Introduction

In reverse engineering, interoperability is the cooperation of two or more systems to enable the exchange and utilization of data [27]. The reverse engineering community has recognized the importance of interoperability among tools [9], as well as the difficulty in facilitating interoperability among these tools [2, 3, 6, 12, 19, 34, 43]. In their roadmap for reverse engineering, Müller et al. identify the lack of adoption of infrastructures as one of the biggest challenges to increasing the interoperability of data so that the effectiveness of reverse engineering approaches hinges on addressing this challenge [44]. An infrastructure is a set of interconnected structural elements, such as tools and schemas, that provide a framework for supporting features and facilities, such as interoperability and reuse. The lack of interoperability among reverse engineering tools and other software utilities has been highlighted as a contributory factor to the lack of adoption of available infrastructures [44].
structure, including a linking process for unifying instances of an API for all translation units in a C++ program, and present the results of a case study for our infrastructure that includes ten popular open source applications and libraries as a test suite. As part of our study, we apply XSLT style sheets to GXL instance graphs. Our implementation of the infrastructure, as well as GXL versions of the schemas in the hierarchy and our XSLT style sheets, are available in our web repository [47].

The contribution of our work is the design and implementation of our infrastructure to support interoperability, as well as a case study that demonstrates the feasibility of our infrastructure. The design of our infrastructure includes a schema hierarchy and details of the interactions among the schemas. The implementation of our infrastructure includes a collection of tools that communicate via a GXL-based pipe-filter architecture. In particular, we provide an application programmers interface (API) and a set of tools that leverage the API to perform reverse engineering tasks including: construction of graphical program representations [35, 33], computation of metrics [28], and static analysis [21]. Thus, our infrastructure operates at levels one and two as specified by Sim [9].

In Section 2 we describe previous research that relates to our infrastructure. In Section 3 we provide details about the infrastructure, including our hierarchy of canonical schemas. In Section 4 we present g4re, our implementation of the infrastructure, and describe the instantiation and linking processes for g4api. In Section 5 we list results of a case study that investigates the feasibility of our infrastructure using ten open source applications and libraries as a test suite. Finally, in Section 6 we draw conclusions and describe future work.

2 Related Work

In this section we describe the work that relates to the design and implementation of our infrastructure. In particular, we describe research on infrastructures for reverse engineering, evaluating reverse engineering tools, and linking in reverse engineering tools.

2.1 Infrastructures for reverse engineering

One of the earliest approaches to providing a general framework for interoperability is the ECMA Reference Model, the “Toaster Model”, which outlines the functionality required to support a tool integration process [48]. The dimensions of functionality addressed by the model include: data integration, provided by the repository manager; control integration, provided by the subsystem interaction manager; presentation integration, provided by the user interaction manager; and process integration, provided by the development manager.

One of the earliest approaches to a reverse engineering infrastructure is the LSME system by Murphy and Notkin [45]. 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.

Kullbach et al. present the EER/GRAL approach to graph-based conceptual modeling of multi-lingual systems [36]. In this approach, models to represent information from a single language are built and then integrated into a unified model. A graph query language is available to perform queries on the unified model.

Dali is a collection of various tools in the form of a workbench for collecting and manipulating architectural information [31]. 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 Mancoridis 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 [51]. 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++ [13]. 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 [22].

Jin and Cordy advocate non-prescriptive integration that focuses on sharing services rather than simply data with the OASIS service-sharing methodology [29]. In the OASIS architecture, each tool in the integration is known as a participant. Each participant offers a set of shared services to the other participants, but not all services offered by a participant must be shared. Two sets of components must be
created in the OASIS methodology: a domain ontology and conceptual service adapters.

Moose is a language-independent reverse- and re-engineering environment that was first developed in the context of FAMOOS [49]. Language independence is achieved by the use of a common metamodel as the core of Moose. Services provided around this core include a metamodel tailoring of the Moose metamodel, a GUI for browsing, querying, and grouping, and metrics evaluation and visualization. Moose uses both the CDIF and XMI exchange formats to interact with external tools.

Al-Ekram and Kontogiannis present an XML-based framework that attempts to represent higher level artifacts in a language-neutral way [1]. The framework includes an XML DTD for each of several artifacts, including control flow graphs, program dependence graphs, and call graphs. The basic elements that are common between the artifacts are represented as Facts and are encoded by another XML DTD, FactML. The framework is multi-layered and follows a “pipe and filter” architectural style.

2.2 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 tool sets to be directly compared. Attempts in this direction include the Dagstuhl Middle Metamodel (DMM) [37, 38], Graph Exchange Language (GXL) [24, 25] and WoSEF [11]. 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 [8].

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 [46]. 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 [5]. 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. [10]. 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 [6, 40]. 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 [59].

2.3 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 PostgreSQL implementation, and note that a naive approach to linking can give rise to linkage anomalies [60]. They describe approaches involving heuristics and build simulation to alleviate these anomalies. Guo et al. describe a method for assigning globally unique identifiers (UIDs) to the declarations and references in a Java program [17]. Each UID is based on scope and file information, and is attached to entity references in the source code using XML markup. While the goal of this work is not linking, the technique for assigning UIDs is directly applicable to linking translation units at the ASG level.

