A Tool Chain for Reverse Engineering C++ Applications

Nicholas A. Kraft[a], Brian A. Malloy[a] and James F. Power[b]

[a]Department of Computer Science, Clemson University,
Clemson, SC 29634, USA

[b]Department of Computer Science, National University of Ireland, Maynooth,
Maynooth, Ireland

In this paper, we describe our tool chain that exploits the gcc *C++ compiler, to enable experimentation and study of real C++ applications. Our tool accepts any C++ application that can be parsed by the* gcc *C++ front-end, including large language processing tools and gaming software. Our tool consists of a chain of applications that enables the user to access the tool at any point in the chain. The easiest point of access is at the end of the chain where an Application Programmers Interface (API) provides easy access to information about the names, classes, namespaces, functions or statements in the C++ application under study.*

1. Introduction

To improve the software development process, researchers must design and implement new techniques and verify that their work is an improvement over previously developed techniques. This verification requires that researchers conduct either controlled experiments or case studies that include the implementation of at least one previously developed technique as a basis of comparison with their own technique. Moreover, the experiments or studies must be conducted on a test suite of programs that include applications of all sizes and a variety of application domains. An important feature of the experimentation is that the newly developed result must be reproducible [12].

However, there are problems associated with the performance of these experiments or studies. First, it can be difficult or impossible to reproduce the results of previous research due to the difficulty of interpreting the previously developed algorithm or technique, with the concomitant lack of confidence in the generated results [10,25,35]. Second, experiments and case studies depend on numerous software-related artifacts, including software systems, such as parsers, and test cases that vary in size, application domain and complexity [12]. We address the first problem in previous research by describing an infrastructure to support interoperability in reverse engineering of C++ applications [24,25]. In this previous work, we describe a hierarchy of canonical schemas that capture minimal functionality for middle-level graph structures. The purpose of the hierarchy is to facilitate an unbiased comparison of experimental results for different tools that implement the same or a similar schema.

In this paper, we focus on the second problem by describing our tool chain that exploits the *gcc* C++ compiler, to enable experimentation and study of real C++ applications. Our tool accepts any C++ application that can be parsed by the *gcc* C++ front-end, including large language processing tools and gaming software [20]. Our tool consists of a chain of applications that enables the user to access the tool at any point in the chain. The preferred point of access is at the end of the chain where an Application Programmers Interface (API) provides easy access to information about the names, classes, namespaces, functions, function calls or statements in the C++ application under study. There are other points of access along the chain where early access points enable lower level access to the information about the program, but this low level access imposes a greater cognitive burden on the user of the tool due to the knowledge required about the details of the implementation of *gcc*.

In the next section we review the terminology and technologies that we use in the design and implementation of our tool chain. In Section 3 we present details about the tool chain and in Section 4 we present some results in using the tool to compute metrics for evaluation of object-oriented applications. In Section 5 we compare our tool to similar systems and in Section 6 we draw conclusions and describe our ongoing work.

2. Background

In this section we review terminology and the major technologies that we use in the design and implementation of the g^4re tool chain. In Section 2.1 we review `GENERIC`, the ASG schema of *gcc* that has recently been utilized by several research tools for program analysis and reverse engineering [2,5,14,20,32,33]. In Section 2.2 we review GXL, the XML-based exchange format 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 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 include an ASG schema known as `GENERIC` [34].

The *gcc* ASG schema, `GENERIC`, consists of over 200 node types whose documentation consists 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 information in a `tu` file, illustrated in Figure 1, can be extracted by a parser and used 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,
- edge tuples consisting of the edge name and a unique identifier for the destination node,

```
@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 the representation of nodes in a `tu` file.

- 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

An important aspect in the design of a reverse engineering tool is the selection of an exchange format that facilitates representation and sharing of the information. Currently, GXL (Graph eXchange Language) is the standard exchange format used by reverse engineering tools [17]. GXL is an XML sublanguage defined by an XML DTD (Document Type Definition) and conceptualized as a typed, attributed, directed graph. GXL is used to describe both instance data and its schema; schemas in GXL are represented by UML class diagrams [17].

3. Description of the Tool

In the sections that follow, we describe the design and implementation of g^4re. In Section 3.1 we describe the TUxformer subsystem, which performs construction, transformation, and serialization of instances of the `GENERIC` ASG schema, as well as validation of the resulting GXL instance graphs. The TUxformer subsystem is illustrated in Figure 2. In Section 3.2 we describe the CppInfo API subsystem, which is responsible for transforming GXL instances of the `GENERIC` ASG schema into instances of the CppInfo API schema, as well as linking the API instances. The CppInfo API subsystem is illustrated in Figure 3.

