Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 69 (2007) 3–13
Science of Computer Programming
www.elsevier.com/locate/scico

A tool chain for reverse engineering C++ applications

Nicholas A. Kraft[a,*], Brian A. Malloy[a], 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*

Received 1 November 2005; received in revised form 1 August 2006; accepted 16 January 2007
Available online 11 October 2007

Abstract

We describe a tool chain that enables experimentation and study of real C++ applications. Our tool chain enables reverse engineering and program analysis by exploiting *gcc*, and thus accepts any C++ application that can be analysed by the C++ parser and front end of *gcc*. Our current test suite consists of large, open-source applications with diverse problem domains, including language processing and gaming. Our tool chain is designed using a GXL-based pipe-filter architecture; therefore, the individual applications and libraries that constitute our tool chain each provide a point of access. The preferred point of access is the g4api Application Programming Interface (API), which is located at the end of the chain. g4api provides access to information about the C++ program under study, including information about declarations, such as classes (including template instantiations); namespaces; functions; and variables, statements and some expressions. Access to the information is via either a pointer to the global namespace, or a list interface.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Reverse engineering; Graph-based tools; GXL; Pipe-filter architecture; Software metrics

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 [10,30,31,42]. 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

* Corresponding address: Department of Computer Science, The University of Alabama, AL 35487-0290 Tuscaloosa, United States.
E-mail address: nkraft@cs.ua.edu (N.A. Kraft).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.01.012

previous research by describing an infrastructure to support interoperability in reverse engineering of C++ applications [29–31]. 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++ parser and front-end 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 [24]. 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 the g4api Application Programming Interface (API), which is located at the end of the chain. g4api provides access to information about the C++ program under study, including information about declarations, such as classes (including template instantiations); namespaces; functions; and variables, statements and some expressions. There are other points of access along the chain that 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 two sample usages. In Section 5 we compare our tool to similar systems. Finally, in Section 6 we draw conclusions and describe our ongoing work.

2. Background

In this section we review terminology and major technologies that we use in the design and implementation of the g^4re tool chain. In Section 2.1 we review GXL, a standard format used to exchange typed, attributed, directed graphs. In Section 2.2 we review `GENERIC`, the internal ASG representation of *gcc* that several research tools have utilized to perform reverse engineering and program analysis [4,17,24,26,39,40].

2.1. Graph eXchange Language

Graph eXchange Language (GXL) is a standard exchange format (SEF) that is an XML language defined by a document type definition (DTD) and conceptualized as a typed, attributed, directed graph. GXL is used to describe both instance data and schemas, which are represented by UML class diagrams [21]. GXL was ratified as the SEF for reverse engineering and reengineering tools at the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [13].

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 [20]. The graph under validation can be a GXL schema or a GXL instance; GXL schemas are validated against the GXL metaschema, which validates against itself. Validating GXL is important; validation can reveal errors in both the modelling and the generation of GXL instances, and valid GXL files are more likely to be accepted by available XML tools than non-valid files.

2.2. GENERIC—The gcc ASG representation

The Abstract Semantic Graph (ASG) is a common program representation used by compiler front ends and other grammarware tools. An ASG is constructed by adding semantic information to an abstract syntax tree (AST). Examples of the added semantic information include edges from variable uses to their declarations, and, for C++, template instantiations. The C++ compiler from the GNU Compiler Collection, *gcc*, uses an ASG to facilitate recognition, analysis and optimization of a program. From version 3.0, *gcc* has included an ASG representation known as `GENERIC` [41].

`GENERIC` consists of 200 concrete node types and 98 concrete edge types and is documented, almost exclusively, in the form of source code comments. Example node types include: `record_type`, `function_decl`, 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 Fig. 1.

A `tu` file contains an ASCII encoding of the information contained in the *gcc* ASG. A node in a `tu` file is represented by:

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

Fig. 1. *Example*: This figure illustrates the format of a node in a `tu` file.

