View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Clemson University: TigerPrints

Clemson University
TigerPrints

All Dissertations | Dissertations

5-2007

An Infrastructure to Support Interoperability in Reverse Engineering

Nicholas Kraft
Clemson University, nkraft@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the Computer Sciences Commons

Recommended Citation
Kraft, Nicholas, "An Infrastructure to Support Interoperability in Reverse Engineering" (2007). *All Dissertations*. 51.
https://tigerprints.clemson.edu/all_dissertations/51

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

AN INFRASTRUCTURE TO SUPPORT INTEROPERABILITY IN REVERSE ENGINEERING

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Nicholas A. Kraft
May 2007

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. Harold C. Grossman
Dr. James F. Power
Dr. Roy P. Pargas

ABSTRACT

An infrastructure that supports interoperability among reverse engineering tools and other software tools is described. The three major components of the infrastructure are: (1) a hierarchy of schemas for low- and middle-level program representation graphs, (2) g^{4}re, a tool chain for reverse engineering C++ programs, and (3) a repository of reverse engineering artifacts, including the previous two components, a test suite, and tools, GXL instances, and XSLT transformations for graphs at each level of the hierarchy. The results of two case studies that investigated the space and time costs incurred by the infrastructure are provided. The results of two empirical evaluations that were performed using the api module of g^{4}re, and were focused on computation of object-oriented metrics and three-dimensional visualization of class template diagrams, respectively, are also provided.

DEDICATION

For C.R.K and J.H.K.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Brian Malloy, for his advice, support, and, most of all, friendship. I would also like to thank the members of my committee, especially Dr. James Power. Thanks to Dr. Jason Hallstrom for his help and friendship, and to Dr. Errol Lloyd for his advice and collaboration. Special thanks to Dr. Baker, Dr. Finkbine, Dr. Hollingsworth, Dr. Lang, and Mr. Manwani for preparing and encouraging me, and to Dr. Murali Sitaraman for bringing me to Clemson. I would also like to thank the faculty and staff of the Computer Science department, especially Dr. Grossman and Dr. Westall. A very special thanks to Jay Harris for his support, trust, friendship, and respect. Also, thanks to Ben Hoipkemier for his multiple contributions to this project, to Daniel Lowhorn for being a great teammate, to Andy Dalton and George Dowding for many late night conversations, to Tony Bowles and John Hunt for helping me survive 822, and to Jennifer and Brian for always taking care of me. Finally, I am grateful to all of my family and friends for their love and support, particularly my parents Jane and Bill, my brother Charlie, my wife Christine, my grandparents Carl and Jean, my oldest friends Chris, Jack, and Sue Robinson, my buddy Ringo, and my in-laws the Stemm family.

I would also like to thank the many anonymous reviewers who read drafts of this work for providing both insightful comments, and encouraging words. I would also like to thank the College of Engineering and Science, and the Graduate School at Clemson University for the awards in recognition of my research. I am truly honored and humbled to have been the first student from the Department of Computer Science to win these awards.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

LIST OF FIGURES

Figure Page

Page

LIST OF SOURCE LISTINGS

LIST OF TABLES

Table Page

Page

Chapter 1

Introduction

In their roadmap for reverse engineering, Müller et al. [2000] explain why reverse engineering tools are critical for controlling the high cost and risk of legacy system evolution. They identify wider adoption as a significant challenge to increased effectiveness of reverse engineering tools, and state that failure to adopt is caused in part by the lack of interoperability among reverse engineering tools and common academic and industrial software tools. Standard exchange formats (SEF) such as the Graph eXchange Language (GXL) have been created to increase interoperability, but previous research has not adequately exploited the semantic specification capabilities of these SEF's. These semantic specification capabilities allow meaning, in addition to structure, to be encoded and exchanged, thus allowing type validation in addition to structural validation.

Accessibility, comparability, reproducibility, and reusability of results are issues tightly coupled to interoperability. To evaluate a new technique, a comparison of the new technique to an existing technique must be performed; to perform the comparison, the reproduction or reuse of the existing results is required. However, researchers have reported considerable difficulty in interpreting and reproducing experimental results. In language design and implementation, for example, Das [2000] and Murphy et al. [1998] reported difficulty in reproducing results in points-to analysis and call graph construction, respectively. In addition, benchmarks are commonly used to evaluate and compare results, but require manual effort if infrastructure support is not provided.

To address the problems of accessibility, comparability, reproducibility, and reusability, we describe an infrastructure that supports interoperability among reverse engineering tools and other software tools [Kraft et al. 2005b; 2007a]. The

three major components of our infrastructure are: (1) a hierarchy of schemas for low- and middle-level program representation graphs, (2) g^{4}re, a tool chain for reverse engineering C++ programs [Kraft et al. 2005a; 2007b], and (3) a repository of reverse engineering artifacts, including the previous two components, a test suite, and tools, GXL instances, and XSLT transformations for graphs at all levels of the hierarchy [Kraft 2006].

1.1 Research Problems

The problems we investigated are in the area of reverse engineering; in particular, we investigated infrastructure support for interoperability. Statements of the problems follow.

Problem 1: Schemas for Low- and Middle-Level Program Representation Graphs

At the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [Lethbridge 2001], GXL [Holt et al. 2006] was ratified as the standard format for exchange of graphs among reverse engineering and reengineering tools. In addition, the participants agreed upon three levels at which interoperability should be applied: (1) low-level graph structures, including abstract syntax trees (AST) and graphs (ASG); (2) middle-level graph structures, including call graphs and control flow graphs (CFG); and (3) high-level graph structures, including architecture descriptions. Two GXL schemas for low-level graphs, but no GXL schemas for middle-level graphs, have been described in the literature.

Only two schemas for low-level graphs have been encoded in GXL and suggested as standard schemas: the Columbus C++ ASG schema [Ferenc et al. 2001], and the Dagstuhl Middle Metamodel (DMM) [Lethbridge et al. 2004]. However, neither of these schemas has been ratified, or widely adopted by tool developers. In addition, neither schema accurately and adequately represents C++ templates, including instantiations, specializations, and partial specializations. Finally, the DMM repre-

sents neither information about function pointers, nor the information necessary to create instances of several schemas for middle-level graphs, including control flow graphs.

Goal

Create GXL schemas for low- and middle-level graphs, and use known transformations between graphs to guide the organization of the schemas into a hierarchy.

Evaluation Criteria

1. The hierarchy is arranged in levels, such that an instance of a schema at one level can be created using only information contained in instances of schemas at previous levels.

2. Instances of low-level schemas contain the information needed to create instances of middle-level schemas, including call graphs, class-centric graphs, control flow graphs, and dependency graphs.

3. Low-level schemas are language-specific, and middle-level schemas are language-independent.

4. Low-level schemas for C++ accurately and adequately represent templates, including instantiations, specializations, and partial specializations.

5. Low-level schemas for C++ represent function pointers, including member function pointers.

Problem 2: Tool Support for Reverse Engineering C++ Programs

Source code based reverse engineering tools require a parser and front end to recognize the application under analysis, and to create a representation of the recognized program, such as an ASG. The difficulties that arise during the construction of a parser and front end for C++ are well documented, and are largely due to the size of the language, and the complexity of the template sublanguage [Bodin et al. 1994; Knapen et al. 1999; Lilley 1997; Power and Malloy 2000; Reiss and Davis 1995;

Roskind 1989; Veldhuizen 2003]. Many research endeavors have focused on the creation of a C++ parser and front end to enable analysis of C++ programs [Ferenc et al. 2002; Malloy et al. 2003a; McPeak 2005]; however, none of these tools can provide a correct parse tree for C++ programs that use templates, including "hello world", which uses the `iostream` library, and thus templates.

The GNU Compiler Collection [2002] includes *gcc*, a public domain, industrial strength compiler for C++ that fully handles templates. Several researchers have used the *gcc* parser and front end to create tools for reverse engineering C++ [Antoniol et al. 2004; Dean et al. 2001; Gschwind et al. 2004; Hennessy et al. 2003]. Until version 3.0 of *gcc* was released, such tools were, of necessity, tightly coupled to the compiler internals. Since then, *gcc* has provided, via a command line flag, a facility for writing the ASG for the given translation unit to a text file. The ASG representation stored in these text files is called GENERIC; instances of GENERIC allow for the construction of a reverse engineering tool for C++ that uses the *gcc* parser and front end without binding to the compiler internals. Despite this facility in *gcc*, there remains no public domain, general purpose tool for reverse engineering C++ programs that can fully handle templates.

Goal

Create a public domain, general purpose tool for reverse engineering C++ programs.

Evaluation Criteria

1. The tool is open-source and available on the Web.
2. The tool correctly parses, instantiates, and specializes templates.
3. The tool consists of loosely coupled, reusable modules.
4. The tool provides a module for linking C++ translation units.
5. The tool provides an API module for accessing information about declarations, statements, and some expressions.
6. The tool exchanges information via conforming instances of GXL schemas.

7. The tool is robust and efficient enough to use on medium-sized C++ programs, which contain up to 500 000 lines of non-commented, non-preprocessed lines of code.
8. The tool is general purpose.

Problem 3: A Repository of Reverse Engineering Artifacts

Empirical results are relevant to both researchers and practitioners; they reveal correlations among software technologies and practices. Accessibility of results and other artifacts has been identified as a key hurdle to the adoption of existing infrastructures [Müller et al. 2000]. Distribution of empirical evaluation artifacts is needed to enhance reproducibility and reusability of results, as well as to allow studies to be expanded [Do et al. 2005]. Researchers in software testing have led the way in providing access to empirical results [Andrews 2004; Graves et al. 2001; Harrold et al. 2001; Jones and Harrold 2005; Orso et al. 2004; Rothermel and Harrold 1998]. However, researchers in reverse engineering and program analysis have not responded to this need [Kraft et al. 2005b; 2007a].

Goal

Create a public repository of reverse engineering artifacts, and populate it with empirical results, including all tools, scripts, and documents needed to reproduce the results.

Evaluation Criteria

1. The repository contains a test suite, including important details about each test case:
   - Version
   - Size metrics
   - Configuration and build information
2. For at least one graph at each level of our hierarchy (see Problem 1), the repository contains:

- A GXL schema
- Tools that exchange information via conforming GXL instances of the schema
- GXL instances of the schema for each test case in the test suite
- A graph transformation that summarizes the information in a GXL instance
- Empirical results that show the space and time costs incurred by the documents and tools, respectively

3. The repository contains all artifacts needed to reproduce the results described in the previous items, including platform information for each experiment.
4. The repository is available to the public, particularly the reverse engineering community.

1.2 Dissertation Outline

In Chapter 2, we provide background information on the Graph eXchange Language (GXL), and various program representation graphs: the abstract syntax graph (ASG), the call graph, three class-centric graphs, and three control flow graphs. We summarize related research, in Chapter 3. In addition, we conclude that chapter with a discussion of the relationship between the summarized research and our own.

In Chapters 4, 5, and 6, we describe the three major components of our infrastructure to support interoperability in reverse engineering. We describe the first component, our hierarchy of schemas for program representation graphs [Kraft et al. 2005b; 2007a], in Chapter 4, and the second component, g^{4}re, our tool chain for reverse engineering C++ programs [Kraft et al. 2005a; 2007b], in Chapter 5. In Chapter 6, we present two case studies in which we use g^{4}re to determine the space and time costs incurred by our infrastructure [Kraft et al. 2007a]. In that same chapter, we also present the final major component of our infrastructure: our repository of artifacts, which includes a test suite, and tools, GXL instance graphs, and XSLT transformations for graphs at all levels of the hierarchy [Kraft 2006].

We present two empirical evaluations performed using g^{4}re, in Chapter 7. The two application areas are software measurement and program comprehension, in particular, computation of object-oriented metrics [Jamieson et al. 2005] and three-dimensional software visualization [Hoipkemier et al. 2006], respectively. Finally, in Chapter 8, we summarize our contributions to the area of reverse engineering.

There are two appendices. In Appendix A, we list the acronyms and abbreviations that we use throughout the dissertation. In Appendix B, we provide links to our online repository, where we make our schemas, our tools, and other artifacts from our infrastructure available to the reverse engineering community.

Chapter 2

Background

In this chapter we describe concepts, technologies, and terms that relate to our work in the area of reverse engineering. In particular, we describe Graph eXchange Language (GXL) in Section 2.1, and various program representation graphs in Section 2.2.

2.1 Graph eXchange Language

Graph eXchange Language (GXL) [Holt et al. 2006] was ratified as the standard exchange format (SEF) for reverse engineering and reengineering tools at the Dagstuhl Seminar on Interoperability of Reengineering Tools [Lethbridge 2001]. GXL is an XML language [Bray et al. 2006] that is defined by a document type definition (DTD), and conceptualized as a typed, attributed, directed graph. GXL describes both instance graphs and schema graphs, which are represented by UML class diagrams [Object Management Group 2005], using the same structural elements (node and edge types).

A UML class diagram is a static representation of a program consisting of rectangles to represent classes in the system, and lines connecting the rectangles to represent relationships among the classes. A rectangle for a class is divided into three horizontal sections. The top section displays the name of the class, which is in italics if the class is abstract. The middle section displays the data members, or *attributes*, of the class, including their types and visibilities (public, protected, private, or package). The bottom section displays the member functions, or *operations*, of the class, including their return types, parameters, and optionally, exception specifiers. A line between two classes can indicate an aggregation, an association, a

composition, or a generalization; a role name and a multiplicity can be assigned to each end of a line that is not a generalization.

A GXL schema is a UML class diagram encoded in XML. GXL provides a common base from which any schema for representing software can be derived; the common base is the GXL metaschema, a schema for E-R graphs that is classified as an explicit-external schema by Jin et al. [2002]. The GXL metaschema gives the structure for all GXL graphs, and like all GXL schemas, is an instance of the GXL metaschema. Thus, the GXL metaschema is its own schema.

All GXL graphs, both instance and schema, are constrained in ways that cannot be expressed by either the GXL DTD, or, for schemas, a UML class diagram. These constraints include: (1) ordered incidences must define a proper ordering, (2) a schema graph must contain at least one GraphClass node, (3) a schema graph must not contain a cycle of isA (generalization) edges, and (4) in a schema graph, an isA must not connect nodes of different types (i.e., a NodeClass must not inherit an EdgeClass, and so on) [Holt et al. 2006]. The GXL Validator was designed to validate GXL graphs against these constraints, as well constraints specified by the GXL DTD, the GXL metaschema, and the specified GXL schema [Kaczmarek 2003].

The GXL Validator is used to validate both instance and schema graphs. GXL schemas are validated against the GXL metaschema, which is validated against itself. Validating GXL is important; validation can reveal errors in both the modeling, and the generation of GXL instances. In addition, valid GXL files are more likely to be accepted by available XML tools than non-valid files.

In Figure 2.1, we illustrate an overview of GXL validation. Input files, shown at the top of the figure, are the GXL metaschema, an optional GXL schema graph, and an optional GXL instance graph. The executable *gxlvalidator*, shown as an ellipse in the middle of the figure, performs several tests that check the constraints described in the previous paragraph, and outputs the results of the validation. We list the tests that *gxlvalidator* performs in the note at the bottom right of the figure.

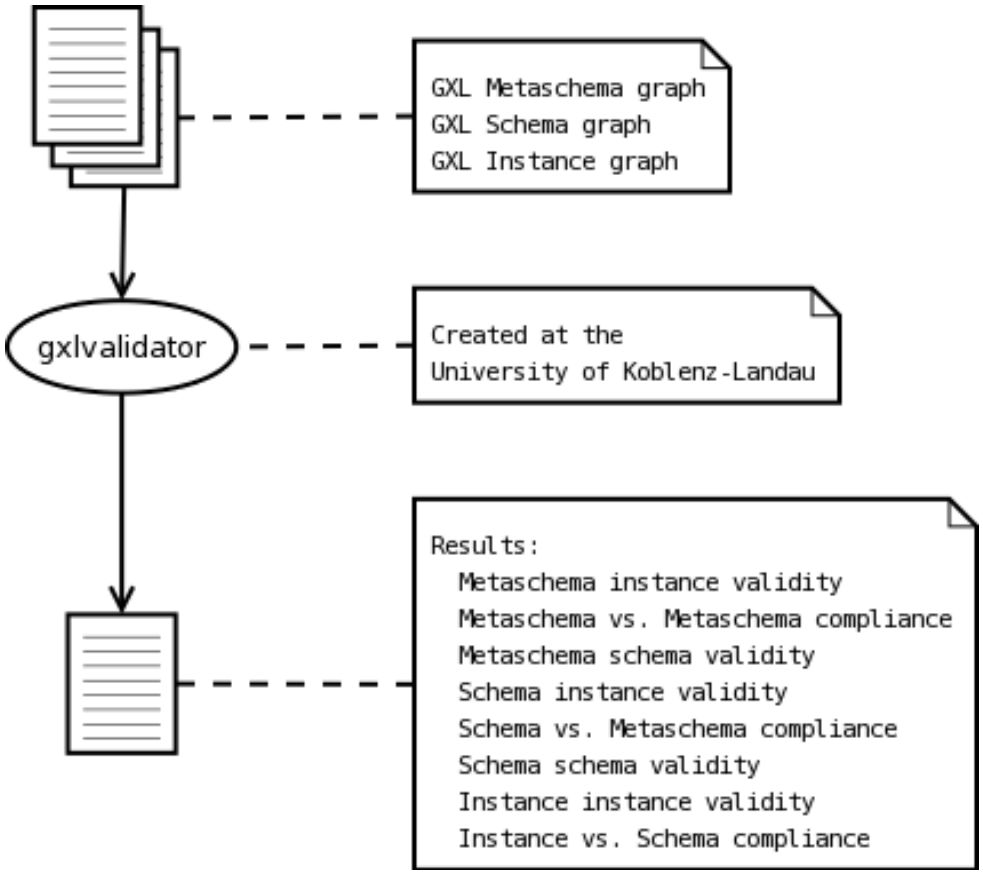

Figure 2.1: Overview of GXL Validation. *The process of validating GXL instance and/or schema graphs Solid edges represent input and output. Dashed edges represent notes.*

2.2 Program Representation Graphs

In this section, we describe several program representation graphs that are commonly used in compilation, program analysis, and program comprehension. We first describe the abstract syntax graph (ASG), which is sometimes called an abstract semantic graph, in Subsection 2.2.1. We then describe the call graph in Subsection 2.2.2, class graphs in Subsection 2.2.3, and control flow graphs in Subsection 2.2.4.

2.2.1 Abstract Syntax Graph

Given an input string, a parser derives a *parse tree* for the string (i.e., it parses the string). An *abstract syntax tree* (AST) is an abridged parse tree; an AST is constructed by a parser in lieu of the unabridged parse tree with non-terminals, keywords, and punctuation explicitly represented. Using the semantic rules for the input language, a semantic analyzer transforms an AST to an *abstract syntax graph* (ASG). An ASG is often the output of a compiler front end, and includes semantic information such as edges from variable uses to their declarations, edges from type uses to their definitions, and for C++, template instantiations and specializations.

```
1 struct Node
2 {
3    int    value;
4    Node*  next;
5 };
```

Source Listing 2.1: Source code for struct Node. *Definition of the C++ struct Node. Node consists of an integer, value, and a pointer to another Node, next.*

In Source Listing 2.1, we list C++ code for the definition of struct `Node`. Node consists of an integer, `value`, on line 3, and a pointer to another `Node`, `next`, on line 4. In Figure 2.2, we illustrate a possible AST for struct `Node`, and in Figure 2.3, we illustrate a possible ASG for struct `Node`. Note that in Figure 2.2 the uses of

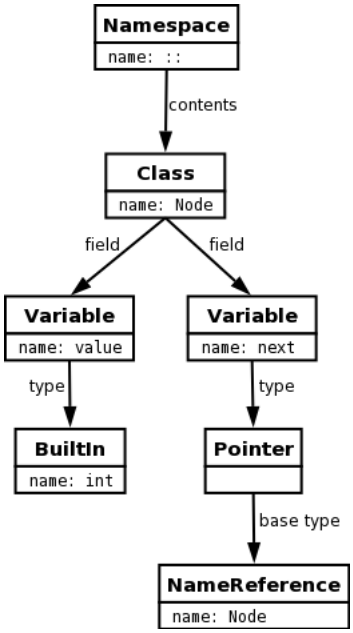

Figure 2.2: Sample AST for struct Node. *An AST for struct Node, which is defined in Source Listing 2.1. Uses of the types int and Node have not yet been resolved to their definitions.*

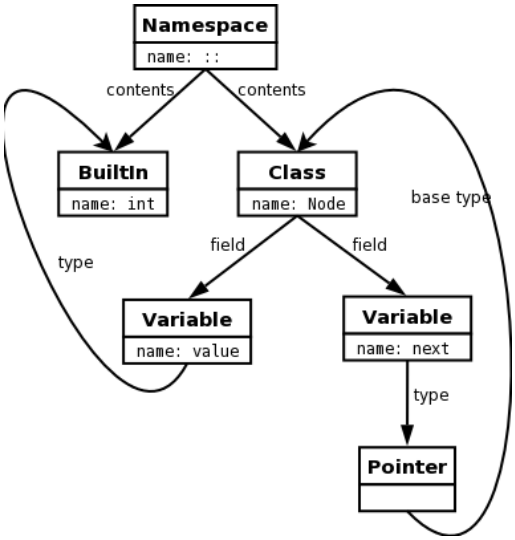

Figure 2.3: Sample ASG for struct Node. *An ASG for struct Node, which is defined in Source Listing 2.1. Uses of the types int and Node have been resolved to their definitions.*

the types `int` and Node have not yet been resolved to their definitions, but that in Figure 2.3 they have.

2.2.2 Call Graph

A *call graph* is a directed graph, $G = (V, E)$. The set of nodes, V, contains the functions in a program. For any two functions $f_0, f_1 \in G$, an edge (f_0, f_1) appears in the set of edges, E, if there is a potential call to f_1 by f_0. The call graph for a program is a directed acyclic graph (DAG) if the program does not use recursion. To reverse engineer a call graph from the source code of a program, information about at least the following constructs is required: function declarations and function calls (sometimes called call sites). Call graphs are commonly used for program profiling [Graham et al. 1982].

In strictly first-order procedural languages, constructing a program call graph is straightforward because, at every call site, the target of the call is directly evident from an inspection of the source code. However, in object-oriented languages such as C++, the target of a call cannot always be precisely determined; rather, the target is partially determined by the data values that reach the call site. For example, in C++, the method invoked by a call to a virtual method through a base class pointer is dependent on the class of the object receiving the call. In general, determining the flow of values needed to build a precise call graph requires an interprocedural data and control flow analysis of the program.

2.2.3 Class Graphs

In this subsection we describe graphs that are centered around the class. We first describe the class diagram, and next describe the object relation diagram (ORD). Finally, we describe the class firewall.

Class Diagram

A *class diagram* is a directed graph, $G = (V, E)$. The set of nodes, V, contains the classes in a program, and the set of edges, E, contains the relationships among the

classes. To reverse engineer a class diagram of low precision from the source code of a program, information about only class declarations is required. However, to reverse engineer class diagrams of high precision, access to information about the following constructs is required: declarations, scopes, types, and control structures.

The classes in a class diagram include generic, or *template*, classes, and may also include instantiated template classes. The edges in a class diagram represent the relationships among the classes in the program, and are specified by the syntax and semantics of data members, and the parameters and local variables of member functions. The edges types in a class diagram are: *aggregation*, *association*, *composition*, *dependency*, *inheritance*, and *ownedElement*, as defined by the UML specification, version 2.0 [Object Management Group 2005].