3.1. The TUxformer subsystem

Figure 2 provides an overview of the TUxformer subsystem, partitioned into three phases: (1) ASG Generation, (2) ASG Construction, Transformation and Serialization, and (3) ASG Validation. A dashed vertical line separates each partition in Figure 2. The leftmost partition of the figure illustrates the first phase, ASG Generation, where we use the C++ source code representation of the application under study as input to the *gcc* compiler. Using the `-fdump-translation-unit-all` option, we obtain a text file containing the ASG for each translation unit (`tu`).

The center partition of Figure 2 illustrates the second stage, where we provide the `tu` files, shown as rectangles in the upper left of the center partition, as input to TUxformer. TUxformer, shown as a solid rectangle on the right side of the center partition, is written in Python, a language ideal for the kind of text-processing we require [41]. TUxformer

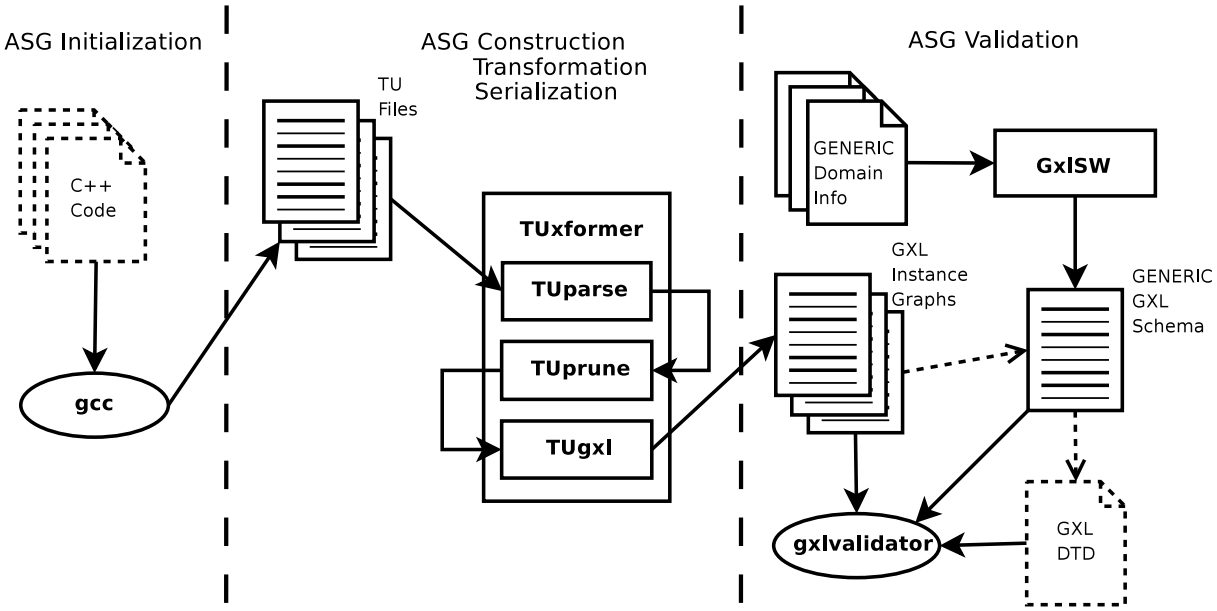

Figure 2. *System Architecture Part I.* This figure illustrates the phases in the TUxformer subsystem of g^4re. This subsystem creates and validates a GXL instance graph for each translation unit in a C++ program. User inputs are shown as tabbed, dashed rectangles; external programs, e.g. *gcc*, are 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 is shown as solid edges with solid arrows and conformance is shown as dashed edges with open arrows.

consists of three modules: TUparse, TUprune and TUgxl. These three modules perform the following actions, respectively: parsing the `tu` file and re-creating the ASG, reduction of the ASG size through transformation, and generation of a GXL representation of the reduced ASG.

The rightmost partition of Figure 2 illustrates the third and final phase of the TUxformer subsystem. In this phase, we use two tools, our GxlSW and the publically available *GXL Validator* [1], to validate the GXL instance graphs that the TUxformer program produces as output. Input to our GxlSW, represented by the three tabbed rectangles in the upper left of the rightmost partition, is `GENERIC` domain information. GxlSW produces a GXL schema that we use, along with the GXL instance graphs and the GXL metaschema [16], as input to the *GXL Validator*.

