Toward an Infrastructure to Support Interoperability in Reverse Engineering

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

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

Abstract

In this paper we present an infrastructure that supports interoperability among various reverse engineering tools and applications. We include an Application Programmer's Interface that permits extraction of information about declarations, including classes, functions and variables, as well as information about scopes, types and control statements in C++ applications. We also present a hierarchy of canonical schemas that capture minimal functionality for middle-level graph structures. This hierarchy facilitates an unbiased comparison of results for different tools that implement the same or a similar schema. We have a repository, hosted by SourceForge.net, where we have placed the artifacts of our infrastructure.

1 Introduction

In their effort to develop new techniques and to improve on previous efforts, most researchers must implement at least one previously developed technique as an unbiased basis for comparison with their technique. However, even after the previously developed technique is implemented the researcher is frequently unsure of the correctness of the implementation or the correctness of the generated results. Thus, comparison of competing approaches is difficult and sometimes impossible. For example, researchers in language design and implementation have reported considerable difficulty in replicating results in generating call graphs and points-to analysis, even for C programs [1, 19].

The issues involved in permitting interoperability among reverse engineering tools and applications have been discussed in the literature and, in particular, at the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [3]. At the seminar, three levels of interoperability were agreed upon: low-level syntax, middle-level graph structures and high-level architectures. Such interoperability is important not only to permit reuse of reverse engineering artifacts but also to facilitate reproduction of prior scientific results.

In this paper, our goal is the practical realization of interoperability through the construction of tools and graphs at each of these levels. Moreover, we wish to address the problem of the plethora of schemas and tools for low-level syntax and the dearth of schemas and tools for middle-level graph structures. Finally, we wish to develop a technique to facilitate unbiased comparison of results from competing tools and technologies. We describe an infrastructure that supports interoperability among various reverse engineering tools and applications. In previous research we presented g^4re, a tool chain that exploits GENERIC, an intermediate format incorporated into the *gcc* C++ compiler, to facilitate analysis of real C++ applications [12]. In this paper we extend g^4re to include an Application Programmer's Interface (API) that permits extraction of information about declarations, including classes, functions and variables, as well as information about scopes, types and control statements in C++ applications. In addition, we describe a hierarchy of canonical schemas that capture minimal functionality for middle-level graph structures. The purpose of this hierarchy is to facilitate an unbiased comparison of results for different tools that implement the same or a similar schema.

Our approach for comparing results from different tools utilizes XSLT style sheets that express transformations on the GXL schema for the graph structure under study. To demonstrate this approach we provide results for the construction of Object Relation Diagrams (ORDs) [15] using GXL representations generated using our API.

We have constructed a repository, hosted by SourceForge.net, and placed the artifacts of our infrastructure in this repository [20]. These artifacts include our g^4re and API tools, a test suite of applications and libraries along with GXL instance graphs for their translation units and ORDs, and XSLT style sheets for comparison of ORDs. In

Accepted for the 12th Working Conference on Reverse Engineering
Pittsburgh, Pennsylvania, USA, November 8-11, 2005

addition, we provide many GXL schemas for artifacts such as call graphs, control flow graphs, and interprocedural control flow graphs. Finally, we include GXL schemas for the GENERIC ASG and our CppInfo API.

In Section 2 we provide detail about the infrastructure, including our hierarchy of canonical GXL schemas. In Section 3 we review g^4re and then describe our API and a procedure for linking GXL representations of each translation unit into a single representation. In Section 4 we list results that describe the efficiency of our linking process and some results for the construction of ORDs for five applications and some test cases from the *gcc 3.3.4* distribution. In Section 5 we describe previous research that relates to our infrastructure. Finally, in Section 6 we draw conclusions.

2 An Overview of the infrastructure

Our goal is to provide an infrastructure that supports interoperability among various reverse engineering tools and applications. The issues involved in the construction of such an infrastructure have been discussed in the literature and at the Dagstuhl Seminar on *Interoperability of Reengineering Tools*, where GXL was ratified as the standard format for the exchange of graphs among reverse engineering and reengineering tools and applications [3]. Moreover, the participants agreed upon three levels at which interoperability should be applied:

- Low-level graph structures: Abstract Syntax Trees (AST) and adorned AST's (ASG);
- Middle-level graph structures: such as call graphs and program dependence graphs;
- High-level graph structures: Architecture descriptions.

However, our goal is the practical realization of interoperability through the construction of tools and graphs at each of these three levels. Moreover, we wish to address the problem of the plethora of schemas and tools for low-level syntax and the dearth of schemas and tools for middle-level graph structures. In particular, the goals of our work are:

1. The practical realization of tools and applications to support interoperability among reverse engineering and reengineering tools and applications;
2. Support for repeatability of results:
   (a) to obviate the need for researchers to repeat the development already achieved by previous researchers, and
   (b) To facilitate comparison of results among tools and applications constructed by different researchers.
3. Provision for a complete infrastructure including all tools required for each of the three levels, together with other infrastructure artifacts such as test cases, results, and support for the comparison of results.