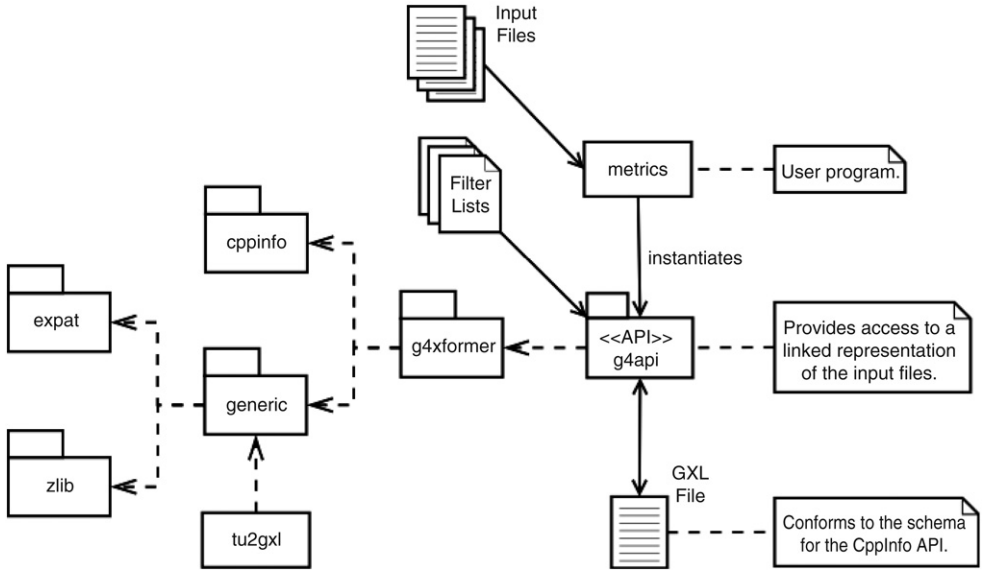

Fig. 2. System overview. *This figure provides an overview of the g^4re tool chain. The dashed lines represent "use" dependencies. The solid lines represent input and output.*

- a unique identifier consisting of '@' concatenated with a unique integer;
- a string representing the `GENERIC` node type;
- edge tuples of the form "edge: dest", where dest is the unique identifier of the destination node;
- field tuples of the form "field: value";
- a set of single word attributes.

For example, in Fig. 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`.

3. Description of the tool

In this section we describe the design and implementation of the g^4re tool chain. We describe g^4re as a *tool chain*, because it is constituted by applications and libraries that may be used individually or en masse. The g^4re tool chain is designed using a GXL-based pipe-filter architecture; each constituent application or library in the chain takes, as input, the output of the preceding application or library in the chain. As a result of using a pipe-filter architecture, our system consists of a set of loosely coupled, reusable modules: the *ASG module*, the *schema module*, the *transformation and linking module*, and the *API module*. All modules in the g^4re tool chain are written in ISO C++.

Fig. 2 provides an overview of the g^4re tool chain. We illustrate the *ASG module*, generic, as a package near the bottom left of the figure, and describe it in Section 3.1. We illustrate the *schema module*, cppinfo, as a package in the upper left of the figure, and describe it in Section 3.2. We illustrate the *transformation and linking module*, g4xformer, as a package in the centre of the figure, and describe it in Section 3.2. Finally, we illustrate the *API module*, g4api, as a package with the stereotype <<API>> to the right of centre of the figure, and describe it in Section 3.4.

Fig. 3 illustrates the process of obtaining input files for the g^4re tool chain. The input files, shown in the far right of Fig. 3 and the top centre of Fig. 2, contain encodings of `GENERIC` ASG instances provided by *gcc*, and may include any combination of: `tu` files, GXL files, and gzipped GXL files. We provide the application tu2gxl, shown in the top centre of Fig. 3 and the bottom left of Fig. 2, with the *ASG module*, generic.