3.1.1. ASG Construction: TUparse

The TUparse module of the TUxformer subsystem provides functionality to parse an input `tu` file and re-create the corresponding ASG. The TUparse module also performs the first stage of our size reduction optimization, pruning the ASG. In this first stage, *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 Algorithm 1.

Algorithm 1 Remove Extraneous Fields

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

Algorithm 1 captures the important actions in removing extraneous fields from an ASG re-created from a `tu` file. In line 1 of Algorithm 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 Algorithm 1 we consider each field f of node n. In line 5 of Algorithm 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.

3.1.2. ASG Transformation: TUprune

The TUprune module of the TUxformer subsystem provides functionality to transform the ASG re-created by TUparse and constitutes the second stage of our second size reduction optimization. In this second stage, we remove nodes that are no longer in the reachable graph.

3.1.3. 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 3.1.4.

3.1.4. GXL Validation

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.

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++ test suite included with *gcc* and an extensive C++ test suite [30] extracted from the ISO C++ standard [19]. 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 tool set 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.

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 [16]. 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 [16]. Generating valid GXL is important because valid GXL files are more likely to be accepted by available XML tools than non-compliant files.

3.2. The CppInfo API subsystem

Figure 3 provides an overview of the CppInfo API subsystem. Input to the subsystem is a set of GXL instance graphs generated by the TUxformer subsystem. The user passes the graphs to the constructor of class ApiInterface, which instantiates the API using the g4xformer package. The g4xformer package contains the following modules:

- a *SAX2 parser* for creating an in-memory representation of a translation unit encoded as a GENERIC conformant GXL instance graph,
- a *transformation module* for creating a CppInfo API instance from the parsed representation of a translation unit,
- a *linking module* that combines API instances for all translation units in a program into a unified representation of the whole program.

In Section 3.2.1 we describe the CppInfo API, and in Section 3.2.2 we describe the algorithm for linking API instances.

3.2.1. The CppInfo API

The CppInfo API, Application Programmers Interface, provides access to information in the unified representation of a complete C++ program. The CppInfo API schema models the implementation of the CppInfo API. The schema currently consists of 42 node classes that represent declarations, scopes, types, control structures, and expressions. The addition of node classes to represent remaining expressions, such as mathematical and memory management operators, remains as future work.

The CppInfo API provides a clear and flexible interface for access to the language elements in a C++ program. The first point of access provided by the CppInfo API is in the form of a pointer to the global namespace. An API user may access the pointer in order to traverse the underlying graph structure of the CppInfo API, or alternatively, may

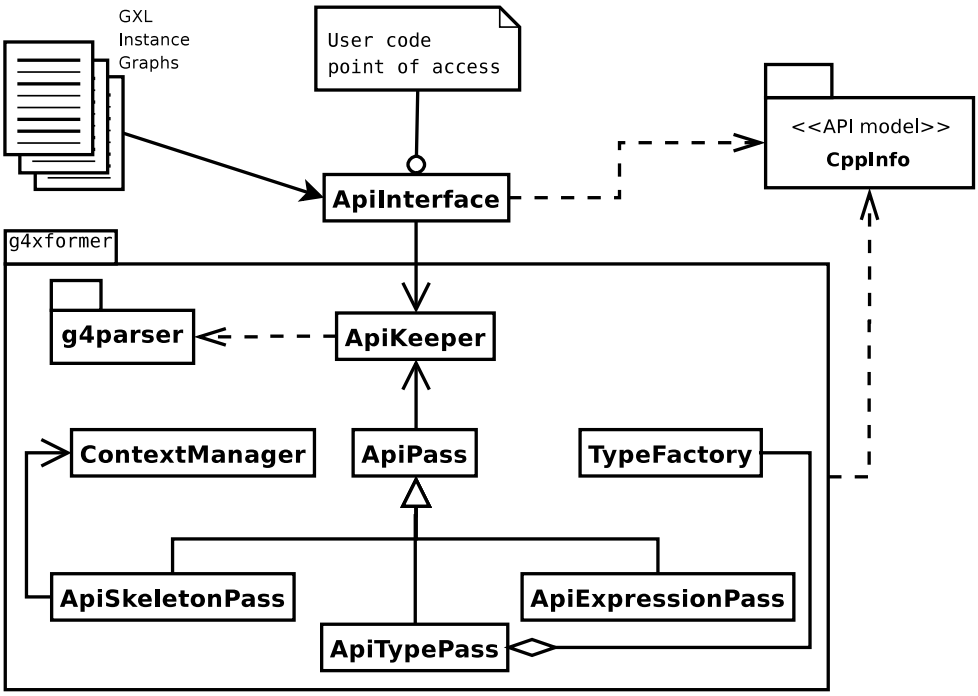

Figure 3. *System Architecture Part II.* This figure illustrates the structure of the CppInfo API subsystem. This subsystem transforms instances of the `GENERIC` ASG schema into instances of the CppInfo API schema; classes in the ApiPass hierarchy perform the transformations. This subsystem also links the API representations for each translation unit in a program; class ApiKeeper performs the linking.