Object Relation Diagram

An *object relation diagram* (ORD), sometimes called a class dependency diagram, is a directed graph, $G = (V, E)$. The set of nodes, V, contains the classes in a program, and the set of edges, E, contains the relationships among the classes. An ORD can be constructed given only the information in a class diagram, and is commonly used to determine an integration order for class-based testing [Kraft et al. 2006].

The classes in an ORD include those that are not template classes, as well as instantiated template classes. The edges in a class diagram represent the relationships among the classes in the program, and are identical to those in a class diagram, but for the addition of polymorphic edges [Labiche et al. 2000; Malloy et al. 2003]. A *polymorphic* edge is generated by each association or dependency edge, e, where the type of the variable or parameter is an indirect type referring to a base class, b. Each edge e generates a set of polymorphic edges with source, $src(e)$, and destination, a derived class of b.

Class Firewall

A *class firewall* is a directed graph, $G = (V, E)$. The set of nodes, V, contains the classes in a program, and the set of edges, E, contains the dependencies between the

classes. A class firewall can be constructed given only the information in an ORD, and is used to define the scope of regression testing required in the presence of a change to a particular class in a system [Kung et al. 1995; Skoglund and Runeson 2005]. For example, assume that a developer finds a fault in class X, and then modifies it in an effort to fix the fault. The firewall for class X defines those classes that must be retested to ensure that no new fault has been introduced into the system.

2.2.4 Control Flow Graphs

In this subsection we describe graphs that express the flow of control for a program. We first describe the control flow graph (CFG), followed by the interprocedural control flow graph (ICFG), and finally, the class control flow graph (CCFG).

Control Flow Graph

A *control flow graph* (CFG) is a directed graph, $G = (V, E)$. The set of nodes, V, contains the basic blocks in a function plus two special nodes, and the set of edges, E, contains the flow of control between the blocks in the function. A basic block is a sequence of statements that has one entry and one exit. The two special nodes, *begin* and *end*, represent the entry and exit points for the function. To reverse engineer a CFG from the source code of a program, information about at least the following constructs is required: function declarations, control structures, and logical expressions. CFGs are commonly used for code generation and optimization [Aho et al. 2006].

Interprocedural Control Flow Graph

To encode control flow for a group of interacting functions that have a single entry point, such as a group of functions that constitute an entire program, an *interprocedural control flow graph* (ICFG) is required. An ICFG for a program, P, contains a CFG for each function in P. Each function is represented as a pair of nodes: the *call* node and the *return* node. Each call node is connected to the entry node of the

called function by a *call edge*, and each exit node is connected to the return node of the calling function by a *return edge*. An ICFG can be created from the information contained in a call graph combined with the information contained in several CFGs. ICFGs are commonly used for whole program optimizations [Aho et al. 2006].

Class Control Flow Graph

A *class control flow graph* (CCFG) consists of the set of CFGs, one CFG for each function in the class. One additional node is the entry node for the class, which is the predecessor node for the begin node of each CFG representing a constructor. Another additional node indicates that functions in the class can be invoked in an arbitrary order by clients of the class. This node is the the predecessor node for the begin node of the CFG for every function, and the successor node of the end node of the CFG for every function. The CCFG allows standard data-flow analyses to be applied to object-oriented programs [Buy et al. 2000]. A CCFG can be created from the information contained in an ICFG combined with the information contained in a class diagram.

Chapter 3

Related Work

In this chapter we summarize research that relates to our work in the area of reverse engineering. In particular, we summarize research on tools and techniques for promoting interoperability in reverse engineering, including: (1) exchange formats and schemas in Section 3.1, (2) infrastructures in Section 3.2, (3) evaluating results in Section 3.3, and (4) linking translation units in Section 3.4. We also summarize research on tools for reverse engineering C++ programs in Section 3.5. Finally, we provide a discussion of the relationship between the summarized research and our work in Section 3.6.

3.1 Exchange Formats and Schemas

Several exchange formats and languages have been proposed by the research community, and discussed at length at the Workshop on Standard Exchange Formats [Elliott Sim and Koschke 2001] and the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [Lethbridge 2001]. GXL (see Section 2.1 for details), the standard format for the exchange of graphs among reverse engineering and reengineering tools, originated from the synthesis of several exchange formats and languages [Holt et al. 2003], including: GRAph eXchange format (GraX) [Ebert et al. 1999], Tuple Attribute language (TA) [Holt 1997], Relation Partition Algebra (RPA) [Feijs and van Ommering 1999], Rigi Standard Format (RSF) [Wong 1998], and the graph format of the PROGRES graph rewriting system [Münch 1999]. More recently, Eichberg et al. [2004] proposed the use of the generic standards XML and XQuery, and Mamas and Kontogiannis [2000] described cppML, an XML DTD that represents the C++ grammar. In addition, the Electronics Industry Association (EIA), the Object Management Group (OMG), and W3C have provided the CASE Data Interchange

Format (CDIF) [Ernst 1997], the XML Metadata Interchange (XMI) [Group 2005], and the Resource Description Framework (RDF), respectively.

Schemas complement exchange formats by giving meaning to the data being exchanged. Many noteworthy schemas exist; some are automatically derived, while others are manually derived. Manually derived schemas are created by software engineers, and are more abstract than their automatically derived (tool generated) counterparts. In addition, some schemas are language-specific, while others are language-independent. Here, the term language-independent indicates that a schema is applicable to more than one, but not necessarily all, languages. We focus on manually derived schemas for AST's and ASG's; first we review language-independent schemas, next we review schemas specific to C++.

Czeranski et al. [2000] present the Bauhaus schema for modeling C and a subset of Ada. Aigner et al. [2006] present the Stanford University Intermediate Format (SUIF), which includes a schema that represents different languages, including C, C++, and Fortran. GENERIC is a schema that is used by GCC [2002] to represent several languages, including C, C++, Objective-C, Fortran, and Java. GENERIC was designed to facilitate semantic analysis and optimizations [Merrill 2003].

The Datrix team at Bell Canada Inc. [2000] presents the Datrix schema for modeling C, C++, Java, and other Algol-based languages. The Datrix schema is not source complete, but does provide mangled names for linking purposes. Datrix attempts to represent several languages with a single representation. Both Bell Canada and Dean et al. [2001] report using the Datrix schema to represent C++. Neither implementation handles C++ templates. Templates are not properly or fully represented by the Datrix schema; this failure is an artifact of attempting to be language-independent.

Ferenc et al. [2001] present the Columbus schema for modeling C++. The Columbus schema is not source complete, but does provide mangled names for linking purposes. The authors indicate that the ISO/IEC C++ standard [ISO/IEC JTC 1 1998] served as the basis for all design decisions. They also claim that a portion of the schema, the *language-independent* portion, could be used as common

root for modeling other programming languages, but these claims have not been investigated. The Columbus implementation [Ferenc et al. 2002], which exchanges data via the Columbus schema, does not handle C++ templates [Gschwind et al. 2004]. In addition, the schema does not properly or fully represent templates; there are several violations of basic object-oriented modeling principles in the template portion of the schema.

Lethbridge et al. [2004] present the Dagstuhl Middle Metamodel (DMM) for representing software in reverse engineering applications. The DMM schema is called a *middle* model because it represents information at a higher level of abstraction than the AST/ASG, but a lower level of abstraction than an architecture description diagram. The schema was specifically designed to be represented as a GXL schema. No reference implementation is provided, and only a handful of tools have leveraged the schema thus far [McQuillan and Power 2006].

3.2 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 [National Institute of Standards and Technology 1993]. 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 [Murphy and Notkin 1996]. This system is based on lexical analysis, and specifically identifies the ability to add additional source languages and extractors as central to the approach. Murphy and Notkin demonstrate this flexibility by applying the approach to extracting source models for ANSI C, CLOS, Eiffel, Modula 3, and TCL.

Kullbach et al. [1998] present the EER/GRAL approach to graph-based conceptual modeling of multi-lingual systems. 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 [Kazman and Carrière 1999]. 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 analysis and visualization phases. They use an SQL database for primary model storage, but then use application specific file formats to facilitate interchange between tools.

Salah and Mancoridis [2003] echo the Dali architecture in their *software comprehension environment*, which has a three-layer architecture. The three layers are composed of: (1) a data gathering subsystem, (2) a repository subsystem, and (3) an analysis and visualization subsystem. The environment supports both static and dynamic analysis of C++ and Java programs. Information can be accessed using either SQL, or a specialized query language.

Finnigan et al. [1997] describe a *Software Bookshelf* that was originally designed to support converting PL/I source code to C++. Their information repository, which describes the content of the bookshelf, is accessed through a web server using object-oriented database technology. The *Portable Bookshelf* (PBS) implementation of these ideas is based around a toolkit that includes a fact extractor, a manipulator, and graph layout tools. This "pipeline philosophy" has since evolved into the SWAG Kit and the LDX/BFX pipeline, each of which emphasizes collections of stand-alone tools communicating only via well-defined inputs and outputs [Holt et al. 2005].

Jin and Cordy [2005] advocate *non-prescriptive* integration that focuses on sharing *services*, rather than simply data, with the OASIS service-sharing methodology. In this approach, each tool in the integration is known as a *participant*. Each par-

ticipant 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 [Nierstrasz et al. 2005]. Language independence is achieved by the use of a common metamodel as the core of Moose. Services provided around this core include a meta-metamodel tailoring of the Moose metamodel; a GUI for browsing, querying, and grouping; and metric evaluation and visualization. Moose uses both the CDIF and XMI exchange formats to interact with external tools.

Al-Ekram and Kontogiannis [2005] present an XML-based framework that attempts to represent higher level artifacts in a language-neutral way. The framework includes an XML DTD for each of several artifacts, including a control flow graph, a program dependence graph, and a call graph. 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-filter" architectural style.

3.3 Evaluating Results from Reverse Engineering

Two important attributes of a reverse engineering infrastructure are: providing for repeatability of results, and allowing comparison of results from different approaches. One way this can be achieved is by agreement on standard exchange formats and schemas for exchanging information (see Section 3.1); this allows output from different tools or tool sets to be directly compared. There can be considerable difficulties involved in comparing results when a standard exchange format is not paired with a schema (standard or otherwise).

Murphy et al. [1998] describe a comparison of nine tools for extracting call graphs from C programs. They compare outputs from three software systems, and find a considerable variance. Das [2000] describes a points-to analysis algorithm, and

compares his results to those from tools implementing competing approaches. He notes that it took his team several months to synchronize the output of the tools so that the results could be compared. In each case, the problem was with different definitions and interpretations of required information, rather than with different output formats.

Sim et al. [2002] note the importance of benchmarks in software engineering in general, and in evaluating fact extractors in particular. They describe the construction of a benchmark suite designed to test the accuracy and robustness of fact extractors, and comparatively evaluate four tools by applying the benchmark suite. Lin et al. [2003] describe a four-level hierarchy of completeness, and use it to validate the CPPX fact extractor [Dean et al. 2001]. 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. [2003] describe an experimentation framework for evaluating C++ and Java disassembly and decompilation tools. The framework includes a layered test suite of programs, and a focused set of reverse engineering tasks.

3.4 Linking in Reverse Engineering Tools

Relatively little work exists on combining information extracted from different translation units. This process is analogous to compile-time linking, where external references in one translation unit are resolved to definitions in another. Wu and Holt [2004] 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. They describe an approach to alleviating these anomalies with heuristics and build simulation. Guo et al. [2003] describe a method for assigning globally unique identifiers (UIDs) to the declarations and references in a Java program. Each UID is based on scope and file information, and is attached via XML markup to entity references in the source code. 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.

3.5 Tools for Reverse Engineering C++ Programs

A reverse engineering tool that accepts C++ source code must have a parser, and likely, a corresponding front end. The difficulties that arise during the construction of a parser for C++ are well documented, and are largely due to the complexity of the template sublanguage [Bodin et al. 1994; Knapen et al. 1999; Lilley 1997; Power and Malloy 2000; Reiss and Davis 1995; Roskind 1989; Veldhuizen 2003]. Consequently, the selection of robust reverse engineering tools that accept C++ programs is inadequate.

Available reverse engineering tools for C++ can be divided into two categories: (1) those that provide their own parser (and possibly front end), and (2) those that utilize the C++ parser and front end from either the GNU Compiler Collection [Free Software Foundation 2002], or the Edison Design Group [2000]. We provide an overview of the first category in Subsection 3.5.1, and an overview of the second category in Subsection 3.5.2.

3.5.1 Tools that Provide a C++ Parser

The C++ parsers provided by reverse engineering tools extract varying levels of information, ranging from limited information, such as class hierarchies, to detailed information, such as statements and expressions. Parsers that extract limited information, known as *fuzzy parsers* [Koppler 1997], are well suited to tasks such as graphical browsing and graph visualization, but are not sufficient for program analysis tasks. Parsers that extract detailed information are ideal for program analysis tasks, but none of the parsers described in this subsection are able to fully accept templates.

Lapierre et al. [2001] present *Datrix*, an analyzer that extracts information from C, C++, or Java programs. Datrix extracts information for each translation unit in accordance with the Datrix ASG Model [Bell Canada Inc. 2000], and output is expressed in either TA (Tuple-Attribute Language) or VCG format. The Datrix

project at Bell Canada ended in the year 2000, and the *Datrix* analyzer is no longer available.

Source–Navigator™from Red Hat is an analysis and graphical browsing framework for C, C++, Java, Tcl, FORTRAN, and COBOL [Source–Navigator Team 2005]. The provided fuzzy parser extracts enough high level information to provide class hierarchies, imprecise call graphs, and include graphs. *Source–Navigator* does not provide statement level information, and the plain text output does not conform to a schema.

Ferenc et al. [2002] present *Columbus*, a fully integrated reverse engineering framework supporting fact extraction, linking, and analysis for C and C++ programs. *Columbus* provides output in a variety of formats, including CPPML, GXL, RSF, and XMI. Nevertheless, *Columbus* is unable to fully accept templates, as noted by [Gschwind et al. 2004].

3.5.2 Tools that Utilize the GCC C++ Parser

Industrial strength C++ parser front ends are provided by the GNU Compiler Collection [Free Software Foundation 2002] and the Edison Design Group [2000]. They both accept virtually all of the constructs defined by the ISO C++ standard, including templates [ISO/IEC JTC 1 1998; Malloy et al. 2003b]. However, *gcc* is in the public domain, which allows the reverse engineering tools that use it to be freely distributable; we summarize only tools that use *gcc* in this subsection.

There are take two common approaches to using *gcc*. The first approach is to modify the source code of the parser, which creates a custom version of *gcc*. The second approach is to use the `tu` files described in Subsection 6.2.1.

Dean et al. [2001] present *CPPX*, a tool that uses *gcc* for parsing and semantic analysis. *CPPX* predates the incorporation of `tu` files into *gcc*, and is built directly into the source code. *CPPX* constructs an ASG that is compliant to the Datrix ASG Schema [Bell Canada Inc. 2000], and can be serialized to GXL, TA, or VCG format. The Datrix ASG Schema is meant to accommodate several languages; this generality makes it difficult to accurately represent many C++ language constructs,

such as template specializations. The last release of *CPPX*, based on version 3.0 of *gcc*, does not properly handle the C++ Standard Library.

Hennessy et al. [2003] present *gccXfront*, a tool that harnesses the *gcc* parser to tag C and C++ source code. The tool annotates source code with syntactic tags in XML by modifying the *bison* parser generator tool, as described by Malloy et al. [2002]. This approach is no longer viable, because the C++ parser in *gcc* has migrated to recursive descent technology.

GCC.XML uses `tu` files to generate an XML representation for class, function, and namespace declarations, but does not propagate information such as function and method bodies [Kitware, Inc. 2005]. As a result, many common program representations, such as the call graph or the ORD, cannot be constructed using the output of *GCC.XML*.

Antoniol et al. [2004] present *XOGASTAN*, a collection of tools that convert a `tu` file to a GXL instance graph, and construct an in-memory representation of the GXL instance graph. *XOGASTAN* fails to create GXL output for certain GENERIC node types, including `try_catch_expr` and `using_directive`. Additionally, *XOGASTAN* has limited analysis capabilities for C++.

Gschwind et al. [2004] present *TUAnalyzer*, a system that uses `tu` files to perform analysis of template instantiations of classes and functions. *TUAnalyzer* performs virtual method resolution by using the '`base`' and '`binf`' attributes, along with the output provided by the compiler switch `-fdump-class-hierarchy`, which reconstructs the virtual method table. The scope of *TUAnalyzer* is limited to analysis of templates; Also, *TUAnalyzer* does not produce an output representation of the `tu` file for exchange with other reverse engineering tools.

3.6 Discussion

Existing schemas do not properly or fully represent C++ templates. A correct representation of templates is critical in reverse engineering, because all non-trivial C++ programs use templates (due to the C++ Standard Library). For example,

"hello world", the famous five line program, uses the `iostream` header, which uses templates extensively. In addition, generic programming is becoming increasingly popular, and the revised version of the ISO C++ standard (due in 2009) will increase the power and flexibility of templates. To address the shortcomings of existing schemas, we present our CppInfo schema for C++ in Section 4.2. The CppInfo schema properly and fully represents C++ templates.

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 in previous research for several languages, including PostgresSQL, Java, and C++. 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, in our SourceForge.net repository [Kraft 2006], GXL instances of unified representations that conform to our CppInfo API schema.

Most currently available tools for reverse engineering C++ do not properly handle templates. However, some tools based on versions of *gcc* greater than 3.0 do handle templates. XOGASTAN is similar to our g^4re tool chain, but has limited analysis capabilities, and does not provide a schema for understanding its GXL output. TUAnalyzer is complementary to g^4re; it uses information acquired from *gcc* using the flag `-fdump-class-hierarchy`. However, the scope of TUAnalyzer is limited to

inspecting template instantiations and reconstructing the virtual function call table. Our $\mathsf{g}^4\mathsf{re}$ tool chain is intended to be both industrial-strength and general-purpose.

Chapter 4

Schemas for Low- and Middle-Level Graphs

In this chapter we present the hierarchy of schemas for low- and middle-level program representation graphs that is central to our infrastructure approach and facilitates interoperability and reuse for reverse engineering tools and applications [Kraft et al. 2005b; 2007a]. In Section 4.1, we present an overview of the hierarchy of schemas. In Section 4.2, we illustrate the low-level graphs in Levels 0 and I of our hierarchy, and in Section 4.3, we illustrate middle-level graphs in Levels II, II, and IV of our hierarchy. Finally, in Section 4.4, we describe an approach to comparing instances of the schemas in our hierarchy.

4.1 Hierarchy of Schemas

In Figure 4.1, we illustrate the hierarchy of schemas that is central to our infrastructure, and facilitates 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 abstract syntax graphs (ASG), and application programming interfaces (API). The solid ellipses in the low-level partition of the figure represent the schemas used in our implementation; we discuss them further in Section 4.2. The solid ellipses in the middle-level partition of the figure represent schemas for graphical representations of code that are language independent, such as call graphs, and control flow graphs; we discuss them further in Section 4.3. The solid edges in Figure 4.1 represent the progression of information from a graphical representation at one level to a graphical representation at a subsequent level [Buy et al. 2000; Harrold et al. 1993; Sinha et al. 1999; Skoglund and Runeson 2005].

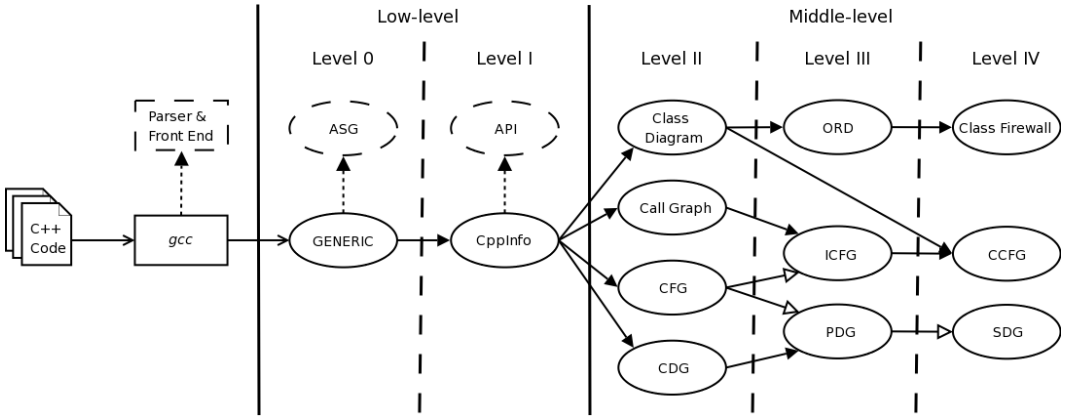

Figure 4.1: Hierarchy of Schemas. *Schemas for program representation graphs organized by levels. Solid edges with open arrows represent input and output. Dashed edges with filled arrows represent realization. Solid edges represent the progression of information from a graphical representation at one level to a graphical representation at a subsequent level. Filled arrows indicate a single instance is needed, empty arrows indicate that a set of instances are needed.*

The filled arrows indicate that a single instance is needed, while empty arrows indicate that a set of instances are needed.

The edge from Level 0 to Level I indicates that the information needed to build an instance of the API schema is present in an instance of the ASG schema, which contains information about a parsed and analyzed translation unit. We use the GENERIC ASG schema, the internal ASG schema used by *gcc*, in our implementation of the infrastructure. In addition, we use an API schema in place of another middle model schema, such as the Dagstuhl Middle Metamodel (DMM) [Lethbridge et al. 2004], which does not retain information about control statements or function calls. Information about control statements and function calls are needed to create instances of schemas at subsequent levels of our hierarchy, in particular, control flow graphs (CFGs) and call graphs. We use the CppInfo API schema in our implementation of the infrastructure.

The edge from CppInfo to Class Diagram indicates that the information needed to build a class diagram [Fowler 2003] is found in the information about classes 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) [Aho et al. 2006], the function declaration and call site information needed to build a call graph [Grove et al. 1997], and the statement and transfer of control information needed to build a control dependence graph (CDG) [Cytron et al. 1991]. The information needed to create instances of all schemas in Level II of Figure 4.1 is present in an instance of the CppInfo schema.

The information needed to create instances of the schemas shown in Level III of Figure 4.1 is present in instances of the schemas shown in Level II. The edge from Class Diagram in Level II to ORD in Level III indicates that the information needed to build an object relation diagram (ORD)[1] is present in an instance of the Class Diagram schema [Kraft et al. 2006; Malloy et al. 2003]. The only ORD edges not readily available in a Class Diagram instance are polymorphic edges. The information

[1]The use of the term ORD is a misnomer, because the nodes are classes, not objects; however, the term is used in previous research, and we continue to use it in this paper.

needed to generate polymorphic edges is extracted from the association, dependency, and inheritance information present in a Class Diagram instance; therefore, building an ORD instance using only information present in a Class Diagram instance is possible.

Two additional schemas are shown in Level III of Figure 4.1: Also shown in Level III of Figure 4.1 are schemas for an interprocedural control flow graph (ICFG) [Aho et al. 2006], and a program dependence graph (PDG) [Ferrante et al. 1987]. The edges from Call Graph and CFG in Level II to ICFG in Level III indicate that the information present in a Call Graph instance and a set of CFG instances can be used to build an ICFG instance [Aho et al. 2006]. Note that our Call Graph schema contains information about each individual call site. A Call Graph instance must contain this information to be used to build an ICFG instance; all solid edges in Figure 4.1 require that instances of the source and sink schemas conform to their respective schemas. Finally, the edges from CDG and CFG in Level II to PDG in Level III indicate that the information present in a CDG instance and a set of CFG instances can be used to build a PDG instance [Harrold et al. 1993].

