, and if so we remove the field f from the node n. In removing these fields, we may be removing the only reference to another node in the ASG. In the next section we describe the actions of TUprune, which prunes extraneous nodes and edges from the remaining reachable nodes of the ASG.

3.2.2 ASG Transformation: TUprune

The TUprune module of the TUxformer subsystem provides functionality to transform the ASG re-created by TUparse and consitutes the second stage of our second size reduction optimization. In this second stage of pruning the ASG, *node and edge removal based on source file, name, and reference*

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

Stage 2 Remove Nodes and Edges based on Source File, Name, and Reference Count

Input: *ASG* the ASG constructed by TUparse
Input: *UserCode* the list of user code source files
Output: P the set of nodes to prune from *ASG*

```
1: function SELECT-NODES(ASG,UserCode)
2:     P ← ∅
3:     R_@1 ← BFS(ASG,'@1')
4:     foreach node n ∈ V[R_@1] do
5:         if SRC-FILE(n) ∈ UserCode then
6:             continue
7:         if IS-SELECTED-NODE(n) then
8:             P ← P ∪ {n}
9:     return P
```

Input: n a reachable ASG node
Output: boolean indicating if the node is selected

```
10: function IS-SELECTED-NODE(n)
11:     name ← NAME(n)
12:     isMangled ← IS-MANGLED(name)
13:     S ← {'__comp_ctor', '__comp_dtor'}
14:     isInternal ← name[0:2] = '__' ∧name ∉ S
15:     if isMangled ∨ isInternal then
16:         return TRUE
17:     if IN-DEGREE(n) ≠ 1 then
18:         return FALSE
19:     E_chain ← {'chan', 'dcls', 'flds', 'fncs'}
20:     if TYPE-OF(edge(src,n)) ∈ E_chain then
21:         return TRUE
22:     return FALSE
```

count information, we select nodes from the reachable graph to be pruned based on the aforementioned information.

Stage 2 captures the important actions in selecting nodes for removal from an ASG based on source file, name, and reference count information. In line 1 of Stage 2 we begin SELECT-NODES, a **function** that takes two inputs: *ASG*, the ASG re-created by TUparse, and *UserCode*, the list of user code source files (shown as input to the system in Figure 2). In line 2 we initialize P, the set of nodes to prune from *ASG* and the output of SELECT-NODES, to $\emptyset$. In line 3 of Stage 2 we use a breadth-first search of *ASG* to compute $R_{@1}$, the subgraph of *ASG* that is reachable from the initial node, which we designate as `'@1'`. The **foreach** loop on line 4 considers each node n in the set $V[R_{@1}]$, the nodes reachable from `'@1'`. In lines 5 and 6 we continue to the next iteration of the **foreach** loop if the source file of n, computed by SRC-FILE, is in *UserCode*. In line 8 of Stage 2 we conditionally add n to P, the set of nodes to be pruned, based on the return value of the line 7 call to IS-SELECTED-NODE, a **function** that indicates if node n is selected for pruning. We return the set P on line 9, the final line of SELECT-NODES.

In line 10 of Stage 2 we begin IS-SELECTED-NODE, a **function** that takes one input, n, a reachable ASG node, and returns a boolean indicating if n is selected for pruning. The variable *name* on line 11 is a string that contains the name of the node under consideration. The variable *isMangled* on line 12 is a boolean that is set if *name* is mangled. In line 13 of Stage 2 we construct a *safe set of nodes*, S, consisting of the names that *gcc* assigns to programmer provided constructors and destructors. The variable *isInternal* on line 14 is a boolean that is set if the name is an internal name used only by *gcc*. Lines 15 and 16 return `TRUE` if the name under consideration is either mangled or internal. Lines 17 and 18 return `FALSE` if the in-degree of the node under consideration is not equal to 1. In line 19 we initialize set E_{chain} to the set of field attributes *'chan', 'dcls', 'flds', 'fncs'*, the field attributes that can constitute the beginning of a chain of nodes in a `GENERIC` ASG. In lines 20 and 21 we return `TRUE` if the type of the exclusive edge with destination n is in E_{chain}. Finally, on line 22 we return the fall-through value, `FALSE`.