---

use the second point of access, the list interface. An API user may access several lists containing all instances of particular CppInfo classes present in the API. Currently, the API provides these lists for Namespace, Class, Enumeration, Enumerator, Function, Variable, Typedef, and FunctionCall. Two lists are provided for each supported language element, a filtered list and an unfiltered list. The filtered lists are configured by the API user and provide the ability to exclude language elements based on the source file in which they are defined.

3.2.2. Linking API instances

Typical C++ programs are spread among tens, hundreds, or even thousands of files, both header and source. A *C++ translation unit* consists of a source file and all of the header files it includes, either directly or transitively. A C++ compiler, such as *gcc*, performs parsing, analysis, and code generation at the translation unit level; the system linker, *ld* on Unix systems, performs linking on the generated object code. The system linker must check for multiple definitions and inconsistencies, e.g. incompatible function declaration and definition, between translation units.

A reverse engineering tool for C++ must also perform parsing and analysis at the trans-

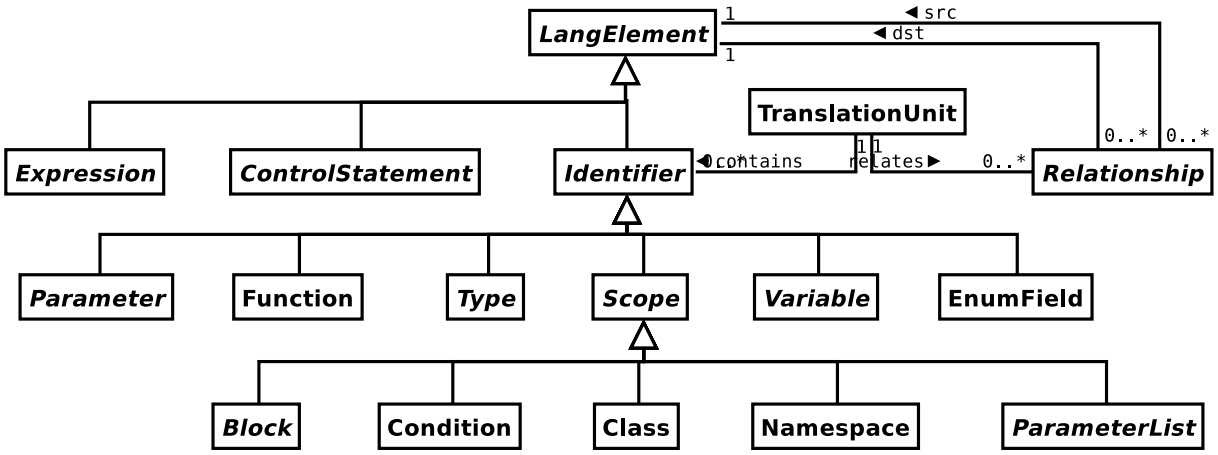

Figure 4. Partial Schema for the CppInfo API. *This figure illustrates some of the main node classes in the CppInfo API, which is used to represent a translation unit as well as the result of linking two translation units.*

lation unit level, but rather than generating code, a reverse engineering tool generates an ASG (or another program representation). Since reverse engineers are principally interested in analyzing whole programs, not individual translation units, a reverse engineering tool for C++ must provide some facility for linking the representations of the individual translation units. A reverse engineering linker may generally assume that the program being analyzed is both compilable and linkable at the object code level; therefore, linking at the ASG (or other program representation) level does not require error checking.