In Level IV of Figure 4.1, we illustrate ellipses representing schemas for a class firewall, a class control flow graph (CCFG), and a system dependence graph (SDG). Instances of these three schemas can be built from information present in instances of schemas in Levels II and III of the hierarchy. The edge from ORD in Level III to Class Firewall in Level IV indicates that the information present in an ORD instance can be used to build a Class Firewall instance [Skoglund and Runeson 2005]. The edges from Class Diagram in Level II and ICFG in Level III to CCFG in Level IV indicate that the information present in a Class Diagram instance and an ICFG instance can be used to build a class control flow graph (CCFG) instance [Buy et al. 2000]. Finally, the edge from PDG in Level III to SDG in Level IV indicates that the information present in a set of PDG instances can be used to build an SDG instance [Sinha et al. 1999].

4.2 Low-Level Schemas: Levels 0 and I

Levels 0 and I in Figure 4.1 comprise the low-level partition of our hierarchy of schemas. We first describe Level 0, which contains the schema for an abstract syntax graph (ASG).We then describe Level I, which contains a schema for an application programming interface (API).

Level 0

In Level 0 of our infrastructure is the schema for an ASG, which contains information about a parsed and analyzed translation unit. We used the GENERIC ASG schema in our implementation of the infrastructure. GENERIC, the internal ASG schema used by *gcc*, is documented almost exclusively by source code and comments. GENERIC consists of 200 concrete node classes and 75 concrete edge classes, and was designed to facilitate semantic analysis and front end optimizations. The key advantage of GENERIC is the accurate and adequate representations of templates, including instantiations, specializations, and partial specializations. The key disadvantage of GENERIC is the complex and low-level representation it uses. For example, GENERIC uses 139 concrete node classes to represent expressions, and includes representations of *artificial*, or compiler generated, statements to manage the stack and heap. We were not the first to create a GXL schema for GENERIC, but unlike previous approaches, which used manually derived domain information to generate GXL schemas for GENERIC [Antoniol et al. 2004; Kitware, Inc. 2005], we used an instrumented parser to automatically reverse engineer domain information. In addition, we were the first to publically distribute a GXL schema for GENERIC.

We wrote a collection of Perl modules named GxlSW to automate the construction of a GXL schema, and used it to construct a GXL schema for GENERIC. Input to GxlSW is a plain-text, simplified UML class diagram, and domain type information; our instrumented parser writes the reverse engineered domain information in this format. To generate domain information for GENERIC, we used our instrumented parser, and two test suites: the C and C++ test suite from *gcc*, and a C++ test

suite [Malloy et al. 2003b] extracted from the ISO C++ standard [ISO/IEC JTC 1 1998]. We wrote two small (approximately 10 line) files that provide domain type information for GXL. We list a link to the generated GXL schema for GENERIC in Appendix B.

Level I

In Level I of our infrastructure is the schema for an API, which contains information about declarations, such as classes (including class templates, class template instantiations, and class template specializations); namespaces; functions (including function templates and function template instantiations); and variables, statements (including control statements and exception statements), and some expressions (function calls). We designed the CppInfo API schema to use in our implementation of the infrastructure. There are 137 classes in CppInfo: 70 nodes (28 abstract), 20 aggregation edges (1 abstract), 26 association edges (1 abstract), and 19 attributes. In addition, there are 2 enumerations, and 6 enumerators. Note that while CppInfo does not currently include representations for most expressions, our preliminary work suggests that the addition of expressions will introduce no more than 20 total node classes (in stark contrast to the 139 concrete node classes used by GENERIC), and 10 concrete edge classes. The key advantages of CppInfo over other schemas proposed by the reverse engineering community are the accurate and adequate representations of templates, including instantiations, specializations, and partial specializations, and function pointers[2].

In Figure 4.2, we illustrate the key classes, aggregations, and associations from CppInfo that relate to templates. We illustrate the primary class, Template, to the right of center in the middle of the figure. A Template is a Scope that is composed of zero or more Instantiations of type TemplateInstantiation, zero or more Specializations of type TemplateSpecialization, and zero or more PartialSpecializations of type TemplatePartialSpecialization. In addition, a Template is associated with a Tem-

[2]Lethbridge [2003] has identified the absence of representations for these language elements to be a key weakness of the Dagstuhl middle metamodel with regards to C++. He also states that practitioners have identified function pointers as being particularly important.

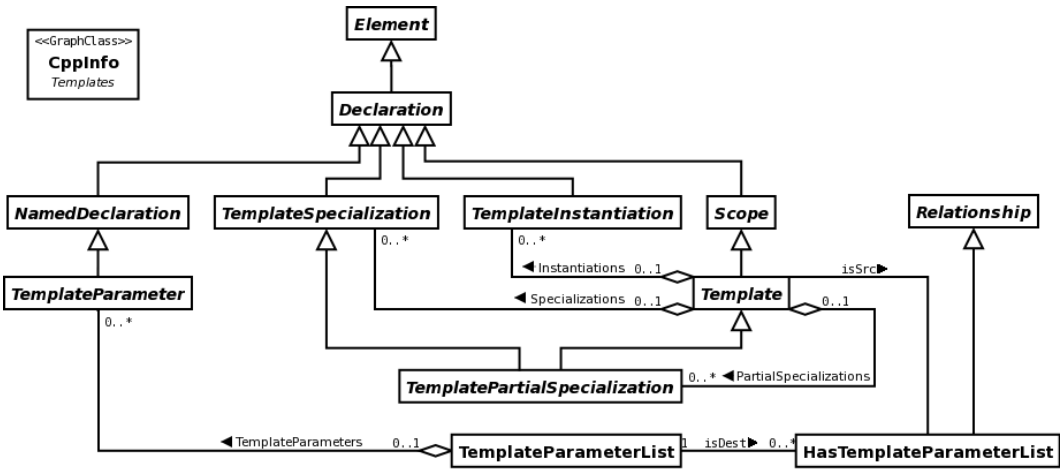

Figure 4.2: Level I: CppInfo API – Templates. *Excerpt from the CppInfo API schema that illustrates the key classes, aggregations, and associations related to templates.*

plateParameterList via a HasTemplateParameterList relationship. Note that a TemplateParameterList is composed of zero or more TemplateParameters.

Not shown in Figure 4.2 are concrete classes for class and function templates, template instantiations, template specializations, and template partial specializations. These concrete classes derive from the abstract classes shown in the figure, along with the abstract class for either a class or function. For example, ClassTemplate inherits from both Class and Template. Also not shown in the figure are concrete classes for template parameters. There are three template parameter classes in CppInfo: ParameterTemplateParameter, TemplateTemplateParameter, and TypeTemplateParameter.

In Figure 4.3, we illustrate the key classes, aggregations, and associations from CppInfo that relate to function types. We illustrate the primary class, FunctionType, in the center of the figure. A FunctionType is associated with a type via the HasReturnType relationship, and is composed of zero or more ParameterTypes of type Type. There are two concrete subclasses of FunctionType: FreeFunctionType and MemberFunctionType, which is associated with a Class via the HasClass relationship.

To the left of center in Figure 4.3, we illustrate the class IndirectType, from which PointerType is derived. IndirectType is associated with a Type via the HasBaseType

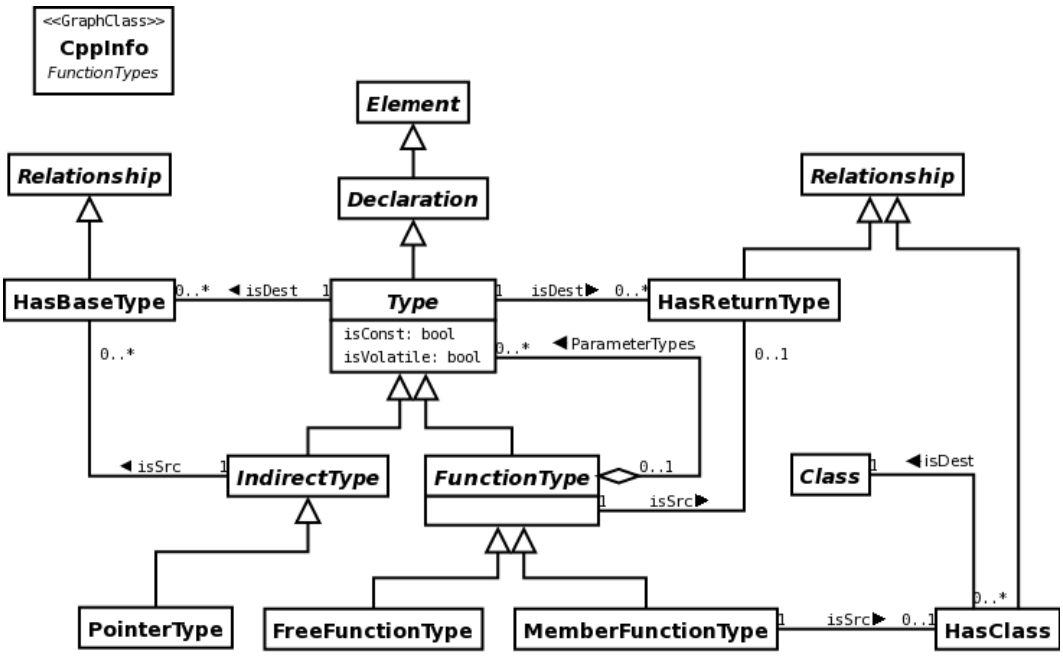

Figure 4.3: Level I: CppInfo API – Function Types. *Excerpt from the CppInfo API schema that illustrates the key classes, aggregations, and associations related to function types.*

relationship. Note that, in addition to C-style function pointers, C++ member function pointers (more commonly known as pointers-to-members) can be accurately represented using CppInfo.

4.3 Middle-Level Schemas: Levels II, III, and IV

Levels II, III, and IV in Figure 4.1 comprise the middle-level partition of our hierarchy of schemas. We first illustrate schemas in Level II, including the class diagram, the call graph, and the CFG. We then illustrate schemas in Level III, including the ORD, and the ICFG. Finally, we illustrate a schema in Level IV, the class firewall. We designed all schemas in this subsection, except for the class diagram schema, which we excerpted from the UML 2.0 specification [Object Management Group 2005]; we wrote GXL schemas for all schemas in this subsection.

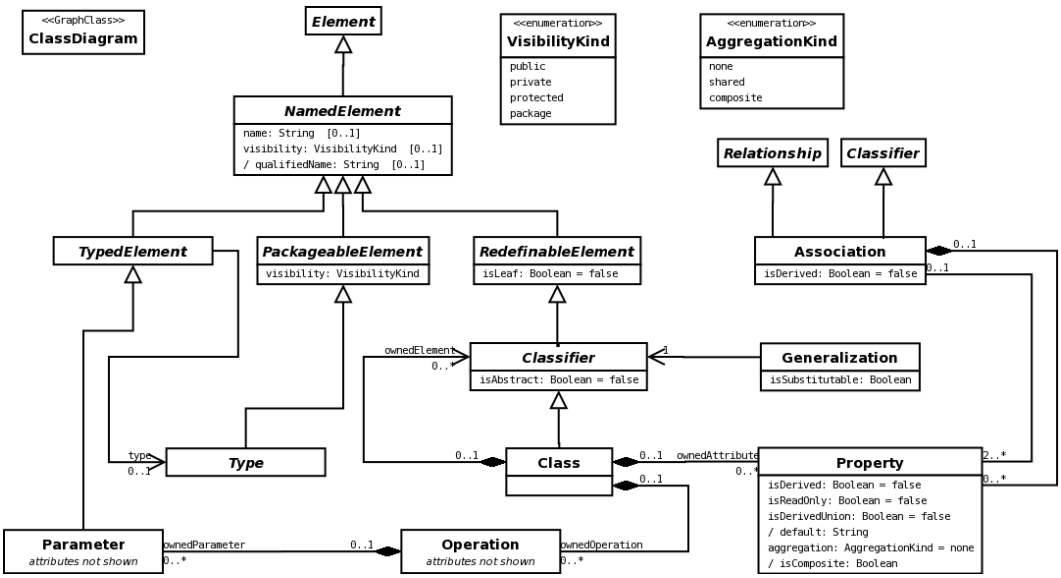

Figure 4.4: Level II: Class Diagram. *Excerpt from the UML 2.0 schema that illustrates the key class-related components in a class diagram.*

Level II

In Figure 4.4, we illustrate an excerpt from the UML 2.0 schema that illustrates the key class-related components in a class diagram [Object Management Group 2005]. We illustrate the primary component, Class, in the center of the figure, near the bottom. A Class is composed of zero or more ownedElements of type Classifier, zero or more ownedOperations of type Operation, and zero or more ownedAttributes of type Property. In addition, a Class can be involved in a Generalization or an Association relationship. Note that an Operation has zero or more typed Parameters, and a Property has attributes to indicate an aggregate or a composite.

In Figure 4.5, we illustrate an excerpt from the UML 2.0 schema that illustrates the key template-related components in a class diagram [Object Management Group 2005]. We illustrate Classifier, to the right of center of the figure, at the bottom. Recall from Figure 4.4 that Class is derived from Classifier, which, as we illustrate in Figure 4.5, can have a TemplateSignature, be used as a formal TemplateParameter, or be used as a boundElement (an actual template parameter).

In Figure 4.6, we illustrate the schema for a call graph. The schema consists of three classes: Function, which represents a function and has a name, FunctionCall,

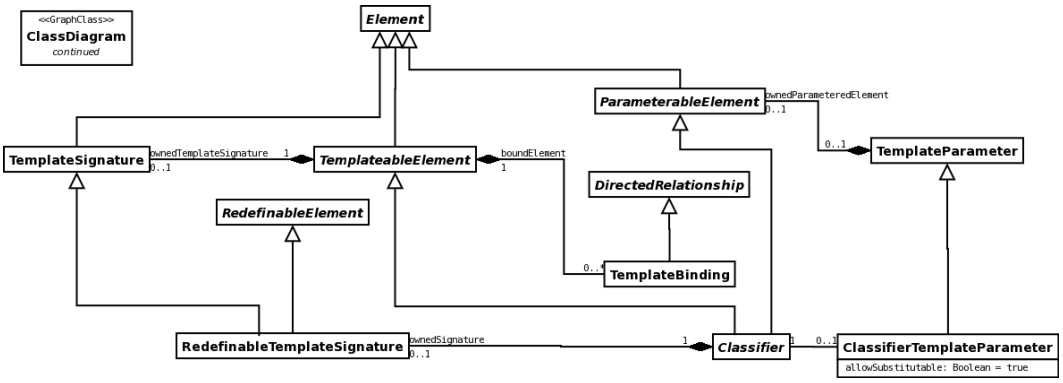

Figure 4.5: Level II: Class Diagram (continued). *Excerpt from the UML 2.0 schema that illustrates the key template-related components in a class diagram.*

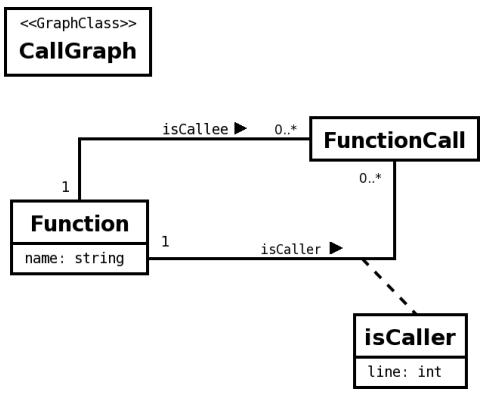

Figure 4.6: Level II: Call Graph. *Schema for a call graph, a graph whose nodes represent functions and function call sites, and whose edges represent function calls.*

which represents a function invocation, and isCaller, which is an association class that specifies the line number for the isCaller association.

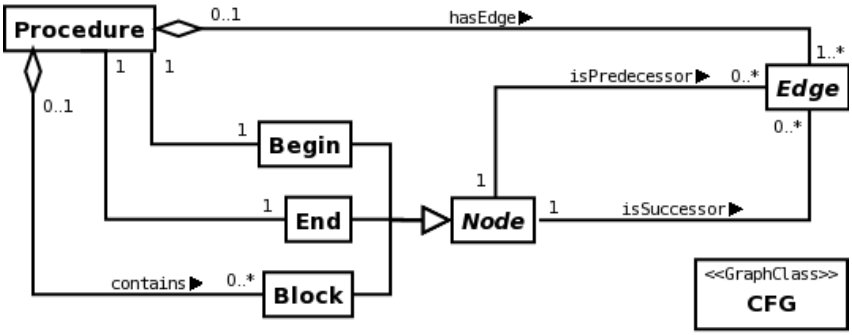

Figure 4.7: Level II: Control Flow Graph (CFG). *Schema for a CFG, a graph whose nodes represent blocks of straight-line code, and whose edges represent flow of control between the blocks.*

In Figure 4.7, we illustrate the schema for a control flow graph (CFG) [Aho et al. 2006]. The schema consists of a Procedure, which is composed of two or more Nodes and one or more Edges. The base class Node has three derived classes: Begin, Block, and End. Minimally, a Procedure contains a Begin, which is a special node whose successor is the Block whose leader is the first statement in the Procedure, and an End, which is a special node whose predecessor is the final Block in the Procedure. A Block represents a sequence of statements that has one entry and one exit. The class Edge represents the isPredecessor and isSuccessor relationships between two Nodes.

Level III

In Figure 4.8, we illustrate the schema for an object relation diagram (ORD). The schema consists of the class Class, along with the edge Edge and its six subclasses: Association, Composition, Dependency, Inheritance, OwnedElement and Polymorphic These six subclasses represent the kinds of relationships between classes [Kraft et al. 2006].

In Figure 4.9, we illustrate the schema for an interprocedural control flow graph (ICFG). The schema for an ICFG is identical to that of a CFG, but for the addition

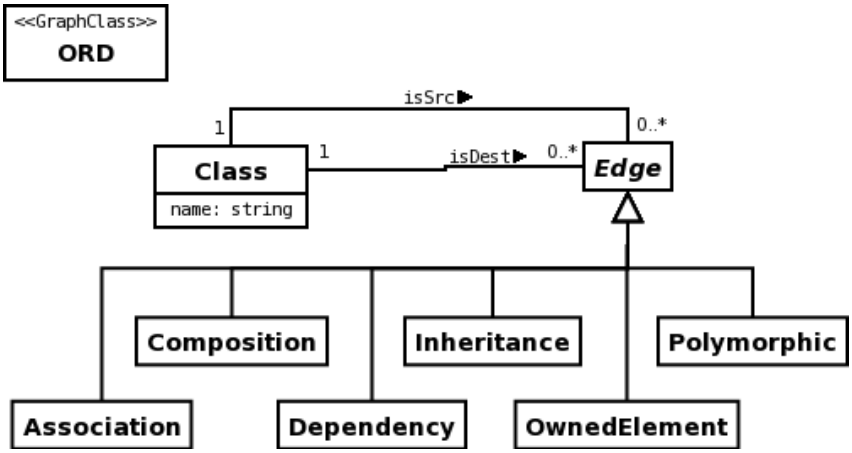

Figure 4.8: Level III: Object Relation Diagram (ORD). *Schema for an ORD, a graph whose nodes represent classes, and whose edges represent relationships, including polymorphic relationships, between the classes.*

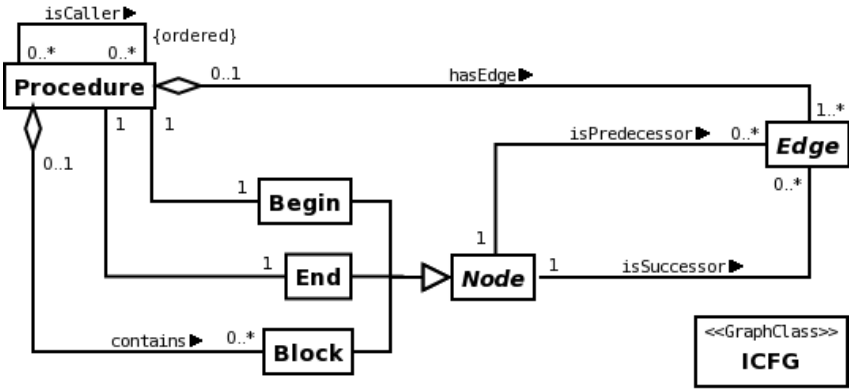

Figure 4.9: Level III: Interprocedural Control Flow Graph (ICFG). *Schema for an ICFG, a graph whose nodes represent blocks of straight-line code, and whose edges represent flow of control between the blocks and caller-callee relationships.*

of the ordered isCaller association. The isCaller association represents the caller-callee relationship between one Procedure and another.

Level IV

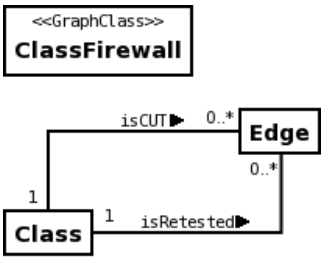

Figure 4.10: Level IV: Class Firewall. *Schema for a class firewall, a graph whose nodes represent classes, and whose edges represent testing dependencies between the classes.*

In Figure 4.10, we illustrate the schema for a class firewall. The schema consists of two classes: Class and Edge. The Class node corresponds to the Class node in the ORD schema. An Edge indicates a dependency between one Class and another; the Class involved in the isRetested association must be retested whenever the Class involved in the isCUT association is changed and must be tested.

4.4 Comparing Schema Instances

The schemas in Figure 4.1 were designed to be minimal yet complete; only the information required to construct a given graph structure is represented. To perform a comparison, the tools under study need not produce instances of the same schema; however, comparison of the instances generated by 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 intermediate transformations are used to create instances of the appropriate schema provided by our infrastructure.

One technique for comparison of GXL instance graphs is to use XSLT style sheets that are specified at the schema level. These style sheets can be applied to conforming instances of a given schema, and can be automatically generated

given domain information about the schema. This technique is not feasible for GXL encoded instances of low-level graphs. XSLT processors use a DOM representation of the input file; instances of low-level graphs are too large to be stored in a DOM tree. However, we successfully applied XSLT style sheets to GXL encodings of middle-level graph instances (see Chapter 6), and preliminary experimental results suggest that we can partially automate the generation of a system to report the differences between instances of the schema.

Chapter 5

g^{4}re – Tool Chain for Reverse Engineering C++ Programs

In this chapter we present the design and implementation of the g^{4}re tool chain [Kraft et al. 2005a; 2007b]. g^{4}re is a *tool chain*, because it is constituted by applications and libraries that are used either individually, or in concert. We designed g^{4}re with a GXL-based pipe-filter architecture; each constituent application or library in the chain takes, as input, the output of the preceding application or library in the chain. An important benefit of this architecture is that g^{4}re consists of a set of loosely coupled, reusable modules: the *ASG module*, the *schema and serialization modules*, the *transformation module*, the *linking module*, and the *API module*. We wrote all modules in ISO C++.

In Section 5.1 we present the architecture of g^{4}re. In this section, we also include an overview of the CppInfo schema. In Section 5.2 we present a sample usage of g^{4}re.