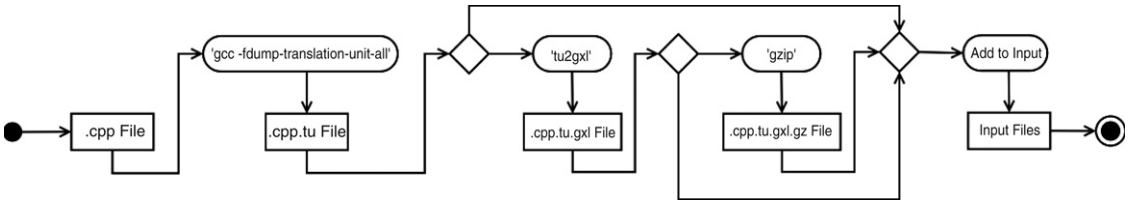

Fig. 3. UML activity diagram for system input. *This figure illustrates the process of creating input files for use with the g^{4}re tool chain. As shown, three different input formats are accepted.*

3.1. The ASG module: generic

The generic package provides parsing, storage, traversal and serialization facilities for working with the `GENERIC` ASG representation of *gcc*. The package provides two parsers: a `tu` file parser that we implemented using a *flex* generated scanner, and a GXL file parser that we implemented using the libraries *expat* and *zlib*. The package provides a simple node list graph representation for storage of the parsed ASG, and provides several parameterized methods for traversing the leftmost-child-right-sibling tree that underlies the ASG. Finally, the package provides an extensible serialization facility that we leverage to create GXL encodings of `tu` files.

The first parser of the generic package provides functionality to parse a `tu` file and to store the corresponding ASG. After parsing a `tu` file, we perform a series of transformations on the stored ASG to remove extraneous information and to make it more suitable for reverse engineering tasks. In particular, we:

- remove fields that store internal information used by the *gcc* backend;
- mark methods whose parameter lists do not contain a *this* pointer as static;
- mark methods whose parameter lists contain a const *this* pointer as const;
- remove the *this* pointer from all method parameter lists.

We use this parser in conjunction with our serialization facility to create GXL instances of `tu` files.

The second parser of the generic package provides functionality to parse a GXL file or gzipped GXL file and to store the corresponding ASG. This parser has three advantages over the `tu` parser:

1. reentrance;
2. the ability to read compressed files;
3. the lack of post-parse transformation overhead.

Note that the `tu` parser is used to create the GXL files accepted by this parser, thus there is a one-time cost associated with its use.

3.2. The schema module: cppinfo

The cppinfo package provides a class hierarchy that implements the CppInfo API schema [30,31], which is partially illustrated in Fig. 4. The package also provides utility classes to read and write GXL instances of the CppInfo API schema, as well as an abstract class that defines the interface for an API that provides access to the information found in an instance of the schema.

The cppinfo package currently contains 62 classes (38 of which are concrete), that provide information about C++ language elements, including declarations, such as classes (including class templates and class template instantiations); namespaces; functions (including function templates and function template instantiations); and variables, statements (including control statements and exception statements) and some expressions. In addition, the package provides *Iterator* classes, and an abstract base *Visitor* class [16] that are both leveraged internally and made available to package users.

3.3. The transformation and linking module: g4xformer

The g4xformer package provides an implementation of the transformation from the ASG representation provided by the generic package to an intermediate ASG representation that contains instances of the classes provided by

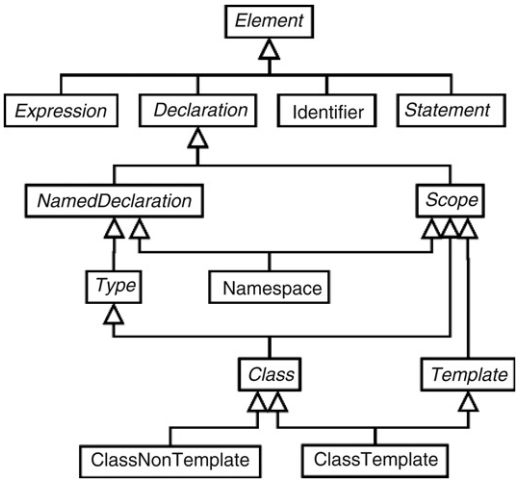

Fig. 4. Partial CppInfo API schema. *This figure illustrates some of the primary classes in the CppInfo API schema. Details, such as attributes and operations, are not shown here, but are available in the full version of the schema. The full version of the schema is available in our web repository as both a GXL schema and a UML class diagram.*

the cppinfo package. In addition, the package provides an implementation of our linking algorithm that operates on instances of the intermediate API representation. The end result of the linking process is a single intermediate API representation that contains a unified representation of a program.