3.2.3 ASG Serialization: TUgxl

The TUgxl module provides methods to perform ASG serialization, i.e. to convert the in-memory ASG to a GXL instance graph stored on disk. TUgxl takes as input the pruned ASG that is output by TUprune and produces a GXL instance graph that complies to the GXL schema graph described in Section 3.3.

3.3 Validation of the generated GXL

One advantage in using an XML technology such as GXL is the outstanding tool support provided by the community. This level of support is due in part to the ease with which an XML processor can be implemented. In this section we describe GxlSW, a system to automatically generate a valid GXL schema graph given a plain-text, simplified UML class diagram and domain type definitions.

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

We have written a collection of Perl modules, GxlSW, to automate the construction of a GXL schema graph for a schema, such as `GENERIC`, given only minimal input. To create our first GxlSW input we reverse engineered a plain-text UML class diagram for `GENERIC` by collecting data from the `tu` files generated by *gcc*. To regenerate as much of the *gcc* `GENERIC` schema as possible, we require a large and varied test suite; thus, we use the C/C++ testsuite included with *gcc* and an extensive C++ testsuite [21] extracted from the ISO C++ standard [14]. The second input, domain type information, consists of two small (approximately 10 line) files that provide mappings from the domain types to GXL primitive types.

We perform, using GxlSW, a direct translation from the simplified UML class diagram to a GXL schema. Using this technique, we gain two distinct advantages over other systems using `GENERIC`. First, the cognitive burden on a reverse engineer who chooses to use the GXL generated by our g^{4}re toolset is reduced, because said user needs only to understand the `GENERIC` ASG schema and not an adaptation of the schema. Second, the implementation of our tool does not require a set of mappings from the `GENERIC` ASG schema to an adapted schema; therefore, the implementation is more flexible with respect to changes to `GENERIC`. *XOGASTAN* [3] uses the adaptation technique, and as a result requires mappings for each node type in the `GENERIC` schema. While performing the experimentation presented in Section 4, we found that *XOGASTAN* is unable to create GXL for certain `GENERIC` node types, including `try_catch_expr` and `using_directive`.

The *GXL Validator* [1] validates a GXL graph against the GXL DTD, the specified GXL schema graph and additional constraints that cannot be expressed by the GXL DTD [11]. We use the *GXL Validator* to demonstrate the compliance of both the TUgxl generated GXL instance graphs to the `GENERIC` GXL schema and the `GENERIC` GXL schema to the GXL metaschema [11]. Generating valid GXL is important, because valid GXL files are more likely to be accepted by available XML tools than non-compliant files.

4 Experiments

In this section we describe the results of a feasibility study of our *g^{4}re* tool chain executed on a workstation with an *AMD Athlon64 3000+* processor, 1024 MB of PC3200 DDR RAM, and a 7200 RPM SATA hard drive, running the Slackware 10.0 operating system. One issue involved in using the *gcc* `GENERIC` system for construction of an ASG is the large size of the files required to store each translation unit (`tu`), and the large number of nodes in the generated ASG. To reduce the size of the generated ASG, we exploit two optimizations: removing extraneous library code and pruning the ASG. In this section we describe some results of these optimizations on a testsuite of application programs.

In the next section we describe five applications that serve as a testsuite in the study. In Section 4.2 we describe the results of the first optimization, the use of stub headers to remove extraneous code from the *gcc* library header files. In Section 4.3 we summarize the results of the second optimization: pruning nodes from the constructed abstract semantic graph (ASG). Finally, in Section 4.4 we compare the time efficiency of our *g^{4}re* tool chain to that of *XOGASTAN*, as well as the space efficiency of the GXL files created by each system.

