# 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

*An infrastructure is a set of interconnected structural elements, such as tools and schemas, that provide a framework for supporting an entire structure. The reverse engineering community has recognized the importance of interoperability, the cooperation of two or more systems to enable the exchange and utilization of data, and has noted that the current lack of interoperability is a contributing factor to the lack of adoption of available infrastructures. To address the problems of interoperability and reproducing previous results, we present an infrastructure that supports interoperability among reverse engineering tools and applications. We present the design of our infrastructure, including the hierarchy of schemas that captures the interactions among graph structures. We also develop and utilize our implementation, which is designed using a GXL-based pipe-filter architecture, to perform a case study that demonstrates the feasibility of our infrastructure.*

## 1 Introduction

In reverse engineering, interoperability is the cooperation of two or more systems to enable the exchange and utilization of data [46]. The reverse engineering community has recognized the importance of interoperability among tools [51], as well as the difficulty in facilitating interoperability among these tools [2, 3, 6, 9, 16, 30, 39]. In their roadmap for reverse engineering, Müeller 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 [40]. An infrastructure is a set of interconnected structural elements, such as tools and schemas, that provide a framework for supporting an entire structure. 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 [40].

The issues involved in promoting interoperability among reverse engineering tools and applications have been discussed at the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [7]. At the seminar, the participants identified three levels at which interoperability should be applied: low-level syntax, middle-level graph structures and high-level architectures. The importance of facilitating interoperability is becoming increasingly recognized for its importance in permitting reuse of reverse engineering artifacts as well as enabling the reproduction of results from previous scientific research.

Two important activities involved in most research endeavors entail the development of an approach that is an improvement on existing approaches and then conducting experiments to show that the new approach is an improvement over existing approaches. To evaluate the new approach, the researcher is typically required to implement at least one previously developed technique as an unbiased basis for comparison with the newly developed 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 all too frequently 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 [5, 42].

To address the problems of interoperability and reproducing previous results, we presented an infrastructure that supports interoperability among reverse engineering tools and applications [31]. In this paper, we expand on the infrastructure in several important directions. We develop and extend a schema hierarchy that is central to our approach,

detail the interactions among instances of the schemas, and illustrate several of the schemas. In addition, we present our implementation of the essential components of the infrastructure, including a linking process for unifying instances of an API for all translation units in a C++ program. In addition, we 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 [43].

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 entails 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 addition, 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 [31, 29], computation of metrics [24], and static analysis [18]. Thus, our infrastructure operates at levels one and two as specified by Sim [51].

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 g$^4$re, 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 [45]. 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.

Another early approach to a reverse engineering infrastructure is the LSME system by Murphy and Notkin [41]. 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 [32]. 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 [27]. 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 [48]. 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++ [10]. 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 [19].

Jin and Cordy advocate *non-prescriptive* integration that focuses on sharing *services* rather than simply data with the OASIS service-sharing methodology [25]. 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 [44]. Language independence is acheived by the use of a common metamodel as the core of Moose. Services provided around this core include meta-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) [33, 34], Graph Exchange Language (GXL) [22, 20] and WoSEF [53]. 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 [42]. 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. [52]. 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, 36]. 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 PostgresSQL 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 [14]. 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 comparision of results to be fully automated.

Linking translation units from a program into a unified representation has been addressed for several languages, including PostgresSQL, 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 publically 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 [43].

## 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, 9, 39] and tools [2, 6, 9, 16, 30] 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 language defined by a 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 [22]. 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 [26].

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 [21].

### 3.2 Low-level graph schemas

Levels 0 and I in Figure 1 comprise the low-level partition of the hierarchy of schemas. Level 0 contains the schema for an abstract semantic 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 I 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 Metamodel (DMM) [33] in that it abstracts the in-