2.1 Front-end and API

Figure 1 illustrates the levels in the infrastructure that we propose, including some of the artifacts in the infrastructure. The lowest level, Level 0, consists of a parser and front-end that builds an Abstract Semantic Graph (ASG), an Abstract Syntax Tree (AST) decorated with semantic information for a C++ application under study. The ASG contains information about declarations, including classes, functions, and variables, as well as information about scopes, types, and control statements. The front-end that we present in this paper is an extension of the g^4re system, described in reference [12]. We have extended g^4re in two important directions: (1) we have linked the ASGs for each translation unit, and (2) we provide an Application Programmer Interface (API) to facilitate access to ASG information. These extensions to g^4re are described further in Section 3.

Our experience in utilizing the ASG motivated the construction of an API to facilitate access to ASG information. Navigating the ASG links and extracting relevant information has proven to be time-consuming and error-prone. The relative sizes of the schemas for the ASG and the API highlight the cognitive burden involved in using each of these structures: the GENERIC ASG schema contains 166 node classes whereas the CppInfo API schema currently contains only 53 classes (both node and edge classes) as illustrated on the lower left side of Figure 1 in the ellipses for the two schemas [1]. Our experience with these two data structures has shown that access to ASG information is greatly facilitated through the use of a corresponding API, illustrated at Level I in Figure 1.

2.2 Graphs as Schemas

In Figure 1, all ellipses represent schemas and the figure illustrates a hierarchy of canonical schemas for various graph structures and analysis information. Levels to the right of Level I in the figure contain schemas that describe graph structures required for analysis and transformation of source code. Level II contains a schema for a Class Diagram, Control Flow Graph (CFG) and Call Graph, illustrated in the middle of Figure 1; these basic graph structures are required, either implicitly or explicitly, for more complex graph and analysis structures whose schemas are placed at higher levels of the infrastructure.

[1]The API schema currently lacks representation for expressions.

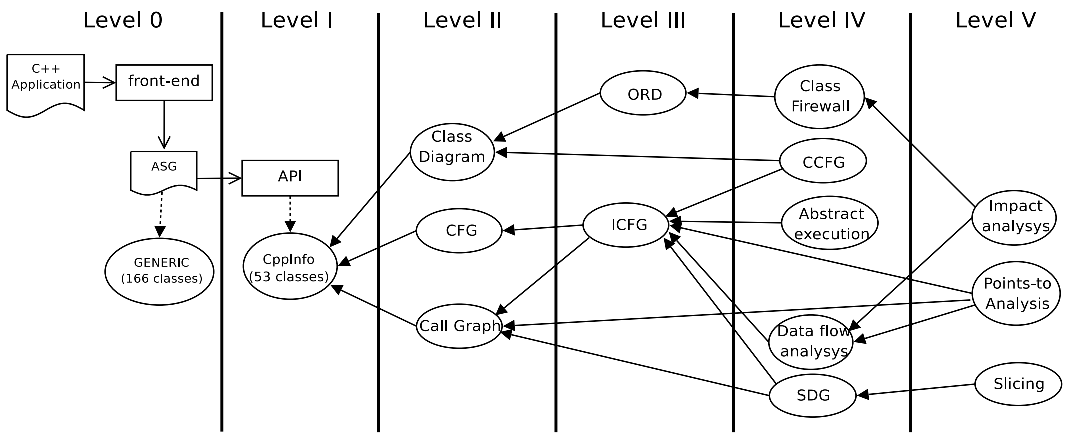


Figure 1. Overview of the Infrastructure and some artifacts. *This figure illustrates the levels in our infrastructure where Level 0 consists of a parser and front-end for syntax analysis and Level 1 consists of the* CppInfo *API. All ellipses in this figure represent schemas and all of the schemas together illustrate our canonical hierarchy of minimal schemas for graph structures and analysis tools.*

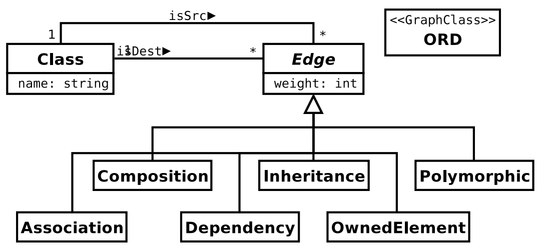


Figure 2. Schema for an ORD. *This figure illustrates a schema for an Object Relation Diagram (ORD). An ORD is a graph consisting of nodes representing classes, and edges representing relationships between the classes.*