4.1 The Testsuite

Table 1 lists the five applications, or test cases, that form the testsuite that we use in our study, together with important statistics about each test case. The top row of the table lists the names that we use to refer to each of the test cases: *Doxygen*, *FluxBox*, *FOX*, *Jikes* and *Keystone*. *Doxygen* is a documentation system for C++, C, and Java [31] and *FluxBox* is a light-weight X11 window management system built for speed and flexibility [8]. *FOX* is a toolkit to facilitate development of graphical user interfaces [30] and *Jikes* is a Java compiler system [13]. The final test case is *Keystone*, a parser and front-end for ISO C++ [15, 20]. The testsuite covers a range of applications including a system for documentation, libraries for graphical user interfaces (GUI), and applications for language implementation and analysis.

The remaining five rows of data in Table 1 describe important details of the test cases. The second row of the table lists the version number and the third row lists the number of source files in each of the test cases. For example, the *FOX* toolkit is version 1.4.6 and includes 474 source files, the largest number of source files for any of the test cases. The

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

Testsuite	Doxygen	FluxBox	FOX	Jikes	Keystone
Version	1.3.9.1	0.9.12	1.4.6	1.22	0.2.3
Source Files	260	229	474	74	113
Translation Units	122	104	245	38	52
Size	7.76 MB	1.93 MB	6.06 MB	3.33 MB	1.38 MB
LOC (≈)	200 K	30 K	125 K	70 K	30 K

Table 1. *Testsuite*. **This table lists the five testcases that we use in our study, together with statistics about the test cases. Our statistics describe the number of source files, translation units, size, and lines of code for each test case.**

`tu` size		Doxygen	FluxBox	FOX	Jikes	Keystone
Lines	STL	14 034 612	23 363 090	25 815 273	13 932 339	12 240 694
	Stub STL	14 034 612	15 988 900	25 815 273	11 036 707	7 452 885
	Diff	0	7 374 190	0	2 895 632	4 787 809
Nodes	STL	7 675 555	11 364 192	11 881 617	7 455 700	6 072 635
	Stub STL	7 675 555	7 016 962	11 881 617	5 797 562	3 256 285
	Diff	0	4 347 230	0	1 658 138	2 816 350

Table 2. *Stub Headers* **The data illustrated in this table provides a comparison of the savings accrued using stub headers rather than the full header files in the standard C++ library.**

fourth row of the table lists the number of translation units constituting each test case and the fifth row lists the size or number of mega-bytes that each test case occupies on disk. For example, the *FOX* toolkit contains 245 translation units, the largest number of translation units, and occupies 6.06 MB of disk space. Finally, the last row of Table 1 lists the number of lines of code in each test case, expressed in thousands of lines of code. For example, the *Doxygen* documentation system contains 200 K lines of code (LOC), the largest number in the test suite.

4.2 Removing extraneous library code

One of the issues involved in using the `GENERIC` system for construction of an ASG is the large size of the files required to store each translation unit (`tu`), and the large number of nodes and edges in the generated ASG. To illustrate this size issue, consider that a Hello World program generates a `tu` file of ten megabytes (10.2 MB) and a lightweight email client, *Mozilla Thunderbird*, generates a set of files 18 gigabytes in size. To reduce the number of nodes and edges in the generated ASG, we exploit the two optimizations described in Section 3.

The results in Table 2 summarize the savings accrued using stub headers rather than the full header files in the standard C++ library. The table is partitioned into three sections: the first partition is the top row of data listing the testcases, the second partition is the middle three rows comparing the number of lines in the full `tu` file with the number of lines in a `tu` file when stub headers are used, and the third partition is the final three rows of data comparing the number of nodes in the ASG constructed when the full library files are used with the number of nodes in the ASG constructed when stub headers are used.

The second partition is illustrated in rows two, three and four of Table 2, showing the number of lines in the `tu` file when the full header files are used, the number of lines in the `tu` file when stub headers are used and the difference between these two values, respectively. The first test case in the table lists results for the *Doxygen* documentation system, which does not use the standard C++ library. With *Doxygen*, the `tu` file consumes 14 034 612 lines of code when the full headers are used and the same number of lines when the stub headers are used.

