8-2007

A Comparison of Class Diagram Construction at Three Different Phases of Compilation

Robert Bailey iii
Clemson University, rrbaile@cs.clemson.edu

A COMPARISON OF CLASS DIAGRAM CONSTRUCTION AT THREE DIFFERENT PHASES OF COMPILATION

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Robert R Bailey III
August 2007

Accepted by:
Dr. Brian A. Malloy, Committee Chair
Dr. Jason O. Hallstrom
Dr. Mark Smotherman

ABSTRACT

In maintaining, designing, and testing of OO systems, understanding the relationships between the classes and corresponding objects is imperative. Class diagrams provide program visualization that promotes easier understanding of large systems. Class diagrams are especially helpful when systems become too large to conceptualize without automated graphical illustration. There are many tools available to software developers that permit generation of class diagrams for large scale computing applications, many of which use various parsing approaches to extract information for a class diagram. In this paper, we compare and evaluate three tools that extract information for class diagrams from different phases of the compilation process: a fuzzy parse of the program, a parse tree representation of the program, and an abstract syntax graph to represent the program. Using three tools we generate class diagrams from a test suite of OO systems, and compute metrics on these diagrams as a means of comparing them.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

Chapter 1

Introduction

The construction of object-oriented systems (OO) has become the norm for system development and the OO approach has replaced the procedural approach as the standard for software development. An important aspect of object-oriented systems is the high degree of interaction among the objects. In the procedural approach, the items of interest are statements and functions; graphs, such as control flow graphs and call graphs, have been developed to facilitate statement and function level analysis [1]. However, in the OO approach the items of interest are classes and objects; graphical program representations have been developed as part of the Unified Modeling Language, including class and object diagrams, to facilitate class and object level analysis [7, 5].

In maintaining, designing, and testing of OO systems, understanding the relationships between the classes and corresponding objects is imperative. Class diagrams provide program visualization that promotes easier understanding of large systems. Class diagrams consist of nodes and edges. The nodes of the diagram represent classes in the system. The edges that connect the nodes represent interactions among the classes in the system. These diagrams are helpful because they aid in visualization, maintenance, and testing of a system. Class diagrams are especially helpful when systems become too large to conceptualize without the assistance of automated graphical illustration.

There are many tools available to software developers that permit generation of class diagrams for large scale computing applications [23, 22, 8, 13]. These tools use various parsing approaches to extract information at different stages of compilation to generate the class diagrams. One problem with the development of tools for C++ applications is that the scope and inherent ambiguity of the grammar make parsing of the C++ language a daunting task and this difficulty is well documented in the literature [4, 9, 10, 14, 15, 17, 19, 20, 21].

Another problem with the development of tools for class diagram construction is that the UML specification is general enough to accommodate all programming languages but is sometimes lacking in specification detail to permit distinction between some of the artifacts for various graphical representations [7]. For example, the distinction between the edges that represent the composition and aggregation relationships in class diagrams for C++ has been difficult to interpret and has led to much ambiguity in the literature.

A final problem with the development of tools for class diagrams is that few studies have been conducted that compare the class diagram generation tools using a standard, objective set of criteria or metrics, most likely because a standard criteria that can accommodate all languages is difficult to develop, especially for the C++ language. One comparison study evaluated several class diagram construction tools for C++, but failed to develop a methodology for standardizing the comparison of the various diagrams [16].

In this paper, we compare and evaluate three tools that extract information for class diagrams from different phases of the compilation process: one tool uses a fuzzy parse of the program, a second tool uses an abstract syntax tree representation of the program, and the third tool uses an abstract syntax graph to represent the program. We compare class diagrams obtained from these tools using a test suite of applications and compare the generated class diagrams using a common schema to specify the nodes and edges in the graph. Our study will use a textual representation of class diagrams to encode them into a graphical exchange language (GXL) file. Using a single schema for all three tools facilitates comparison of the class diagrams constructed using a test suite of medium-sized applications.

In the next section, we review related work in the field of class diagram generation and measuring class diagrams with a given set of metrics. In Section 3, we define the key terms and concepts used in our study. In Section 4 we describe our methodology for generating class diagrams from different tools that all conform to the same schema. This section also describes the manner in which metrics are computed for these graphs. In the final section, Section 5, we describe our case study involving a test-suite of object-oriented applications that we use as input to the three tools for computation of class diagrams and metrics. This section will also draw conclusions from the case study.

Chapter 2

Related Work

Matzko et al. investigated class diagram construction through a tool called Reveal [16]. Reveal used Keystone as a front-end to parse the input and to reverse-engineer a class diagram for C++ applications [15]. The tool included some UML extensions to accomodate language constructs peculiar to C++ and utilized a full parse of the input to provide a more precise class diagram. The Reveal tool used *graphviz* to provide visualization of class diagrams to accomodate maintenance tasks for applications that are not bundled with design artifacts [2].