Real C++ programs, such as the open-source applications and libraries in our test suite, consist of multiple files, both header and source. A *C++ translation unit* consists of a source file and all of the files it includes, either directly or transitively. A C++ compiler, such as *gcc*, operates on a single translation unit at a time; the generated object code for all translation units in a program is linked by the system linker, e.g. *ld* on Unix systems. A reverse engineering tool for C++, such as g^4re, also operates on a single translation unit at a time; however, the generated output is not object code, but rather a program representation such as an ASG or API.

Unique identifiers that are not specific to a particular translation unit, such as mangled names or fully-qualified names, allow programs to be linked at the program representation level [31,50]. The g4xformer package serially transforms each API provided by the generic package to an intermediate API representation consisting of a set of dictionaries that map unique identifiers to instances of classes provided by the cppinfo package. We implemented g4xformer to link these dictionary representations each time a pair becomes available. Thus, linking in g^4re is performed $n - 1$ times, where n is the number of translation units.

We achieve linking by performing a traversal of the most recently constructed intermediate API instance, adding or appending cppinfo class instances to the existing intermediate API instance if the existing instances are missing or incomplete. A cppinfo class instance is incomplete if it is missing a required element (as defined by the schema) or contains another incomplete instance. Using this definition of incomplete, we also resolve function declarations to their corresponding definitions with our implementation of the linking algorithm.

3.4. The API module: g4api

The g4api package provides a concrete implementation of the API interface (provided by the schema module, cppinfo) for accessing information in the unified representation of a C++ program. Our implementation of the API provides the capability to serialize this representation to a GXL instance that conforms to the CppInfo API schema. In addition, the implementation can deserialize such a GXL instance into an API instance, thereby eliminating the need to repeatedly link all translation units of a C++ program.

Our API module, g4api, provides a clear and flexible interface for accessing information about language elements in a C++ program. The g4api package provides two points of access that allow users to access information about a C++ program. The first point of access is a pointer to the global namespace. Using this pointer, a user can traverse the ASG that underlies the API. We provide *Iterator* classes, as well as an abstract base *Visitor* class, for users to leverage when accessing the API in this fashion. Alternatively, a user may access several lists containing

```
(01) class Shape { };
(02) class Circle    : public Shape { };
(03) class Rectangle : public Shape { };
(04)
(05) class Square : public Rectangle { };
(06)
(07) class Visitor { };
(08) class ComputationVisitor   : public Visitor { };
(09) class SerializationVisitor : public Visitor { };
(10)
(11) class AreaComputationVisitor     : public ComputationVisitor { };
(12) class PerimeterComputationVisitor : public ComputationVisitor { };
(13)
(14) class XmlSerializationVisitor : public SerializationVisitor { };
```

Fig. 5. Sample C++ program. *This figure illustrates two disjoint inheritance hierarchies containing ten classes.*

instances of particular CppInfo schema classes present in the API. These lists are provided for Namespace, Class, Enumeration, Enumerator, Function, Variable, and Typedef. These lists are available in two forms. The first form provides all instances of the particular schema class; the second form provides *filtered* instances of the particular schema class. *Filtered* instances are determined by user-provided *filter lists*, shown near the top of the center column in Fig. 2. *Filter lists* contain the names of source files from which instances should be ignored. The g4api package comes with a script that generates these filter lists.

4. Sample tool usage

In this section we provide two sample usages of the g^4re tool chain. In the first sample usage, we present the source code for both a simple C++ program, and a simple analysis of that program. In the second sample usage, we review our metrics computation system [24] that we use to measure programs from our test suite of open-source software.