The second test case in Table 2 lists results for the *Fluxbox* window manager, which uses the standard C++ library. With *Fluxbox*, the `tu` files consume 23 363 090 lines when the full header files are used

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

		Doxygen	FluxBox	FOX	Jikes	Keystone
Nodes	Before	7 675 555	7 016 962	11 881 617	5 797 562	3 256 285
	After	5 133 305	2 221 299	6 967 120	4 162 046	880 350
	Diff	2 542 250	4 795 663	4 914 497	1 635 516	2 375 935
Edges	Before	13 350 444	12 745 573	22 785 781	10 425 650	5 769 231
	After	9 512 107	4 278 372	13 680 285	7 819 724	1 749 339
	Diff	3 838 337	8 467 201	9 105 496	2 605 926	4 019 892

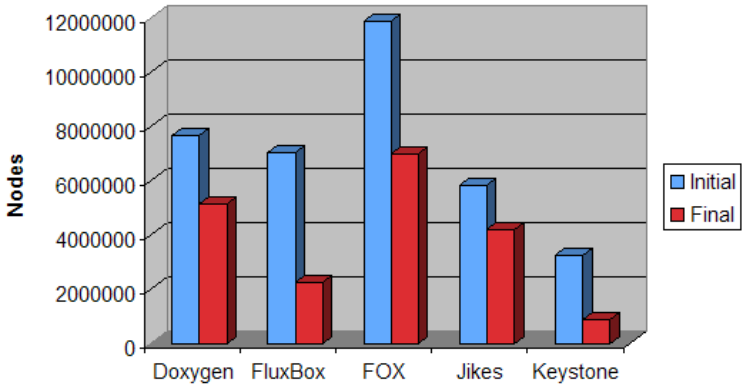

Figure 3. *Summary of size reduction optimizations.*

and 15 988 900 lines when the stub headers are used: a savings of 32 percent. The fifth test case in the table lists results for the *Keystone* parser, which makes heavier use of the standard C++ library. With *Keystone*, the `tu` files consume 12 240 694 lines when the full header files are used and 7 452 885 lines when the stub headers are used: a savings of 40 percent.

Table 2 demonstrates that the savings accrued using stub headers for the C++ standard library are dependent on the source code characteristics of each application. The first test case, *Doxygen*, does not use the C++ standard library, but rather a combination of the *Qt* library and the C standard library. For this reason, the use of stub headers yields no size reduction for the `tu` files generated for *Doxygen*. The third test case, *FOX*, uses only the C standard library, and again we see no size reduction for its `tu` files. The fourth test case, *Jikes*, utilizes the C++ standard library exclusively for input and output; thus, the use of stub headers does yield a size reduction for this test case. Finally, test cases two and five, *FluxBox* and *Keystone*, make extensive use of the C++ standard library, including classes providing input, output, containers, and algorithms, and as a result the use of stub headers yields substantial size reductions in both cases.

4.3 Removing extraneous fields and nodes

In this section we describe the results of pruning the ASG, the second size reduction optimization we perform on the *gcc* generated ASG. It would be easier to simply "filter" non-essential nodes when reading the `tu` files. However, non-essential nodes can appear among essential nodes in the `tu` file. Thus, the non-essential nodes must be "unlinked" from the ASG, with essential nodes relinked so that the optimized ASG remains a single connected graph.

The algorithm for *field removal*, Stage 1, in Section 3.2.1, summarizes our actions in removing extraneous fields from the ASG. These fields hold internal information that the *gcc* compiler uses to facilitate recognition or optimization of the input C++ program. Since they are not needed in analysis of the reverse-engineered code, we remove them. After removing these fields, there are nodes and subtrees that are no longer connected to the ASG. The Stage 2 algorithm, in Section 3.2.2, summarizes the actions taken on the remaining reachable graph to perform further size optimization.