5.1 Architecture

In Figure 5.1, we provide an overview of the packages in the tool chain.We illustrate implementation artifacts, which we indicate with bold text, and third party libraries, which we indicate with italic text, at the left of the figure. We illustrate the *ASG module*, generic, as a package in the large g4re package at the right of the figure, and describe it in Section 5.1.1. We illustrate the *schema and serialization modules*, schema and serialization, as packages in the large cppinfo package at the center of the figure, and describe them in Section 5.1.2. We illustrate the *transformation module*, g4xformer, as a package in the large g4re package at the right of the figure, and describe it in Section 5.1.3. Finally, we illustrate the *linking module* and the *API*

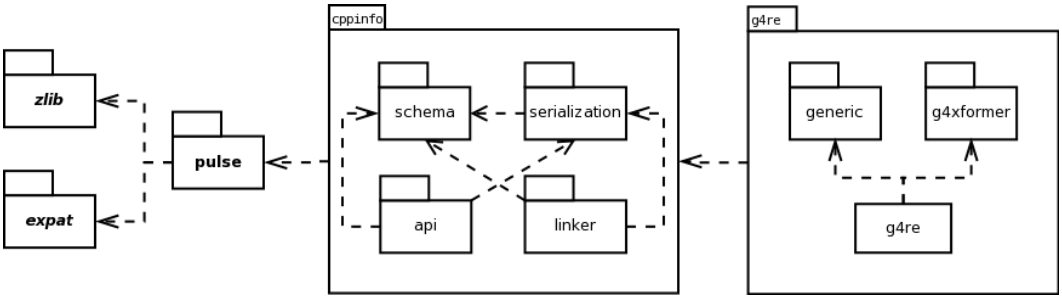

Figure 5.1: Overview of g^{4}re. *Dashed lines represent "use" dependencies. Bold text indicates an implementation artifact. Italic text indicates a third party library.*

module, linker and api, as packages in the large cppinfo package at the center of the figure, and describe them in Sections 5.1.4 and 5.1.5, respectively.

5.1.1 The ASG module

In the generic package, we provide parsing, storage, traversal, and serialization facilities for working with the GENERIC ASG representation of *gcc*. The input to this package is a `tu` file, or a GXL encoding of a `tu` file. The output of this package is a gzipped GXL encoding of the input file, or an in-memory representation of the GENERIC ASG.

We implemented two parsers: a `tu` file parser that uses a scanner generated by *flex*, and a GXL file parser that uses the *expat* XML parsing library and the *zlib* compression library via the pattern and utility library with standard extensions (*pulse*). We also implemented a simple node list representation for storage of the parsed ASG, and several parameterized methods for traversing the leftmost child right sibling (LCRS) tree that underlies the ASG. Finally, we implemented an extensible serialization facility that we use to create GXL encodings of `tu` files.

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

- remove fields that store internal information used by the *gcc* back end,
- mark methods as static if their parameter lists do not contain a *this* pointer,
- mark methods as const if their parameter lists contain a const *this* pointer,
- remove the *this* pointer from all method parameter lists.

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

The second parser we wrote provides functionality to parse a GXL file or gzipped GXL file and to store the corresponding ASG. Three advantages of this parser over the `tu` parser are:

1. reentrance,
2. the lack of post-parse transformation overhead,
3. the compression rate is higher for GXL files than for `tu` files.

We use the `tu` parser (in conjunction with our serialization facility) to create the GXL file(s) accepted by this parser; thus, there is a one-time cost associated with its use.

5.1.2 The Schema and Serialization Modules

In the schema package, we provide a class library that implements the CppInfo schema[1] for the ISO C++ programming language. In the current implementation, we provide 72 classes, 42 of which are concrete, that provide information about C++ language elements. Language elements include declarations, such as classes (including class templates and class template instantiations); namespaces; functions (including function templates and function template instantiations); and variables, statements (including control statements and exception statements), and some expressions.

In the serialization package, we provide serialization facilities for working with instances of the schema representation of C++. We implemented a parser to read GXL encodings; gzipped files are also accepted. We implemented visitor [Gamma et al. 1995] classes to write gzipped GXL encodings. We used C++ templates to allow the package to read and write both individual and linked instances of the schema representation.

5.1.3 The Transformation Module

In the g4xformer package, we provide an implementation of the transformation from the ASG representation that we provide in the generic package to the representation that we provide in the schema package. The input to this package is a `tu` file, or a GXL encoding of a `tu` file; gzipped files of either type are also accepted. The

[1] We describe the CppInfo schema in more detail in Section 4.2.

output of this package is a gzipped GXL encoding of the instance of the schema representation that corresponds to the GENERIC ASG in the input file.

We implemented the transformation in three passes. In the first pass, we traverse the generic ASG in program order, and create the *core* of the instance of the schema representation. The core consists of all declaration, declarator, and statement nodes, as well as structural edges. In the second pass, we adorn the core with edges that indicate the use of a type; these edges include inheritance edges. In addition, in the second pass, we build all cv-qualified types. We also resolve uses of bound template template parameters to their template declarations.[2]. Finally, in the third pass, we adorn the graph that results from the second pass with edges that indicate uses of expressions, including function calls[3]

5.1.4 The Linking Module

In the linker package, we provide an implementation of our linking algorithm. The input to this package is a set of GXL encodings of instances of the schema representation for all C++ translation units in a program; gzipped files are also accepted. The input files need not be created by the g4xformer package. The output of this package is a gzipped GXL encoding of the linked, or *unified*, instance of the schema representation for all C++ translation units in the program.

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

We perform linking $n-1$ times, where n is the number of translation units, when

[2]This task is not needed for compilation, and is not performed by *gcc*

[3] Calls to virtual functions are designated as such in `tu` files, but sets of possible targets are not identified. These sets are available via the *gcc* compiler flag `-fdump-class-hierarchy`.

n is greater than one. Otherwise, we perform linking one time. We achieve linking by performing a traversal of the most recently constructed instance of the schema representation. We add or append a schema class instance to the unified instance of the schema representation if the class instance does not exist, or is incomplete, in the unified instance. A schema class instance is *incomplete* if it is missing a required element (as defined by the CppInfo schema) or contains another incomplete instance. Using our definition of incomplete, we resolve function declarations to their corresponding definitions.

There is a special case for linking function parameters. A function parameter from a function declaration (prototype), is not always identical to the corresponding function parameter from the function definition. A function parameter may only have an initial value in a function declaration. In addition, the name of the function parameter may differ, e.g., anonymous function parameters are commonly used in header files.

5.1.5 The API Module

In the api package, we provide an abstract class that defines the interface for an API that provides access to information about language elements in a C++ program. In addition, we provide a concrete implementation of the API. The input to this package is a GXL encoding of a linked instance of the schema representation; gzipped files are also accepted. The input files need not be created by the linker package. The output of this package is an API, an in-memory representation of the linked instance that may be queried by a user program.

We designed the api package to provide a clear and flexible interface. We provide two points of access. The first point of access is a pointer to the global namespace, from which a user can traverse the ASG that underlies the API. We provide iterator classes, as well as an abstract visitor class, to use when accessing the API in this fashion [Gamma et al. 1995]. The second point of access is a collection of lists that each contain instances of a particular schema class. We provide, in two forms, the lists for Namespace, Class, Enumeration, Enumerator, Function, Variable, and Type-

def. 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* that contain the names of source files from which schema class instances should be ignored. We provide a script that generates filter lists.

5.2 Sample Usage

In Figure 5.2 we provide an overview of API usage. We illustrate a GXL file containing a linked instance of the schema representation at the top of the figure. Next, we illustrate a sample user program, metrics, that instantiates then queries the API to compute metrics. We illustrate the API, the abstract class cppinfo::api::Interface, in the middle of the figure. Finally, we illustrate filter lists at the bottom of the figure.

The user program instantiates the API with the name of the GXL file; when the API is instantiated, it reads the filter lists. The user program queries the instantiated API to perform a reverse engineering task, such as a program analysis. In Section 5.2.1 we describe the process of acquiring the GXL files needed to instantiate the API. In Section 5.2.2 we present a sample user program that instantiates and queries an API to perform a simple program analysis.

5.2.1 Input

In Figure 5.3 we illustrate the process of using *gcc*, and optionally g^4re and/or *gzip*, to create a set of files that contain instances of the GENERIC schema. We show the input, a C++ source file, at the left of the figure. We show the output, a set of files to transform, at the right of the figure (see Subsection 5.1.3 for details). This set may contain any combination of the four possible encodings of the input.

We show the use of the *gcc* command line flag `-fdump-translation-unit-all` to obtain a plain text representation of the GENERIC instance for each translation unit in a program. We show the creation of these representations, known as `tu`

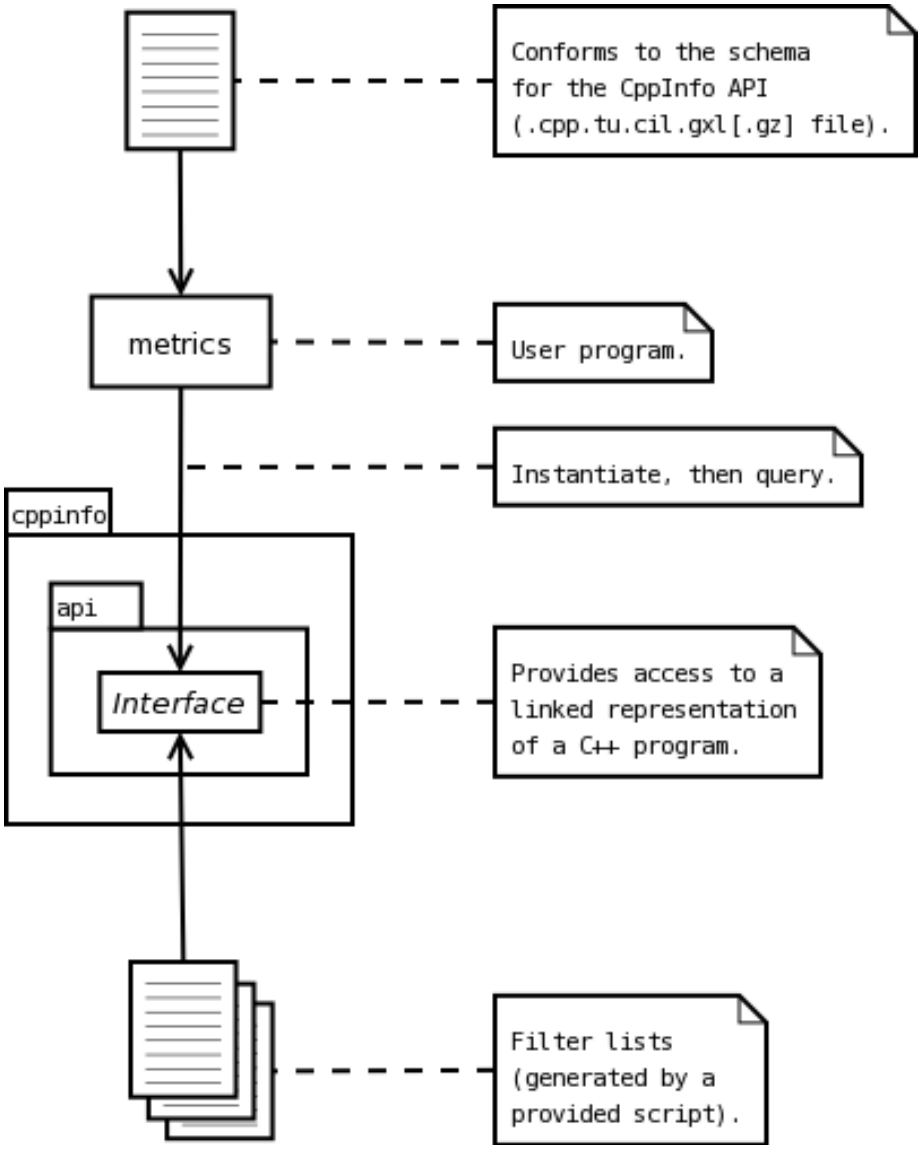

Figure 5.2: Overview of API usage. *Solid, directed lines show input, unless otherwise noted. Dashed lines show notes.*

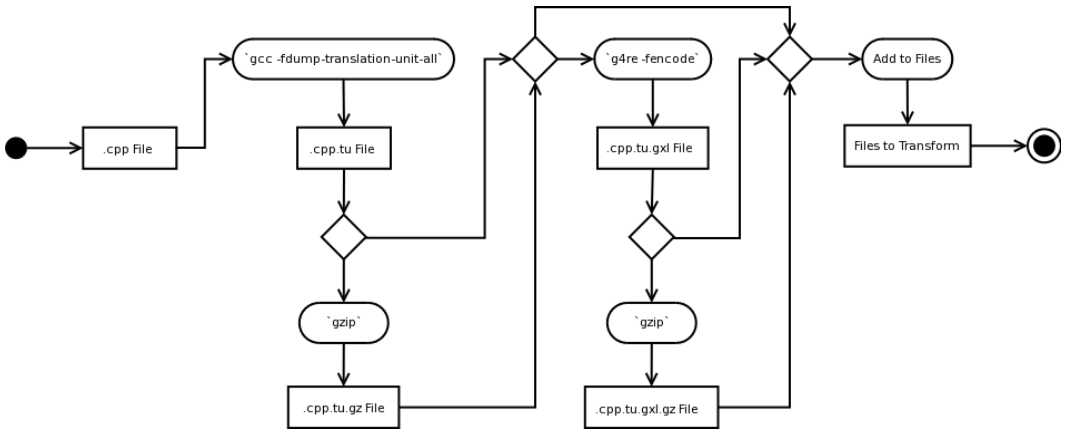

Figure 5.3: UML Activity Diagram for Transformer Input. *The process of creating a set of files to transform.*

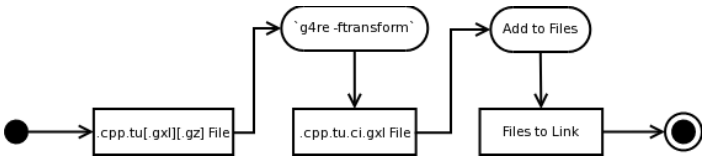

Figure 5.4: UML Activity Diagram for Linker Input. *The process of creating a set of files to link.*

files, in the upper left of Figure 5.3. We use `tu` files rather than hard-coding our solution into the *gcc* source code. This provides flexibility, and fits our theme of exchange among reverse engineering tools. In the upper right of the figure, we show the optional use of the g^{4}re command line flag `-fencode` to obtain, for each `tu` file, a GXL encoding of an instance of the GENERIC schema. At the bottom of the figure, we show the optional use of *gzip* to compress either a `tu` file, or a GXL instance,

In Figure 5.4 we illustrate the process of using g^{4}re to create a set of GXL files that contain instances of the CppInfo schema. We show the input, the set of files to transform (obtained as shown in Figure 5.3), at the left of the figure. We show the output, a set of files to link, at the right of the figure (see Subsection 5.1.4 for details).

We show the use of the g^{4}re command line flag `-ftransform` to obtain, for each GENERIC instance, a GXL encoding of a temporary instance of the CppInfo schema. We show the creation of these temporary instances, which use string encodings of

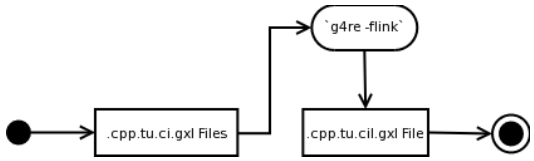

Figure 5.5: UML Activity Diagram for API Input. *The process of creating a file for use with the API.*

the unique names for the contained instances of CppInfo classes, in the center of Figure 5.4. We omit showing the optional use of *gzip* in this figure.

In Figure 5.5 we illustrate the process of using g^{4}re to create a GXL file that contains a linked instance of the CppInfo schema. We show the input, the set of files to link (obtained as shown in Figure 5.4), at the left of the figure. We show the output, a GXL encoding of the linked instance of the CppInfo schema, at the right of the figure (see Subsection 5.1.5 for details).

We show the use of the g^{4}re command line flag `-flink` to obtain, for a set of temporary instances of the CppInfo schema, a GXL encoding of the linked instance of the CppInfo schema. We show the creation of the linked instance, which uses unique integers to identify the contained instances of CppInfo classes, at the right of Figure 5.5. We omit showing the optional use of *gzip* in this figure.

5.2.2 Usage

```
1  class Shape { };
2  class Circle    : public Shape { };
3  class Rectangle : public Shape { };
4
5  class Square : public Rectangle { };
6
7  class Visitor { };
8  class ComputationVisitor   : public Visitor { };
9  class SerializationVisitor : public Visitor { };
10
11 class AreaComputationVisitor       : public ComputationVisitor { };
12 class PerimeterComputationVisitor : public ComputationVisitor { };
13
14 class XmlSerializationVisitor : public SerializationVisitor { };
```

Source Listing 5.1: Sample C++ program. *Two disjoint inheritance hierarchies that consist of ten classes.*

In Source Listing 5.1, we list a small C++ program that consists of ten classes. We list two *root classes*, Shape and Visitor, on lines 1 and 7, respectively. Root classes do not have base classes. We list three *interior classes*, Rectangle, ComputationVisitor, and SerializationVisitor, on lines 3, 8, and 9, respectively. Interior classes have one or more base classes, and one or more derived classes. Finally, we list five *leaf classes*, Circle, Square, AreaComputationVisitor, PerimeterComputationVisitor, and XmlSerializationVisitor, on lines 2, 5, 11, 12, and 14, respectively. Leaf classes have one or more base classes, but no derived classes.

```
1  void countClasses ( const cppinfo::api::Filename_t& filename ) {
2    using cppinfo::api::Interface;
3    using cppinfo::api::LinkedInterface;
4    using cppinfo::ConstClassPtrListIterator_t;
5     Interface* interface = new LinkedInterface( filename );
6     unsigned root = 0, interior = 0, leaf = 0;
7     ConstClassPtrListIterator_t i = interface->getClasses().createIterator();
8     while ( true == i->isValid() ) {
9        const cppinfo::ConstClassPtr_t c = i->getCurrent();
10       unsigned baseCount = c->getBaseClasses().size();
11       unsigned derivedCount = c->getDerivedClasses().size();
12       if ( 0 == baseCount ) {
13          ++root;
14       }
15       else {
16          if ( 0 < derivedCount )
17             ++interior;
18          else
19             ++leaf;
20       }
21       i->moveNext();
22    }
23    delete i;
24 }
```

Source Listing 5.2: Sample user program. *A simple program analysis that counts the number of root, interior, and leaf classes.*

In Source Listing 5.2, we list a C++ function that instantiates and queries an API instance to compute the number of root, interior, and leaf classes in a C++ program. We list the function declaration on line 1, where the parameter filename denotes the input program (see Subsection 5.2.1 for details). We list the API instantiation on line 5, where the filename is passed to the constructor of class cppinfo::api::LinkedInterface. On line 7, we use the list point of access provided by

the API to obtain an iterator that accesses each class in the input program[4]. Finally, we list a *while* loop on lines 8–21 that computes the number of root, interior, and leaf classes.

[4]The list contains all `ClassNonTemplate`, `ClassTemplate`, and `ClassTemplateInstantiation` instances. A trivial addition to the loop is required to exclude instances of one or two of these classes.

Chapter 6

Case Studies: Realizing the Infrastructure with g^{4}re

In this chapter we present two case studies in which we use the g^{4}re tool chain to realize our infrastructure. We designed our case studies to determine the space and time costs incurred by the use of our infrastructure. We measure space in two dimensions: size on disk, and size of graph(s), i.e., the number of nodes and edges. We measure times for parsing and building in-memory representations, as well as for the linking process, and the application of XSLT style sheets.

First, in Section 6.1, we describe the twelve applications and libraries that serve as the test suite in our case studies. In Section 6.2, we exchange low-level graphs, and measure the space and time costs incurred. In Section 6.3, we exchange middle-level graphs, and again measure the space and time costs incurred. In this section we also apply XSLT style sheets to each middle-level graph. We use style sheets that summarize the contents of each middle-level graph instance; the process of writing the style sheets, which requires knowledge of only the schema, is automatable.

6.1 Test Suite

In Table 6.1, we list the twelve open source applications and libraries, or test cases, that form the test suite for our studies.In the first column, we list the names that we use to refer to the test cases. In the next three columns of the table, we list relevant data about the test cases. We list the version numbers in the second column, the number of C++ translation units in the third, and the approximate number of thousands of lines of non-commented, non-preprocessed lines of code in the fourth.

Test Case	Version	C++ Translation Units	NCLOC (≈ K)
AvP	CVS 07/22/05	95	295
CppUnit	1.10.2	51	4
Doxygen	1.4.4	69	170
FluxBox	0.9.14	107	32
FOX	1.4.17	245	110
HippoDraw	1.15.8	249	55
Jikes	1.22	38	70
Keystone	0.2.3	52	16
Licq	1.3.0	28	36
Pixie	1.5.2	78	80
Scintilla	1.66	78	35
Scribus	1.2.3	110	80

Table 6.1: Test suite. *The 12 test cases that we use in our study. For each test case, we list the version, the number of C++ translation units, and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). The test suite contains 1,200 C++ translation units and approximately one million lines of code.*

The twelve applications and libraries that form our test suite are widely used, are freely available on the Web, and consist of approximately one million lines of non-commented, non-preprocessed code. *AvP* is a Linux port of the Fox Interactive/Rebellion Developments game Aliens vs Predator (Gold Edition) [Rebellion 2005]. *CppUnit* is a C++ port of the JUnit framework for unit testing [CppUnit Project 2006]. *Doxygen* is a documentation system for C, C++, and Java [van Heesch 2006]. *FluxBox* is a light-weight X11 window manager built for speed and flexibility [FluxBox Project 2006]. *FOX* is a toolkit to facilitate development of graphical user interfaces [van der Zijp 2006]. *HippoDraw* provides a highly interactive data analysis environment [Kunz 2006]. *Jikes* is a Java compiler system from IBM [IBM Jikes Project 2006]. *Keystone* is a parser and front end for ISO C++ [Keystone Project 2005; Malloy et al. 2003a]. *Licq* is a multi-threaded ICQ clone [Licq Project 2006]. *Pixie* is a RenderMan ® like photorealistic renderer [Arikan 2006]. *Scintilla* is a source code editing component that includes support for syntax styling, error indicators, code completion, and call tips [Hodgson 2006]. The final test case, *Scribus*, is a professional, cross-platform desktop publishing system [Scribus Project 2006].

We executed all experiments on a custom workstation with a *Dual Core AMD Opteron ™ 165* processor, 2048 MB of PC3200 DDR RAM, and a 250 GB, 7200 RPM SATA II hard disc on which we installed the Slackware 10.2 operating system after formatting with version 3.6 of the *ReiserFS* filesystem. We performed the experiments with version 1.5.0 of g^4re, which we compiled with version 4.1.1 of *gcc*. We created all `tu` files with *gcc* version 3.3.6.

6.2 Case Study: Exchanging Low-Level Graphs

In this section we describe the results of our first case study, in which we examine low-level graphs from our infrastructure. g^4re exchanges multiple formats, as discussed in Subsection 5.2.1. In Subsections 6.2.1 and 6.2.2, we describe the formats that g^4re exchanges, and present results for exchanging GXL encoded instances of schemas at

Level 0 and I of our infrastructure, respectively. We discuss the results of the case study in Subsection 6.2.3.

6.2.1 Exchanging Graphs at Level 0

In this subsection we investigate the costs associated with exchanging instances of low-level graphs; in particular, we investigate the costs of exchanging instances of the GENERIC ASG schema, in both `tu` and GXL formats. First, we illustrate the two exchange formats. Second, we measure the space and time costs incurred by exchanging ASGs, which are found in Level 0 of our infrastructure.

```
1 class Base { };
2 class Parser : public Base { };
```

Source Listing 6.1: Source code for class Parser. *Definition of the C++ class Parser. Parser inherits from the class Base.*

In Source Listing 6.1, we list C++ code for the definition of class `Parser`. We list a base class, `Base`, on line 1, and the class `Parser` on line 2. The inheritance relationship between `Parser` and `Base` is public and non-virtual.

```
@3      type_decl         name: @4        type: @5        srcp: Parser.cpp:2
                          artificial      chan: @6        addr: b7e0a460
@4      identifier_node   strg: Parser    lngt: 6         addr: b66b3ac0
@5      record_type       name: @3        size: @7        algn: 8
                          base: @8        public          struct
                          flds: @9        fncs: @10       binf: @11
                          addr: b7e0a310
```

Source Listing 6.2: Instance of a `tu` file. *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.*

We list the definition of a C++ class, `Parser`, in the GENERIC `tu` file format in Source Listing 6.2, and the corresponding definition as a GXL encoded instance of the GENERIC schema in Source Listing 6.3. GXL is clearly more verbose than the *gcc* `tu` file format; the respective character counts for the text in the figures are 447 and 1178.