2.4 Discussion

Previous research on infrastructures has leveraged standard exchange formats (SEF) such as GXL, but has not adequately exploited the semantic specification capabilities of these SEFs. In addition, previous research has not addressed the problem of delineating interactions among schemas at the semantic level. Our infrastructure utilizes the semantic specification capabilities of GXL.

The benchmark approach to evaluating reverse engineering tools has been used in previous research for evaluation and comparison, but requires manual comparison of the results. The approach that we support with our schema hierarchy imposes an additional requirement that the tool output must conform to a common schema, or be translated to conform to a common schema. This additional requirement permits comparison of results to be fully automated.

Linking translation units from a program into a unified representation has been addressed for several languages, including PostgreSQL, Java, and C++, in previous research. We have adopted certain elements of these approaches, such as a variation of UIDs. In addition, to address the current lack of a publicly accessible repository containing representations of linked translation units for C++ programs, we
provide GXL instances of unified representations that conform to our CppInfo API schema on our SourceForge.net repository [47].

3 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 [7]. At the seminar, the participants identified three levels at which interoperability should be applied:

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

Our goal is the practical realization of interoperability through the design of schemas and the construction of tools and instance graphs at each of these three levels. Moreover, we wish to complement the plethora of schemas [3, 12, 43] and tools [2, 6, 12, 19, 34] for low-level graph structures by addressing the dearth of schemas and publicly available 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.

Figure 1 provides an overview of the hierarchy of schemas in our infrastructure that facilitate interoperability and reuse for reverse engineering tools and applications. There are two major partitions in our hierarchy: low-level and middle-level; there are five minor partitions in our hierarchy: Levels 0 through 4. The dashed ellipses in the figure represent schemas for graphical representations of code that differ for disparate languages, such as an abstract syntax graph (ASG) and an application programmer’s interface (API). The solid ellipses in the low-level partition of the figure represent the schemas used in our implementation, and are discussed further in Section 3.2. The solid ellipses in the middle-level partition of the figure represent schemas for graphical representations of code that are language independent, such as a call graph and a control flow graph. The middle-level partition of Figure 1 is discussed in Section 3.3.

3.1 Graph eXchange Language

Graph eXchange Language (GXL) is a standard exchange format (SEF) that 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 can be represented by UML class diagrams [25]. GXL provides a common base, from which any schema for representing software can be derived, through the use of explicit-external schemas and a metaschema for E-R graphs [30].

GXL was ratified as the standard format for the exchange of graphs among reverse engineering and reengineering tools at the Dagstuhl Seminar on Interoperability of Reengineering Tools. To this point, some tools have made instance graphs available in GXL, but those same tools have not made GXL schemas available. In addition, tools have previously not been designed around a GXL-based pipe-filter architecture. However, our infrastructure is designed around a GXL-based pipe-filter architecture, and the constituent components of our infrastructure, including GXL schemas and instance graphs, are available in our web repository [23].

3.2 Low-level graph schemas

Levels 0 and 1 in Figure 1 comprise the low-level partition of the hierarchy of schemas. Level 0 contains the schema for an abstract syntax graph (ASG), which contains information about a parsed and analyzed translation unit. In our implementation of the infrastructure, we use the GENERIC ASG schema, the internal ASG schema used by gcc. The GENERIC ASG schema consists of 200 concrete node classes and 75 concrete edge classes.

Level 1 of figure 1 contains a schema for an application programmer’s interface (API). In our infrastructure, an API schema is similar to a middle model such as the Dagstuhl Middle Model (DMM) [37] in that it abstracts the information found in an ASG, but differs in that it retains
more detail about declarations, statements, and some expressions (such as function calls). Despite retaining significant low-level information, the API schema lessens the cognitive burden on a user who requires access to this low-level information. The CppInfo API schema, partially illustrated in Figure 2, consists of 63 total node classes, including only 38 concrete node classes, and 38 concrete edge classes. Note that while the CppInfo API schema currently does not include representations for expressions, our preliminary work suggests that the addition of expressions will introduce no more than 20 total node classes and 10 concrete edge classes. This is in stark contrast to the GENERIC ASG schema, which uses 139 concrete node classes to represent expressions.

3.3 Middle-level graph schemas

The solid edges in Figure 1 represent the progression of information from a graphical representation at one level to a graphical representation at a subsequent level [4, 18, 54, 55]. The edge from CppInfo to Class Diagram shows that the information needed to build a class diagram can be gathered from the information about classes found in an instance of the CppInfo API schema. Similarly, an instance of the CppInfo API schema provides: the statement level information needed to build a control flow graph (CFG), the function declaration and call site information needed to build a call graph, and the statement and transfer of control information needed to build a control dependence graph (CDG). Thus, all of the schemas at Level II of Figure 1 can be built from information gathered by accessing the information in a CppInfo API instance.