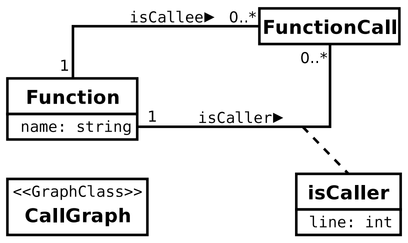


Figure 3. Schema for a Call Graph. *This figure illustrates a schema for a call graph, a graph whose nodes represent either functions or function call sites and whose edges represent function invocations.*

Level III contains schemas for an Object Relation Diagram (ORD) and an Interprocedural Control Flow Graph (ICFG). An ORD is an extension of a Class Diagram and an ICFG is an abstraction of a CFG and a Call graph; the edges from the ORD schema to the Class Diagram schema and from the ICFG schema to the CFG and Call Graph schemas capture this hierarchy of abstraction. Level IV contains schemas for a Class Firewall, a Class Control Flow Graph (CCFG), and schemas to describe graphs for Abstract Execution, Data Flow Analysis and a System Dependence Graph (SDG). Level V contains schemas for Impact Analysis, Points-to Analysis and Slicing.

Figures 2, 3 and 4 illustrate schemas for an ORD, Call Graph and CFG, respectively. For example, an ORD is a graph whose nodes represent classes and whose edges represent relationship between the classes. The ORD schema in Figure 2 consists of eight classes, two classes for nodes, Class, and edges, Edge, and six classes derived from Edge representing the six kinds of relationships between classes. These relationships consist of Association, Composition, Dependency, Inheritance, OwnedElement and Polymorphic edges [15]. The call graph schema in Figure 3 consists of three classes representing a function, Function, a function invocation, FunctionCall and the relationship isCaller, which specifies the line number where the function invocation occurred. Finally, the control flow graph (CFG) schema in Figure 4 consists of six classes representing the flow of control between basic blocks in a program.

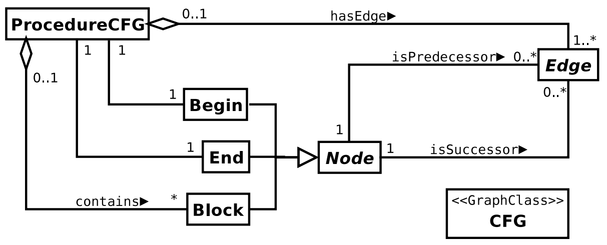


Figure 4. Schema for a Control Flow Graph. *This figure illustrates a schema for a control flow graph, whose nodes represent blocks of straight-line code and whose edges represent flow of control between the blocks.*

2.3 Generating Data and Comparing ORDs

Each of the schemas in Figure 1 can be used to facilitate comparison of results for a given graph structure. The tools under study are not required to use the same schema; however, comparison of the results generated for each tool can only be undertaken for those parts of the schema that are common to both tools. To generate results for a graph structure, an XSLT style sheet is used with a GXL instance of the graph. All of the results in Section 4.3 were obtained in this manner.

3 Realizing the infrastructure: g^4re

Our g^{4}re tool chain exploits GENERIC, the ASG representation incorporated into the *gcc* C++ compiler, to facilitate analysis of real C++ programs. In [12], we describe our approach to encoding this representation of a translation unit as a GXL instance graph conformant to the GENERIC schema. Recently, we have added several extensions to the g^{4}re tool chain, including:

- 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,
- a *C++ API* for accessing information in the unified representation.

In this section, we describe the two most recent additions to the g^{4}re tool chain, the linking module and the C++ API.

3.1 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; linking is performed on the generated object code by the system linker, e.g. *ld* on Unix systems. 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 translation 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.

In our infrastructure, described in section 2, facilities for linking are provided by a Level I schema. Level I schemas must provide some form of *unique name*, which is not specific to a particular translation unit, for each schema entity that has an associated name attribute. Unique names, such as a mangled names or fully-qualified names, enable a module, such as a standalone linker or an API builder, to link individual translation units into a unified representation of the whole program. Requiring linking facilities to be present in a Level I schema obviates the need to provide such facilities in schemas at subsequent levels of the infrastructure.