Note that the text in Source Listing 6.2 contains information not present in Source Listing 6.3. Extraneous information, such as an address or string length,

```
<node id="n3">
  <type xlink:href="GENERIC.gxl#type_decl"/>
  <attr name="attr"><set><string>artificial</string></set></attr>
  <attr name="srcp"><string>Parser.cpp:2</string></attr>
</node>
<edge from="n3" to="n4"><type xlink:href="GENERIC.gxl#name"/></edge>
<edge from="n3" to="n5"><type xlink:href="GENERIC.gxl#type"/></edge>
<edge from="n3" to="n6"><type xlink:href="GENERIC.gxl#chan"/></edge>
<node id="n4">
   <type xlink:href="GENERIC.gxl#identifier_node"/>
   <attr name="attr"><set></set></attr>
   <attr name="strg"><string>Parser</string></attr>
</node>
<node id="n5">
   <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="n5" to="n8">
   <type xlink:href="GENERIC.gxl#base"/>
   <attr name="base">
      <tup><bool>false</bool><string>public</string></tup>
   </attr>
</edge>
<edge from="n5" to="n3"><type xlink:href="GENERIC.gxl#name"/></edge>
<edge from="n5" to="n7"><type xlink:href="GENERIC.gxl#size"/></edge>
<edge from="n5" to="n10"><type xlink:href="GENERIC.gxl#fncs"/></edge>
<edge from="n5" to="n11"><type xlink:href="GENERIC.gxl#binf"/></edge>
```

Source Listing 6.3: GXL instance of the GENERIC schema. *Definition of class Parser as represented in a GXL encoded 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.*

	.cpp.tu[.gxl][.gz]	.cpp.tu[.gz]	.cpp.tu.gxl[.gz]
Test Case	Nodes	Edges	Edges
AvP	3 286 604	8 607 856	8 509 901
CppUnit	4 574 861	10 983 481	10 911 237
Doxygen	7 558 527	17 894 321	17 724 872
FluxBox	12 016 093	30 111 171	29 852 859
FOX	12 139 219	32 260 488	31 953 355
HippoDraw	18 835 420	44 662 239	44 338 296
Jikes	7 543 803	17 437 798	17 321 098
Keystone	6 159 791	15 152 153	15 047 146
Licq	2 663 307	6 813 822	6 751 433
Pixie	3 278 791	7 665 603	7 620 166
Scintilla	1 414 562	3 456 874	3 427 785
Scribus	17 418 294	44 859 563	44 426 635

Table 6.2: Level 0: Numbers of nodes and edges. *The numbers of nodes and edges for ASGs that represent the test cases.*

is omitted from the GXL encoding. Empty lists are detected and removed during encoding; the `flds` edge is omitted in this example. The `fncs` edge is not omitted, because *gcc* provides a constructor, copy constructor, and assignment operator for each class.

It is well known that XML imposes significant storage costs; however, this fact has not hindered its wide spread adoption. Due to the prevalence of XML, there are several tools, available in popular languages such as C, C++, and Java, that were designed with these costs in mind. We designed and implemented a wrapper for the XML parser *expat* [eXpat Project 2005] that uses *zlib* [zlib Project 2005] to read compressed files. We also implemented a subclass of the C++ standard library class `ostream` to write compressed files. To provide a complete comparison, we instrumented our *flex* scanner to read compressed `tu` files.

In Table 6.2, we list the numbers of nodes and edges for ASGs that represent the test cases. In column 1, we list the test cases. In column 2, we list the number of nodes in the possibly GXL-encoded instance of the GENERIC schema. In columns 3 and 4, we list the numbers of edges in the `tu` files and GXL encoded `tu` files, respectively.

Test Case	.cpp.tu[.gz]		.cpp.tu.gxl[.gz]	
AvP	809	84	1 376	122
CppUnit	567	98	1 784	116
Doxygen	863	152	2 794	172
FluxBox	1 540	250	4 842	312
FOX	1 643	230	5 162	303
HippoDraw	2 283	376	7 222	469
Jikes	872	145	2 795	181
Keystone	773	126	2 439	157
Licq	341	56	1 081	69
Pixie	414	56	1 202	71
Scintilla	177	27	554	34
Scribus	2 184	352	6 967	440

Table 6.3: Level 0: Size on disk (MB). *The size on disk, in megabytes, for ASGs that represent the test cases.*

We apply the pruning algorithm discussed in Subsection 5.1.1 during the parse of a `tu` file. We show the effects of our pruning algorithm in Table 6.2. Our pruning algorithm does not remove any nodes, but it does remove edges. In the table, we show the difference between the numbers of edges in the `tu` files and the GXL encodings of the `tu` files. Next, we investigate the storage costs introduced by the use of GXL, and the saving that can be achieved by compressing files of each format.

In Table 6.3, we list the sizes on disk, in megabytes, for ASGs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed tu files, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded tu files, respectively.

A comparison of columns 2 and 4 of the table shows the significant storage cost introduced by the use of uncompressed GXL. For all but one of the test cases, the uncompressed GXL encodings of the `tu` files more than double 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 is 2 795 megabytes; the `tu` files are 3.2 times smaller than the GXL encodings. The outlier is *AvP*, for which the `tu` files, at 809 megabytes, are only 1.7 times smaller than the GXL encodings. On average, uncompressed `tu` files are 3.02 times smaller than the GXL encodings of the `tu` files,

Test Case	.cpp.tu[.gz]		.cpp.tu.gxl[.gz]	
AvP	97.39	112.62	136.58	155.71
CppUnit	123.47	142.85	174.66	199.56
Doxygen	206.07	238.65	279.39	322.82
FluxBox	341.50	388.24	472.39	552.80
FOX	347.15	411.66	503.60	577.47
HippoDraw	514.72	584.73	715.28	829.80
Jikes	208.93	233.54	253.77	291.35
Keystone	171.89	194.75	239.23	275.83
Licq	76.06	87.47	90.77	105.75
Pixie	86.74	104.06	125.20	144.57
Scintilla	38.63	46.65	56.30	64.99
Scribus	508.73	572.60	600.87	703.67

Table 6.4: Level 0: Time (s). *The running time, in seconds, to parse and build in-memory representations of ASGs that represent the test cases.*

with a standard deviation of 0.42. Columns 3 and 5 show the significant savings in storage cost that compression introduces when compared to columns 2 and 4, respectively. In addition, the gap between the storage costs of the two file formats is significantly reduced when compression is used. On average, compressed `tu` files are 1.25 times smaller than the GXL encodings of the `tu` files, with a standard deviation of 0.08. GXL, and XML in general, compresses at a higher ratio than other text formats. Next, we investigate the run-time costs introduced by the use of compression and GXL.

In Table 6.4, we list the running times, in seconds, to parse and build in-memory representations of ASGs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total times for the uncompressed and compressed tu files, respectively. In columns 4 and 5, we list the total times for the uncompressed and compressed GXL encoded tu files, respectively.

As stated in Subsection 5.1.1, we parse `tu` files using a *flex* generated scanner, GXL files using *expat*, and compressed files using *zlib*. We use the same node list graph data structure to store each graph instance in memory. A comparison of columns 2 and 4 of the table shows the run-time cost introduced by the use of GXL. The running times for GXL inputs are consistently higher than those for `tu` inputs, but the run-time costs introduced by GXL are much lower than the corresponding

storage costs. On average, parsing the uncompressed `tu` files is 1.36 times faster than parsing the uncompressed GXL encodings of the `tu` files, with a standard deviation of 0.10. The average time for uncompressed `tu` files is 226.77 seconds, with a standard deviation of 164.86. On average, parsing the compressed `tu` files is 1.36 times faster than parsing the compressed GXL encodings of the `tu` files, with a standard deviation of 0.08. The average time for compressed `tu` files is 259.82 seconds, with a standard deviation of 186.81.

6.2.2 Exchanging Graphs at Level I

In this subsection we continue to investigate the costs associated with exchanging instances of low-level graphs; in particular, we investigate the costs of exchanging instances of the CppInfo schema. First, we illustrate a GXL encoded instance of the CppInfo schema. Second, we measure the space and time costs incurred by exchanging APIs, which are found in Level I of our infrastructure.

We list the definition of C++ class `Parser` (see Source Listing 6.1 for details) as a GXL encoded, linked instance of the CppInfo schema in Source Listing 6.4. The character count for the text in the figure is 1307, which is larger than even the GXL encoding of the original `tu` file. However, we implemented maximal sharing of strings, such as file and identifier names, and integers, such as line and column numbers, to improve the scalability of this format.

We show the effects of our linking process in Table 6.5. In the table, we show the differences between the numbers of nodes and edges in the intermediate (unlinked) instances and the linked instances of the CppInfo schema. In columns 2 and 3, we list the sums of nodes and edges, respectively, for all intermediate instances for each test case. The numbers of nodes and edges for intermediate instances vary widely. The minimum number of nodes is 780 024 for *Scintilla*, and the maximum number of nodes is 10 164 005 for *HippoDraw*. The minimum number of edges is 2 391 321 for *Scintilla*, and the maximum number of edges is 34 941 134 for *HippoDraw*. The average numbers of nodes and edges are 4 262 119 and 14 445 413, with standard deviations of 3 402 982 and 11 469 128, respectively.

```
<node id="n81">
  <type xlink:href="CppInfo.gxl#ClassNonTemplate"/>
  <attr name="visibility"><enum></enum></attr>
  <attr name="isConst"><bool>false</bool></attr>
  <attr name="isVolatile"><bool>false</bool></attr>
  <attr name="key"><enum>class</enum></attr>
</node>
<edge from="n81" to="n82">
   <type xlink:href="CppInfo.gxl#HasSourceLocation"/>
</edge>
<edge from="n81" to="n1"><type xlink:href="CppInfo.gxl#HasScope"/></edge>
<edge from="n81" to="n84"><type xlink:href="CppInfo.gxl#HasName"/></edge>
<edge from="n81" to="n58" toorder="24">
  <type xlink:href="CppInfo.gxl#Bases"/>
  <attr name="inheritanceSpecifier">
    <tup><enum>public</enum><bool>false</bool></tup>
  </attr>
</edge>
<edge from="n81" to="n85" toorder="28">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<edge from="n81" to="n90" toorder="29">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<edge from="n81" to="n95" toorder="30">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<node id="n82">
  <type xlink:href="CppInfo.gxl#SourceLocation"/>
</node>
<edge from="n82" to="n60"><type xlink:href="CppInfo.gxl#HasFilename"/></edge>
<edge from="n82" to="n83"><type xlink:href="CppInfo.gxl#HasLine"/></edge>
<edge from="n82" to="n4"><type xlink:href="CppInfo.gxl#HasColumn"/></edge>
<node id="n83">
  <type xlink:href="CppInfo.gxl#SourcePosition"/>
  <attr name="number"><int>2</int></attr>
</node>
<node id="n84">
  <type xlink:href="CppInfo.gxl#Identifier"/>
  <attr name="string"><string>Parser</string></attr>
</node>
```

Source Listing 6.4: GXL instance of the CppInfo schema. *Definition of class Parser as represented in the GXL encoded, linked instance of the* CppInfo *schema.*

	.cpp.tu.ci.gxl[.gz]		.cil.gxl[.gz]	
Test Case	Nodes	Edges	Nodes	Edges
AvP	2 059 850	6 321 574	148 972	631 882
CppUnit	2 657 601	9 208 857	85 355	330 845
Doxygen	2 234 210	7 956 801	208 463	805 926
FluxBox	6 562 227	23 026 116	215 846	1 264 464
FOX	9 631 093	29 647 216	221 383	1 016 806
HippoDraw	10 164 005	34 941 134	254 270	1 470 270
Jikes	2 932 380	10 204 160	154 132	554 202
Keystone	3 314 379	11 731 213	139 570	625 173
Licq	1 142 403	3 996 935	128 045	541 960
Pixie	1 538 147	4 832 153	109 408	491 839
Scintilla	780 024	2 391 321	129 658	437 110
Scribus	8 129 110	29 087 482	330 537	1 510 133

Table 6.5: Level I: Numbers of nodes and edges. *The numbers of nodes and edges for APIs that represent the test cases.*

In columns 4 and 5 of Table 6.5, we list the numbers of nodes and edges, respectively, for the linked instance for each test case. These numbers are substantially smaller than those for the intermediate instances. The minimum number of nodes is 177 355 for *CppUnit*, and the maximum number of nodes is 254 270 for *HippoDraw*. The minimum number of edges is 330 845 for *CppUnit*, and the maximum number of edges is 1 470 270 for *HippoDraw*. The average numbers of nodes and edges are 177 136 and 806 717, with standard deviations of 70 277 and 409 918, respectively. The substantial reductions indicate a high ratio of duplication among translation units for all test cases. Recall that duplication is the result of compiler-specific information, as well as header files, being present in multiple translation units. Next, we investigate the savings in storage costs introduced by the linking process.

In Table 6.6, we list the sizes on disk, in megabytes, for APIs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed GXL encoded, intermediate instances of the CppInfo schema, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded, linked instances of the CppInfo schema, respectively.

Test Case	.cpp.tu.ci.gxl[.gz]		.cil.gxl[.gz]	
AvP	1 586	62	99	5
CppUnit	3 443	103	54	3
Doxygen	2 102	80	126	7
FluxBox	8 609	258	188	10
FOX	7 270	279	149	8
HippoDraw	12 826	389	219	11
Jikes	3 425	111	89	5
Keystone	4 404	132	98	5
Licq	1 380	44	85	5
Pixie	1 212	47	73	4
Scintilla	625	24	71	4
Scribus	7 932	289	231	12

Table 6.6: Level I: Size on disk (MB). *The size on disk, in megabytes, for APIs that represent the test cases.*

Test Case	.cpp.tu.ci.gxl[.gz]		.cil.gxl[.gz]	
AvP	110.81	116.17	8.43	9.47
CppUnit	202.19	217.50	4.70	5.19
Doxygen	143.85	150.62	10.89	11.53
FluxBox	521.57	548.98	16.01	16.52
FOX	516.11	542.16	12.46	13.56
HippoDraw	774.65	815.89	18.23	20.29
Jikes	211.79	223.77	7.68	8.12
Keystone	264.85	270.78	8.84	9.05
Licq	85.50	88.86	7.51	7.75
Pixie	83.88	87.96	6.10	6.52
Scintilla	42.72	44.70	6.08	6.58
Scribus	534.36	445.43	19.46	21.58

Table 6.7: Level I: Time (s). *The running time, in seconds, to parse and build in-memory representations of APIs that represent the test cases.*

A comparison of columns 2 and 3 of the table to columns 4 and 5 of the table, respectively, shows the significant savings introduced by the linking process. For all test cases, the uncompressed GXL encoding of the linked instance is at least 8.8 times smaller than the uncompressed GXL encodings of the intermediate instances. For example, the total storage cost of the linked instance for *Jikes* is 89 megabytes, but the total storage cost of the intermediate instances is 3 425 megabytes; the linked instance is 38.5 times smaller than the intermediate instances. *CppUnit* shows the biggest difference in storage costs, with the linked instance 63.8 times smaller than the intermediate instances. *Scintilla* shows the smallest difference in storage costs. The savings for the compressed GXL encodings are similar, although the ratios drop slightly due to the high rate of compression. A large reduction in size indicates a high level of duplication among translation units (intermediate instances), likely caused by poor compiler firewalling. Next, we investigate the savings in run-time costs introduced by the linking process.

In Table 6.7, we list the running times, in seconds, to parse and build in-memory representations of APIs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total times for the uncompressed and compressed GXL encoded, intermediate instances of the CppInfo schema, respectively.

In columns 4 and 5, we list the total times for the uncompressed and compressed GXL encoded, linked instances of the CppInfo schema, respectively.

A comparison of columns 2 and 4 shows a significant savings in run-time costs when dealing with a linked representation of a program. This result follows directly from the significant savings in storage costs shown in Tables 6.5 and 6.6. The time to parse a linked instance is well under 30 seconds for all test cases, whether or not the GXL encoding is compressed. The time to parse the intermediate instances is under 60 seconds for only one test case, and over half of the test cases take over three minutes to parse. The maximum time to parse compressed GXL encodings of intermediate instances is nearly 15 minutes, for *HippoDraw*.

6.2.3 Discussion

The results for exchanging low-level graphs show that the storage costs can be prohibitive. The largest files recorded in this case study, uncompressed GXL encodings of intermediate instances of the CppInfo schema, total over 53 gigabytes of disc space for the 12 test cases. However, compressed GXL encodings of linked instances of the CppInfo schema, the smallest files recorded in this case study, total only 79 megabytes of disc space for the 12 test cases.

The results also show that the run-time costs for low-level graphs can also be prohibitive. The slowest parsing times in this case study were for compressed GXL encodings of `tu` files. For the 12 test cases, these files took over 70 minutes to parse. The fastest parsing times in this case study were for uncompressed GXL encodings of linked instances of the CppInfo schema. For the 12 test cases, these files took just over 2 minutes to parse.

We presented results that show the importance of a linker for C++ reverse engineering tools, and presented the first experimental evidence which shows the significant savings that can be achieved by linking C++ translation units. Unfortunately, the smallest files recorded in this case study are still too large to be exchanged via email or newsgroups. This is important, as accessibility of results has been identified as a key hurdle to the adoption of existing infrastructures [Müller et al. 2000].

6.3 Case Study: Exchanging Middle-Level Graphs

In this section we describe the results of our second case study, in which we examine middle-level graphs from our infrastructure. In Subsection 6.3.1, we present results for exchanging GXL encoded instances of schemas at Levels II, III, and IV of our infrastructure. In Subsection 6.3.2, we extract results from GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas by applying XSLT style sheets. We discuss the results of the case study in Subsection 6.3.3.

6.3.1 Exchanging Graphs at Levels II, III, and IV

In this subsection we investigate the costs associated with exchanging instances of middle-level graphs. In particular, we investigate the storage costs of exchanging GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas. First, we illustrate GXL encoded instances of the ORD and Class Firewall schemas. We omit an instance of the Class Diagram, because it would be nearly identical to the ORD instance. Second, we measure the space costs incurred by exchanging graphs at Levels II, III, and IV.

```
<?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="c0" to="e0"><type xlink:href="ORD.gxl#isDest"/></edge>
    <edge from="c1" to="e0"><type xlink:href="ORD.gxl#isSrc"/></edge>
  </graph>
</gxl>
```

Source Listing 6.5: GXL encoded ORD instance. *A GXL encoded instance of the ORD schema containing two classes,* `::A` *and* `::B`*, and one* `Inheritance` *edge. The edge indicates that* `B` *inherits from* `::A`*.*

In Source Listing 6.5, we list a prototypical GXL encoded instance of the ORD schema. We list two classes, `::A` and `::B`. In addition, we list an Inheritance edge,

which indicates that `::B` inherits from `::A`. In this case, the Class Diagram instance would be identical, but for the references to the schema (shown as `xlink:href` attributes in `type` tags).

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

Source Listing 6.6: GXL encoded Class Firewall instance. *A GXL encoded instance of the Class Firewall schema containing two classes,* `::A` *and* `::B`*, and one edge that indicates that if* `::A` *has changed and must be tested, then* `::B` *must be retested as well.*

In Source Listing 6.6, we list a prototypical GXL encoded instance of the Class Firewall schema. We again list two classes, `::A` and `::B`. We also list one Edge edge, which indicates that if `::A` has changed and must be tested, then `::B` must be retested as well. This edge results from the Inheritance edge in the ORD instance.

In Table 6.8, we list the sizes on disk, in megabytes, for class diagrams, ORDs, and class firewalls that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed GXL encoded, instances of the Class Diagram schema, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded, instances of the ORD schema, respectively. In columns 6 and 7, we list the total sizes of the uncompressed and compressed GXL encoded, instances of the Class Firewall schema, respectively.

Test Case	.cd.gxl[.gz]		.ord.gxl[.gz]		.cfw.gxl[.gz]	
AvP	4 207	227	5 845	301	2 735	140
CppUnit	183	10	186	10	76	8
Doxygen	4 530	245	4 309	217	2 038	100
FluxBox	1 297	71	899	49	400	24
FOX	2 922	158	582	28	953	52
HippoDraw	1 706	93	4 016	200	1 065	52
Jikes	1 345	73	4 561	221	1 041	52
Keystone	1 066	58	3 246	156	813	40
Licq	908	49	1 366	68	264	16
Pixie	1 301	71	1 988	101	693	36
Scintilla	399	22	218	12	68	4
Scribus	1 329	72	1 371	69	320	20

Table 6.8: Levels II, III, and IV: Size on disk (kB). *The size on disk, in kilobytes, for class diagrams, ORDs, and class firewalls that represent the test cases.*

In columns 2, 4, and 6, we list the size in kilobytes[1] for compressed GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas, respectively. The average number of kilobytes for the compressed GXL encodings of instances of the Class Diagram, ORD, and Class Firewall schemas, are 95.75, 119.33, and 45.33, with standard deviances of 75.13, 96.76, and 39.61, respectively. Neither the contents, nor the sizes of these instances are directly comparable. However, the results show that none of the compressed GXL encodings for the 12 test cases is larger than 301 kilobytes, and that 25 of the 36 compressed files are no more than 100 kilobytes in size.

6.3.2 Transforming GXL Graphs with XSLT

In this subsection we apply XSLT style sheets to the GXL instances of the middle-level graphs. In particular, we investigate the run-time costs of the transformations, and present the results for instances of the Class Diagram, ORD, and Class Firewall schemas. First, we illustrate a representative XSLT style sheet for summarizing GXL instances, in this case, instances of the ORD schema. Second, we apply XSLT style sheets to the instances of each of the three schemas, and summarize the results. We used *xsltproc* [xsltproc Project 2005] to apply the style sheets to the GXL graphs.

[1]This table uses kilobytes. The similar tables in Section 6.2 use megabytes.

```
<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>
```

Source Listing 6.7: XSLT for summarizing ORD instances. *The XSLT style sheet we used to generate the results listed in Table 6.10. We used similar style sheets to generate the results listed in Tables 6.9 and 6.11.*

Test Case	Time (s)	Classes	Association	Composition	Dependency	Inheritance	OwnedElement	Total Edges
AvP	0.66	2 099	1 353	388	6 128	371	523	8 763
CppUnit	0.02	59	27	3	349	28	6	413
Doxygen	0.57	752	406	577	6 372	492	33	7 880
FluxBox	0.15	318	163	349	1 603	233	43	2 391
FOX	0.53	500	387	352	6 311	224	203	7 477
HippoDraw	0.24	272	379	27	3 289	195	1	3 891
Jikes	0.38	433	749	150	4 645	180	55	5 779
Keystone	0.15	163	173	22	2 120	111	4	2 430
Licq	0.09	224	32	17	1 249	161	1	1 460
Pixie	0.19	309	405	30	2 296	146	50	2 927
Scintilla	0.04	89	52	79	2 198	14	1	2 813
Scribus	0.17	243	1 154	33	1 568	17	25	2 797

Table 6.9: Class Diagram sizes for the test suite. *The number of classes and edges, by type, in the 12 instances of the Class Diagram schema constructed for the applications and libraries in our test suite.*

In Source Listing 6.7, we list an XSLT style sheet for summarizing the information in a GXL encoded instance of the ORD schema. As we noted in the introduction to this chapter, writing such a style sheet requires knowledge of only the schema, and not any particular instance. We list nine variables that contain the sets of instances of classes, edges, association edges, composition edges, dependency edges, inheritance edges, owned element edges, and polymorphic edges, respectively. We also list nine statements that print the sizes of the sets.