The research also involved a comparison of the class diagrams produced by Reveal with those produced by three other tools: *Rational Rose*, *Together* and *SuperWomble* [8, 22, 23]. However, the research did not investigate or compare the class diagrams produced at different phases of the compilation process [16].

Tong Yi et al. present a comparison of UML class diagram metrics [25]. The authors compare the usefulness of several metrics used to measure class diagrams. The metrics compared are Marchesi's Metric, Genero's Metric, In's Metric, Rufai's Metric, Kang's Metric, and Zhou's Metric. These various metrics rate a UML class diagram based on ratios of number of classes, association, composition, and inheritance edges, along with several other metrics. The authors use hand written class diagrams of small scale systems to test the various metrics under study. The conclusion of the work suggests that each metric provides insight for various system qualities.

Eichelberger presents discussion of metrics and aesthetics in class diagram generation in [6]. The author is mainly concerned with the improvement of aesthetics in class diagram generation. However, he does describe the necessity of using certain "design criteria" for aesthetic diagrams [6]. Class diagrams are measured on such qualities as number of inheritances, number of children, and complexity of associations. The work proposes changes to the UML specification for class diagrams, which would enhance the presentation of class

diagrams. This applies to our study, because the author describes the set of important metrics in the study of class diagrams.

Chapter 3

Background

In this section we review background information and definitions of concepts and constructs used in our paper. In the next section we review *class diagrams* including a discussion of the kinds of edges that describe the relationships between classes in these diagrams. In Section 3.2 we describe and distinguish parse trees and abstract semantic graphs (ASG), and in Section 3.3 we discuss *fuzzy parsing*. We conclude this section with a discussion of *schemas* and their use in validation of the syntax and semantics of a program.

3.1 UML Class Diagrams

Because of the complexity inherent in large programs, a developer, or group of developers, may not be able to grasp the overall design of the system, even if those involved are familiar with the source code. The Unified Modeling Language (UML) includes graph structures to represent aspects of the program, providing an overview of the programs structure and behavior[5]. The UML can help even non-programmers get a better grasp of the overall functionality of the system.

A UML class diagram is a static representation of the program consisting of rectangles to represent classes in the system and lines connecting the rectangles to represent the relationships between the classes. In UML, a class is represented as a box with three vertical sections. The top section shows the name of the class. The middle section displays the variables belonging to the class, with symbols representing the visibility (public, protected, or private) and properties (constant or static). The bottom section contains the member functions of the class. Each method has a name, signature, and properties.

There are four types of relationships in a class diagram: association, dependency, generalization and realization. Each relationship is represented in the diagram by a different type of arrow.

An *association* is a structural relationship that describes a set of links, where a link is a connection among objects. An association can be a one-way or two-way relationship. Class A has an association with class B if class A has a data member of type B. If class B also has a data member of type A, then the relationship is bi-directional. A directed arrow to the associated class specifies a one-way relationship. A line (no arrowhead) signifies a bi-directional relationship. Associations are often adorned with numbers or symbols representing the *multiplicity* of the relationship, or how many objects of one class the other is using.

Dependency is a semantic relationship between two classes in which a change to one class (the independent class) may affect the semantics of the other class (the dependent class). A method is said to use an object of a class if the object is passed in as a parameter, created in the method, or returned from a method. In UML, dependencies are designated by a dashed arrow from the dependent class to the class on which it depends. If both classes depend on the other, a double-headed dashed arrow is used.

Dependency may also be used to show a special type of relationship between classes: friendship. If class A is a friend of class B, class A can access class B's private information. Friend classes are denoted by a dependency arrow with the stereotype <<friend>> on the arrow.

Generalization, or inheritance, is a specialization/generalization relationship where objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). *Realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. In C++, an abstract base class, parent, has one or more purely virtual methods that every immediate child must overwrite. The contract between the parent and the child is that the child will overwrite these methods. Abstract classes are written in italics. Children of abstract classes are connected to the class by a dashed arrow with a hollow arrowhead.

3.2 Graphical Representations of Code: Parse Tree & ASG

Given an input string, a parser derives a *parse tree* for the string (i.e., it parses the string). As the parser progresses through the input program, it builds a tree of the productions that are invoked. 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.