To build instances of the schemas shown in Level III of Figure 1, the information found in instances of Level II schemas can be reused. For example, there is an edge from Class Diagram in Level II to ORD in Level III. This indicates that the information found in a Class Diagram instance can be reused to build an object relation diagram (ORD) [41]: the only edges not readily available in a Class Diagram are polymorphic edges, and the information needed to generate polymorphic edges can be extracted from the information in the Class Diagram instance. Thus, construction of an ORD can be accomplished by reusing the information in a Class Diagram instance.

Also at Level III of Figure 1 are schemas for an Interprocedural Control Flow Graph (ICFG) and a Program Dependence Graph (PDG). The edges from CFG and Call Graph in Level II to ICFG in Level III indicate that the information contained in a CFG instance and a Call Graph instance can be reused to build an ICFG instance; however, in this case, such reuse requires that the Call Graph instance contain information about each individual call site, as shown in Figure 4. In fact, all solid edges in Figure 1 require that instances of the source and sink schemas conform to their respective schemas. Moreover, the edge from CDG in Level II to PDG in Level III indicates that the information contained in a CDG instance can be reused to build a PDG instance [18].

Level IV of Figure 1 contains ellipses representing schemas for a Class Firewall, a Class Control Flow Graph (CCFG), and a System Dependence Graph (SDG). In-

---

1The use of the term ORD is a bit of a misnomer, since the nodes are classes, not objects; however, since the term is used in previous research, we continue its use in this paper.
Figure 2. Partial CppInfo API schema. This figure illustrates some of the main node classes in the CppInfo API schema. Details, such as attributes and operations, are elided 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.

Figure 3. 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 4. 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.

Instances of the schemas in Level IV can be built from information found in instances of schemas at Levels II and III. The edge from ORD at Level II to Class Firewall at Level III indicates that the information in an instance of the ORD schema can be reused to build an instance of the Class Firewall schema [55], while the edges from Class Diagram at Level II and ICFG at Level III to CCFG at Level IV indicate that the information contained in a Class Diagram instance and an ICFG instance can be reused to build a CCFG instance [4], and the edges from CFG at Level II and PDG at Level III to SDG at Level IV indicate that the information contained in multiple CFG instances and multiple PDG instances can be reused to build an SDG instance [54].

Figures 3, 4 and 5 illustrate schemas for an ORD, Call Graph and CFG, respectively. In our schemas, we use association classes [15] to model edges that represent associations. The use of association classes allows for subclassing and the addition of attributes and operations. For example, the ORD schema in Figure 3 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 [41]. The call graph schema in Figure 4 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 5 consists of six classes representing the flow of control between basic blocks in a program.
4 Implementation of the Infrastructure: \textit{g}^{4}\textit{re}

Our \textit{g}^{4}\textit{re} \textit{tool chain} exploits \textsc{generic}, the \textsc{asg} \textit{module}, \textsc{cppinfo} API, \textsc{transformation and linking module}, and \textsc{api} module.

The \textsc{asg} \textit{module}, \textsc{generic}, is shown as a package near the bottom left of Figure 6. The \textsc{generic} \textit{package} provides parsing of \texttt{tu} files using a scanner generated by \textsc{flex} (not shown). In addition, the \textit{package} provides parsing of \textsc{gxl} and gzipped \textsc{gxl} encodings of \texttt{tu} files using the popular libraries \textsc{expat} [56] and \textsc{zlib} [62]. The application \texttt{tu2gxl}, shown on the far left of the bottom row of Figure 6, uses the \textsc{generic} \textit{package} to transform plain text \texttt{tu} files to equivalent \textsc{gxl} files that conform to the \textsc{generic} \textsc{gxl} schema.

The \textsc{schema} \textit{module}, \textsc{cppinfo}, is shown as a package in the upper left of Figure 6. The \textsc{cppinfo} \textit{package} provides a class hierarchy, written in ISO C++, that implements the \textsc{cppinfo} API schema. The \textit{package} also provides utility classes to read and write \textsc{gxl} instances and an abstract base class that defines the interface for an API that provides access to the information found in an instance of the \textsc{cppinfo} API schema.

The \textsc{transformation and linking module}, \textsc{g4xformer}, is shown as a package in the center of Figure 6. The \textsc{g4xformer} \textit{package} provides an implementation of the transformation from the \textsc{asg} representation provided by the \textsc{generic} \textit{package} to an intermediate API representation that consists of instances of the classes provided by the \textsc{cppinfo} package. In addition, the \textit{package} implements the linking of the intermediate API instances into a single intermediate API instance that contains a unified representation of a program.

The \textsc{api} \textit{module}, \textsc{g4api}, is shown as a package with the stereotype $<$API$>$ to the right of center in the middle of Figure 6. The \textsc{g4api} \textit{package} provides a concrete implementation of the API interface provided by the schema module, and is discussed in the following section.