In Table 6.9, we present results from applying the XSLT style sheet PrintCdSummary.xslt to GXL encoded instances of the Class Diagram schema that represent each of the 12 test cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that *xsltproc* took less than one second to apply the style sheet to each of the test cases. In column 3, we list the total number of classes found in each instance; this class count includes all instances of the CppInfo schema classes ClassNonTemplate, ClassTemplate, and ClassTemplateInstantiation. In columns 4 through 8, we list the number of each in-

Test Case	Time (s)	Classes	Association	Composition	Dependency	Inheritance	OwnedElement	Polymorphic	Total Edges
AvP	3.33	2 082	1 346	381	6 075	367	381	15 872	24 422
CppUnit	0.06	56	27	3	349	26	6	409	820
Doxygen	2.36	724	390	575	6 267	475	31	11 144	18 882
FluxBox	0.37	307	161	346	1 600	226	40	1 470	3 843
FOX	9.76	499	387	352	6 311	223	203	27 716	35 192
HippoDraw	2.10	271	379	27	3 289	195	1	14 043	17 934
Jikes	2.50	427	748	147	4 640	179	53	14 533	20 300
Keystone	1.78	162	173	22	2120	111	4	12 185	14 615
Licq	0.60	224	32	17	1 249	161	1	4 613	6 073
Pixie	0.91	299	398	30	2 271	142	45	5 938	8 824
Scintilla	0.24	89	52	79	2 198	14	1	469	2 813
Scribus	0.57	243	1 154	33	1 568	17	25	3 293	6 090

Table 6.10: ORD sizes for the test suite. *The number of classes and edges, by type, in the 12 instances of the ORD schema constructed for the applications and libraries in our test suite.*

dividual edge type from the schema. Finally, in column 9, we list the total number of edges for each test case. On average, Class Diagram instances for our test cases contain hundreds of classes, and thousands of edges. Dependency edges are most common.

In Table 6.10, we present results from applying the XSLT style sheet PrintOrdSummary.xslt to GXL encoded instances of the ORD schema that represent each of the 12 test cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that, for half of the test cases, *xsltproc* took less than one second to apply the style sheet; the maximum running time was 9.76 seconds for *FOX*. In column 3, we list the total number of classes found in each instance. This class count includes all instances of the CppInfo schema classes ClassNonTemplate and ClassTemplateInstantiation. In columns 4 through 9, we list the number of each individual edge type from the schema. Finally, in column 10, we list the total number of edges for each instance. On average, ORD

Test Case	Time (s)	Classes	Edges	Min	Max	Avg
AvP	2.25	2 082	9 695	1	724	182.67
CppUnit	0.25	56	275	1	40	21.66
Doxygen	3.12	724	7 888	1	623	369.34
FluxBox	0.66	307	1 436	1	216	154.76
FOX	6.00	499	3 636	1	231	41.79
HippoDraw	0.83	271	4 200	1	210	116.07
Jikes	1.52	427	3 994	1	330	297.95
Keystone	3.33	162	3 242	1	140	87.89
Licq	0.50	224	959	1	172	26.90
Pixie	0.76	299	2 665	1	162	83.43
Scintilla	0.50	89	219	1	38	21.53
Scribus	0.62	243	1 175	1	107	64.40

Table 6.11: Class Firewall sizes for the test suite. *The numbers of classes and edges in the 12 instances of the Class Firewall schema. In addition, the minimum, maximum, and average class firewall sizes for each of the instances. Class firewall sizes are expressed as number of classes.*

instances for our test cases contain hundreds of classes, and tens of thousands of edges. Polymorphic edges are, by far, the most common.

In Table 6.11, we present results from applying the XSLT style sheet PrintCfw-Summary.xslt to GXL encoded instances of the Class Firewall schema that represent each of the 12 test cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that, for half of the test cases, *xsltproc* took less than one second to apply the style sheet; the maximum running time was 6.00 seconds for *FOX*. In column 3, we list the total number of classes found in each instance. These classes are the same set of classes found in the corresponding ORD instance. In column 4, we list the total number of edges for each instance. On average, Class Firewall instances for our test cases contain hundreds of classes, and thousands of edges.

In columns 5 and 6, we list the minimum and maximum number of classes, respectively, found in the class firewall for any class from the particular test case. For each of the 12 test cases the minimum class firewall size is one (1). The maximum class firewall size is as small as 38 classes in *Scintilla*, and as large as 724 classes in *AvP*. The average class firewall size for all 12 test cases is 122 classes, with a standard deviation of 112.

6.3.3 Discussion

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, no compressed GXL encoding of a middle-level graph is greater than 301 kilobytes for any of the 12 test cases. In addition, it took no more than 9.76 seconds to apply, using *xsltproc*, a style sheet that summarizes the contents of the given graph.

An application that builds a class firewall can take advantage of the savings that we highlight in this case study by taking ORD instances, rather than ASG or API instances, as input. This is the technique that we used to create GXL encoded instances of the Class Firewall schema for this case study. Other applications of these savings are described in Chapter 4.

We demonstrate the use of XSLT to extract information from GXL encoded instances of three different schemas. We show that this process is efficient, and present experimental results for the 12 test cases in our test suite. All GXL files that we created for this case study are available in our SourceForge.net repository, and are available for use by practitioners and other researchers.

Chapter 7

Applications: Empirical Evaluation with g^{4}re

In this chapter we present empirical evaluations with g^{4}re in two areas: (1) software measurement and (2) program comprehension. In Section 7.1, we present a system for computation of object-oriented metrics, and a case study that examines the use of object technology in games and language processing tools. In Section 7.2, we present a system for three-dimensional visualization of class template diagrams, and a case study in which we visualize ten open source C++ applications. In each section, we note the platform and g^{4}re version that we used to conduct the respective case study.

7.1 Application: Computing Object-Oriented Metrics

In this section we present our system for computing object-oriented metrics, and a case study using the system [Jamieson et al. 2005]. In Subsection 7.1.1, we provide an overview that includes motivation and background information on game application programming interfaces (APIs) and object-oriented metrics. In Subsection 7.1.2, we summarize research that relates to our system and case study. In Subsection 7.1.3, we define the metrics that we apply to the test suite in our case study. Finally, in Subsection 7.1.4, we describe our case study, and present the results.

7.1.1 Overview

Game developers have migrated from traditional approaches to the use of object technology to take advantage of its extensibility, and ease of modification and reuse. The object-oriented paradigm has a natural application to the domains of graphics, graphical user interfaces, and games. Early games such as Quake and Doom were implemented in C, because of its small learning curve and its fast compilation and

execution speeds. However, current game development teams are large, and consist of not only programmers, but also analysts, artists, and musicians. In addition, there have been considerable recent gains in compilation and execution speeds for C++; most modern games are developed using C++.

In this section, we evaluate the exploitation of object technology as it is used in a test suite of games and language processing tools. We developed a metric computation system using g^4re, and used it to apply several well-known object-oriented metrics to the test suite. The results formed a basis of comparison for the use of object technology in the two groups of programs. We used the results to draw conclusions about the modularity, the use of inheritance, and method complexity in the two groups of programs in the test suite. First, we provide some background information about game APIs and object-oriented metrics.

Game APIs

In early game development, DOS-based games were generally implemented with commands issued directly to the computer's hardware. These early DOS games used calls to device drivers for input devices, such as a mouse or a joystick, and calls to specific sound cards, such as Creative Labs' Sound Blaster. Video programming was the most difficult aspect of game development; the generation of fast and smooth graphics required significant programming skill. Graphics code frequently exploited the speed of assembly language programming, and depended on certain hardware-level features of the VGA graphics adapter.

Currently, few game developers write register-level video code, instead relying on prewritten application programming interfaces (APIs) that form a layer of software between the game and the hardware. The most popular API in current use is DirectX, which is implemented in C++ [Parberry 2000; Parberry et al. 2005]. The DirectX API provides low-level access to multimedia hardware in a device-independent manner. New versions of DirectX are released to permit game developers to take advantage of hardware advances as they occur, even after games have shipped. Unfortunately, the DirectX API is specific to Windows.

With the popularity of the Linux operating system, game developers became interested in platform-independent game programming. Several APIs have been introduced, including ClanLib [2005], and the Simple DirectMedia Layer (SDL) [2005]. SDL is a multimedia library that has been used to port a number of Windows-based games to Linux, and is the most popular of the platform-independent game APIs [Pazera 2003],

SDL supports virtually all of the major operating systems including FreeBSD, Linux, MacOS, Solaris, Windows, and Solaris. In addition to graphics support, SDL provides interfaces for playing sound, accessing CD-ROM drives, and achieving portable multi-threaded applications. SDL is released under the GNU LGPL and has accumulated a collection of user-contributed libraries that provide additional functionality.

Object-Oriented Metrics

Software metrics are quantitative measures that enable software developers, testers, and maintainers to evaluate the static properties of a software system [Fenton and Pfleeger 1998]. Software metrics are computed and the resultant data are collected, analyzed, and compared throughout the lifetime of a software system to evaluate improvement or deterioration of the software system. Software metrics are also useful for identifying problem modules of a software system.

Object-oriented metrics were introduced to measure software properties specific to object-oriented software systems, including properties pertaining to classes and their object instances [Chidamber and Kemerer 1994; Fenton and Pfleeger 1998]. The primary focus of object-oriented metrics is measuring properties of classes and their instances. Properties of interest include scope of properties, object complexity, coupling, and cohesion.

7.1.2 Related Work

The literature on object-oriented software metrics is extensive; Chidamber and Kemerer [1994] and Fenton and Pfleeger [1998] present detailed surveys. By contrast,

there has been relatively little work focused on applying metrics to assess the relative design characteristics of systems in different application domains. Further, our work is unique in its consideration of gaming applications. Other researchers have, however, considered comparisons that are similar in spirit to our own.

Paulson et al. [2004] perform an empirical evaluation of the differences between open-source and proprietary software. Their goal is to evaluate the validity of common perceptions regarding open-source projects. Their study considers five dimensions of comparison: (1) system growth, (2) design creativity, (3) complexity, (4) reliability, and (5) modularity. For each dimension of comparison, they apply a series of metrics to a test suite consisting of three open-source projects and three proprietary projects. Based on the resulting figures, the authors conclude that relative to their proprietary counterparts, open-source projects: (1) do not grow faster, (2) foster more creativity, (3) are more complex, (4) are more reliable, and (5) are less modular. We note that we share the authors' interest in complexity and modularity, and use a similar metric for evaluating complexity.

MacCormack et al. [2004] focus on the modularity of open-source software. Their approach is novel in its use of metrics defined over *design structure matrixes* [Steward 1981]. Each matrix captures the dependencies between the source files in a given implementation. A value of one at position (i, j) denotes the existence of a call from a function defined in file i to a function defined in file j. Similarly, a zero value denotes the absence of such a call. The authors consider two metrics. The first metric estimates the number of files affected, on average, by an arbitrary system change. The second metric is based on a clustering algorithm that groups collections of interdependent files. The metric estimates the cost of coordination between the individuals responsible for implementing the various elements by allocating a higher cost to *inter*-cluster dependencies, and a lower cost to *intra*-cluster dependencies. When applied to their test suite, which consists of one open-source system and one proprietary system, the resulting figures contradict the results of Paulson et al.: the open-source system appears more modular. Later, however, the authors evaluate a major redesign of the proprietary system which fares better: it is more modular than

the open-source system. We note that the size of the test suite makes it difficult to draw definitive conclusions.

Ferrett and Offutt [2002] report an empirical evaluation that considers different implementation paradigms, rather than different application domains. The authors analyze programs written in Fortran, C, C++, and Java to compare the modularity of programs written in a procedural style with those written in an object-oriented style. Ferrett and Offutt define a *module* as a function (or subroutine) in Fortran or C, and as a method in C++ or Java. They measure modularity by counting implementation modules, lines of code per module, and number of parameters per module. The assumption is that modularity increases with each of these measures. Their test suite includes 38 programs, with eight programs written in Java, and ten programs each written in Fortran, C, and C++. Based on the test suite, the authors conclude that object-oriented programs typically have more modules that are smaller and have fewer parameters. It is interesting to note that despite the aggregate conclusion, C++ programs appear to be no more modular than their C counterparts. The authors provide a possible explanation by observing that C++ programs are often developed by programmers trained in C; the programs need not be object-oriented. One final point is that the test suite contains programs that are only partially developed, and target different domains. The authors note that without experimental controls to account for these factors, the conclusions may be biased.

7.1.3 Methodology

In this subsection, we define the object-oriented metrics that we computed for the case study in Section 7.1.4. We define one metric measuring complexity, Weighted Methods per Class. We define three metrics measuring the use of inheritance: Depth of Inheritance Tree, Number of Ancestors, and Number of Children.

Metric 1: Weighted Methods per Class (WMC). This metric measures the complexity of an object, and is an indicator of the time and effort required to develop

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

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

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

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

Metric 2: Depth of Inheritance Tree (DIT). This metric is the length of the maximum path from a class to the root of its inheritance hierarchy, relates to scope of properties, and is an indicator of the number of ancestor classes that can potentially affect a class. Given a class C with a set of base classes BC, the metric is computed as

$$DIT(C) = \begin{cases} 0 & \text{if } |BC| = 0 \\ max(\{DIT(B_i) : B_i \in BC\}) + 1 & \text{if } |BC| > 0 \end{cases}$$

Metric 3: Number of Ancestors (NOA). This metric is the total number of ancestor classes of a class. In the absence of multiple inheritance, NOA is equivalent to DIT. In the presence of multiple inheritance, care must be taken to avoid counting an ancestor class more than once, due to the possibility of a diamond-shaped inheritance hierarchy.

Test Case	Version	Source Files	Translation Units		NCLOC (≈ K)
			Total	C++	
ASC	1.16.1.0	436	199	194	130
AvP	CVS 07/22/05	509	222	95	295
Freespace2	CVS 07/22/05	652	220	220	365
Scorched3D	38.1	1 069	513	492	110
Doxygen	1.3.9.1	260	122	90	170
g^4re	1.0.4	128	60	60	10
Jikes	1.22	75	38	38	70
Keystone	0.2.3	123	52	52	16

Table 7.1: Test suite. *The eight test cases that we use in our study: four SDL games and four language processing tools. For each test case, we list the version, the number of source files, the number of translation units, the number of C++ translation units, and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC).*

Metric 4: Number of Children (NOC). This metric is the number of immediate successors of a class, and measures the breadth of inheritance. Given a class C with a set of derived classes DC, the metric is computed as

$$NOC(C) = |DC|$$

7.1.4 Case Study

In this subsection we describe the results that we obtained using our metrics tool to evaluate game application software. We evaluated game software by comparing metrics computed for four games and four language processing tools. The results that we report in this section capture information about the sizes of the programs, and the exploitation of object technology as measured by the metrics described in Subsection 7.1.3.

We first provide information about the eight (8) test cases that form our test suite: four games implemented using the SDL API, and four language processing tools. We then provide results describing: (1) modularity and delegation, (2) the use of inheritance, and (3) complexity of methods.

Test Suite

In Table 7.1, we list the eight test cases that form our test suite, and size statistics for each test case. In column 1 of the table, we list the names that we use to refer to each of the test cases. We list the SDL games in the first four rows. The four games are: *Allied Strategic Command* (*ASC*), Aliens vs Predator (*AvP*), Freespace 2 (*Freespace2*), and Scorched 3d (*Scorched3D3D*). We list the language processing tools in the last four rows. The four language processing tools are: *Doxygen*, $\mathsf{g}^4\mathsf{re}$, *Jikes*, and *Keystone*. *Doxygen* is a documentation system for C, C++, and Java [van Heesch 2006]. *Jikes* is a Java compiler system from IBM [IBM Jikes Project 2006]. *Keystone* is a parser and front end for ISO C++ [Keystone Project 2005; Malloy et al. 2003a].

In column 2 of Table 7.1, we list the version number for each test case. In column 3 through 5, we list the number of source files, the number of total translation units, and the number of C++ translation units for each test case, respectively. Three of the games and one of the language processing tools (*Doxygen*) use C and/or assembly language in addition to C++. Finally, in column 6 of the table, we list the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC) for each test case.

The games are larger than the language processing tools. For example, the average NCLOC for the games is 231 000, but the average NCLOC for the language processing tools is 78 000; therefore, the average game in our test suite is three times as large as the average language processing application.

We executed all experiments on a custom workstation with an *AMD Athlon64* ^{TM} *3000+* processor, 1024 MB of PC3200 DDR RAM, and an 80 GB, 7200 RPM SATA hard disc on which we installed the Slackware 10.1 operating system after formatting with version 3.6 of the *ReiserFS* filesystem. We performed the experiments with version 1.0.4 of $\mathsf{g}^4\mathsf{re}$, which we compiled with version 4.0.2 of *gcc*. We created all `tu` files with *gcc* version 3.3.4.

	Classes				Functions			
Test Case	Total	Abstract	Root	Leaf	Total	Member	Virtual	Pure
ASC	1 389	58	901	390	8 693	7 775	2 170	208
AvP	1 732	28	1 369	327	11 548	9 350	1 216	90
Freespace2	332	0	320	12	9 468	1 687	48	0
Scorched3D	799	50	405	364	8 432	7 210	1 907	112
Doxygen	315	9	153	157	5 422	4 570	2 159	249
g^4re	78	17	27	37	849	798	303	106
Jikes	378	5	210	158	5 717	5 685	602	16
Keystone	160	14	49	87	2 354	2 306	1 178	189

Table 7.2: Modularity and delegation. *The number of classes and functions in each test case; the classes and function are broken down into categories that relate to modularity and delegation.*

Modularity and Delegation

In Table 7.2, we present results that capture information about modularity of and delegation in the eight test cases. In column 1 we list the test cases, again, we list the SDL games in the first four rows, and the language processing tools in the last four rows. In columns 2 through 5, we list the number of total, abstract, root, and leaf classes, respectively. In columns 6 through 9, we list the number of total, member, virtual, and pure virtual functions, respectively.

The games *AvP* and *ASC* contain the most and second most classes, respectively, of the eight test cases. *AvP* contains 1 732 classes; *ASC* contains 1 389 classes. It is somewhat surprising that the *AvP* game contains a large number of classes in view of the large number of C files in the program. In Table 7.1, we indicated that *AvP* contains 222 translation units, but only 95 of these are C++ translation units; over half of the translation units consist of C or assembly code. However, g^4re uses an ASG representation of the input program; all template classes are instantiated in an ASG for C++. So, the count of classes that we listed in column 2 of Table 7.2 includes class template instantiations in addition to classes that are not templates. We use this set of classes when computing the metrics in the following subsections.

The results in Table 7.2 suggest that, for our test suite of four games and four language processing tools, the games may be more modular than the language processing tools. For example, the four games contain a total of 4 252 classes in 923 000

Test Case	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	4	0.645	0.991	0	0
AvP	0	5	0.304	0.707	0	0
Freespace2	0	1	0.036	0.187	0	0
Scorched3D	0	9	0.931	1.496	0	0
Doxygen	0	4	0.927	1.122	1	0
g^4re	0	5	1.897	1.592	2	0
Jikes	0	3	0.714	0.960	0	0
Keystone	0	5	1.794	1.450	2	0

Table 7.3: Depth Of Inheritance Tree (DIT). *Statistics about the DIT metric for each test case.*

Test Case	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	6	0.687	1.097	0	0
AvP	0	5	0.307	0.712	0	0
Freespace2	0	1	0.036	0.187	0	0
Scorched3D	0	9	1.036	1.632	0	0
Doxygen	0	7	1.041	1.214	1	0
g^4re	0	5	1.910	1.605	2	0
Jikes	0	4	0.728	0.979	0	0
Keystone	0	5	1.794	1.450	2	0

Table 7.4: Number of Ancestors (NOA). *Statistics about the NOA metric for each test case.*

NCLOC; the class to one thousand NCLOC ratio is 4.6. However, the four language processing tools contain a total of 931 classes in 310 000 NCLOC; the class to one thousand NCLOC ratio is 3.0

Inheritance

In Tables 7.3, 7.4, and 7.5, we list statistics about the Depth of Inheritance Tree (DIT), Number of Ancestors (NOA), and Number of Children (NOC) metrics, respectively. Column 1 of each table lists the test cases. Columns 2 and 3 list the minimum, maximum values, respectively. Columns 4 and 5 list the mean and standard deviation of the values, respectively. Columns 6 and 7 list the median and mode values, respectively. The median is the value for which an equal number of values lie above and below; the mode is the most common value.

The DIT metric equals the length of the longest path from a class to the root of its inheritance tree. The NOA metric equals the total number of ancestor classes of a

Test Case	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	27	0.314	1.550	0	0
AvP	0	30	0.121	0.850	0	0
Freespace2	0	12	0.036	0.659	0	0
Scorched3D	0	11	0.200	0.912	0	0
Doxygen	0	48	0.359	2.767	0	0
g^4re	0	6	0.487	1.171	0	0
Jikes	0	26	0.455	2.474	0	0
Keystone	0	10	0.606	1.656	0	0

Table 7.5: Number of Children (NOC). *Statistics about the NOC metric for each test case.*

class. The NOC metric measures the number of immediate successors of a class in its inheritance tree. Intuitively, the DIT metric measures the depth of the inheritance tree, and the NOC metric measures the breadth of the inheritance tree. The NOA metric indicates whether or not multiple inheritance is used, when compared to the DIT metric.

In Table 7.3, we list statistics about DIT that we computed for the eight test cases. *Scorched3D* had a maximum value of 9, which gave it the deepest inheritance tree in the test suite. In addition, *Scorched3D* had the largest mean value, 0.931, but also the largest standard deviation, 1.496, which means that *Scorched3D* had the largest amount of variation among the depths of inheritance trees for its classes. *Freespace2* is the only test case that did not have an inheritance tree with a depth greater than one (1).

In Table 7.4, we list statistics about NOA that we computed for the eight test cases. A comparison of the mean NOA values to the mean DIT values listed in Table 7.4 indicates that multiple inheritance is used by three of the four games, and two of the four language processing tools. However, the differences between the mean values for NOA and the mean values for DIT indicate that the two language processing tools made heavier use of multiple inheritance than the three games.

In Table 7.5, we list statistics about NOC that we computed for the eight test cases. *AvP* had a maximum value of 30, which gave it the second broadest inheritance tree in the test suite. However, *AvP* also had the second lowest mean, and the second lowest variation among the numbers of children for its classes. The median

Test Case	Min	Max	Mean	Std Dev	Median	Mode
ASC	0	561	12.977	30.365	4	0
AvP	0	107	7.290	10.594	3	3
Freespace2	0	123	6.660	15.707	3	3
Scorched3D	0	240	17.372	19.058	12	3
Doxygen	0	430	27.676	57.497	7	7
g^4re	0	206	17.756	30.270	13	0
Jikes	0	2 016	32.397	119.124	13	10
Keystone	0	557	24.388	52.474	15	14

Table 7.6: Weighted Methods per Class (WMC). *Statistics about the WMC metric for each test case.*

and mode values for *AvP* are zero (0), as are the median and mode values for all test cases.

When taken together with the statistics listed in Tables 7.3 and 7.4, the statistics listed in Table 7.5 show that: (1) *Freespace2* makes very little use of inheritance, (2) on average, the four language processing tools make more use of inheritance than the four games, (3) on average, maximum depth to number of classes and maximum breadth to number of classes ratios are much lower for the four language processing tools than for the four games.

Complexity

In Table 7.6, we list statistics about the Weighted Methods per Class (WMC) metric, which equals the sum of the cyclomatic complexities for the methods in a class. Column 1 lists the test cases. Columns 2 and 3 list the minimum, maximum values, respectively. Columns 4 and 5 list the mean and standard deviation of the values, respectively. Columns 6 and 7 list the median and mode values, respectively.