We observe that, as is the case with the use of stub headers, the savings accrued using our two

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

stage ASG pruning are application dependent, as demostrated by Figure 3. For three of our test cases, *FluxBox*, *FOX*, and *Keystone*, substantial savings accrue using our pruning algorithm. These savings result from both the heavy uses of the C and C++ standard libraries, and the large amount of library code that is included but not referenced by user code. For *Doxygen* we observe a moderate savings accrued by the pruning algorithm. *Doxygen* uses the *Qt* library and references a large portion of the included code; therefore, there are savings, but to a lesser degree. Finally, *Jikes* makes only light use of the C++ standard library, and provides wrapper classes for the functionality it uses. These wrapper classes ensure that nearly all of the included library code is referenced. Thus, the savings accrued for the *Jikes* test case are the least significant for any of our test cases.

The table and graph in Figure 3 summarize the space saved using the algorithms in Stages 1 and 2. The last five columns of data in the table at the top of Figure 3 list results for each of the five test cases before and after these two stages. The rows of data in the table can be partitioned into two sets of data: the first three rows list optimization results for nodes and the last three rows list optimization results for edges removed from the generated ASG.

For example, the column of data labeled *Doxygen* and the first row of data illustrates that the number of nodes in the ASG before field and node removal is 7 675 555; the second row of data shows that the number of nodes after removal is 6 679 399, for a total edge savings of 996 156. The third row of data, labeled Diff, lists this total node savings. The column of data labeled *Doxygen* and the fifth row illustrate that the number of edges in the ASG before field and node removal is 13 350 444 and the number of nodes after removal is 12 623 626 for a total edge savings of 726 818. The third row of data, labeled Diff, lists this total edge savings.

The bar graph in Figure 3 consists of five pairs of bars, graphically illustrating the total node savings accrued from pruning the ASG. The first bar in the pair is the number of nodes in the ASG before pruning and the second bar in the pair is the number of nodes in the ASG after pruning. For example, the first pair of bars illustrates node savings for *Doxygen* and the second set of bars illustrates node savings for *FluxBox*. The third set of bars illustrates the node savings for *FOX*, the improvement in total number of nodes removed. The final two sets of bars illustrate node savings for *Jikes* and *Keystone*, respectively.

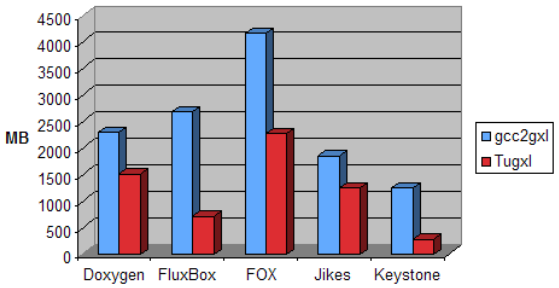

Figure 4. *Space comparison*.

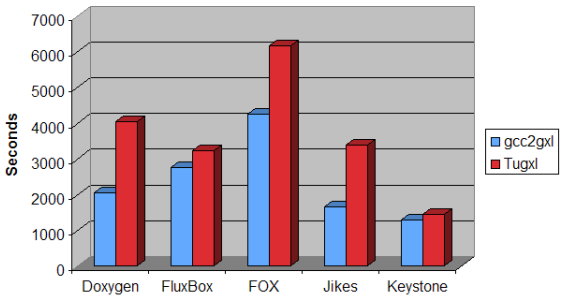

Figure 5. *Time comparison*.

4.4 Comparison of gcc2gxl and TUgxl

In this section we present a comparison of our g^4re tool chain to the gcc2gxl subsystem of *XOGASTAN* [3]. Specifically, we compare two properties of the GXL output of each system: the time required to generate the GXL, and the space occupied on disk by the GXL. The two systems that we compare are written in different languages: the g^4re tool chain is written in Python [32] and the *XOGASTAN* system is written in Perl [33]. Version 2.3.4 of the Python interpreter and version 5.8.4 of the Perl interpreter were, respectively, used to execute the systems. We obtained our results using `tu` files generated using stub headers and the testsuite described in Table 1.