We have recently extended our reverse engineering front-end, g^{4}re, to link all translation units from a given C++ 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 conformant to the GENERIC schema. The API builder serially transforms each ASG to an internal API representation instance, consisting of dictionaries mapping mangled 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 5 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 lan-

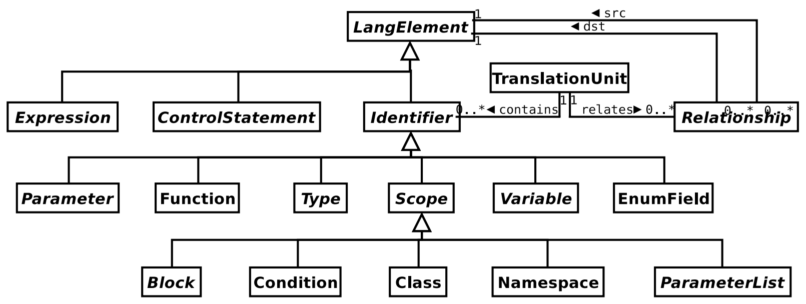


Figure 5. Partial schema for the CppInfo API. *This figure illustrates some of the main node classes in the CppInfo API, which is used to represent a translation unit as well as the result of linking two translation units.*

```
context TranslationUnit
inv:
  −− A unique name is unique in that translation unit
  self.contains→forAll( i:Identifier |
                    not self.contains→exists( j:Identifier |
                                       i<>j and uniqueName(i) = uniqueName(j)))
  −− Omitted: A unique name for static data is unique in all translation units

context TranslationUnit::link(other : TranslationUnit)
post:
  −− The new identifiers are the old ones, plus any from the other TU that weren't in here
  self.contains→forAll( id:Identifier | self.contains@pre→includes(id)
    or
  other.contains→exists( i2:Identifier | uniqueName(id) = uniqueName(i2)))
  −− The new relationships are the old ones, plus any from the other TU that weren't in here
  self.relates→forAll( r:Relationship | self.relates@pre→includes(r)
    or
  other.relates→exists( r2:Relationship | uniqueName(r.src) = uniqueName(r2.src)
                              and uniqueName(r.dst) = uniqueName(r2.dst)))
```

Figure 6. An OCL Specification of the Linking Algorithm. *Here we define the linking process as one of adding in those nodes whose unique name is new, and of adding in new relationships, suitably translated.*

guage 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 incomplete if they contain incomplete Function or Class elements.

Figure 6 presents our linking algorithm as an OCL specification over the CppInfo API schema. Since all nodes relevant to linking are subclasses of Identifier, we can define the linking process in terms of adding in new identifiers from another schema instance, and adding in new relationships to match. Since we generate canonical internal identifiers for function bodies, this process also has the effect of resolving function declarations with their corresponding definitions.

3.2 The CppInfo API

The g^{4}re tool chain provides the CppInfo API, Application Programmers Interface, for accessing information in the unified representation of a whole C^{++} program. The CppInfo API schema, a Level I schema in our infrastructure, is used to model the implementation of the CppInfo API. The schema currently consists of 39 node classes that represent declarations, scopes, types, and control structures, and 14 edge classes that represents the relationships between the node classes. The addition of node and edge classes to represent expressions remains as future work.

The CppInfo API attempts to provide a clear and flexible interface for access to the language elements in a C^{++} program. The major 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

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
tu/GXL files	65	104	245	38	52
tu size (MB)	7.76	1.93	6.06	3.33	1.38
LOC (≈)	200 K	30 K	125 K	70 K	30 K

Table 1. Testsuite. *This table lists the five test cases 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.*

the underlying graph structure of the CppInfo API, or alternatively, may access several lists containing all instances of particular CppInfo classes present in the API. Currently, these lists are provided for Namespace, Class, and Function, but we do intend to extend this point of access to include lists of additional CppInfo classes.

We used our CppInfo API to conduct the case study of Section 4. Specifically, we used the API to construct ORD's for several real C++ programs. The repository for our infrastructure [20] contains the full source code for two example programs that use the CppInfo API; these two example programs are the ORD builder and a graphical source code browser.

4 Using the infrastructure: a case study

In this section we describe the results of a feasibility study of our infrastructure. The results that we report in this section capture information about the run-time efficiency of our linking process, described in Section 3, and about classes and edges in generated ORDs, described in Section 2. All 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.

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 its corresponding GXL file, 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. These optimizations are described in reference [12].

In the next section we describe five applications that serve as a testsuite in the study. In Section 4.2 we describe some results for linking instances of the CppInfo API. In Section 4.3 we summarize results for the ORDs that we generate for the five applications and finally in Section 4.4 we apply the technique to some of the test cases included in the *gcc 3.3.4* distribution.