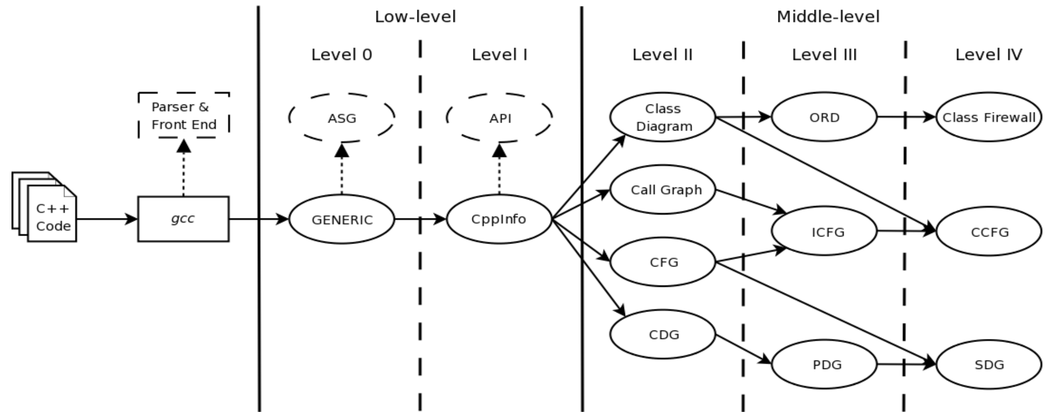


**Figure 1. Overview of the Schemas in the Infrastructure.** *This figure illustrates the levels in our infrastructure. The dashed edges represent realization. The solid edges represent the progression of information from a graphical representation at one level to a graphical representation at a subsequent level.*

formation 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, 15, 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) [1] [37]; 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 [15].

Level IV of Figure 1 contains ellipses representing schemas for a Class Firewall, a Class Control Flow Graph

[1] The 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.

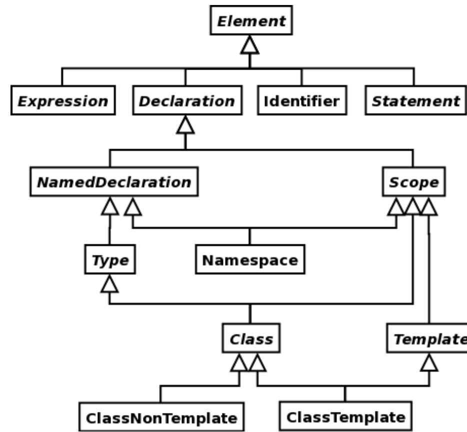


**Figure 2. Partial CppInfo API schema.** *This figure illustrates some of the main node classes in the CppInfo API schema. Details, such as attributes and operations, are elided here, but are available in the full version of the schema. The full version of the schema is available in our web repository as both a GXL schema and a UML class diagram.*

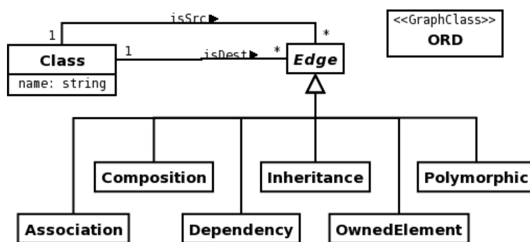


**Figure 3. Schema for an ORD.** *This figure illustrates a schema for an Object Relation Diagram (ORD). An ORD is a graph consisting of nodes representing classes, and edges representing relationships between the classes.*

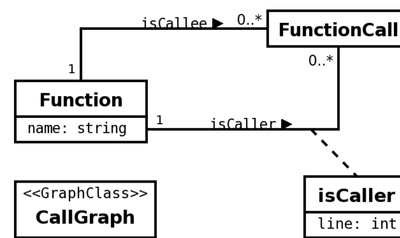


**Figure 4. Schema for a Call Graph.** *This figure illustrates a schema for a call graph, a graph whose nodes represent either functions or function call sites and whose edges represent function invocations.*

(CCFG), and a System Dependence Graph (SDG). 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 a 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* [12] 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 [37]. 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 ba-

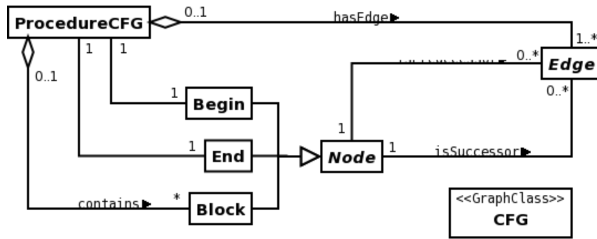


**Figure 5. Schema for a Control Flow Graph.** *This figure illustrates a schema for a control flow graph, whose nodes represent blocks of straight-line code and whose edges represent flow of control between the blocks.*

sic blocks in a program.