Unique names, such as a mangled names or fully-qualified names, enable a module, such as a stand-alone linker or an API builder, to link individual translation units into a unified representation of the whole program. In g^{4}re, linking is performed pairwise by the API builder on the internal representations of API instances. When instantiating the API, a user provides all translation units from a C++ program, each in the form of a GXL encoded ASG [1] conformant to the `GENERIC` schema. The API builder serially transforms each ASG to an internal API representation instance, consisting of dictionaries mapping unique names to their CppInfo API schema node class instances, and performs pairwise linking of the API instances each time a pair becomes available. Therefore, linking in g^{4}re is performed $n-1$ times, where n is the number of translation units.

Figure 4 illustrates part of the CppInfo API schema. Of central interest here is the TranslationUnit, which contains a set of identifier definitions and declarations, along with a set of relationships between these and the other language elements it contains. Intuitively, we achieve linking of schema elements by performing a traversal of the most recently constructed API instance, adding or appending elements in the existing API instance if they are not found or are incomplete. For example, the element Function is incomplete if one of its instances does not contain a body, while the elements Namespace and Class are

[1] *g^{4}re* is capable of reading gzipped GXL files in addition to plain-text GXL files.

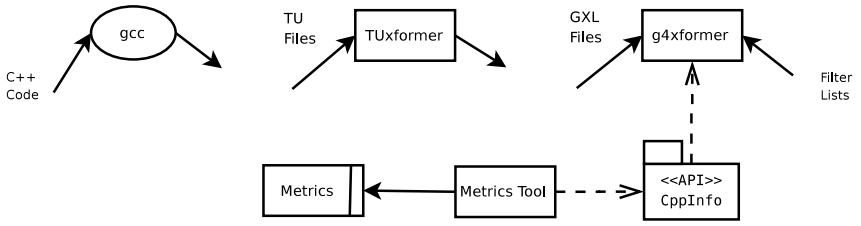

Figure 5. System overview. This figure illustrates the important components in our metrics computation system that we constructed to compute metrics for C++ applications. The metrics computation system consists of the g[4]re tool chain, including the CppInfo API, and a Metrics Tool that interacts with the API to extract information about a C++ program.

---

incomplete if they contain incomplete Function or Class elements.

4. Sample Tool Usage

In this section we review our metrics computation system that we use to evaluate the exploitation of object technology in game application software [20]. Our purpose is to illustrate one possible usage of the *g[4]re* tool chain, and we chose this example because it illustrates analysis of C++ applications at the level of the *namespace*, *class*, *method* and *statement*. All of the experiments were 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 10.1 operating system. The programs were compiled using *gcc* version 3.3.4.

In Section 4.1 we describe details of the metric computation system and its use of the g[4]re tool chain [24,25]. In Section 4.2 we describe the testsuite of applications including some popular game applications written in the Simple Directmedia Layer (SDL), and some language processing applications. In Section 4.3 we describe some results about the ability of game software to exploit the object-oriented methodology.

4.1. Overview of the Metrics Computation System

Figure 5 illustrates our metrics computation system, which consists of the g[4]re tool chain, including the CppInfo API, and a Metrics Tool that interacts with the API to extract information about a C++ program. Output of our system is a set of statistics for each computed metric.

Input to our system is the source code for a C++ program, shown in the far left of the top row of the figure, which is used as input to the *gcc* compiler. Using the `-fdump-translation-unit-all` option, we obtain a plain text representation of the ASG for each C++ translation unit in the program. We use these plain text ASG representations, known as `tu` files, as input to our TUxformer subsystem, shown in the middle of the top row of the figure. For each `tu` file, the TUxformer subsystem creates an in-memory representation of the encoded ASG, prunes the ASG, and serializes the ASG to GXL.

	SDL Game Applications				Language Processing Applications			
	ASC	AvP	Freespace2	Scorched3D	Doxygen	g^{4}re	Jikes	Keystone
Version	1.16.1.0	cvs	cvs	38.1	1.3.9.1	1.0.4	1.22	0.2.3
Source Files	436	509	652	1069	260	128	75	123
Translation Units	199	222	220	513	122	60	38	52
C++ Translation Units	194	95	220	492	90	60	38	52
LOC (≈)	130 K	318 K	365 K	110 K	200 K	10 K	70 K	30 K

Table 1
Testsuite of SDL Game Application Software and Language Processing Tools.