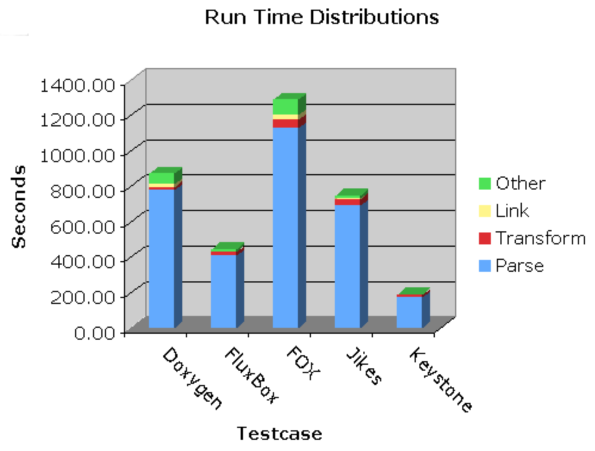


Figure 7. Timing results for linking GXL files. *The bars in this graph represent the wall-clock timings for linking the five applications in our test suite.*

4.1 The Testsuite of Applications and Libraries

Table 1 lists five applications, or test cases, that form the first 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 [25] and *FluxBox* is a light-weight X11 window management system built for speed and flexibility [6]. *FOX* is a toolkit to facilitate development of graphical user interfaces [24] and *Jikes* is a Java compiler system [9]. The final test case is *Keystone*, a parser and front-end for ISO C++ [11, 16]. The testsuite covers a range of applications including a system for documentation, an X11 window manager, libraries for creating a graphical user interface (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 fourth row of the table lists the number of GXL encoded translation units (*tu*) for each test case [2] and the

[2]The number of files for translation units listed in Table 1 differs slightly from the numbers listed in reference [12] because in this paper we no longer count external libraries packaged with the test cases.

		Doxygen	FluxBox	FOX	Jikes	Keystone
Name-spaces	Total	189	595	991	152	219
	Unique	3	30	7	4	7
Classes	Total	4 925	35 321	58 006	15 158	21 585
	Unique	538	1 901	736	677	1 060
Functs.	Total	42 684	57 357	63 463	78 892	22 321
	Unique	10 657	5 115	8 220	6 344	3 609

Table 2. Reduction in language elements due to linking. *This table shows the difference between the total number of namespaces, classes and functions in each unlinked ASG and the eventual number of unique elements in the linked ASG.*

	Doxygen	FluxBox	FOX	Jikes	Keystone
Classes	532	834	759	417	257
Association	153	100	123	486	122
Composition	104	188	45	79	40
Dependency	2 032	1 803	2 182	4 757	2 608
Inheritance	204	159	61	171	101
OwnedElement	17	82	174	47	8
Polymorphic	6 201	111	125	14 152	6 281
Total	8 711	2 444	2 718	19 693	9 160

Table 3. Application testsuite ORD Sizes. *This table shows the number of nodes (classes) and edges in the 5 ORD schema instances constructed for the applications in our test suite.*

fifth row lists the size or number of mega-bytes in the *tu* files that each test case occupies on disk. For example, the *FOX* toolkit contains 245 translation units (or GXL files), 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 Linking GXL Files for each Translation Unit

The graph in Figure 7 illustrates some timing results for our linking of GXL files for each translation unit, described in Section 3. The height of each bar in the graph represents wall-clock timings for the corresponding test case, listed along the X-axis. Each bar consists of four partitions: the lowest partition, Parse, measures the time to read and parse the GXL files for the translation units; the second lowest partition, Transform, measures the time to transform the ASGs for each translation unit into a CppInfo API instance; the second partition from the top of the bar, Link, measures the time to link the CppInfo API instances; and the topmost partition, Other, measures the time to perform other tasks not associated with parsing, transforming or linking.

To interpret the data in Figure 7 consider the middle bar representing timings for the FOX test case. The FOX test case, consisting of 245 GXL files, took the longest time to process with the parsing time dominating the entire processing time. Moreover, the time to Transform and Link the internal representations of the GXL files for the FOX test case is a minimal part of the entire processing time. This trend is reflected in each of the five test cases. Since GXL representations are large, the Parse phase of the processing, which includes time to read the GXL files into memory, dominates the total processing time, while the linking phase requires minimal time.

Table 2 lists results for linking Namespaces, Classes, and Functions for each of the five applications in our first testsuite. There are six rows of data in the table, for each of the three language elements there are two rows, Total and Unique. The row labeled Total lists the total number of each respective language element found in all GXL files for that test case and Unique lists the number of each respective language element found in the linked CppInfo API instance.

For example, the GXL files for the FOX test case collectively contained 991 namespaces, 58,006 classes and 63,463 functions; however, the linked API contained only 7 namespaces, 736 classes and 8,220 functions. This dramatic difference is due to the large amount of repetition among the individual translation units for a C++ program; this repetition is due to C++ inclusion model implemented by the preprocessor.

4.3 Generating ORD Information

In this section we present some results for the ORDs generated for the five applications in the test suite. All of the results in this section were generated with XSLT style sheets using the technique described in Section 2.3.