### 3.4 Comparing graph instances

Each of the schemas in Figure 1 can be expressed as a GXL [21] schema and used to facilitate comparison of results for a given graph structure. The schemas are designed to be minimal yet complete, and hence to represent only the information required to construct a given graph structure. To perform a comparison, the tools under study are not required to produce instances of the same schema; however, comparison of the instances generated for each tool can only be undertaken for those parts of the schema that are common to both tools. Alternatively, comparison of instances can be undertaken if the instances are transformed to instances of the appropriate schema provided by our infrastructure.

One technique for comparison of GXL instance graphs is the use of XSLT style sheets. These style sheets represent transformations that are specified at the schema level, that is, the transformations can be applied to any conforming instance of the given schema. All of the results in Section 5.4 were obtained in this manner.

## 4 Implementation of the Infrastructure: $g^4re$

Our g4re *tool chain* exploits GENERIC, the ASG representation incorporated into the *gcc* C++ compiler, to facilitate analysis of real C++ programs. We use a command line flag, `-fdump-translation-unit-all`, to obtain a plain text representation of the GENERIC instance for each translation unit in a program from *gcc*. These representations are known as `tu` files. The use of `tu` files provides flexibility over hard-coding our solution into the *gcc* source code, and additionally, fits the theme of exchange among reverse engineering tools.

We describe g4re as a *tool chain*, because the applications and libraries that constitute g4re can be used individually or in concert. Each application or library in the chain takes, as input, the output of the preceding application or library in the chain. As a result, our implementation is modular; the modules in our system include: the *ASG module*, the *schema module*, the *transformation and linking module*, and the *API module*.

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

The *schema module*, cppinfo, is shown as a package in the upper left of Figure 6. The cppinfo package provides a class hierarchy, written in ISO C++, that implements the CppInfo API schema. The package also provides utility classes to read and write 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 CppInfo API schema.

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

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

### 4.1 g4api

The g4re tool chain provides g4api, an Application Programmer's Interface (API) for accessing information in the unified representation of a C++ program. The CppInfo API schema, described in Section 3.2, is used to model the implementation of g4api. Our implementation of the API provides the capability to serialize the unified representation of a program to a GXL instance that conforms to the CppInfo API schema. Additionally, the implementation is capable of deserializing the GXL instance into an API instance, thereby eliminating the need to repeatedly link all