3.3 Fuzzy Parsing

A *fuzzy parser* is the term given to a tool that parses a subset of a given language. Fuzzy parsers were first described by Sniff; Koppler first attempted to formalize a definition of the tool in [11]. Fuzzy parsers are used to parse a subset of a language to be used in software tools needing only selected parts of an input. These tools are useful because they alleviate the need for programmers to implement full parsers that perform a set of small tasks. Fuzzy parsers are language specific tools that perform only the necessary actions on necessary parts of an input source. A major disadvantage of fuzzy parsers is their limited reuseability. These tools are implemented to fulfill specific tasks, and adding extra capabilities would push the parser closer to a full implementation of the language parser. The other disadvantage of fuzzy parsers is that they ignore semantic information; thus, any context sensitive information is ignored by the tools created [11].

Fuzzy parser construction requires that the developer partition the set of input symbols into those that will be recognized by the fuzzy parser and those that will not be recognized. Additionally, the operations that the language will be permitted to perform on the chosen set of input symbols must also be similarly partitioned. The input symbols that the fuzzy parser requires are a subset of the full input language and are known as *anchors* of the

parser. The fuzzy parser performs its functions in the same manner that a normal parser works: the lexical analyzer searches an input file for the given anchors. When the analyzer encounters an anchor, it calls the parser to parse the input until the scanner encounters the last token for the desired input string [11].

3.4 Schemas

A schema is a UML class diagram encoded in XML. XML provides a common base from which any schema for representing software can be derived. A schema constrains the ways that a class diagram can be represented. An XML validator is used to validate an instance of a schema graph. Validating XML and the corresponding graph is important: validation can reveal errors in both the modeling, and the generation of XML instances. In addition, valid XML files are more likely to be accepted by available XML tools than non-valid files.

Chapter 4

Methodology

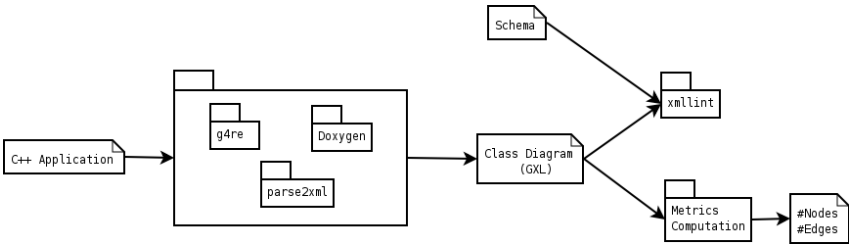

Figure 4.1: Overview of the Comparison System.

The goal of our work is to compare class diagrams generated at different levels, or phases, of the compilation process. To make an accurate comparison of the generated class diagrams, all graphs should conform to the same schema so that nodes and edges in each graph will have the same semantics. Using the schema conforming class diagrams, we can compute the number of nodes and edges in class diagrams generated for the same application at different phases of the compilation process.

The basic flow of our system is illustrated in Figure 4.1, where the input to our system, a C++ Application, is shown on the left side of the figure. The C++ Application is then processed by one of the three tools, listed in the tabbed box in the middle of Figure 4.1; the three tools used in our comparison system are *g*4*re*, Doxygen, and parse2xml [3, 12, 13, 18, 24]. The *g*4*re* tool uses *gcc* to construct an ASG representation of the C++ Application and we then traverse the ASG to build a class diagram. Doxygen is a fuzzy parser that produces a class diagram as one form of output. Finally, parse2xml is a tool that generates a parse tree in XML format as part of the compilation process. We then use a SAX parser to traverse the parse tree and build a class diagram.

Each of the construction tools that we use produces a class diagram in GXL format and the GXL will be syntactically and semantically validated against the GXL schema for a class diagram using *xmllint*; this process is shown in the upper right corner of Figure 4.1.

Finally, the GXL representation of the class diagram is used as input to a tool that computes some metrics for each class diagram. These metrics include the count of the number of nodes and edges in the C++ Application. The edge count is further partitioned into the number of inheritance, association, and composition edges.

4.1 Construction of a Parse Tree

To build a class diagram from the parse tree representation of an application, we traverse the tree and process the token nodes in the tree. The parse tree uses an XML representation of the C++ application. This methodology uses an parse tree, which is generated before linking, so that some type information generated in normal compilation will not be included in the parse tree output.

The tool, *parse2xml* takes as input the source code of an application, and its function is to produce an XML representation of the parse tree for the application. The XML is generated during the parsing phase of compilation. To do this, the file, parser.c, in GNUs C++ compiler is modified to output each production in the parse. This step results in a file containing many of the grammar productions, however there is some recursive descent information from the parse included in the XML. This information is present because some statements in an application generate a tentative parse using certain subtrees, which may be used or rolled back. These statements describing tentative parsing and rollbacks are removed using the tool postprocess. The postprocess tool produces the final version of the parse tree, with only the necessary information in the tree.