4.1. A simple example

In Fig. 5, we list a small C++ program that consists of ten classes. We list two *root classes*, Shape and Visitor, on lines 1 and 7, respectively. Root classes do not have base classes. We list three *interior classes*, Rectangle, ComputationVisitor, and SerializationVisitor, on lines 3, 8, and 9, respectively. Interior classes have one or more base classes, and one or more derived classes. Finally, we list five *leaf classes*, Circle, Square, AreaComputationVisitor, PerimeterComputationVisitor, and XmlSerializationVisitor, on lines 2, 5, 11, 12, and 14, respectively. Leaf classes have one or more base classes, but no derived classes.

In Fig. 6, we list a C++ function that instantiates and uses an API instance to compute the number of root, interior, and leaf classes in a C++ program. We list the function signature on line 1, where the parameter filenames denotes the input program (see Fig. 3 for details). We list the API instantiation on line 2, where the list of file names is passed to the constructor of class api::Interface. On line 4, we use the list point of access provided by the API to obtain an iterator that accesses each class in the input program. Finally, we list a *while* loop on lines 5–19 that computes the number of root, interior, and leaf classes.

4.2. The metrics computation system

In this section we review our metrics computation system, metrics, that we use to measure and characterize the exploitation of object technology in game application software [24]. We chose this example because it illustrates an analysis of C++ applications at the levels of the *class*, *method*, and *statement*. The relationship of metrics to the g^4re tool chain is illustrated in Fig. 2, where it is shown in the upper right.

The output of metrics 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 [24]; the computed metrics include the number of classes

```
(01)  void countClasses ( const api::FilenameList_t& filenames ) {
(02)    api::Interface interface( filenames );
(03)    unsigned root = 0, interior = 0, leaf = 0;
(04)    cppinfo::ClassListIterator* cli = interface.getClasses().createIterator();
(05)    while ( cli->isValid() ) {
(06)      const cppinfo::schema::Class* c = cli->getCurrent();
(07)      unsigned baseCount = c->getBaseClasses().size();
(08)      unsigned derivedCount = c->getDerivedClasses().size();
(09)      if ( baseCount == 0 ) {
(10)        ++root;
(11)      }
(12)      else if ( baseCount > 0 && derivedCount > 0 ) {
(13)        ++interior;
(14)      }
(15)      else if ( baseCount > 0 && derivedCount == 0 ) {
(16)        ++leaf;
(17)      }
(18)      cli->moveNext();
(19)    }
(20)    delete cli;
(21)  }
```

Fig. 6. Sample analysis. *This figure illustrates a simple program analysis that counts the number of root, interior and leaf classes.*

and methods, depth of inheritance, breadth of inheritance, and weighted method per class. In this paper, we present only results for weighted methods per class—a measure of complexity.

In Section 4.2.1 we describe our test suite of open-source applications, including popular games written using the Simple Directmedia Layer (SDL), as well as language processing tools. In Section 4.2.2 we describe some results about the ability of game software to exploit the object-oriented methodology.

4.2.1. The test suite of SDL games and language processing tools

Table 1 lists eight applications, or test cases, that form the test suite we use in our study. The top row of the table lists the names that we use to refer to each of the test cases. We list the games in the first four columns and the language processing tools 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 Programming Interface (API) used for the four games is the Simple Directmedia Layer (SDL) [32]. The four language processing tools are: *Doxygen* [48], g^4re, *Jikes* [22], and *Keystone* [25,36].

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[1]; the second row lists the number of source files, Source Files; 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; and finally, the last row of the table lists the (approximate) number of thousands of lines of code (KLOC) for each test case, not counting blank or comment lines.

Table 1 shows that, for the test cases that we have chosen for our study, the SDL games are larger than the language processing tools. For example, the average number of KLOC for the games is 231, whereas the average number of KLOC for the language processing tools is 78. Thus, the average game in our test suite is three times as large as the average language processing tool.

4.2.2. Complexity in game application software

The weighted methods per class (WMC) metric measures the complexity of an object, and is an indicator of the time and effort required to develop and maintain a class [8,14].

[1] We performed CVS checkouts on July 22, 2005.