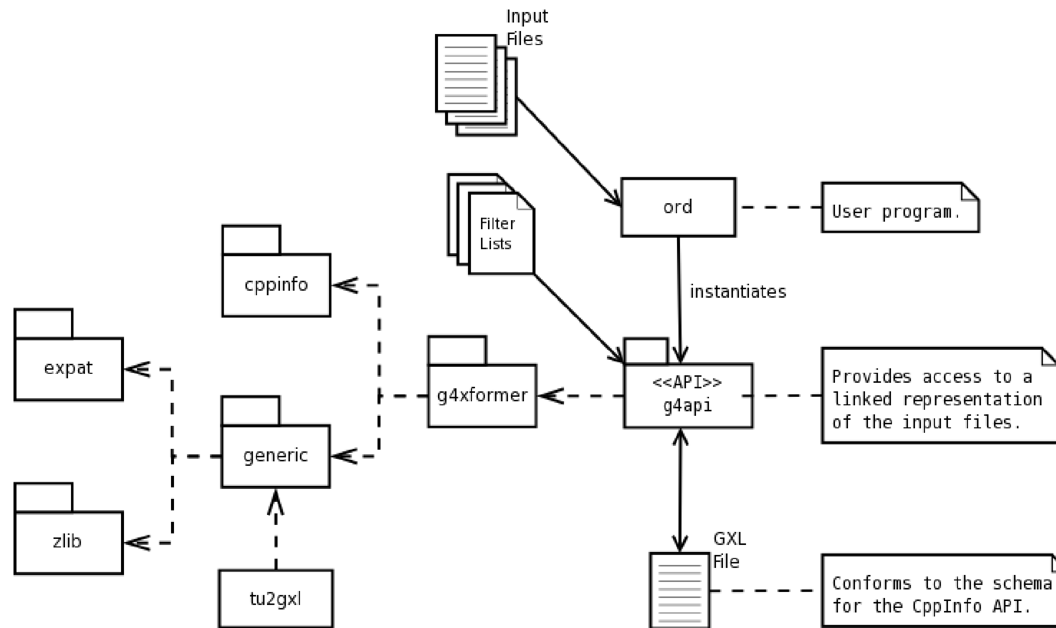


**Figure 6. System Overview.** *This figure illustrates a system diagram for g$^4$re. The dashed lines represent "use" dependencies. The solid lines represent input and output.*

translation units for a C++ program.

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 [13], 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 [43] 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` files, GXL files, 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

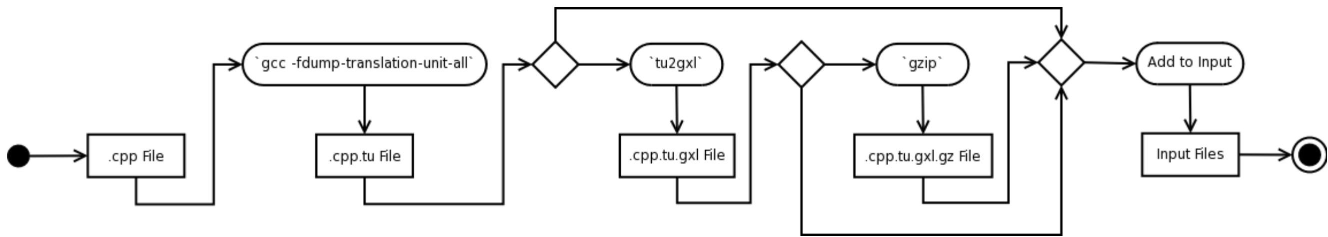


**Figure 7. UML Activity Diagram for g4api Input.** *This figure illustrates the process of creating input files for use with the g4api API in any of three different formats.*

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 ($C$ is $D$), composition ($C$ is an attribute of $D$), or containment ($C$ is contained by $D$). Because of the existance 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 g4xformer, 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; g4xformer 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.

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| Version | 1.4.4 | 1.4.17 | 0.9.14 | 1.15.8 | 1.22 | 0.2.3 | 1.3.0 | 1.5.2 | 1.66 | 1.2.3 |
| C++ Translation Units | 69 | 245 | 107 | 249 | 38 | 52 | 28 | 78 | 78 | 110 |
| NCLOC (≈ K) | 170 | 110 | 32 | 55 | 70 | 16 | 36 | 80 | 35 | 80 |

**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).*

### 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 [11]. *HippoDraw* provides a highly interactive data analysis environment [17]. *Jikes* is a Java compiler system from IBM [23]. *Keystone* is a parser and front end for ISO C++ [28, 38]. *Licq* is a multi-threaded ICQ clone [35]. *Pixie* is a RenderMan like photorealistic renderer [47]. *Scintilla* is a source code editing component that includes support for syntax styling, error indicators, code completion, and call tips [49]. The final test case is *Scribus*, a desktop publishing system for Unix-like platforms [50].

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 C++ 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 g$^4$re. 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 g$^4$re tool chain accepts multiple input formats, as shown in Figures 6 and 7. The definition of a C++ class, `Parser`, in the GENERIC `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, we see that GXL (and hence XML) is more verbose than the *gcc* `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 storage 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 `tu` files in GXL more than doubled the storage costs; for example, the total storage cost of the `tu` files for Jikes is 872 megabytes, but the total storage cost of the GXL encodings of the `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