Table 3 lists results for the five application programs described in Section 4.1. The five columns on the right of the table list results for each test case, the first row lists the number of classes, Classes, in each test case, rows 2–7 list results for the kinds of edges in each ORD, and the final row of the table lists the total number of edges in the ORD for the respective test case. For example, the second column from the left in Table 3 lists results for the Doxygen test case, containing 532 classes (row 2) and 6 201 polymorphic edges (row 7). Polymorphic edges are generated by data attributes of a class that are references to base classes, since these data attributes can polymorphically refer to any of the classes derived from the base class in the inheritance hierarchy. The 6 201 polymorphic edges in the ORD for Doxygen indicate that this test case makes extensive use of

		g++.dg			g++.old-deja		
		inherit	lookup	template	g++.gb	g++.law	g++.oliva
Positive Test Cases		18	16	103	12	149	18
Classes	min	0	0	0	0	0	0
	max	16	5	29	4	44	3
	avg	2	2	2	2	3	1
Edges	min	0	0	0	0	0	0
	max	61	12	28	12	41	10
	avg	8	4	3	6	4	2

Table 4. gcc Testsuite ORD Summary. *This table summarises the number of nodes (classes) and edges in the 316 ORD schema instances constructed for the programs in the gcc test suite.*

inheritance and polymorphism. However, the ORD for the FluxBox test case contained only 111 polymorphic edges (second column of data, seventh row), indicating that the FluxBox application does not make extensive use of inheritance and polymorphism. The table also includes results for the number of OwnedElement edges for the test cases, where an OwnedElement edge indicates a class nested inside another class. There are several causes of a high number of OwnedElement edges, the most common being the frequent use of iterator classes, which are nested in the corresponding container class. Two other causes we observed in our first test suite are the presence of a nested class inside a base class, as seen in the FOX test case, and the presence of a nested class inside a template class, as seen in several of the test cases.

4.4 Robustness of the approach

To demonstrate the robustness of our approach, and to produce further reproducible results based on freely-available software, we generated ORDs for relevant programs from the *gcc 3.3.4* test suite. Table 4 lists summaries for three directories of test cases from each of the g++.dg and g++.old-deja directories, chosen because of the inclusion of test cases for inheritance and template functionality and for the predominance of positive test cases.[3]

The first row of data in Table 4 shows that the inherit, lookup and template subdirectories of the g++.dg directory contained 18, 16 and 103 positive test cases respectively. Similarly, the g++.gb, g++.law and g++.oliva subdirectories of the g++.old-deja directory contained 12, 149 and 18 positive test cases respectively. The remaining six rows of data provide information about the minimum, maximum and average number of classes and edges in the positive test cases in the respective subdirectory. For example, for the inherit subdirectory, some ORDS for positive test cases contained no classes, one ORD had 16 classes and the average number of classes in the ORDs in this subdirectory was 2 classes. Similarly, the last three rows of the table show that for the inherit subdirectory, some ORDs for positive test cases contained no edges, one ORD contained 61 edges and the average number was 8 edges.

[3]A positive test case passes if it executes successfully; a negative test case passes if it fails to execute successfully

5 Related Work

In their roadmap for reverse engineering, Müeller et al. identify the lack of adoption as one of the biggest challenges to increasing the effectiveness of reverse engineering, and note the lack of integration between reverse engineering tools and other software utilities as a contributory factor [17]. They assert that completely integrated environments ran against the trend for more modular toolsets, and prohibit easy integration of reverse engineering tools into these toolsets. They also note the significant progress in developing reverse engineering infrastructures and tools during the 1990s, particularly in relation to parsers, repositories, and visualization engines.

5.1 Infrastructures for reverse engineering

One of the earliest approaches to a reverse engineering infrastructure is the LSME system by Murphy and Notkin [18]. This system is based on lexical analysis and specifically identifies the ability to add additional source languages and extractors as central to the approach. This flexibility is demonstrated by applying the approach to extracting source models for ANSI C, CLOS, Eiffel, Modula 3 and TCL.

Dali is a collection of various tools in the form of a workbench for collecting and manipulating architectural information [10]. The Dali workbench was designed to be *open*, so that new tools could be easily integrated, and *lightweight*, so that such integration would not unnecessarily impact unrelated parts of the workbench. Kazman et al. identify an extraction phase, encompassing both parsing and profiling, accumulating information in a repository, which then feeds visualization and analysis phases. They use an SQL database for primary model storage, but then use application specific file formats to facilitate interchange between tools.

The Dali architecture is echoed by Salah and Mancoridids in their *software comprehension environment*, which has a three-layer architecture composed of a data gathering subsystem, a repository subsystem, and an analysis and visualization subsystem [21]. Their environment supports both static and dynamic analysis of Java and C++ programs, and information can be accessed using either SQL or a specialized higher-level query language.