Table 1
Test suite of SDL games and language processing tools

	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 2
Weighted methods per class

	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^{4}re	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

Given a method M with control flow graph $G = (V, E)$, let D equal the set of decision nodes in V, where a decision node represents one of {*if, switch, for, while, do while, catch*}. The cyclomatic complexity, c, of M is the number of linearly independent paths in G and is computed as

$$c(M) = |D| + 1.$$

Given a class C with methods $M_1, M_2, \ldots, M_n$, weighted with cyclomatic complexity $c_1, c_2, \ldots, c_n$, respectively, the metric is computed as

$$WMC(C) = \sum_{i=1}^{n} c_i.$$

Table 2 presents results for the 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. We executed all of the experiments 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 `tu` files were generated using *gcc* version 3.3.6.

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 WMC metric for language processing tools is 802.25, whereas the maximum value for the games is only 257.75. Similarly, the average Mean value for the language processing tools is 25.55, whereas the average Mean value for the games 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 the construction of a parser for the C++ language are well documented, and are largely

due to the complexity of the template sublanguage [7,27,34,44–46,49]. Consequently, the availability of tools that provide 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* [28], 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.

LaPierre, et al. present *Datrix*, an analyser that extracts information from C, C++, or Java programs [33]. Datrix extracts information for each translation unit in accordance with the Datrix ASG Model [6], 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 [47]. 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 does not conform to a schema.

Ferenc, et al. present *Columbus*, a fully integrated reverse engineering framework supporting fact extraction, linking, and analysis for C and C++ programs [15]. *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 [17].

5.2. Tools that utilize the GCC parser

Some reverse engineering tools use the C++ parser included in *gcc*, the C++ compiler from the GNU Compiler Collection. There are two common approaches taken by these tools. The first approach is to modify the source code of the parser, thus creating a custom version of *gcc*. The second approach is to use the `tu` files described in Section 2.2. *gcc* is an industrial strength compiler that accepts virtually all of the constructs defined by the ISO C++ standard including templates [23,37].

Dean, et al. present *CPPX*, a tool that uses *gcc* for parsing and semantic analysis [11]. *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 [6] 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.

Hennessy, et al. present *gccXfront*, a tool that harnesses the *gcc* parser to tag C and C++ source code [18]. The tool annotates source code with syntactic tags in XML by modifying the *bison* parser generator tool, as described by Malloy, et al. [38]. However, this approach is no longer viable because the *gcc* C++ compiler has migrated to recursive descent technology.

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 [26]. As a result, many common program representations, such as the call graph or the ORD, cannot be constructed using the output of *GCC.XML*.

Antoniol, et al. present *XOGASTAN*, a tool chain similar to our g^4re tool chain [5]. The provided tools convert a *gcc* tu file to a GXL instance graph and construct an in-memory representation of the GXL instance graph. However, *XOGASTAN* fails to create GXL for certain GENERIC node types, including try_catch_expr and using_directive. Additionally, the *XOGASTAN* analytical capabilities for C++ are limited.

Gschwind, et al. present *TUAnalyzer*, a system complementary to g^4re [17]. 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.

6. Conclusions and future work

In this paper we have described our tool chain, g^4re, 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++ parser and front-end, including common open-source applications such as *Scribus* and *LyX*, as well as games and language processing tools [2,3,24]. Our tool consists of a chain of applications, thus allowing the user to access the tool at any point in the chain, with the preferred point of access being the g4api Application Programming Interface (API), which is located at the end of the chain. g4api provides access to information about the C++ program under study, including information about declarations, such as classes (including template instantiations); namespaces; functions; and variables, statements and some expressions.

We have used g^4re to build object relation diagrams (ORDs), and a taxonomy of classes for maintenance [9,30,31, 35]. In addition, we have used g^4re to construct systems to facilitate software visualization and to compute metrics to evaluate object-oriented applications [19,24,39]. We have also used g^4re in the classroom at Clemson University, where graduate students wrote C++ programs that accessed the g4api to build class diagrams.