[2] Additional information about each test case is available in our online repository.

```
@48049  type_decl        name: @50737   type: @50738   scpe: @13938
                         srcp: parser.h:76             artificial
                         chan: @50739   addr: b66969a0

@50737  identifier_node  strg: Parser   lngt: 6        addr: b66b3ac0

@50738  record_type      name: @48049   size: @53498   algn: 32
                         base: @23733   public         struct
                         flds: @53499   fncs: @53500   binf: @53501
                         addr: b6696850
```

**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.*

---

```
<node id="n48049">
   <type xlink:href="GENERIC.gxl#type_decl"/>
   <attr name="attr"><set><string>artificial</string></set></attr>
   <attr name="srcp"><string>parser.h:76</string></attr>
</node>
<edge from="n48049" to="n50739"><type xlink:href="GENERIC.gxl#type_decl_chan"/></edge>
<edge from="n48049" to="n50738"><type xlink:href="GENERIC.gxl#type_decl_type"/></edge>
<edge from="n48049" to="n50737"><type xlink:href="GENERIC.gxl#type_decl_name"/></edge>

<node id="n50737">
   <type xlink:href="GENERIC.gxl#identifier_node"/>
   <attr name="attr"><set></set></attr>
   <attr name="strg"><string>Parser</string></attr>
</node>

<node id="n50738">
   <type xlink:href="GENERIC.gxl#record_type"/>
   <attr name="attr"><set><string>struct</string></set></attr>
   <attr name="qual"><string></string></attr>
</node>
<edge from="n50738" to="n23733">
   <type xlink:href="GENERIC.gxl#record_type_base"/>
   <attr name="base"><tup><bool>false</bool><string>public</string></tup></attr>
</edge>
<edge from="n50738" to="n53500"><type xlink:href="GENERIC.gxl#record_type_fncs"/></edge>
<edge from="n50738" to="n53501"><type xlink:href="GENERIC.gxl#record_type_binf"/></edge>
<edge from="n50738" to="n53499"><type xlink:href="GENERIC.gxl#record_type_flds"/></edge>
<edge from="n50738" to="n48049"><type xlink:href="GENERIC.gxl#record_type_name"/></edge>
```

**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.*

---

```
<node id="n31873">
   <type xlink:href="CppInfo.gxl#Identifier"/>
   <attr name="string"><string>parser.h</string></attr>
</node>

<node id="n53656">
   <type xlink:href="CppInfo.gxl#ClassNonTemplate"/>
</node>
<edge from="n53656" to="n41484">
   <type xlink:href="CppInfo.gxl#Base"/>
   <attr name="inheritanceSpec"><tup><bool>false</bool><string>public</string></tup></attr>
</edge>
<edge from="n53656" to="n4"><type xlink:href="CppInfo.gxl#DefinedIn"/></edge>
<edge from="n53656" to="n31873"><type xlink:href="CppInfo.gxl#SourceFile"/></edge>
<edge from="n53656" to="n53657"><type xlink:href="CppInfo.gxl#Name"/></edge>
<edge from="n53656" to="n53658"><type xlink:href="CppInfo.gxl#Enumerations"/></edge>
<edge from="n53656" to="n53661"><type xlink:href="CppInfo.gxl#Enumerators"/></edge>
<edge from="n53656" to="n53664"><type xlink:href="CppInfo.gxl#Functions"/></edge>
<edge from="n53656" to="n54657"><type xlink:href="CppInfo.gxl#Functions"/></edge>
<edge from="n53656" to="n54742"><type xlink:href="CppInfo.gxl#Variables"/></edge>

<node id="n53657">
   <type xlink:href="CppInfo.gxl#Identifier"/>
   <attr name="string"><string>Parser</string></attr>
</node>
```

**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.*

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| *.cpp.tu* | 863 | 1 643 | 1 540 | 2 283 | 872 | 773 | 341 | 414 | 177 | 2 222 |
| *.cpp.tu.gxl* | 2 274 | 4 510 | 3 737 | 6 679 | 2 289 | 1 895 | 990 | 1 059 | 509 | 5 198 |
| *.cpp.tu.gxl.gz* | 191 | 323 | 323 | 486 | 205 | 201 | 71 | 70 | 36 | 450 |
| *.api.gxl* | 1 289 | 4 081 | 2 989 | 4 269 | 1 289 | 1 290 | 545 | 692 | 378 | 4 277 |
| *.api.gxl.gz* | 59 | 192 | 135 | 192 | 60 | 59 | 25 | 31 | 17 | 194 |