\subsection{\textsc{g4api}}

The \textsc{g}^{4}\textsc{re} \textit{tool chain} provides \textsc{g4api}, an Application Programmer’s Interface (API) for accessing information in the unified representation of a C++ program. The \textsc{cppinfo} API schema, described in Section 3.2, is used to model the implementation of \textsc{g4api}. Our implementation of the API provides the capability to serialize the unified representation of a program to a \textsc{gxl} instance that conforms to the \textsc{cppinfo} API schema. Additionally, the implementation is capable of deserializing the \textsc{gxl} instance into an API instance, thereby eliminating the need to repeatedly link all translation units for a C++ program.
**Figure 6. System Overview.** This figure illustrates a system diagram for g4re. The dashed lines represent “use” dependencies. The solid lines represent input and output.

**g4api** provides a clear and flexible interface for accessing the language elements in a C++ program. The first point of access provided by **g4api** is in the form of a pointer to the global namespace. Using this pointer, a user may traverse the underlying graph structure. We provide several Iterator classes, as well as an abstract base Visitor class [16], for users to leverage when accessing the API in this fashion. Alternatively, a user may access several lists containing instances of particular CppInfo schema classes present in the API. Currently, these lists are provided for Namespace, Class, Enumeration, Enumeration, Function, Variable, and Typedef, and we do intend to extend this point of access to include lists of instances for additional CppInfo schema classes. 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 Figure 6. Filter lists contain the names of source files from which instances should be ignored.

We used **g4api** to conduct the case study of Section 5. Specifically, we used the API to construct ORDs for several real C++ programs. The repository for our infrastructure [47] contains the full source code for several example programs that use **g4api**; these example programs include the ORD builder, a metrics computation system, a Class Diagram builder, and a graphical source code browser.

### 4.2 Creating API instances

**g4api** is instantiated by a user program, as shown in the upper right of Figure 6. To instantiate the API, the user program provides a list containing the names of the input files, which are shown at the top of Figure 6. The input files contain GENERIC ASG instances and may include any combination of tu, GXL, and gzipped GXL files. The process of obtaining a list of input files is illustrated in the UML Activity Diagram shown in Figure 7. We discuss the advantages and disadvantages of each input format in the case study of Section 5. **g4api** may also be instantiated by a single GXL file that encodes an instance of the CppInfo API schema, as discussed in Section 4.1.

### 4.3 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). Because reverse engineers are principally interested in analyzing whole programs, not individual translation units, a reverse engineering tool for C++ must provide a 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 3, 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 can appear in multiple translation units. 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 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.

In an instance of the CppInfo API schema, each class instance, C, is related to at least one Declaration instance, D, by one of the following relationships: identity (C \(\equiv\) D), composition (C is an attribute of D), or containment (C is contained by D). Because of the existence of one of these relationship for each class instance, we provide linking facilities in the CppInfo API schema by requiring that each Declaration be assigned a unique name. In addition, because FunctionBody is a subclass of Declaration, we also use this process to resolve function declarations to their corresponding definitions.

The implementation of our reverse engineering front end, g\(^4\)re, links all translation units from a given C++ program. In g\(^4\)re, linking is performed by g\(^4\)xformer, the transformation and linking module, on an intermediate form of the API. When instantiating the API, a user provides all translation units from a C++ program; g\(^4\)xformer serially transforms each ASG to an intermediate API instance, consisting of dictionaries mapping unique names to their CppInfo API schema node class instances, and performs linking of the intermediate 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. Intuitively, we achieve linking of schema elements by performing a traversal of the most recently constructed intermediate API instance, adding or appending elements in the existing intermediate 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.

5 Case Study

In this section we describe the results of a feasibility study of our infrastructure. 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 formatted with version 3.6 of the ReiserFS filesystem, running the Slackware 10.1 operating system. The programs were compiled using gcc version 3.3.6.

In Section 5.1 we describe ten applications and libraries that serve as the test suite in our study. In Section 5.2 we describe results for exchanging instances of low-level graph schemas. In Section 5.3 we describe results for exchanging instances of middle-level graph schemas. Finally, in Section 5.4 we extract results from GXL instances of the ORD schema by applying XSLT style sheets.

5.1 The test suite of applications and libraries

Table 1 lists the ten open source applications and libraries, or test cases, that form the test suite that we use in our study, together with important statistics about each
test case \(^2\). The header of the table lists the names that we use to refer to each of the test cases: Doxygen, FOX, FluxBox, HippoDraw, Jikes, Keystone, Licq, Pixie, Scintilla, and Scribus. Doxygen is a documentation system for C++, C, and Java [58]. FOX is a toolkit to facilitate development of graphical user interfaces [57]. FluxBox is a lightweight X11 window manager built for speed and flexibility [14]. HippoDraw provides a highly interactive data analysis environment [20]. Jikes is a Java compiler system from IBM [26]. Keystone is a parser and front end for ISO C++ [32, 42]. Licq is a multi-threaded ICQ clone [39]. Pixie is a RenderMan like photorealistic renderer [50]. Scintilla is a source code editing component that includes support for syntax styling, error indicators, code completion, and call tips [52]. The final test case is Scribus, a desktop publishing system for Unix-like platforms [53].