Development of the g^4re tool chain is ongoing. We are currently in the process of dividing g4xformer, the transformation and linking module, into two separate modules. Our new linker module will work on GXL instances of the CppInfo API schema rather than our (internal) intermediate API representation. Our goal is to allow external tools to take advantage of our linker. The source code for g^4re is available on the project homepage [43].

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.01.012.

References

[1] GXL Validator. http://www.uni-koblenz.de/FB4/Contrib/GUPRO/Site/Downloads/index_html?project=gupro_all, January 2003.
[2] LyX. http://www.lyx.org/, October 2005.
[3] Scribus. http://www.scribus.org.uk/, October 2005.
[4] G. Antoniol, M. Di Penta, G. Masone, 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.
[5] G. Antoniol, M. Di Penta, G. Masone, U. Villano, Compiler hacking for source code analysis, Software Quality Journal 12 (4) (2004) 383–406.
[6] Bell Canada Inc, DATRIX—Abstract Semantic Graph Reference Manual, 1.4 edition, Bell Canada Inc, Montreal, Canada, May 2000.
[7] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, 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, Sunriver, Oregon, USA, 1994, pp. 122–136.
[8] Shyam R. Chidamber, Chris F. Kemerer, A metrics suite for object oriented design, IEEE Transactions of Software Engineering 20 (6) (1994) 476–493.
[9] P.J. Clarke, B.A. Malloy, J. Ding, D. Babich, A tool to automatically map implementation-based testing techniques to classes, International Journal of Software Engineering and Knowledge Engineering 16 (4) (2006) 585–614.
[10] Manuvir Das, Unification-based pointer analysis with directional assignments, in: Programming Language Design and Implementation, Vancouver, BC, Canada, May 2000, pp. 35–46.
[11] T.R. Dean, A.J. Malton, R.C. Holt, Union schemas as a basis for a c++ extractor, in: Working Conference on Reverse Engineering. www.cppx.com, October 2001.
[12] H. Do, S. Elbaum, G. Rothermel, Infrastructure support for controlled experimentation with software testing and regression testing techniques, in: Proceedings of the International Symposium on Empirical Software Engineering, August 2004, pp. 60–70.
[13] J. Ebert, K. Kontogiannis, J. Mylopoulus (Eds.), Seminar No. 01041: Interoperability of Reengineering Tools, Schloss Dagstuhl, Germany, 21–26 January 2001.
[14] Norman E. Fenton, Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Co., Boston, MA, USA, 1998.
[15] R. Ferenc, A. Beszedes, M. Tarkiainen, T. Gyimothy, Columbus—reverse engineering tool and schema for c++, in: Proceedings of the 18th International Conference on Software Maintenance, Montreal, Canada, October 2002, pp. 172–181.
[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995.
[17] T. Gschwind, M. Pinzger, H. Gall, TUAnalyzer—analyzing templates in C++ code, in: Proceedings of the Eleventh Working Conference on Reverse Engineering, IEEE, 2004.
[18] M. Hennessy, B.A. Malloy, 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, IEEE, Portland, Oregon, USA, May 2003, pp. 298–299.
[19] B.N. Hoipkemier, N.A. Kraft, B.A. Malloy, 3d visualization of class template diagrams for deployed open source applications, in: Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, USA, July 2006.