**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.*

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| *.cpp.tu* | 206.07 | 347.15 | 341.50 | 514.72 | 208.93 | 171.89 | 76.06 | 86.74 | 38.63 | 508.73 |
| *.cpp.tu.gxl* | 230.73 | 379.16 | 354.50 | 589.41 | 228.14 | 178.55 | 88.01 | 99.34 | 45.47 | 510.96 |
| *.cpp.tu.gxl.gz* | 238.18 | 393.72 | 364.37 | 618.10 | 236.06 | 185.64 | 91.07 | 103.92 | 48.24 | 527.74 |
| *.api.gxl* | 82.86 | 288.16 | 192.32 | 274.87 | 83.17 | 84.10 | 35.43 | 44.19 | 24.08 | 275.13 |
| *.api.gxl.gz* | 91.97 | 303.35 | 213.50 | 306.31 | 92.06 | 91.66 | 38.93 | 46.36 | 27.03 | 306.13 |

**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.*

low-level graphs representing each of the test cases. As stated in Section 4, we parse `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 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 `tu` 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 `tu` 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$^4$re system at Level I, including the CppInfo API, can be compiled and used with any version of the gcc

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
   <graph id="OrdInstance" edgemode="directed">
      <type xlink:href="ORD.gxl#ORD"/>
      <node id="c0">
         <type xlink:href="ORD.gxl#Class"/>
         <attr name="name">
            <string>::A</string>
         </attr>
      </node>
      <node id="c1">
         <type xlink:href="ORD.gxl#Class"/>
         <attr name="name">
            <string>::B</string>
         </attr>
      </node>
      <node id="e0"><type xlink:href="ORD.gxl#Inheritance"/></node>
      <edge from="c1" to="e0"><type xlink:href="ORD.gxl#isSrc"/></edge>
      <edge from="c0" to="e0"><type xlink:href="ORD.gxl#isDest"/></edge>
   </graph>
</gxl>
```

**Figure 11. ORD GXL instance.** *This figure illustrates a GXL instance of the ORD schema containing two classes and one edge.*

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| *.ord.gxl* | 10 | 12 | 2 | 4 | 9 | 0.175 | 3 | 3 | 0.614 | 2 |
| *.ord.gxl.gz* | 0.441 | 0.532 | 0.082 | 0.171 | 0.378 | 0.011 | 0.125 | 0.125 | 0.031 | 0.058 |

**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.*

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| *.ord.gxl* | 0.58 | 0.69 | 0.10 | 0.22 | 0.51 | 0.01 | 0.16 | 0.16 | 0.04 | 0.08 |
| *.ord.gxl.gz* | 0.63 | 0.74 | 0.10 | 0.24 | 0.58 | 0.01 | 0.17 | 0.18 | 0.04 | 0.08 |

**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.*

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                             xmlns:xlink="http://www.w3.org/1999/xlink">
   <xsl:output method="text" indent="no" encoding="ISO-8859-1"/>
   <xsl:strip-space elements="*"/>

   <xsl:template match="/gxl/graph">
      <xsl:variable name="nodes"
                    select="node[type/@xlink:href = 'ORD.gxl#Class']"/>
      <xsl:variable name="edges"
                    select="node[type/@xlink:href != 'ORD.gxl#Class']"/>

      <xsl:variable name="association"
                    select="node[type/@xlink:href = 'ORD.gxl#Association']"/>
      <xsl:variable name="composition"
                    select="node[type/@xlink:href = 'ORD.gxl#Composition']"/>
      <xsl:variable name="dependency"
                    select="node[type/@xlink:href = 'ORD.gxl#Dependency']"/>
      <xsl:variable name="inheritance"
                    select="node[type/@xlink:href = 'ORD.gxl#Inheritance']"/>
      <xsl:variable name="ownedElement"
                    select="node[type/@xlink:href = 'ORD.gxl#OwnedElement']"/>
      <xsl:variable name="polymorphic"
                    select="node[type/@xlink:href = 'ORD.gxl#Polymorphic']"/>

      <xsl:text>Nodes:        </xsl:text>
         <xsl:value-of select="count($nodes)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Edges:        </xsl:text>
         <xsl:value-of select="count($edges)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Association:  </xsl:text>
         <xsl:value-of select="count($association)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Composition:  </xsl:text>
         <xsl:value-of select="count($composition)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Dependency:   </xsl:text>
         <xsl:value-of select="count($dependency)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Inheritance:  </xsl:text>
         <xsl:value-of select="count($inheritance)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>OwnedElement: </xsl:text>
         <xsl:value-of select="count($ownedElement)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Polymorphic:  </xsl:text>
         <xsl:value-of select="count($polymorphic)"/>
      <xsl:text>&nl;</xsl:text>
   </xsl:template>
</xsl:transform>
```