The classes (and hence methods) in the four language processing tools are more complex than the classes in the four games. For example, the average maximum value for the language processing tools is 802.250, but the average maximum value for the games is only 257.750. Similarly, the average mean value for the language processing tools is 25.554, but the average mean value for the games is only 11.074.

Concluding Remarks

We have shown that, for our test suite, the games are more modular than the language processing tools, but that the language processing tools make more use of inheritance. In addition, we have shown that the methods in the games are less complex than the methods in the language processing tools. However, our metrics computation system does not include library code in its analysis, and it is likely that much of the method complexity in games is found in the SDL API.

7.2 Application: Visualizing Class Template Diagrams

In this section we present our system for visualizing class template diagrams, and a case study using the system [Hoipkemier et al. 2006]. In Subsection 7.2.1, we provide an overview that includes motivation and background information on generic programming. In Subsection 7.2.2, we summarize research that relates to our system and case study. In Subsection 7.2.3, we provide an overview of our methodology for the case study. Finally, in Subsection 7.2.4, we describe our case study, and present the results.

7.2.1 Overview

Generic programming deals with finding abstract representations of efficient algorithms and data structures, and expressing them in an adaptable interoperable manner [Jazayeri et al. 2000]. The generic programming paradigm is a popular ancillary tool to object technology; conferences, seminars, and other literature have recently appeared to address problems and concerns related to this paradigm [Eichelberger and v. Gudenberg 2000; Glück and Lowry 2005; Jazayeri et al. 2000; Lengauer et al. 2004; Veldhuizen 2000].

The canonical example of generic programming is the C++ Standard Library (STL). In addition, other libraries, such as Boost, Loki and Blitz++, rely heavily on both generic and generative programming to produce code that is more general, more efficient, and more easily incorporated into existing applications than their

non-generic counterpart [Alexandrescu 2001; Siek and Lumsdaine 2005; Veldhuizen and Gannon 1998]. In recognition of the importance of generics, they have been recently introduced into both the C# 2.0 [Microsoft Corporation 2006], and the Java 5 [Sun Microsystems Inc 2006] programming languages.

One problem with generic programming is the dearth of technology to facilitate comprehension, documentation, and debugging of programs that utilize generic programming [Jazayeri et al. 2000]. This may be true, in part, because C++, the most mature language vehicle for generic programming thus far, is notoriously difficult to parse [Bodin et al. 1994; Klint et al. 2005; Knapen et al. 1999; Lilley 1997; Malloy et al. 2003a]. We were unable to find any tool description in the literature that provides a facility for the comprehension and debugging of generic C++ code.

Various graphical program representations have been used to reverse engineer class diagrams; most of these have used either the source code or Java bytecode as input, although some representations have used object code as input [Duffy and Malloy 2005; Eng 2002; Gutwenger et al. 2003; van Heesch 2006]. Unlike these systems, we used an abstract syntax graph (ASG). An ASG for a C++ program contains additional information about templates not included in the source code; in particular, information about class template instantiations and specializations is available. This is import, because C++ programmers often specialize types to provide space and/or time savings [Vandevoorde and Josuttis 2002].

In this section, we evaluate the utilization of generic programming in a test suite of ten deployed open source programs. We developed a visualization system using g^{4}re, and applied it to the test suite. We designed the system to visualize classes (non-templates), class templates, class template instantiations, and class template specializations in three dimensions. We placed classes (non-templates and templates) in the X-Y plane, and class template instantiations and specializations along the Z axis. We used the results to draw conclusions about the frequency and efficiency of generic programming in these applications.

7.2.2 Related Work

The literature on software visualization can be partitioned into three categories: (1) data visualization [Jones et al. 2004; Marcus et al. 2003], (2) class diagram layout [Eiglsperger et al. 2004; Gutwenger et al. 2003], and (3) code visualization. There has been no reported work that provides either visualization of templates, or three-dimensional visualization of class diagrams. Next, we describe relevant research on code visualization.

Malloy and Power [2005] exploit a molecular metaphor for three-dimensional visualization of visualize, in three dimensions, the dynamic object relationships in Java applications. They instrument Java bytecode to collect trace data, and then analyze it and visualize it in three dimensions using VRML. Their quantitative and graphical results include analyses of the SPEC JVM98 and JOlden benchmark suites; they reveal interesting relationships among the data structures in these applications. Unfortunately, their approach does not accommodate Java generics.

Lewerentz and Simon [2002] present a metrics based approach to software visualization that supports quality assessment of large object-oriented software systems written in C++ and Java. They use a combination of software metrics data and program structure information to form a virtual information space. They then visualize the information space using three-dimensional graph structures that represent, in a uniform manner, many aspects and views of the application under study. They layout the graphs by using a generic similarity measure to calculate geometric distances between graph nodes, and a force-directed mapping into three-dimensional space. Their approach does not include the visualization of C++ class templates or Java generics.

Eng [2002] presents a framework to visualize intermediate representations of Java programs that are constructed by an optimizing compiler. He visualizes both the software, and the characteristics of the execution platform. His visualization interface illustrates classes, including data and methods, and profile information in

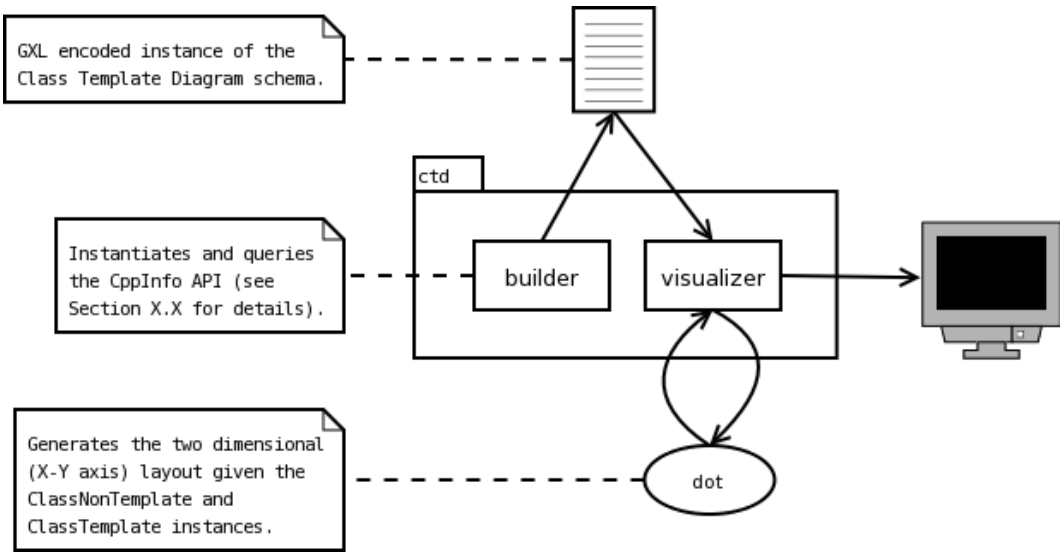

Figure 7.1: System overview. *Solid, directed lines show input. Dashed lines show notes.*

a single framework. His approach does not include visualization in three dimensions, or the visualization of Java generics.

7.2.3 Methodology

In this subsection, we describe the system that we used to conduct the case study in Section 7.2.4. We describe both of the components in the system: builder and visualizer. In addition, we illustrate the schema for a Class Template Diagram (CTD), which is the subject of our visualization.

In Figure 7.1, we provide a overview of the system. We illustrate the builder component to the left of center at the middle of the figure. The builder component, part of the ctd package, uses the API provided by g^4re to build and write a GXL encoded instance of a class diagram subset, which we termed a Class Template Diagram (CTD). The nodes in a CTD represent classes (non-templates), class templates, and class template instantiations and specializations. The nodes contain information about classes, including names, attributes, and operations. The edges in a CTD represent either inheritance, including information about access and the virtual specifier, or one of the aggregations: hasInstantiation and hasSpecialization. We illustrate the CTD schema in Figure 7.2, and list a link to it in Appendix B.

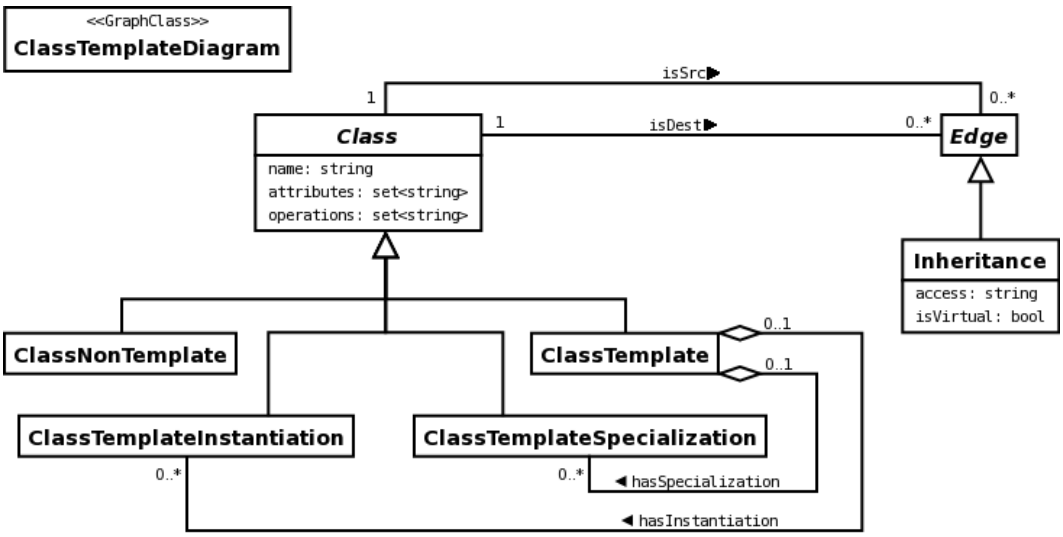

Figure 7.2: Schema for Class Template Diagram (CTD). *A UML class diagram representation of the CTD schema.*

We illustrate the visualizer component to the right of center at the middle of Figure 7.1. The visualizer component, part of the ctd package, takes the GXL encoded instance of the CTD schema (generated by the builder component) as input. The visualizer first sends an inheritance hierarchy, which includes only instances of ClassNonTemplate and ClassTemplate, to *dot*, which computes the two-dimensional layout for the inheritance hierarchy. The visualizer then reads the layout that *dot* has computed, and places instances of ClassTemplateInstantiation and ClassTemplate-Specialization on the Z axis, behind the corresponding instances of ClassTemplate. Finally, the visualizer then connects inheritance edges involving instantiations and specializations, and uses OpenGL to draw the three-dimensional diagram to the screen.

The three-dimensional diagram generated by our visualizer component is interactive, and can be manipulated in real time. All classes are represented by boxes, on which attribute and operation information is drawn as it would appear in a UML Class Diagram. Instances of ClassNonTemplate are colored yellow, instances of ClassTemplate are colored light blue, and instances of ClassTemplateInstantiation and ClassTemplateSpecialization are colored red. The inheritance edges are colored green

Test Case	.ctd.gxl	.dot
Doxygen	2 904	1 048
FluxBox	824	336
FOX	1 336	848
HippoDraw	808	488
Jikes	1 248	536
Keystone	264	176
Licq	624	384
Pixie	616	304
Scintilla	216	136
Scribus	720	432

Table 7.7: Size on disk (kB). *The size on disk, in kilobytes, for uncompressed GXL encodings of instances of the Class Template Diagram schema, and the uncompressed dot encodings of the inheritance hierarchies.*

at the isDest end, and brown at the isSrc end. The aggregation edges hasInstantiation and hasSpecialization are colored blue.

7.2.4 Case Study

In this subsection we describe the results that we obtained using our system to visualize the use of generic programming in open source, C++ software. The results that we report in this section capture the space and time costs of our system, and information about the use of generic programming in our test suite of 10 open source programs [1]

We executed all experiments on a *Sun* TM workstation with an *AMD Opteron* TM *148* processor, 1024 MB of PC3200 DDR RAM, and an 80 GB, 7200 RPM SATA hard disc on which we installed the Fedora Core 4 operating system after formatting the *ext3* filesystem. We performed the experiments with version 1.0.8 of g^4re, which we compiled with version 4.0.0 of *gcc*. We created all `tu` files with *gcc* version 3.3.6.

	.ctd.gxl		.dot			
Test Case	Generate	Parse	Generate	Parse	Render	Total
Doxygen	6.43	1.03	0.34	231.55	0.21	239.56
FluxBox	2.08	0.32	0.06	59.20	0.05	61.71
FOX	3.07	0.40	0.11	91.87	0.06	95.51
HippoDraw	1.77	0.28	0.09	49.51	0.04	51.69
Jikes	2.98	0.44	0.10	80.14	0.08	83.74
Keystone	0.50	0.09	0.03	17.80	0.01	18.43
Licq	1.18	0.20	0.04	38.75	0.03	40.20
Pixie	1.26	0.23	0.06	36.51	0.04	38.10
Scintilla	0.49	0.06	0.03	12.89	0.02	13.49
Scribus	1.84	0.24	0.06	44.11	0.03	46.28

Table 7.8: Time (s). *The running time, in seconds, for each phase of our visualization system, and for the entire visualization system.*

Space and Time Costs

In Table 7.7, we list the sizes on disk, in kilobytes, for graphs that are generated, parsed, and used by our visualization system to represent the test cases. In column 1, we list the test cases. In column 2, we list the total sizes of the uncompressed GXL encoded instances of the Class Template Diagram (CTD) schema. In column 3, we list the total sizes of the dot encoded inheritance hierarchies.

The CTD instances are comparable [2] in size to the middle-level graphs described in Section 6.3. The dot files are, on average, half the size of the corresponding CTD instance. Next, we examine the time costs for generating and parsing these files.

In Table 7.8, we list the running times, in seconds, for our visualization system. In column 1, we list the test cases. In columns 2 and 3, we list the times to generate and to parse GXL encoded instances of the Class Template Diagram schema, respectively. Recall that the builder component generates the file, and the visualizer component parses the file. In columns 4 and 5, we list the times to generate and to parse dot encodings of inheritance hierarchies, and the *dot* generated two-dimensional layout, respectively. In column 6, we list the times to render the

[1]The test suite for this study is a subset of the test suite for the case studies in Chapter 6, from which *AvP* and *CppUnit* have been omitted. See Subsection 6.1 or our online repository for information about the test suite. Furthermore, see Subsection 6.2.1 and 6.2.2 for information about the space and time costs of the input to the system.

[2]Unlike the rest of g^{4}re, the visualization system does not use the *pulse* library to write GXL files. Differences in formatting can cause sizes on disk to vary widely for GXL files.

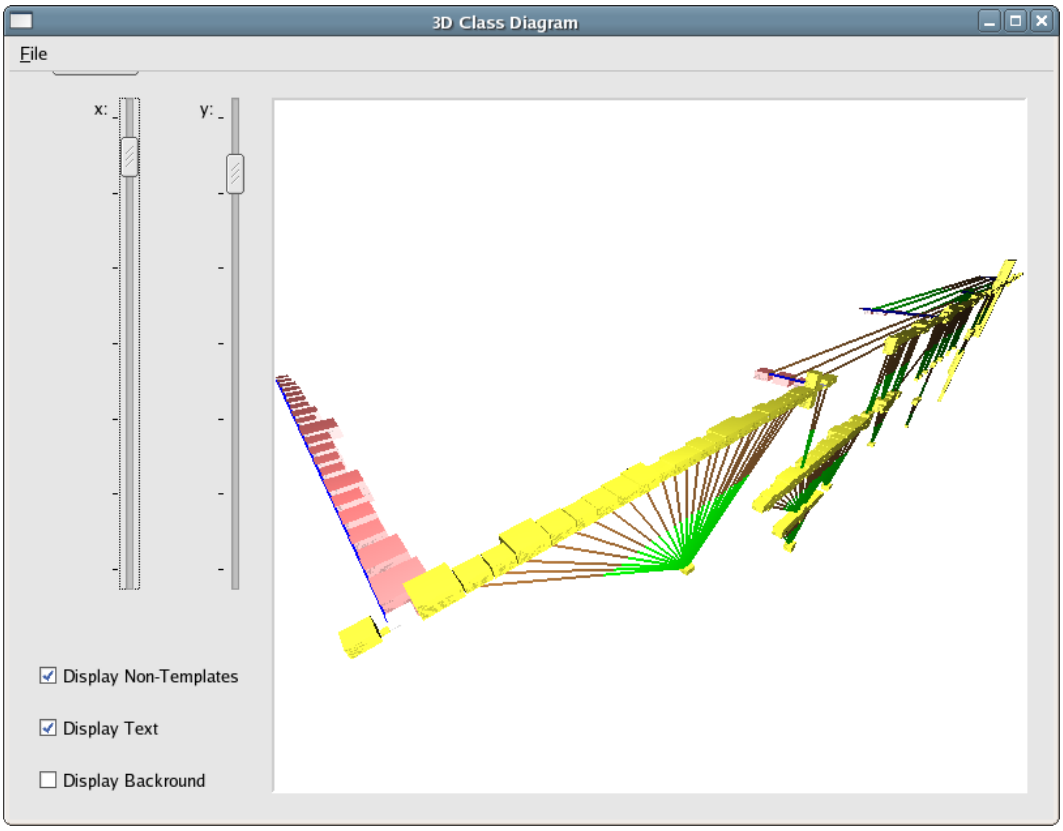

Figure 7.3: Visualization of CTD for *Pixie*. *The interface to the visualization system, and and a three-dimensional CTD diagram for Pixie. The two sliders control the azimuth and the elevation. The keyboard controls horizontal and vertical movement, and zooming.*

OpenGL three-dimensional representation of the diagram. Finally, in column 7, we list the total time for the visualization system.

The running time for each test case is dominated by the time spent parsing the *dot* generated two-dimensional layout. Even with this cost, which could be greatly reduced by eliminating the dependency on *dot*, the running time is under one minute for over half of the test cases. Clearly, our use of three dimensions adds minimal costs to the system as a whole.

Visualization Results

Three of the test cases did not contain any template classes: *Licq*, *Scintilla*, and *Scribus*. Also, only three of the test cases contained ten or more template classes: *Doxygen*, *FluxBox*, and *Pixie*. However, the number of class template instantiations

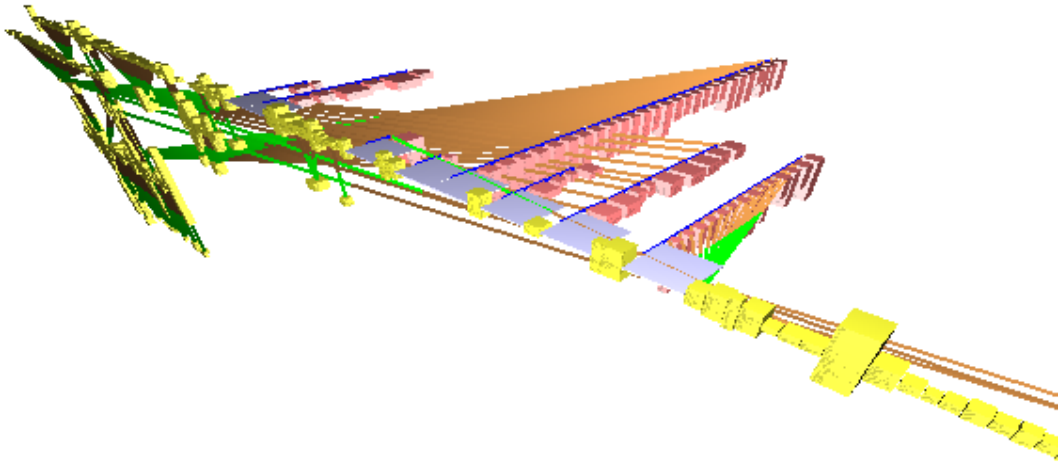

Figure 7.4: Visualization of CTD for *FluxBox*. *A three-dimensional CTD diagram for FluxBox.*

can be large; for example, in *Doxygen*, each class template averages 10 instantiations. In addition, in *Jikes* the average number of instantiations per class template is 20.

Our visualization system allowed us to obtain information about the test cases that was not readily available from inspection of the source code. In Figure 7.3, we show the interface to the system. The two sliders that control the azimuth and the elevation. The keyboard controls horizontal and vertical movement, and zooming. The default view shows all classes in the system, together with inheritance and template instantiation and specialization relationships. We provide options that allow the user to view only the templates in the system, including instantiations and specializations; the check boxes allow the user to hide the background and text.

To the right of the controls in Figure 7.3, we illustrate a three-dimensional CTD diagram for *Pixie*. The cubes aligned at the left of the diagram lie on the Z-axis; these cubes represent instantiations of the $CArray < T >$ class template. The majority of instantiations in *Pixie* are instantiations of the $CArray < T >$ class template; this indicates that modifications to $CArray < T >$ have far reaching effects on *Pixie*. This information is not readily available from inspection of the source code. We illustrate an additional three-dimensional CTD diagram, for *FluxBox*, in Figure 7.4.

Concluding Remarks

We have described our approach to three-dimensional visualization of class template diagrams for 10 open source, C++ programs. Unlike systems that use the source code to construct UML class diagrams, our system uses an ASG; therefore, our system has access to information about templates that is not readily available from inspection of the source code. We have also shown that our approach is feasible for medium-sized programs.

Chapter 8

Conclusion

In Section 1.1, we identified the problems we investigated in the area of reverse engineering; the problems focus on infrastructure support for interoperability. In this chapter, for each problem, we first restate the original goal, and highlight our publications that addressed the goal. We then restate the original evaluation criteria, and state how we satisfied the criteria.

Problem 1: Schemas for Low- and Middle-Level Program Representation Graphs

Goal

Our goal for this problem was to create GXL schemas for low- and middle-level graphs, and to use known transformations between graphs to guide the organization of the schemas into a hierarchy. We published work that addressed this goal in the *Proceedings of the 12th Working Conference on Reverse Engineering* [Kraft et al. 2005b], and in the journal *Information and Software Technology* [Kraft et al. 2007a]. We presented this work in Chapter 4.

Evaluation Criteria

1. The hierarchy is arranged in levels, such that an instance of a schema at one level can be created using only information contained in instances of schemas at previous levels.
2. Instances of low-level schemas contain the information needed to create instances of middle-level schemas, including call graphs, class-centric graphs, control flow graphs, and dependency graphs.

3. Low-level schemas are language-specific, and middle-level schemas are language-independent.

4. Low-level schemas for C++ accurately and adequately represent templates, including instantiations, specializations, and partial specializations.

5. Low-level schemas for C++ represent function pointers, including member function pointers.

In Section 4.1, to address evaluation criteria 1 and 2, we illustrated the hierarchy of schemas, which consists of two major partitions, low-level and middle-level, and five minor partitions, Levels 0 through 4. We also described the structure of the hierarchy, which represents the progression of information from schemas at one level to schemas at a subsequent level.

In Sections 4.2 and 4.3, to address evaluation criterion 3, we presented our low-level schemas. The abstract syntax graph (ASG) and the application programming interface (API), are language-specific; we focused on the C++ language. We also presented several language-independent middle-level schemas, including those for the call graph and the class firewall.

In Section 4.2, to address evaluation criteria 4 and 5, we described the *gcc* ASG schema for C++, GENERIC, and our API schema for C++, CppInfo. Both schemas provide an accurate and adequate representation of templates, including instantiations, specializations, and partial specializations. In addition, both schemas provide a representation of function pointers, including member function pointers (sometimes called pointers to members).

Problem 2: Tool Support for Reverse Engineering C++ Programs

Goal

Our goal for this problem was to create a public domain, general purpose tool for reverse engineering C++ programs. We published work that addressed this goal in the *Proceedings of the Dagstuhl Seminar on Transformation Techniques in Software*

Engineering [Kraft et al. 2005a], in the *Proceedings of Future Play 2005* [Jamieson et al. 2005], in the *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering* [Hoipkemier et al. 2006], and in the *Special Issue on Experimental Software and Toolkits* of the journal *Science of Computer Programming* [Kraft et al. 2007b]. We presented this work in Chapters 5, 6, and 7.