The remaining three rows of data in Table 1 list relevant details of the test cases. The first row of the table lists the version number and the second row lists the number of translation units. Finally, the third row lists the approximate number of thousands of lines of non-commented, non-preprocessed lines of code.

### 5.2 Exchanging low-level graph instances

In this section we investigate the costs associated with exchanging instances of low-level graphs; in particular, we investigate the costs of exchanging instances of both the GENERIC ASG schema and the CppInfo API schema. First, we illustrate and discuss the different exchange formats used in g4re. Next, we measure and discuss the storage costs of exchanging low-level graphs. Finally, we measure and discuss the run-time costs of exchanging low-level graphs, and summarize the results of this section.

Recall that our g4re tool chain accepts multiple input formats, as shown in Figures 6 and 7. The definition of a C++ class, 

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Doxygen</th>
<th>FOX</th>
<th>FluxBox</th>
<th>HippoDraw</th>
<th>Jikes</th>
<th>Keystone</th>
<th>Licq</th>
<th>Pixie</th>
<th>Scintilla</th>
<th>Scribus</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ Translation Units</td>
<td>1.4.4</td>
<td>1.4.17</td>
<td>0.9.14</td>
<td>1.15.8</td>
<td>1.22</td>
<td>0.2.5</td>
<td>1.3.0</td>
<td>1.5.2</td>
<td>1.66</td>
<td>2.45</td>
</tr>
<tr>
<td>NCLOC (≈ K)</td>
<td>69</td>
<td>245</td>
<td>103</td>
<td>249</td>
<td>38</td>
<td>52</td>
<td>28</td>
<td>78</td>
<td>78</td>
<td>110</td>
</tr>
</tbody>
</table>

**Table 1. Testsuite.** This table lists the ten test cases that we use in our study. For each test case, we list the version, the number of translation units, and the approximate number of lines of non-commented, non-preprocessed lines of code (NCLOC).

we see that GXL (and hence XML) is more verbose than the gcc \(\text{tu}\) format; the respective character counts for the text in the three figures are 497, 1503, and 1307. Also, note that the text in Figure 8 contains information not present in Figures 9 and 10, including addresses and string lengths.

It is well known that XML imposes significant storage costs; however, this fact has not hindered the wide spread adoption of XML. Due to the prevalence of XML, there are several tools, available in several popular languages such as C, C++, and Java, that were designed with these costs in mind. In particular, several XML parsers, including expat, allow for easy integration with libraries that read and write compressed files, including zlib.

Table 2 presents a summary of the run-time costs for the low-level graphs representing each of the test cases. A comparison of rows 1 and 2 of the table shows the significant storage cost introduced by the use of an XML dialect (in this case GXL). For each test case, encoding the \(\text{tu}\) files in GXL more than doubled the storage costs; for example, the total storage cost of the \(\text{tu}\) files for Jikes is 872 megabytes, but the total storage cost of the GXL encodings of the \(\text{tu}\) files is 2,289 megabytes. A comparison of rows 2 and 4 of Table 2 shows the savings in storage cost achieved through the elimination of duplicated information by the linking process. The savings vary by test case; higher savings suggest that the size of the set of header files included in multiple translation units is large, while lower savings suggest that the size of the set is small. A comparison of rows 1 and 4 of the table again shows the significant storage cost introduced by GXL; even with the elimination of duplicated information, the storage costs shown in row 4 are larger than those in row 1 for all test cases. Rows 3 and 5 show the significant savings in storage cost that compression introduces when compared to rows 2 and 4, respectively. Next, we investigate the run-time costs introduced by the use of GXL and compressed GXL files.

Table 3 presents a summary of the run-time costs for parsing and building an in-memory representation for the low-level graphs representing each of the test cases. As stated in Section 4, we parse \(\text{tu}\) files using a flex generated scanner, GXL files using expat, and compressed GXL files using the combination of expat and zlib. We use the same node, edge, and graph data structures to store each graph instance of the GENERIC in Figure 8, the corresponding definition as a partial GXL class, in the\(\text{tu}\) file format is shown in Figure 8, the corresponding definition as a partial GXL instance of the GENERIC schema is shown in Figure 9, and the corresponding definition as a partial GXL instance of the CppInfo API schema is shown in Figure 10. Immediately,
Figure 8. Instance of a tu file. This figure illustrates the definition of class Parser as represented in a tu file. A node definition in a tu file consists of: a unique integer prepended with “@”, a string representing the node type, edges of the form “edge: dest”, fields of the form “field: value”, and a set of single word attributes.

Figure 9. GXL instance of the GENERIC schema. This figure illustrates the definition of class Parser as represented in a GXL instance of the GENERIC schema. The GENERIC GXL schema is a direct encoding of the tu file format, but with internal gcc information, such as addresses and string lengths, omitted. The “@” symbol is translated to “n” to conform to XML standards.
Figure 10. GXL instance of the CppInfo schema. This figure illustrates the definition of class Parser as represented in a GXL instance of the CppInfo schema.