**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*

| Test Case | Doxygen | FOX | FluxBox | HippoDraw | Jikes | Keystone | Licq | Pixie | Scintilla | Scribus |
|---|---|---|---|---|---|---|---|---|---|---|
| Time (s) | 6.15 | 8.00 | 0.86 | 1.92 | 5.28 | 0.07 | 1.40 | 1.35 | 0.30 | 0.59 |
| Classes | 562 | 465 | 243 | 253 | 322 | 62 | 225 | 263 | 89 | 208 |
| Association | 397 | 385 | 158 | 173 | 647 | 49 | 32 | 323 | 52 | 247 |
| Composition | 585 | 353 | 226 | 29 | 78 | 8 | 24 | 78 | 79 | 33 |
| Dependency | 14 141 | 18 460 | 5 499 | 8 631 | 7 292 | 554 | 3 793 | 4 588 | 2 198 | 5 078 |
| Inheritance | 335 | 220 | 198 | 180 | 164 | 25 | 161 | 139 | 14 | 6 |
| OwnedElement | 10 | 173 | 21 | 0 | 18 | 3 | 1 | 29 | 1 | 22 |
| Polymorphic | 26 618 | 31 116 | 1 311 | 7 143 | 28 762 | 137 | 7 760 | 6 383 | 469 | 168 |
| Total | 42 086 | 50 707 | 7 413 | 16 156 | 36 961 | 776 | 11 771 | 11 540 | 2 813 | 5 554 |

**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.*

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 and their students have exploited our infrastructure, including use of the g$^4$re system, the GXL encodings of `tu` files, and the schemas stored in the sourceforge repository [18, 31]. 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 researchers.

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 g$^4$re 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 corresponding `tu` 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 results 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 fortifying 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 corresponding 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 acheived by interoperating at the middle, rather than low, level. In addition, in Table 6 we observe that processing middle-level graph instances, such as ORDs, with widely available tools, such as *xsltproc*, is efficient. For example, 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 conformant instance of our ORD schema, not just those instances created by our g$^4$re system.

## 6 Conclusions and Future Work

In this paper, we have expanded on an infrastructure, first introduced in [31], that supports interoperability among reverse engineering tools and applications. We have described

the schema hierarchy that is central to our approach, detailed the interactions among instances of the schemas, and illustrated several of the schemas. We have implemented the infrastructure, described our implementation, and presented the results of a case study for our infrastructure.

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 the GXL class diagram schema. In the second assignment, the students parsed and built in-memory representations of GXL instances of the 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

[1] R. Al-Ekram and K. Kontogiannis. An xml-based framework for language neutral program representation and generic analysis. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, Manchester, UK, March 21–23 2005.

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

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

[4] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, Portland, OR, USA, August 2000.

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

[6] 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.

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

[8] 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.

[9] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the 18th International Conference on Software Maintenance*, pages 172–181, Montreal, Canada, October 2002.

[10] 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.

[11] Fluxbox Project. FluxBox version 0.9.14. Available at http://www.fluxbox.org.

[12] M. Fowler. *UML Distilled*. Addison Wesley, third edition, 2004.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters*. Addison-Wesley, 1995.

[14] X. Guo, J. R. Cordy, and T. R. Dean. Unique renaming of java using source transformation. In *SCAM*, pages 151–160, 2003.