Figure 4 illustrates some space comparisons for the GXL produced for each of the five test cases. The figure consists of five pairs of bars, where the first bar in the pair is the space occupied by the GXL output of *XOGASTAN* and the second bar in the pair is the space occupied by the GXL output of g^4re. The figure further illustrates that for the

FluxBox, *FOX* and *Keystone* test cases, the output of g^4re occupies considerably less space, and the average space improvement of g^4re over *XOGASTAN* is 53%. This improvement results entirely from our removal of extraneous nodes and fields from the generated ASG, as described in Section 3.

Figure 5 illustrates some timing comparisons for each of the five test cases. The figure consists of five pairs of bars, where the first bar in the pair is the time requirements for *XOGASTAN* and the second bar in the pair is the time requirements for g^4re. Here we see that *XOGASTAN* is consistently faster than g^4re and is considerably faster for the *Doxygen* and *Jikes* test cases.

5 Related Work

Antoniol, et al. present a toolset, *XOGASTAN*, that is similar to our g^4re tool chain [3]. *XOGASTAN* includes tools to convert a *gcc* `tu` file to a GXL instance graph and to construct an in-memory representation of the GXL instance graph [12]. However, *XOGASTAN* does not include facility to reduce the ASG, resulting in large GXL instance graphs with extraneous information that is not useful to the user of the toolset.

Gschwind, et al. present *TUAnalyzer*, a system that is complementary to g^4re [9]. The focus of *TUAnalyzer* is on using a *gcc* `tu` file to perform analysis of template instantiations of functions and classes. *TUAnalyzer* performs virtual method resolution by using the '`base`' and '`binf`' attributes, along with the output provided by the compiler switch `-fdump-class-hierarchy`, to reconstruct the virtual method table. However, the functionality of the tool has a narrow scope and does not include a representation of the *gcc* `tu` file for exchange with other reverse engineering tools.

The *GCC.XML* toolset also uses the *gcc* `tu` file approach [2]. *GCC.XML* generates an XML representation for class, function, and namespace declarations, but does not propogate information such as function and method bodies. As a result, many common program representations, such as the call graph or the ORD, cannot be constructed using *GCC.XML*.

The *CPPX* tool relies on *gcc* for parsing and semantic analysis [7]; however, it predates `GENERIC` and is built directly into the *gcc* code base. *CPPX* constructs an ASG that is compliant to the Datrix ASG Schema [4] and can be serialized to GXL, TA, or VCG format. The Datrix ASG Schema is more general than the `GENERIC` schema to accomodate C++ and other languages; however, this generality makes it difficult to accurately represent many C++ language constructs.

gccXfront In previous work we developed an approach to annotate source code with syntactic tags in XML based on modifying the *bison* parser generator tool [22]. We exploited this approach in developing the `gccXfront` tool, which harnessed the *gcc* parser to tag C and C++ source code [10]. However, since the migration of the *gcc* C++ compiler to recursive descent technology, this latter approach is no longer directly applicable.

6 Conclusions

In this paper, we have described g^4re, our tool chain that exploits `GENERIC`, an intermediate format incorporated into the *gcc* C++ compiler, to facilitate analysis of real C++ applications. Since `tu` files can be prohibitively large, we also describe our approach for reducing the size of the generated ASG using transformations on the ASG. Using the transformations, we were able to reduce the average size of the generated ASGs for our test suite by 53%.

We have used g^4re for modeling and visualizing the dynamic interactions among objects in a C++ application [23]. Our visualizer uses UML graphs and diagrams, typically used during the requirements and design phases of the life cycle, to expressively visualize both the static and dynamic properties of the application under study. The visualizer is especially useful for comprehension and debugging of graphical applications such as games and GUIs.

As part of our ongoing research, we are using g^4re to build Object Relation Diagrams (ORD) [19], and to compute a firewall for a modified class in the ORD. We are also in the process of incorporating an Application Programmers Interface (API) into g^4re to provide accessibility to its functionality while relieving the user of the burden of understanding the intricacies of the `GENERIC` schema or the generated ASG.

Presented at "Transformation techniques in software engineering"
Daghstuhl seminar # 05161, April 17-22, 2005.