Table 2. Size on disk (MB). This table lists the size on disk, in megabytes, for low-level graphs. Row 1 lists the total size of the tu files for each test case, rows 2 and 3 list the total sizes of the uncompressed and compressed GXL encoded tu files, respectively, and rows 4 and 5 list the total sizes of the uncompressed and compressed GXL instances of the CppInfo schema, respectively.

Table 3. Time (s). This table lists the running time, in seconds, to parse and build in-memory representations of low-level graphs for each test case. Row 1 lists the total time for the tu files for each test case, rows 2 and 3 list the total times for the uncompressed and compressed GXL encoded tu files, respectively, and rows 4 and 5 list the times for the uncompressed and compressed GXL instances of the CppInfo schema, respectively.
instance in memory. A comparison of rows 1 and 2 of Table 3 shows the run-time cost introduced by the use of GXL. The running times for GXL input are consistently higher than those for XML input, but the run-time cost introduced by GXL is much lower than the corresponding storage cost. A comparison of rows 1 and 4 shows a significant savings in run-time cost when dealing with a linked representation of a program. In addition, a comparison of rows 2 and 4 with rows 3 and 5, respectively, shows that while a run-time cost is introduced when dealing with compressed files, the cost is minimal when the corresponding savings in storage cost is considered.

The results for exchanging low-level graphs show that the storage costs for uncompressed files are prohibitive, as are the run-time costs for unlinked graphical program representations. Despite the savings achieved through the exchange of compressed and linked low-level graphs, both the storage and run-time costs are comparatively high, as the results in the next section show.

5.3 Exchanging middle-level graph instances

In this section we investigate the costs associated with exchanging instances of middle-level graphs; in particular, we investigate the costs of exchanging GXL instances of the ORD schema presented in Section 3.3. First, we measure and discuss the storage costs of exchanging middle-level graphs. Next, we measure and discuss the run-time costs of exchanging middle-level graphs, and summarize the results of this section.

Figure 11 illustrates a prototypical GXL instance of the ORD schema. Table 4 presents a summary of the storage costs for the ORD instances representing each of the test cases. Immediately, we see that the storage costs for the uncompressed ORD instances are an order of magnitude smaller than those of the compressed and linked low-level graphs from the previous section. In addition, the storage costs for the compressed ORD instances are at most 0.532 megabytes for FOX and as little as 0.011 megabytes for Keystone.

Table 5 presents a summary of the run-time costs for parsing and building an in-memory representation for the ORD instances representing each of the test cases. We use the same tools and data structures for parsing and in-memory representation that we described in Section 5.2. As with storage costs, the run-time costs for ORD instances are at least one order of magnitude smaller than those for the compressed and linked low-level graphs from Section 5.2. In addition, the run-time costs introduced when dealing with compressed files are insignificant.

The results for exchanging middle-level graphs show, for both storage and run-time costs, savings of at least one order of magnitude when compared to the results for exchanging low-level graphs. Thus, the results indicate significant savings in the costs of exchange for applications that do not require full low-level information about a program. For example, an application that builds a class firewall can take advantage of these savings by using ORD instances, rather than ASG or API instances, as input. Other applications of these savings are described in Section 3.

5.4 Transforming graph instances using XSLT

In this section we apply transformations to the GXL instances of the ORD schema that is discussed in the previous section. First, we illustrate an XSLT style sheet for summarizing GXL instances of the ORD schema. Next, we apply the XSLT style sheet to the ORD instances discussed in Section 5.3.

Figure 12 illustrates an XSLT style sheet for summarizing the information in an ORD GXL instance. As mentioned in Section 3.4, the style sheet is specified at the schema level. The style sheet defines nine variables that contain the sets of classes, edges, association edges, composition edges, dependency edges, inheritance edges, owned element edges, and polymorphic edges, respectively. The style sheet contains nine statements that output the sizes of the sets.

Table 6 summarizes the results of applying the XSLT style sheet to the ORD GXL instances described in Section 5.3. We applied the style sheet to the ORD GXL instance for each test case using xsltproc [61], which is freely available and runs on many platforms. Row 1 of Table 6 lists the running time in seconds for xsltproc to apply the style sheet to each test case. The running times are all small, ranging from 0.07 seconds for Keystone to 8.00 seconds for FOX. Row 2 of the table lists the number of classes found in the ORD for each test case. Rows 3 through 8 of the table list the number of the respective edge types found in the ORD for each test case. The bottom row of Table 6 lists the total number of edges found in the ORD for each test case.

5.5 Threats to validity

There are several threats to the validity of our studies. An external threat to our studies ensues from our use of the gcc compiler at Level 0 of the infrastructure: generation of C++ files that work with our system is dependent on versions 3.3.0 through 3.4.0 of the gcc compiler. However, the artifacts of the g^re system at Level I, including the CppInfo API, can be compiled and used with any version of the gcc C++ compiler, and the schemas and artifacts at levels higher than Level 1 are independent of any compiler.