[15] M. J. Harrold, B. A. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 139–148, Boston, MA, USA, June 1993.

[16] M. Hennessy, B. A. Malloy, and J. F. Power. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. In *Proceedings of International Workshop on Program Comprehension*, pages 298–299, Portland, Oregon, USA, May 2003. IEEE.

[17] HippoDraw. Hippodraw version 1.15.8. Available at http://www.slac.stanford.edu/grp/ek/hippodraw/.

[18] B. N. Hoipkemier, N. A. Kraft, and B. A. Malloy. 3d visualization of class template diagrams for deployed open source applications. In *Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, USA, July 2006.

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

[20] R. Holt, A. Schürr, S. Sim, and A. Winter. GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*, 60(4):149–170, 2006. Special Issue of the Journal Science of Computer Programming.

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

[22] 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.

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

[24] A. C. Jamieson, N. A. Kraft, J. O. Hallstrom, and B. A. Malloy. A metric evaluation of game application software. In *Proceedings of Future Play 2005*, East Lansing, MI, USA, October 2005.

[25] D. Jin and J. Cordy. Ontology-based software analysis and reengineering tool integration: The oasis service-sharing methodology. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 613–616, Budapest, Hungary, September 2005.

[26] D. Jin, J. R. Cordy, and T. R. Dean. Where's the schema? a taxonomy of patterns for software exchange. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 65–75, Washington, DC, USA, 2002. IEEE Computer Society.

[27] 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.

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

[29] N. A. Kraft, E. L. Lloyd, B. A. Malloy, and P. J. Clarke. The implementation of an extensible system for comparison and visualization of class ordering methodologies. *Journal of Systems and Software*, to appear.

[30] 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.

[31] N. A. Kraft, B. A. Malloy, and J. F. Power. Toward an infrastructure to support interoperability in reverse engineering. In *Proceedings of the Twelfth Working Conference on Reverse Engineering*, Pittsburgh, PA, USA, November 2005.

[32] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Working Conference on Reverse Engineering*, October 1998.

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

[34] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Proceedings of the 1st International Workshop on Meta-Models and Schemas for Reverse Engineering*, pages 7–18, Amsterdam, 2004. Elsevier Science.

[35] Licq. Licq version 0.2.3. Available at http://www.licq.orq.

[36] 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.

[37] 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.

[38] 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.

[39] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *GCC Developers Summit*, pages 171–180, Ottawa, Canada, 2003.

[40] 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.

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

[42] 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.

[43] N. A. Kraft. $g^4re$ reverse engineering infrastructure, version 1.0.8. Available at http://g4re.sourceforge.net.

[44] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York, New York, USA, September 2005. ACM Press.

[45] NIST. Ecma: Reference model for frameworks of software engineering environments. Technical Report, 1993.

[46] I. of Electrical and E. Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, New York, New York, USA, 1990.

[47] Pixie. Pixie version 1.5.2. Available at http://pixie.sourceforge.net.

[48] 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.

[49] Scintilla. Scintilla version 1.66. Available at http://www.scintilla.org.

[50] Scribus. Scribus version 1.2.3. Available at http://www.scribus.net.

[51] S. E. Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *Working Conference on Reverse Engineering*, pages 278–280, 2000.

[52] 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.

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

[54] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the Twenty-First International Conference on Software Engineering*, Los Angeles, CA, USA, May 1999.

[55] M. Skoglund and P. Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *Proceedings of the International Symposium on Emperical Software Engineering*, Australia, 2005. IEEE.

[56] The Expat XML Parser. Expat version 1.95.8. Available at http://expat.sourceforge.net.

[57] J. van der Zijp. The FOX Toolkit Library version 1.4.17. Available at http://www.fox-toolkit.org.

[58] D. van Heesch. Doxygen version 1.4.4. Available at http://stack.nl/ dimitri/doxygen.

[59] 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.

[60] 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.

[61] xsltproc. xsltproc version 1.1. http://xmlsoft.org/XSLT/xsltproc2.html.

[62] zlib. zlib version 1.2.3. Available at http://www.zlib.net.