[20] R. Holt, A. Schürr, S.E. Sim, A. Winter, GXL—Graph eXchange Language. http://www.gupro.de/GXL, January 2003.
[21] R.C. Holt, A. Walter, A. Schürr, GXL: Toward a standard exchange format, in: Working Conference on Reverse Engineering, Queensland, Australia, November 2000, pp. 162–171.
[22] IBM Jikes Project, Jikes version 1.22. Available at: http://jikes.sourceforge.net.
[23] ISO/IEC JTC 1, International standard: Programming languages—C++. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.
[24] A.C. Jamieson, N.A. Kraft, J.O. Hallstrom, 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.
[25] Keystone Project, Keystone version 0.2.3. Available at: http://keystone.sourceforge.net.
[26] Kitware, Inc. GCC-XML. http://www.gccxml.org, February 2005.
[27] Gregory Knapen, Bruno Lague, Michel Dagenais, Ettore Merlo, Parsing C++ despite missing declarations, in: 7th International Workshop on Program Comprehension, Pittsburgh, PA, USA, 5–7 May 1999.
[28] R. Koppler, A systematic approach to fuzzy parsing, Software—Practice and Experience 27 (6) (1997) 637–649.
[29] N.A. Kraft, B.A. Malloy, J.F. Power, g^4re: Harnessing gcc to reverse engineer C++ applications, in: Seminar No. 05161: Transformation Techniques in Software Engineering, Schloss Dagstuhl, Germany, 17–22 April 2005.
[30] N.A. Kraft, B.A. Malloy, J.F. Power, Toward an infrastructure to support interoperability in reverse engineering, in: Proceedings of the 12th Working Conference on Reverse Engineering, Pittsburgh, PA, November 2005.
[31] N.A. Kraft, B.A. Malloy, J.F. Power, An infrastructure to support interoperability in reverse engineering, Information and Software Technology 49 (3) (2007) 292–307.
[32] S. Lantinga, Simple directmedia layer. http://www.libsdl.org, October 2005.
[33] S. Lapierre, B. Lague, C. Leduc, Datrix source code model and its interchange format: Lessons learned and considerations for future work, ACM SIGSOFT Software Engineering Notes 26 (1) (2001) 53–56.
[34] John Lilley, PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5, February 1997.
[35] B.A. Malloy, P.J. Clarke, E.L. Lloyd, A parameterized cost model to order classes for integration testing of C++ applications, in: International Symposium on Software Reliability Engineering, Denver, CO, USA, Nov 2003, pp. 353–364.
[36] B.A. Malloy, T.H. Gibbs, J.F. Power, Decorating tokens to facilitate recognition of ambiguous language constructs, Software, Practice & Experience 33 (1) (2003) 19–39.
[37] B.A. Malloy, T.H. Gibbs, J.F. Power, Progression toward conformance for C++ language compilers, Dr. Dobbs Journal (November) (2003) 54–60.
[38] B.A. Malloy, J.F. Power, Program annotation in XML: A parser-based approach, in: Proceedings of the Ninth Working Conference on Reverse Engineering, IEEE, Richmond, Virginia, USA, October 2002, pp. 190–198.
[39] Brian A. Malloy, James F. Power, Exploiting UML dynamic object modeling for the visualization of C++ programs, in: ACM Symposium on Software Visualization, May 2005.
[40] Brian A. Malloy, 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.
[41] J. Merrill, GENERIC and GIMPLE: A new tree representation for entire functions, in: GCC Developers Summit, Ottawa, Canada, 2003, pp. 171–180.
[42] Gail C. Murphy, David Notkin, William G. Griswold, Erica S. Lan, An empirical study of static call graph extractors, ACM Transactions on Software Engineering and Methodology 7 (2) (1998) 158–191.
[43] N.A. Kraft, g^4re reverse engineering infrastructure, version 1.0.8, May 2006. Available at: http://g4re.sourceforge.net.
[44] J.F. Power, 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, Sydney, Australia, November 2000, pp. 57–68.
[45] S.P. Reiss, T. Davis, Experiences writing object-oriented compiler front ends, Technical Report, Brown University, January 1995.
[46] J.A. Roskind, A YACC-able C++ 2.1 grammar, and the resulting ambiguities, Independent Consultant, Indialantic FL, 1989.
[47] Source–Navigator Team. The Source–Navigator IDE. http://sourcenav.sourceforge.net, June 2005.
[48] D. van Heesch, Doxygen version 1.3.9.1. Available at: http://stack.nl/~dimitri/doxygen.
[49] T.L. Veldhuizen, C++ templates are turing complete, Technical Report, Indiana University, 2003.
[50] Jingwei Wu, Richard C. Holt, Resolving linkage anomalies in extracted software system models, in: International Workshop on Program Comprehension, Bari, Italy, 24–26 June 2004, pp. 241–245.