A second external threat to our study derives from the fact that, to our knowledge, no researchers other than the authors of this paper have exploited our infrastructure, in-
Figure 11. ORD GXL instance. This figure illustrates a GXL instance of the ORD schema containing two classes and one edge.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Doxygen</th>
<th>FOX</th>
<th>FluxBox</th>
<th>HippoDraw</th>
<th>Jikes</th>
<th>Keystone</th>
<th>Licq</th>
<th>Pixie</th>
<th>Scintilla</th>
<th>Scribus</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ord.gxl</td>
<td>10</td>
<td>12</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>0.175</td>
<td>3</td>
<td>3</td>
<td>0.614</td>
<td>2</td>
</tr>
<tr>
<td>.ord.gxl.gz</td>
<td>0.441</td>
<td>0.532</td>
<td>0.082</td>
<td>0.171</td>
<td>0.378</td>
<td>0.011</td>
<td>0.125</td>
<td>0.125</td>
<td>0.031</td>
<td>0.058</td>
</tr>
</tbody>
</table>

Table 4. Size on disk (MB). This table lists the size on disk, in megabytes, for ORD GXL instances. Row 1 lists the size of the ORD GXL instances for each test case, and row 2 lists the size of the compressed ORD GXL instances for each test case.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Doxygen</th>
<th>FOX</th>
<th>FluxBox</th>
<th>HippoDraw</th>
<th>Jikes</th>
<th>Keystone</th>
<th>Licq</th>
<th>Pixie</th>
<th>Scintilla</th>
<th>Scribus</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ord.gxl</td>
<td>0.58</td>
<td>0.69</td>
<td>0.10</td>
<td>0.22</td>
<td>0.51</td>
<td>0.01t</td>
<td>0.16</td>
<td>0.16</td>
<td>0.04</td>
<td>0.08</td>
</tr>
<tr>
<td>.ord.gxl.gz</td>
<td>0.63</td>
<td>0.74</td>
<td>0.10</td>
<td>0.24</td>
<td>0.58</td>
<td>0.01t</td>
<td>0.17</td>
<td>0.18</td>
<td>0.04</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 5. Time (s). This table lists the running time, in seconds, to parse and build in-memory representations of ORD GXL instances for each test case. Row 1 lists the time for the ORD GXL instances for each test case, and row 2 lists the time for the compressed ORD GXL instances for each test case.
Figure 12. XSLT style sheet for summarizing ORD instances. This figure illustrates the XSLT style sheet used to generate the results listed in Table 6
including use of the g4re system, the GXL encodings of 

files, and the schemas stored in the sourceforge repository 

[21, 35]. However, our results have been available for less 

than a year and we believe that our publication and use 

of the infrastructure will promote its use among other re-

searchers.

Threats to the internal validity consist of possible errors 

in our implementation and measurement tools that might 

affect outcomes. To control these threats, we continually 

validated both the implementation and the timings using 

internal validity checks. For example, our g4re tool chain 

accepts multiple input formats, and the size of a 

tu file is always smaller than its GXL formatted counterpart, the time 

to parse and build an in-memory representation of a low 

level graph stored in 

tu format is always less than the time 

for its GXL counterpart, and we have validated that parsing 

and building an in-memory representation from each of 

those instances created by our g4re and GXL files yields the same graph.

5.6 Discussion

The results of our studies demonstrate the feasibility of 

our GXL-based infrastructure. In particular, we report re-

sults for ten medium sized, popular, open-source programs.

These programs cover a range of applications including 

graphical interface APIs, language and text processing tools 

and desktop publishing. Our ongoing work includes fortify-

ing our repository with additional programs and artifacts 

to facilitate comparison and reproduction of results among 

the community of researchers.

A comparison of rows 1 and 2 of Table 2 illustrates the 

storage cost introduced by the use of GXL. In particular, 

the GXL representations of 

tu files are at least 2.34 times larger than the corresponding 
tu files in all cases, and as much as 2.93 times larger for the HippoDraw test case. 

However, a comparison of rows 1 and 3 of the same table 

shows that when we compress the GXL, the storage cost for 

these compressed files is less than the storage cost of the original 
tu files. The 
tu files are at least 3.84 times larger 

than the corresponding compressed GXL representations of 
tu files in all cases, and as much as 5.91 times larger for the Pixie test case.

A comparison of rows 1 and 3 of Table 3 illustrates the 

run-time cost introduced by the use of compressed GXL. In 

particular, parsing and building in-memory representations 

for 
tu files is at least 1.03 times faster than for the cor-

responding compressed GXL representations in all cases, 

and as much as 1.24 times faster for the Scintilla test case. Clearly, the run-time cost introduced by compressed GXL is much lower than the corresponding introduced storage cost. This result is important for working with graphs at all 

levels, and is particularly important for working with low-level graphs such as ASGs or APIs.

Finally, Tables 4 and 5 illustrate that the run-time and 

storage costs for instances of the ORD, a middle-level 

graph, are at least one order of magnitude smaller than those 

for the compressed and linked low-level graphs. This is not 

particularly surprising, but it does underscore the savings 

that can be achieved by inter-operating at the middle, rather 

than low, level. In addition, in Table 6 we observe that pro-

cessing middle-level graph instances, such as ORDs, with 

widely available tools, such as xsltproc, is efficient. For ex-

ample, we extracted information from an ORD with over 

450 classes and thousands of edges in less than ten seconds. Furthermore, our XSLT ORD transformations can be applied to any conforming instance of our ORD schema, not 

just those instances created by our g4re system.