References

[1] GXL Validator. http://www.uni-koblenz.de/FB4-/Contrib/GUPRO/Site/Downloads/index_html-?project=gupro_all, January 2003.

[2] GCC-XML. http://www.gccxml.org, February 2005.

[3] G. Antoniol, M. D. Penta, G. Masone, and U. Villano. XOGastan: XML-oriented GCC AST analysis and transformation. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation*. IEEE, 2003.

[4] Bell Canada Inc. *DATRIX - Abstract Semantic Graph Reference Manual*. Bell Canada Inc., Montreal, Canada, 1.4 edition, May 2000.

[5] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *The second annual object-oriented numerics conference (OON-SKI)*, pages 122–136, Sunriver, Oregon, USA, 1994.

[6] S. F. Cohen. Quest for Java. *Communications of the ACM*, 41(1):81–83, January 1997.

[7] T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a c++ extractor. In *Proceedings of the Eighth Working Conference on Reverse Engineering*. IEEE, October 2001. www.cppx.com.

[8] Fluxbox Project. FluxBox version 0.9.12. Available at http://www.fluxbox.org.

[9] T. Gschwind, M. Pinzger, and H. Gall. TUAnalyzer - analyzing templates in C++ code. In *Proceedings of the Eleventh Working Conference on Reverse Engineering*. IEEE, 2004.

[10] 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 International Workshop on Program Comprehension*, pages 298–299, Portland, Oregon, USA, May 2003. IEEE.

[11] R. Holt, A. Schürr, S. E. Sim, and A. Winter. GXL - Graph eXchange Language. http://www.gupro.de/GXL, January 2003.

[12] R. C. Holt, A. Walter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 162–171. IEEE, November 2000.

[13] IBM Jikes Project. Jikes version 1.22. Available at http://jikes.sourceforge.net.

[14] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[15] Keystone Project. Keystone version 0.2.3. Available at http://keystone.sourceforge.net.

[16] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. Draft, Submitted for journal publication; Online since July 2003, 47 pages, Feb.22 2005.

[17] G. Knapen, B. Lague, M. Dagenais, and E. Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, May 5-7 1999.

[18] J. Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.

[19] B. A. Malloy, P. J. Clarke, and E. L. . Lloyd. A parameterized cost model to order classes for integration testing of C++ applications. In *Proceedings of the 14th International Symposium on Reliability Engineering (ISSRE '03)*, pages 353–364, Los Alamitos, CA, Nov. 2003. IEEE Computer Society Press.

[20] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 33(1):19–39, 2003.

[21] B. A. Malloy, T. H. Gibbs, and J. F. Power. Progression toward conformance for C++ language compilers. *Dr. Dobbs Journal*, pages 54–60, November 2003.

[22] B. A. Malloy and J. F. Power. Program annotation in XML: A parser-based approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 190–198, Richmond, Virginia, USA, October 2002. IEEE.

[23] B. A. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualization (SoftViz'05)*, St. Louis, MO, May 2005. (to appear).

[24] S. McPeak. Elkhound: A Fast, Practical GLR Parser Generator. Technical Report UCB/CSD-2-1214, Berkeley University, Apr.2 2005. Technical Report.

[25] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *37th International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS Pacific 2000)*, pages 57–68, Sydney, Australia, November 2000.

[26] S. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.

[27] J. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.

[28] S. Singhal and B. Nguyen. The Java factor. *Communications of the ACM*, 41(6):34–37, June 1998.

[29] P. Tyma. Why are we using Java. *Communications of the ACM*, 41(6):38–42, June 1998.

[30] J. van der Zijp. The FOX Toolkit Library version 1.4.6. Available at http://www.fox-toolkit.org.

[31] D. van Heesch. Doxygen version 1.3.9.1. Available at http://stack.nl/ dimitri/doxygen.

[32] G. van Rossum. *Python Library Reference*. Python Software Foundation, 2001.

[33] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl: Third Edition*. O'Reilly & Associates, third edition, 2000.