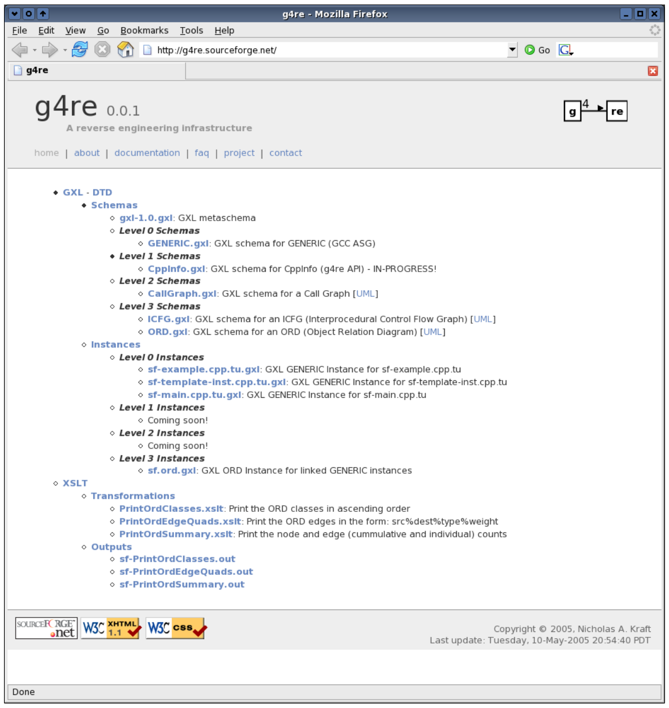
Finnigan et al. describe a *Software Bookshelf*, that was originally designed to support converting PL/I source code to C++ [5]. Their information repository, describing the content of the bookshelf, is accessed through a web server using object-oriented database technology. An implementation of these ideas as the *Portable Bookshelf* (PBS) is based around a toolkit that includes a fact extractor, manipulator and layout tools. This “pipeline philosophy” has since evolved into the SWAG Kit and the LDX/BFX pipeline, each emphasizing collections of stand-alone tools communicating only via well-defined inputs and outputs [7].

5.2 Evaluating reverse engineering tools

An important attribute of any reverse engineering infrastructure is that it provide for repeatability of results, and allow comparison of results from different approaches. One way this can be achieved is by agreement on standard schemas for representing information, which would allow output from different tools or toolsets to be directly compared. Attempts in this direction include the Dagstuhl middle model [13], GXL [8] and WoSEF [23]. Recent work by Eichberg et al. seeks to exploit the more generic standards, XML and XQuery, to provide a uniform approach to extracting information from reverse engineering tasks [4].

Even with an agreed output schema (or conversion to such a schema) there can still be considerable difficulties involved in comparing results. Murphy et al. describe a comparison of nine tools for extracting C call graphs from three software systems, and finds a considerable variance in the outputs [19]. In a paper describing a novel points-to analysis algorithm, Das notes that it took his team several months to synchronize the output from tools implementing competing approaches, so that the results could be compared [1]. In both cases, the problem was with different definitions and interpretations of the information that was required, rather than with the output format.

The importance of benchmarks in software engineering in general, and in evaluating fact extractors in particular, has been noted by Sim et al. [22]. They describe the construction of a benchmark suite designed to test the accuracy and robustness of fact extractors, and apply it to comparatively evaluate four tools. In a similar vein, Lin et al. describe a four-level hierarchy of completeness, and use this to validate the CPPX fact extractor [2, 14]. They use a test suite consisting of programs used to demonstrate the Datrix model, as well as test cases from the gcc test suite. Vinciguerra et al. describe a framework for evaluating C++ and Java disassembly and decompilation tools based around an experimentation framework that includes a layered test suite of programs as well as a focused set of reverse engineering tasks [26].


g4re - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://g4re.sourceforge.net/
Go
g4re

g4re 0.0.1

A reverse engineering infrastructure

home | about | documentation | faq | project | contact

- GXL - DTD
  - Schemas
    - gxl-1.0.gxl: GXL metaschema
    - *Level 0 Schemas*
      - GENERIC.gxl: GXL schema for GENERIC (GCC ASG)
    - *Level 1 Schemas*
      - CppInfo.gxl: GXL schema for CppInfo (g4re API) - IN-PROGRESS!
    - *Level 2 Schemas*
      - CallGraph.gxl: GXL schema for a Call Graph [UML]
    - *Level 3 Schemas*
      - ICFG.gxl: GXL schema for an ICFG (Interprocedural Control Flow Graph) [UML]
      - ORD.gxl: GXL schema for an ORD (Object Relation Diagram) [UML]
  - Instances
    - *Level 0 Instances*
      - sf-example.cpp.tu.gxl: GXL GENERIC Instance for sf-example.cpp.tu
      - sf-template-inst.cpp.tu.gxl: GXL GENERIC Instance for sf-template-inst.cpp.tu
      - sf-main.cpp.tu.gxl: GXL GENERIC Instance for sf-main.cpp.tu
    - *Level 1 Instances*
      - Coming soon!
    - *Level 2 Instances*
      - Coming soon!
    - *Level 3 Instances*
      - sf.ord.gxl: GXL ORD Instance for linked GENERIC instances
- XSLT
  - Transformations
    - PrintOrdClasses.xslt: Print the ORD classes in ascending order
    - PrintOrdEdgeQuads.xslt: Print the ORD edges in the form: src%dest%type%weight
    - PrintOrdSummary.xslt: Print the node and edge (cummulative and individual) counts
  - Outputs
    - sf-PrintOrdClasses.out
    - sf-PrintOrdEdgeQuads.out
    - sf-PrintOrdSummary.out