6 Conclusions and Future Work

In this paper, we have expanded on an infrastructure, first 

introduced in [35], that supports interoperability among re-

verse engineering tools and applications. We have described 

the schema hierarchy that is central to our approach, de-

tailed the interactions among instances of the schemas, and 

illustrated several of the schemas. We have implemented 

the infrastructure, described our implementation, and pre-

sented the results of a case study for our infrastructure.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Doxygen</th>
<th>FOX</th>
<th>FluxBox</th>
<th>HippoDraw</th>
<th>Jikes</th>
<th>Keystone</th>
<th>Liq</th>
<th>Pixie</th>
<th>Scintilla</th>
<th>Scribus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>6.15</td>
<td>8.00</td>
<td>0.86</td>
<td>1.92</td>
<td>5.28</td>
<td>0.07</td>
<td>1.40</td>
<td>1.35</td>
<td>0.30</td>
<td>0.59</td>
</tr>
<tr>
<td>Classes</td>
<td>562</td>
<td>465</td>
<td>243</td>
<td>253</td>
<td>322</td>
<td>62</td>
<td>225</td>
<td>263</td>
<td>89</td>
<td>208</td>
</tr>
<tr>
<td>Association</td>
<td>397</td>
<td>385</td>
<td>138</td>
<td>173</td>
<td>647</td>
<td>49</td>
<td>32</td>
<td>323</td>
<td>52</td>
<td>247</td>
</tr>
<tr>
<td>Composition</td>
<td>585</td>
<td>353</td>
<td>226</td>
<td>29</td>
<td>78</td>
<td>8</td>
<td>26</td>
<td>78</td>
<td>79</td>
<td>33</td>
</tr>
<tr>
<td>Dependency</td>
<td>1414</td>
<td>18460</td>
<td>5499</td>
<td>8631</td>
<td>7292</td>
<td>554</td>
<td>3793</td>
<td>4588</td>
<td>2198</td>
<td>5078</td>
</tr>
<tr>
<td>Inheritance</td>
<td>335</td>
<td>220</td>
<td>198</td>
<td>180</td>
<td>164</td>
<td>25</td>
<td>161</td>
<td>139</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>OwnedElement</td>
<td>10</td>
<td>173</td>
<td>21</td>
<td>0</td>
<td>18</td>
<td>3</td>
<td>1</td>
<td>29</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Polymorphic</td>
<td>26168</td>
<td>31116</td>
<td>1311</td>
<td>7143</td>
<td>28762</td>
<td>137</td>
<td>7760</td>
<td>6383</td>
<td>469</td>
<td>168</td>
</tr>
<tr>
<td>Total</td>
<td>42086</td>
<td>50707</td>
<td>7413</td>
<td>16156</td>
<td>36961</td>
<td>776</td>
<td>11771</td>
<td>11540</td>
<td>2813</td>
<td>5554</td>
</tr>
</tbody>
</table>

Table 6. ORD sizes for the testsuite. This table lists the number of classes and edges in the 10 ORD schema instances constructed for the applications and libraries in our testsuite.
For this feasibility study, we use ten popular open source applications and libraries as a test suite. As part of our study, we applied XSLT style sheets to GXL instance graphs. The results of the study for exchanging middle-level graphs show, for both storage and run-time costs, savings of at least one order of magnitude when compared to the results for exchanging low-level graphs. Our implementation of the infrastructure, as well as GXL versions of the schemas in the hierarchy and our XSLT style sheets, are available in our web repository. Our web repository is available at http://g4re.sourceforge.net/.

To evaluate the usability of our tool and the techniques presented in the infrastructure, we introduced some of the concepts found in this paper into a graduate course in program analysis techniques. The Clemson University graduate students were given two assignments that made use of the infrastructure. In the first assignment, the students wrote C++ programs that accessed the g4api to build conforming instances of our GXL class diagram schema. In the second assignment, the students parsed and built in-memory representations of GXL instances of our ORD schema, and computed class firewalls using only the information obtained from the GXL instance. The students then output conforming GXL instances of the class firewall schema. For the second assignment, the students submitted solutions in an array of languages, including C, C++, Java, and Lisp.

Our future work includes further investigation into the hierarchy of canonical schemas described in Section 3. The hierarchy in Figure 1 illustrates the use of instances of schemas at Level I to build instances of schemas at Level II, the reuse of instances of schemas at Levels II to build instances of schemas at Level III, and the reuse of instances of schemas at Levels II and III to build instances of schemas at Level IV. Our ongoing work includes a formalization of these schemas and relationships as well as an investigation into schemas for other graph structures required for reverse engineering of code, such as the points-to escape graph.

7 Acknowledgements

We would like to thank the anonymous reviewers for their careful read of our paper and for their thoughtful, perceptive and helpful comments.

References