We use the set of GXL files produced by TUxformer as input to the g4xformer subsystem, shown in the far right of the top row of the figure. The g4xformer subsystem parses each GXL file and creates an in-memory representation of the encoded ASG. The subsystem then links the representations of each individual ASG to create a unified representation of the entire C++ program. After linking is complete, the subsystem filters language elements that are identified as defined in a filename contained in the filter lists, shown in the far right of the top row of the figure.

The CppInfo API, shown in the far right of the bottom row of the figure, provides access to information from the unified representation of a whole C++ program created by the g4xformer subsystem. Our Metrics Tool, shown in the middle of the bottom row of the figure, instantiates and queries the API to gain access to the information about classes and functions needed to compute the metrics. Output of the Metrics Tool, shown in the far left of the bottom row of the figure, is available in a variety of formats and consists of a set of statistics for each computed metric. The complete results of our study can be found in reference [20] and the metrics include information about the number of classes, methods, depth of inheritance, breadth of inheritance and complexity of the methods in the respective applications. In this paper, we only present results about the modularity and complexity of the respective applications.

4.2. The Test suite of Game Applications and Language Processing Tools

Table 1 lists eight applications, or test cases, that form the test suite that we use in our study, together with size 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. The game applications are listed in the first four columns and the language processing applications are listed in the last four columns. The four game applications are: *Allied Strategic Command* (ASC), *Alien vs Predator* (AvP), *Freespace 2* (Freespace2), and *Scorched 3d* (Scorched3D). The Application Programmer's Interface (API) used for the four games is the Simple Directmedia Layer (SDL), described in Section 2. The four language processing applications, listed in the last four columns of Table 1, are: *Doxygen*, g^4re, *Jikes*, and *Keystone*. *Doxygen* is a documentation system for C++, C, and Java [40] and g^4re is part of the infrastructure for reverse engineering that we use to construct our metrics tool [24,25]. *Jikes* is a Java compiler system [18] and *Keystone* is a parser and front-end for ISO C++ [21,29].

	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	561	12.9770	30.3646	4	0
AvP	0	107	7.2898	10.5944	3	3
Freespace2	0	123	6.6596	15.7072	3	3
Scorched3D	0	240	17.3717	19.0581	12	3
Doxygen	0	430	27.6762	57.4967	7	7
g^4re	0	206	17.7564	30.2694	13	0
Jikes	0	2016	32.3968	119.1240	13	10
Keystone	0	557	24.3875	52.4735	15	14

Table 2
Weighted Methods per Class.

The rows of Table 1 list some statistics and coarse-grained size metrics for the test cases: the first row lists the version number, Version; the second row lists the number of source files, Source Files, for each test case; the third row lists the number of translation units, Translation Units, which includes both C++ and C translation units; the fourth row lists the number of C++ translation units (C++ Translation Units), which is only C++ code; and finally, the last row of the table lists the (approximate) thousands of lines of code (KLOC) for each test case, not counting blank or comment lines. For example, the largest game in our test suite is Freespace 2, a Version that we obtained from a cvs repository (on July 22, 2005), consisting of 652 source files, 220 Translation Units and 220 C++ Translation Units. Since the number of Translation Units is the same as the number of C++ Translation Units, the Freespace 2 test case contains no C code. The Freespace 2 test case consists of 365 *KLOC*, as illustrated on the last row, third column of Table 1.

The results in Table 1 suggest that, for the test cases that we have chosen for our study, the game applications are larger than the language processing applications. For example, the average number for the game applications is 231 KLOC, whereas the average number of KLOC for the language processing applications is 78 KLOC; thus, the average game application in our test suite is three times as large as the average language processing application.

4.3. Complexity in Game Application Software

Table 2 presents results for the *Weighted Methods per Class* (WMC) metric. The rows in the table list the test cases. The columns list results for the WMC metric, where the first three columns list the minimum, Min, the maximum, Max and the mean, Mean, values for weighted methods. The final three columns in the tables list the standard deviation from the mean, Std Dev, the median, Median and the mode, Mode.

The results in Table 2 show that the methods in the language processing tools are more complex than the methods in the game application software. For example, the average maximum value of the language processing tools is 802.25, whereas the maximum value of the game applications is only 257.75. Similarly, the average Mean value for the language

processing tools is 25.55, whereas the avereage Mean value for the game applications is only 11.07.

5. Comparison with Similar Tools

The construction of source-based reverse engineering tools for C++ requires a parser, and possibly, a corresponding front-end. The difficulties in construction of a parser for the C++ language are well documented, and are largely due to the complexity of the template sublanguage [8,22,27,36–38,42]. Consequently, the availability of tools that require source-based reverse engineering of C++ programs is inadequate.