Copyright © 2005, Nicholas A. Kraft
Last update: Tuesday, 10-May-2005 20:54:40 PDT

Done

Figure 8. Main web page for the repository. *This screen capture shows the contents of the SourceForge repository where the artifacts described in this paper can be accessed.*

5.3 Linking in reverse engineering tools

There has been relatively little work on combining information extracted from different translation units, a process analogous to compile-time linking, where external references in one unit are resolved to definitions in another. Wu et al. describe a study of linking information extracted from a PostgresSQL implementation, and note that a naive approach to linking can give rise to linkage anomalies [27]. They describe approaches involving heuristics and build simulation to alleviate these anomalies.

6 Concluding Remarks

In this paper, we have presented an infrastructure that supports interoperability among reverse engineering tools and applications. We have built an Application Programmer’s Interface that permits extraction of information about declarations, including classes, functions and variables, as well as information about scopes, types and control statements in C++ applications. We have described our linking process for unifying API instances for each translation unit in a C++ application and provided results that describe the efficiency in linking five common C++ applications. We have also described a hierarchy of canonical schemas that capture minimal functionality for middle-level graph structures and presented an approach, using a GXL instance of a graph together with an XSLT style sheet, to permit generation of information about the graph. This approach permits an unbiased comparison of results for different tools that implement the same or a similar schema. We have constructed a repository, hosted by SourceForge.net at `http://g4re.sourceforge.net/`, and placed the artifacts of our infrastructure in this repository [20]. Figure 8 illustrates a screen capture of our web page where the repository can be accessed.

References

[1] M. Das. Unification-based pointer analysis with directional assignments. In *Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, Canada, May 2000.

[2] 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.

[3] J. Ebert, K. Kontogiannis, and J. Mylopoulus, editors. *Seminar No. 01041: Interoperability of Reengineering Tools*, Schloss Dagstuhl, Germany, January 21-26 2001.

[4] M. Eichberg, M. Mezini, K. Ostermann, and T. Schafer. XIRC: a kernel for cross-artifact information engineering in software development environments. In *Working Conference on Reverse Engineering*, pages 182–191, The Netherlands, November 8-12 2004.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

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

[7] R. Holt, M. Godfrey, and A. Malton. Swag: Software architecture group. http://swag.uwaterloo.ca/tools.html, 2005.

[8] 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.

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

[10] R. Kazman and S. J. Carrire. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.

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

[12] 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.

[13] T. C. Lethbridge, E. Plodereder, S. Tichelaar, C. Riva, P. Linos, and S. Marchenko. The Dagstuhl Middle Model (DMM), 2002. Version 0.006.

[14] Y. Lin, R. C. Holt, and A. Malton. Completeness of a fact extractor. In *Working Conference on Reverse Engineering*, pages 196– 205, British Columbia, Canada, November 13-17 2003.

[15] 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.

[16] 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.

[17] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *Conference on The Future of Software Engineering*, pages 47–60, Limerick, Ireland, June 4-11 2000.

[18] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[19] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.

[20] N. A. Kraft. g^4re reverse engineering infrastructure, version 0.0.1. Available at http://g4re.sourceforge.net.

[21] M. Salah and S. Mancoridis. Toward an environment for comprehending distributed systems. In *Working Conference on Reverse Engineering*, pages 238–247, British Columbia, Canada, November 13-17 2003.

[22] S. E. Sim, S. Easterbrook, and R. C. Holt. On using a benchmark to evaluate C++ extractors. In *International Workshop on Program Comprehension*, pages 114–123, Paris, June 2002.

[23] S. E. Sim and R. Koschke. WoSEF: Workshop on standard exchange format. *ACM SIGSOFT Software Engineering Notes*, 26:44–49, January 2001.

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

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

[26] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra. An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java. In *Working Conference on Reverse Engineering*, pages 14–23, British Columbia, Canada, November 13-17 2003.

[27] J. Wu and R. C. Holt. Resolving linkage anomalies in extracted software system models. In *International Workshop on Program Comprehension*, pages 241–245, Bari, Italy, June 24-26 2004.