Evaluation Criteria

1. The tool is open-source and available on the Web.
2. The tool correctly parses, instantiates, and specializes templates.
3. The tool consists of loosely coupled, reusable modules.
4. The tool provides a module for linking C++ translation units.
5. The tool provides an API module for accessing information about declarations, statements, and some expressions.
6. The tool exchanges information via conforming instances of GXL schemas.
7. The tool is robust and efficient enough to use on medium-sized C++ programs, which contain up to 500 000 lines of non-commented, non-preprocessed lines of code.
8. The tool is general purpose.

To address evaluation criterion 1, we placed our tool chain for reverse engineering C++ programs, g^{4}re, in our SourceForge.net repository [Kraft 2006]. In Section 5.1, we described the architecture of g^{4}re, which uses *gcc* to correctly parse, instantiate, and specialize templates, and to address evaluation criterion 2. We also described the six constituent modules: the ASG module, the schema and serialization modules, the transformation modules, the linking module, and the API module. The modular GXL-based pipe-filter architecture of g^{4}re addresses evaluation criterion 3 and 6; notable modules are the linking module and the API module, which address evaluation criteria 4 and 5, respectively. We plan to reuse the linking and API modules for another project in the immediate future.

In Chapters 6 and 7, we described many experiments that we have performed with g^{4}re. In Section 6.1, we described our most recent test suite, which contains 12 popular, open-source applications and libraries. The test suite consists of 1,200 C++ translation units and approximately 1 000 000 lines of non-commented, non-preprocessed code. The largest test case that we used with g^{4}re contained approximately 365 000 lines of code; we described this test case in Subsection 7.1.4. In Sections 6.2 and 6.3, we described case studies that measure the costs of common reverse engineering and program analysis tasks, and address criterion 7. Finally, to address evaluation criterion 8, we presented empirical evaluations that we performed with g^{4}re in two areas: (1) software measurement, specifically, computation of object-oriented metrics, in Section 7.1, and (2) program comprehension, specifically, three-dimensional visualization of class template diagrams, in Section 7.2.

Problem 3: A Repository of Reverse Engineering Artifacts

Goal

Our goal for this problem was to create a public repository of reverse engineering artifacts, and to populate it with empirical results, including all tools, scripts, and documents needed to reproduce the results. We published work that addressed this goal in the *Proceedings of the 12th Working Conference on Reverse Engineering* [Kraft et al. 2005b], and in the journal *Information and Software Technology* [Kraft et al. 2007a]. We presented the repository artifacts in the case studies of Chapter 6. We present links to the repository in Appendix B.

Evaluation Criteria

1. The repository contains a test suite, including important details about each test case:
    - Version
    - Size metrics
    - Configuration and build information

2. For at least one graph at each level of our hierarchy (see Problem 1), the repository contains:
   - A GXL schema
   - Tools that exchange information via conforming GXL instances of the schema
   - GXL instances of the schema for each test case in the test suite
   - A graph transformation that summarizes the information in a GXL instance
   - Empirical results that show the space and time costs incurred by the documents and tools, respectively
3. The repository contains all artifacts needed to reproduce the results described in the previous items, including platform information for each experiment.
4. The repository is available to the public, particularly the reverse engineering community.

We created a repository at SourceForge.net to hold our reverse engineering artifacts [Kraft 2006]. We populated our repository with the test suite information and artifacts specified in evaluation criteria 1 through 3. The specific graphs for which we created GXL schemas are: the ASG, the API, the class diagram, the call graph, the CFG, the ORD, the ICFG, and the class firewall. The specific graphs for which we created tools are: the ASG, the API, the class diagram, the ORD, and the class firewall[1]. All of the tools are part of the GXL-based pipe-filter architecture of g^{4}re. Finally, to address evaluation criterion 4, we described our repository in the two publications mentioned above, cited it in several other publications, and discussed it in our November 2005 presentation at WCRE.

[1]Special thanks to the CpSc 829 students in Fall 2005, particularly Ben Hoipkemier, for providing the class firewall tool.

APPENDICES

A Acronyms and Abbreviations

API	–	Application Programming Interface
ASG	–	Abstract Syntax Graph
CCFG	–	Class Control Flow Graph
CDG	–	Control Dependence Graph
CFG	–	Control Flow Graph
CTD	–	Class Template Diagram
DIT	–	Depth of Inheritance Tree
DTD	–	Document Type Definition
GCC	–	GNU Compiler Collection
GXL	–	Graph eXchange Language
ICFG	–	Interprocedural Control Flow Graph
NOA	–	Number of Ancestors
NOC	–	Number of Children
OO	–	Object-Oriented
ORD	–	Object Relation Diagram
PDG	–	Program Dependence Graph
SDG	–	System Dependence Graph
SEF	–	Standard Exchange Format
TU	–	Translation Unit
UML	–	Unified Modeling Language
WCRE	–	Working Conference on Reverse Engineering
WMC	–	Weighted Methods per Class
XML	–	Extensible Markup Language
XSL	–	Extensible Stylesheet Language
XSLT	–	XSL Transformations

B Repository of Reverse Engineering Artifacts

GXL Schemas

Level 0	`GENERIC.gxl`
Level I	`CppInfo.gxl`
Level II	`ClassDiagram.gxl`
	`CallGraph.gxl`
	`CFG.gxl`
Level III	`ORD.gxl`
	`ICFG.gxl`
Level IV	`ClassFirewall.gxl`

GXL Instances

Level 0	`ASG/`
Level I	`API/`
Level II	`ClassDiagram/`
Level III	`ORD/`
Level IV	`ClassFirewall/`

Other Artifacts

	`tools/`
	`transformations/`
	`results/`

BIBLIOGRAPHY

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2006. *Compilers: Principles, Techniques, and Tools*, Second ed. Addison-Wesley.

Aigner, G., Diwan, A., Heine, D. L., Lam, M. S., Moore, D. L., Murphy, B. R., and Sapuntzakis, C. 2006. An overview of the SUIF2 compiler infrastructure. http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/overview.ps.

Al-Ekram, R. and Kontogiannis, K. 2005. An XML-based framework for language neutral program representation and generic analysis. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Manchester, UK.

Alexandrescu, A. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Andrews, J. H. 2004. Relevant empirical testing research: Challenges and responses. *ACM SIGSOFT Software Engineering Notes 29,* 5 (September), 1–4.

Antoniol, G., Penta, M. D., Masone, G., and Villano, U. 2004. Compiler hacking for source code analysis. *Software Quality Journal 12,* 4 (December), 383–406.

Arikan, O. 2006. Pixie version 1.5.2. http://pixie.sourceforge.net.

Bell Canada Inc. 2000. *DATRIX - Abstract Semantic Graph Reference Manual*, 1.4 ed. Bell Canada Inc., Montreal, Canada.

Bodin, F., Beckman, P., Gannon, D., Gotwals, J., Narayana, S., Srinivas, S., and Winnicka, B. 1994. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*. Sunriver, OR, USA, 122–136.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. 2006. Extensible markup language (XML) 1.0. W3C recommendation, W3C.

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

Chidamber, S. R. and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering 20,* 6, 476–493.

ClanLib Project. 2005. ClanLib Game SDK. http://www.clanlib.org.

CppUnit Project. 2006. CppUnit version 1.10.2. http://cppunit.sourceforge.net.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (October), 451–490.

Czeranski, J., Eisenbarth, T., Kienle, H., Koschke, R., Plödereder, E., Simon, D., Girard, J. F., and Würthner, M. 2000. Data exchange in bauhaus. In *Proceedings of the Seventh Working Conference on Reverse Engineering.* IEEE Computer Society, Brisbane, Australia, 293–295.

Das, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* Vancouver, BC, Canada, 35–46.

Dean, T. R., Malton, A. J., and Holt, R. C. 2001. Union schemas as a basis for a C++ extractor. In *Proceedings of the Eighth Working Conference on Reverse Engineering.* IEEE Computer Society, Stuttgart, Germany. www.cppx.com.

Do, H., Elbaum, S., and Rothermel, G. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10,* 4 (October), 405–435.

Duffy, E. B. and Malloy, B. A. 2005. A language and platform-independent approach for reverse engineering. In *Proceedings of the Third ACIS International Conference on Software Engineering Research, Management & Applications.* 415–422.

Ebert, J., Kullbach, B., and Winter, A. 1999. GraX – an interchange format for reengineering tools. In *Proceedings of the Sixth Working Conference on Reverse Engineering.* IEEE Computer Society, Atlanta, GA, USA, 89–98.

Edison Design Group. 2000. C++ Front End. http://www.edg.com/cpp.html.

Eichberg, M., Mezini, M., Ostermann, K., and Schafer, T. 2004. XIRC: A kernel for cross-artifact information engineering in software development environments. In *Proceedings of the Eleventh Working Conference on Reverse Engineering.* IEEE Computer Society, Delft, The Netherlands, 182–191.

Eichelberger, H. and v. Gudenberg, J. W. 2000. UML description of the STL. In *Proceedings of the First Workshop on C++ Template Programming.* Erfurt, Germany. http://oonumerics.org/tmpw00/.

Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., and Siebenhaller, M. 2004. Automatic layout of uml class diagrams in orthogonal style. *Information Visualization 3,* 3, 189–208.

Elliott Sim, S., Easterbrook, S., and Holt, R. C. 2002. On using a benchmark to evaluate C++ extractors. In *Proceedings of the Tenth International Workshop on Program Comprehension.* Paris, France, 114–123.

Elliott Sim, S. and Koschke, R. 2001. WoSEF: Workshop on standard exchange format. *ACM SIGSOFT Software Engineering Notes 26,* 1 (January), 44–49.

Eng, D. 2002. Combining static and dynamic data in code visualization. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*. Charleston, SC, USA, 43–50.

Ernst, J. 1997. Introduction to cdif. http://www.eigroup.org/-cdif/intro.html.

eXpat Project. 2005. The Expat XML Parser version 1.95.8. http://expat.sourceforge.net.

Feijs, L. M. G. and van Ommering, R. C. 1999. Relation partition algebra – mathematical aspects of uses and part-of relations. *Science of Computer Programming 33,* 2 (February), 163–212.

Fenton, N. E. and Pfleeger, S. L. 1998. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA.

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

Ferenc, R., Elliott Sim, S., Holt, R. C., Koschke, R., and Gyimothy, T. 2001. Towards a standard schema for C/C++. In *Proceedings of the Eighth Working Conference on Reverse Engineering*. Stuttgart, Germany, 49–58.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9,* 3 (July), 319–349.

Ferrett, L. K. and Offut, J. 2002. An empirical comparison of modularity of procedural and object-oriented software. In *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, Greenbelt, MD, USA, 173–182.

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

FluxBox Project. 2006. FluxBox version 0.9.14. http://www.fluxbox.org.

Fowler, M. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third ed. Addison-Wesley.

Free Software Foundation. 2002. GNU Compiler Collection. http://www.gnu.org/software/gcc/.

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

Glück, R. and Lowry, M. R., Eds. 2005. *Generative Programming and Component Engineering*. Lecture Notes in Computer Science, vol. 3676. Tallinn, Estonia.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. 1982. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices 17,* 6 (June), 120–126.

Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. 2001. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology 10,* 2 (April), 184–208.

Group, O. M. 2005. MOF 2.0/XMI mapping specification, v2.1. Tech. rep. September.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices 32,* 10 (October), 108–124.

Gschwind, T., Pinzger, M., and Gall, H. 2004. TUAnalyzer - analyzing templates in C++ code. In *Proceedings of the Eleventh Working Conference on Reverse Engineering*. IEEE Computer Society, Delft, The Netherlands, 48–57.

Guo, X., Cordy, J. R., and Dean, T. R. 2003. Unique renaming of java using source transformation. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, Amsterdam, The Netherlands, 151–160.

Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., and Mutzel, P. 2003. A new approach for visualizing uml class diagrams. In *Proceedings of the 2003 ACM Symposium on Software Visualization*. San Diego, CA, USA, 179–188.

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

Harrold, M. J., Rosenblum, D., Rothermel, G., and Weyuker, E. 2001. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering 27,* 3 (March), 248–263.

Hennessy, M., Malloy, B. A., and Power, J. F. 2003. gccXfront: Exploiting gcc as a front end for program comprehension tools via XML/XSLT. In *Proceedings of the Eleventh International Workshop on Program Comprehension*. IEEE Computer Society, Portland, OR, USA, 298–299.

Hodgson, N. 2006. Scintilla version 1.66. http://www.scintilla.org.

Hoipkemier, B. N., Kraft, N. A., and Malloy, B. A. 2006. 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.

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

Holt, R., Schürr, A., Elliott Sim, S., and Winter, A. 2003. GXL - Graph eXchange Language. http://www.gupro.de/GXL.

Holt, R., Schürr, A., Elliott Sim, S., and Winter, A. 2006. GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming 60,* 4, 149–170.

Holt, R. C. 1997. An introduction to TA: The tuple-attribute language. http://www.swag.uwaterloo.ca/pbs/papers/ta.html.

IBM Jikes Project. 2006. Jikes version 1.22. http://jikes.sourceforge.net.

ISO/IEC JTC 1. 1998. *International Standard: Programming Languages – C++*, First ed. Number 14882:1998(E) in ASC X3. ANSI.

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

Jazayeri, M., Loos, R., and Musser, D., Eds. 2000. *Generic Programming*. Lecture Notes in Computer Science, vol. 1766. Springer-Verlag, Heidelberg, Germany.

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

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

Jones, J. A. and Harrold, M. J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Long Beach, CA, USA, 273–282.

Jones, J. A., Orso, A., and Harrold, M. J. 2004. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization 3,* 3, 173–188.

Kaczmarek, A. 2003. Gxl validator, validierung von gxl-dokumenten auf instanz-, schema, und metaschema-ebene. Studienarbeit, Universität Koblenz, Fachbereich Informatik, Koblenz, Germany.

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

Keystone Project. 2005. Keystone version 0.2.3. http://keystone.sourceforge.net.

Kitware, Inc. 2005. GCC-XML. http://www.gccxml.org.

Klint, P., Lämmel, R., and Verhoef, C. 2005. Towards an engineering discipline for grammarware. Draft, Submitted for publication; Online since July 2003, 47 pages.

Knapen, G., Lague, B., Dagenais, M., and Merlo, E. 1999. Parsing C++ despite missing declarations. In *Proceedings of the Seventh International Workshop on Program Comprehension*. Pittsburgh, PA, USA.

Koppler, R. 1997. A systematic approach to fuzzy parsing. *Software – Practice and Experience 27,* 6 (June), 637–649.

Kraft, N. A. 2006. Repository of reverse engineering artifacts. http://g4re.sourceforge.net.

Kraft, N. A., Lloyd, E. L., Malloy, B. A., and Clarke, P. J. 2006. The implementation of an extensible system for comparison and visualization of class ordering methodologies. *Journal of Systems and Software 79,* 8, 1092–1109.

Kraft, N. A., Malloy, B. A., and Power, J. F. 2005a. g^4re: Harnessing gcc to reverse engineer C++ applications. In *Transformation Techniques in Software Engineering*. Number 05161 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany.

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

Kraft, N. A., Malloy, B. A., and Power, J. F. 2007a. An infrastructure to support interoperability in reverse engineering. *Informating and Software Technology 49,* 3, 292–307.

Kraft, N. A., Malloy, B. A., and Power, J. F. 2007b. A tool chain for reverse engineering C++ applications. *Science of Computer Programming, Special Issue on Experimental Software and Toolkits*. (accepted for publication).

Kullbach, B., Winter, A., Dahm, P., and Ebert, J. 1998. Program comprehension in multi-language systems. In *Proceedings of the Fifth Working Conference on Reverse Engineering*. IEEE Computer Society, Honolulu, HI, USA, 135–143.

Kung, D., Gao, J., and Hsia, P. 1995. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming 8,* 2 (May), 51–65.

Kunz, P. F. 2006. HippoDraw 1.15.8. http://www.slac.stanford.edu/grp/ek/hippodraw/.

Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., and Durand, M.-H. 2000. Testing levels for object-oriented software. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, Limerick, Ireland, 136–145.

Lapierre, S., Lague, B., and Leduc, C. 2001. Datrix source code model and its interchange format: Lessons learned and considerations for future work. *ACM SIGSOFT Software Engineering Notes 26,* 1 (January), 53–56.

Lengauer, C., Batory, D., Consel, C., and Odersky, M., Eds. 2004. *Domain-Specific Program Generation*. Number 3016 in Lecture Notes in Computer Science. Springer-Verlag.

Lethbridge, T. C. 2001. Report from the dagstuhl seminar on interoperability of reengineering tools. In *Proceedings of the Ninth International Workshop on Program Comprehension*. Toronto, ON, Canada, 119.

Lethbridge, T. C. 2003. The dagstuhl middle model: An overview. Presented at the First International Workshop on Metamodels and Schemas for Reverse Engineering.

Lethbridge, T. C., Tichelaar, S., and Plödereder, E. 2004. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science 94*, 7–18.

Lewerentz, C. and Simon, F. 2002. Metrics-based 3d visualization of large object-oriented programs. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis*. Paris, France, 70–77.

Licq Project. 2006. Licq version 0.2.3. http://www.licq.orq.

Lilley, J. 1997. PCCTS-based LL(1) C++ parser: Design and theory of operation. Version 1.5.

Lin, Y., Holt, R. C., and Malton, A. 2003. Completeness of a fact extractor. In *Proceedings of the Tenth Working Conference on Reverse Engineering*. IEEE Computer Society, Victoria, BC, Canada, 196–205.

MacCormack, A., Rusnak, J., and Baldwin, C. 2004. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Working Paper 05-016, Harvard Business School, Boston, MA, USA.

Malloy, B. A., Clarke, P. J., and Lloyd, E. L. 2003. A parameterized cost model to order classes for integration testing of c++ applications. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*. Denver, CO, USA, 353–364.

Malloy, B. A., Gibbs, T. H., and Power, J. F. 2003a. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience 33,* 1, 19–39.

Malloy, B. A., Gibbs, T. H., and Power, J. F. 2003b. Progression toward conformance for C++ language compilers. *Dr. Dobbs Journal*, 54–60.

Malloy, B. A. and Power, J. F. 2002. Program annotation in XML: A parser-based approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering*. IEEE Computer Society, Richmond, VA, USA, 190–198.

Malloy, B. A. and Power, J. F. 2005. Using a molecular metaphor to facilitate comprehension of 3d object diagrams. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Dallas, TX, USA, 233–240.

Mamas, E. and Kontogiannis, K. 2000. Towards portable source code representations using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering*. IEEE Computer Society, Washington, DC, USA, 172.

Marcus, A., Feng, L., and Maletic, J. I. 2003. 3d representations for software visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization*. San Diego, CA, USA, 27–36, 207–208.

McPeak, S. 2005. Elkhound: A fast, practical GLR parser generator. Tech. Rep. UCB/CSD-2-1214, University of California, Berkeley. April.

McQuillan, J. A. and Power, J. F. 2006. Experiences of using the dagstuhl middle metamodel for defining software metrics. In *Proceedings of the Fourth International Conference on Principles and Practices of Programming in Java*. Mannheim, Germany, 194–198.

Merrill, J. 2003. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*. Ottawa, Canada, 171–180.

Microsoft Corporation. 2006. C# language specification. http://msdn.microsoft.com/netframework/ecma/.

Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. 2000. Reverse engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. Limerick, Ireland.

Münch, M. 1999. Programmed graph rewriting system progres. *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance 1779*, 441–448.

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

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

National Institute of Standards and Technology. 1993. ECMA: Reference model for frameworks of software engineering environments. Technical Report.

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

Object Management Group. 2005. *UML Superstructure Specification, v2.0*.

Orso, A., Apiwattanapong, T., Law, J. B., Rothermel, G., and Harrold, M. 2004. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, Scotland, 491–500.

Parberry, I. 2000. *Learn Computer Game Programming with DirectX 7.0*. Woodward Publishing Co., Plano, TX, USA.

Parberry, I., Roden, T., and Kazemzadeh, M. B. 2005. Experience with an industry-driven capstone course on game programming: Extended abstract. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. St. Louis, MO, USA, 91–95.

Paulson, J. W., Succi, G., and Eberlein, A. 2004. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering 30,* 4, 246–256.

Pazera, E. 2003. *Focus on SDL.* Premier Press, Cincinnati, OH, USA.

Power, J. F. and Malloy, B. A. 2000. Symbol table construction and name lookup in ISO C++. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems.* Sydney, Australia, 57–68.

Rebellion. 2005. Aliens vs Predator version CVS 07/22/05. http://www.icculus.org/avp.

Reiss, S. and Davis, T. 1995. Experiences writing object-oriented compiler front ends. Tech. rep., Brown University. January.

Roskind, J. 1989. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL.

Rothermel, G. and Harrold, M. J. 1998. Empirical studies of a safe test selection technique. *IEEE Transactions on Software Engineering 24,* 6 (June), 401–419.

Salah, M. and Mancoridis, S. 2003. Toward an environment for comprehending distributed systems. In *Proceedings of the Tenth Working Conference on Reverse Engineering.* IEEE Computer Society, Victoria, BC, Canada, 238–247.

Scribus Project. 2006. Scribus version 1.2.3. http://www.scribus.net.

Siek, J. G. and Lumsdaine, A. 2005. Essential language support for generic programming. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation.* Chicago, IL, USA, 73–84.

Simple DirectMedia Layer Project. 2005. Simple DirectMedia Layer. http://www.libsdl.org.

Sinha, S., Harrold, M. J., and Rothermel, G. 1999. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering.* Los Angeles, CA, USA.

Skoglund, M. and Runeson, P. 2005. 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.

Source–Navigator Team. 2005. The Source–Navigator IDE. http://sourcenav.sourceforge.net.

Steward, D. V. 1981. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management 28,* 3, 71–84.

Sun Microsystems Inc. 2006. Java language specification. Version 5.

van der Zijp, J. 2006. The FOX Toolkit Library version 1.4.17. http://www.fox-toolkit.org.

van Heesch, D. 2006. Doxygen version 1.4.4. http://stack.nl/ dimitri/doxygen/.

Vandevoorde, D. and Josuttis, N. M. 2002. *C++ Templates: The Complete Guide*. Addison Wesley.

Veldhuizen, T. L. 2000. Five compilation models for C++ templates. In *Proceedings of the First Workshop on C++ Template Programming*. Erfurt, Germany. http://oonumerics.org/tmpw00/.

Veldhuizen, T. L. 2003. C++ templates are turing complete. Tech. rep., Indiana University.

Veldhuizen, T. L. and Gannon, D. 1998. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM Press, Yorktown Heights, NY, USA.

Vinciguerra, L., Wills, L., Kejriwal, N., Martino, P., and Vinciguerra, R. 2003. An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java. In *Proceedings of the Tenth Working Conference on Reverse Engineering*. IEEE Computer Society, Victoria, BC, Canada, 14–23.

Wong, K. 1998. Rigi user's manual – version 5.4.4. http://www.rigi.cs.uvic.ca/downloads/rigi/doc/user.html.

Wu, J. and Holt, R. C. 2004. Resolving linkage anomalies in extracted software system models. In *Proceedings of the Twelfth IEEE International Workshop on Program Comprehension*. Bari, Italy, 241–245.

xsltproc Project. 2005. xsltproc version 1.1. http://xmlsoft.org/XSLT/xsltproc2.html.

zlib Project. 2005. zlib version 1.2.3. http://www.zlib.net.