5.1. Tools that provide C++ parsing capability

Some reverse engineering tools include their own C++ parser. These included parsers extract information ranging from limited information, such as class hierarchies, to detailed information, such as statements and expressions. Parsers that extract limited information, known as *fuzzy parsers* [23], are well suited to tasks such as graphical browsing and graph visualization, but are not sufficient for program analysis tasks. Parsers that extract detailed information are ideal for program analysis tasks, but none of the parsers described in this subsection are able to fully accept templates.

Ferenc, et al. present *Columbus*, a fully integrated reverse engineering framework supporting fact extraction, linking, and analysis for C and C++ programs [13]. *Columbus* provides output in a variety of formats, including CPPML, GXL, RSF, and XMI. Nevertheless, *Columbus* is unable to fully accept templates, as noted in reference [14]. Also, *XOGASTAN* fails to create GXL for certain GENERIC node types including try_catch_expr and the using_directive. However, the g^4re tool chain accepts any program that can be parsed by the *gcc* compiler, which has performed well in tests measuring conformance to the ISO C++ standard including template programming and template metaprogramming [30].

LaPierre, et al. present *Datrix*, an analyzer that extracts information from C, C++, or Java programs [26]. Datrix extracts information for each translation unit in accordance with the Datrix ASG Model [7], and output is expressed in either TA (Tuple-Attribute Language) or VCG format. The Datrix project at Bell Canada ended in the year 2000, and the *Datrix* analyzer is no longer available.

Source Navigator (TM) from Red Hat is an analysis and graphical browsing framework for C, C++, Java, Tcl, FORTRAN, and COBOL [39]. The provided parser is a fuzzy parser that extracts enough high level information to provide class hierarchies, imprecise call graphs, and include graphs. *Source Navigator* does not provide statement level information and the plain text output is not conformant to a schema.

5.2. Tools that utilize the GCC parser

Some reverse engineering tools use the C++ parser included in the *gcc* GNU project by using the *tu* files described in Section 2.1. *gcc* is an industrial strength compiler that accepts virtually all of the constructs defined by the ISO C++ standard including templates [19,30].

Antoniol, et al. present *XOGASTAN*, a tool chain similar to our g^4re tool chain [6]. The provided tools convert a *gcc* tu file to a GXL instance graph and construct an in-memory

representation of the GXL instance graph. *XOGASTAN* does not provide a facility to reduce the ASG, resulting in large GXL instance graphs with extraneous information that is not useful to the user of the tool set. Additionally, the *XOGASTAN* analysis capabilities for C++ are limited.

Gschwind, et al. present *TUAnalyzer*, a system complementary to g^4re [14]. The *TUAnalyzer* uses a *gcc* tu file to perform analysis of template instantiations of functions and classes. The *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 scope of the tool is restricted to analysis of templates and does not produce a representation of the *gcc* tu file for exchange with other reverse engineering tools.

GCC.XML uses tu files to generate an XML representation for class, function, and namespace declarations, but does not propagate information such as function and method bodies [2]. As a result, many common program representations, such as the call graph or the ORD, cannot be constructed using the output of *GCC.XML*.

Hennessy, et al. present *gccXfront*, a tool that harnesses the *gcc* parser to tag C and C++ source code [15]. The tool annotates source code with syntactic tags in XML by modifying the *bison* parser generator tool, as described by Malloy, et al. [31]. However, this approach is no longer viable because the *gcc* C++ compiler has migrated to recursive descent technology.

Dean, et al. present *CPPX*, a tool that uses *gcc* for parsing and semantic analysis [11]. However, *CPPX* predates the incorporation of *tu* files into *gcc* and is built directly into the *gcc* code base. *CPPX* constructs an ASG that is compliant to the Datrix ASG Schema [7] and can be serialized to GXL, TA, or VCG format. The Datrix ASG Schema is more general than the GENERIC schema to accommodate C++ and other languages; this generality makes it difficult to accurately represent many C++ language constructs. The last release of *CPPX*, based on version 3.0 of *gcc*, does not properly handle the C++ Standard Library.

6. Conclusions and Future Work

In this paper we have described our tool chain that exploits the *gcc* C++ compiler, to enable experimentation and study of real C++ applications. Our tool accepts any C++ application that can be parsed by the *gcc* C++ front-end, including commonly used applications such as *Scribus* and *LyX* as well as large language processing tools and gaming software [3,4,20]. Our tool consists of a chain of applications that enables the user to access the tool at any point in the chain. The preferred point of access is at the end of the chain where an Application Programmers Interface (API) provides easy access to information about the names, classes, namespaces, functions, function calls or statements in the C++ application under study. Our tool has been used to build class diagrams, object relation diagrams (ORDs), a taxonomy of classes for maintenance, to facilitate software visualization and to compute metrics to evaluate object-oriented applications [9,25,20,28,32].

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. LyX. http://www.lyx.org/, October 2005.
4. Scribus. http://www.scribus.org.uk/, October 2005.
5. G. Antoniol, M. Di 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.
6. G. Antoniol, M. Di Penta, G. Masone, and U. Villano. Compiler hacking for source code analysis. *Software Quality Journal*, 12(4):383–406, December 2004.
7. Bell Canada Inc. *DATRIX - Abstract Semantic Graph Reference Manual*. Bell Canada Inc., Montreal, Canada, 1.4 edition, May 2000.
8. 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.
9. P. J. Clarke, J. D. Djuradj Babich, and B. Malloy. A tool to automatically map implementation-based testing techniques to classes. *International Journal of Software Engineering and Knowledge Engineering*, 2005. to appear.
10. Manuvir Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, Canada, May 2000.
11. T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a c++ extractor. In *Working Conference on Reverse Engineering*, October 2001. www.cppx.com.
12. H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 60–70, August 2004.
13. R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the 18th International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, October 2002.
14. 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.
15. 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.
16. R. Holt, A. Schürr, S. E. Sim, and A. Winter. GXL - Graph eXchange Language. http://www.gupro.de/GXL, January 2003.
17. R. C. Holt, A. Walter, and A. Schürr. GXL: Toward a standard exchange format. In *Working Conference on Reverse Engineering*, pages 162–171, Queensland, Australia, November 2000.
18. IBM Jikes Project. Jikes version 1.22. Available at http://jikes.sourceforge.net.
19. ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.
20. A. C. Jamieson, N. A. Kraft, J. O. Hallstrom, and B. A. Malloy. A metric evaluation of game application software. *Future Play 2005: The International Academic Conference on the Future of Game Design and Technology*, October 2005.
21. Keystone Project. Keystone version 0.2.3. Available at http://keystone.sourceforge.net.
22. Gregory Knapen, Bruno Lague, Michel Dagenais, and Ettore Merlo. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension*, Pittsburgh, PA, USA, May 5-7 1999.
23. R. Koppler. A systematic approach to fuzzy parsing. *Software – Practice and Experience*, 27(6):637–649, June 1997.
24. N. A. Kraft, B. A. Malloy, and J. F. Power. g^4re: Harnessing gcc to reverse engineer C++ applications. In *Seminar No. 05161: Transformation Techniques in Software Engineering*, Schloss Dagstuhl,

Germany, April 17-22 2005.
25. N. A. Kraft, B. A. Malloy, and J. F. Power. Toward an infrastructure to support interoperability in reverse engineering. In *Proceedings of the 12th Working Conference on Reverse Engineering, WCRE'05*, Pittsbrugh, PA, November 2005.
26. S. Lapierre, B. Lague, and C. Leduc. Datrix source code model and its interchange format: Lessons learned and considerations for future work. *ACM SIGSOFT Software Engineering Notes*, 26(1):53–56, January 2001.
27. John Lilley. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.
28. B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for integration testing of C++ applications. In *International Symposium on Software Reliability Engineering*, pages 353–364, Denver, CO, USA, Nov 2003.
29. 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.
30. 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.
31. 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.
32. Brian A. Malloy and James F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualization*, May 2005.
33. Brian A. Malloy and James F. Power. Using a molecular metaphor to facilitate comprehension of 3d object diagrams. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, September 2005.
34. J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, pages 171–180, Ottawa, Canada, 2003.
35. Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.
36. 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.
37. S.P. Reiss and T. Davis. Experiences writing object-oriented compiler front ends. Technical report, Brown University, January 1995.
38. J.A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1989.
39. Source–Navigator Team. The Source–Navigator IDE. http://sourcenav.sourceforge.net, June 2005.
40. D. van Heesch. Doxygen version 1.3.9.1. Available at http://stack.nl/ dimitri/doxygen.
41. Guido van Rossum. *Python Library Reference*. Python Software Foundation, 2001.
42. T. L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University, 2003.