To use the *parse2xml* tool on our test suite, we made changes to the manner in which the files are compiled. When building a large test case such as fluxbox, compiling with our modified parser resulted in an explosion of XML productions. This was a result of all included files, such as standard library files, being parsed each time they are encountered in a system. To prevent the large size of XML output, we compile all header files at the same time, with a main file that includes all header information. We assert that this will result in the same number of classes, inheritance edges, composition edges, and association edges. If our work computed the number of dependence edges, the results may not be considered

valid. By compiling all header files at once, we can obtain an XML output file of reasonable size.

After the *parse2xml* tool has produced an XML representation of the parse tree, we break the overall XML file into separate files for each class declaration. We use a SAX parser written in perl to recognize the desired tree nodes. This is done so that our DOM parser, which produces a class diagram, can be run quicker and more efficiently. Our tool *parse2xml.pl* traverses each class declaration XML file and finds all the necessary information for class diagrams. The first step in class diagram construction using the tool is to recognize class declarations. These class declarations can be found as part of the 'class specifier' productions. From this point, parent classes can be found using the 'base clause' productions in the class specifier's subtree. Finally, association and composition relationships are found using the 'member declaration' subtrees of a class. Only class attributes that are of user defined types are used to produce edges in the generated class diagram. Association edges are found when the member variable's declarator subtree contains a *ptroperator* node. Composition edges are found when the member variable's declarator subtree does not contain a *ptroperator* node. All class declaration nodes and edge nodes are stored as nodes in a intermediate XML document. The final step in our perl program for class diagram construction is to parse the document and convert it to the GXL schema.

4.2 Using Doxygen to Build a Class Diagram

Doxygen uses a fuzzy parse of an application's source code to create a class diagram for the input application. Using the doxygen configuration file, we direct doxygen to generate an inheritance graph for each class in the system in XML format. The inheritance graph maps each child class to its parent in the application under study. Our responsibility in this project is to convert the doxygen output format so that it conforms to our GXL schema for class diagrams. We use doxygen version 1.5.1 in our study. Doxygen generates an inheritance graph for each class in an application; our goal is to combine all of these diagrams, interpret composition and association edges, and present them in the schema conforming XML format. The first step in solving this problem is to run doxygen on all header and

source files in a program, using the doxygen configuration file. When this step is complete, all classes in the system will have a corresponding XML file, containing information about its members, as well as the inheritance graph that corresponds to the class.

We use our perl tool[1] to create a class diagram in our GXL schema using two steps. The first step in this process is to traverse each class's XML file and map class names to the reference numbers in the XML file. This step creates an intermediate XML document in memory that stores all classes, and edges between classes. All inheritance edges are found using the inheritance graphs from the doxygen-generated XML files. Composition and Association edges are found in a class's *memberdef* tag. The second step in our perl tool construction is to create a GXL file that conforms to our schema. The most intricate step in this process is to ensure that edges are represented only once in the GXL file. Redundant edges exist due to the replication of the inheritance graphs for each doxygen-generated XML file.

4.3 Using *g4re* to Build an ASG

The general structure of creating a class diagram using an ASG is done by utilizing the *g4re* tool chain. This tool uses an application programmer interface (API) for the ASG created by *g4re* when an application is compiled using g++. From this API, the tool extracts class declarations, inheritances, associations, and compositions, and presents them in either GXL or Dot format. Since the GXL is a native output format for g4re, there is no conversion step for the graph created by this tool. Since it makes use of the ASG created by the GNU compiler, type information that is not present in the other two methods of extracting class diagrams is available using the *g4re* tool.

The *g4re* tool creates a class diagram using the following method. A C++ application's source code is run through the *gcc* parser and front end, producing an ASG from gcc's GENERIC schema for its internal ASG. This ASG is then represented by the CppInfo API, which is an abstraction of the ASG that an application can access to obtain information about the original application. The *g4re* tool creates a class diagram by obtaining a list of

[1]The perl tool is dox2gxl.pl

all user classes in a system from the API. For each class obtained, all base classes, member functions, and member variables are obtained from the API.

4.4 Metric Computation using GXL Representation

For each class diagram created by the above methods, we compute a set of metrics. This serves as a means for comparing the effectiveness of each method in finding the maximum number of edges for each application. The metrics that we calculate are: number of association edges, number of composition edges, and number of inheritance edges. These metrics are calculated simply by traversing the GXL representation of a class diagram, and counting the number of each type of node. Our tool gxlMetrics.pl does this traversal. The tool first searches for all nodes in the GXL. If the node is a class node, the class count is increased. Edge nodes are counted by finding all nodes that are either composition, association, or inheritance nodes according to the GXL schema.

Chapter 5

Case Study

In this section we describe the results of a comparison of the three tools used to build class diagrams. All of our test cases were run on Intel Pentium 4 2.0 GHz processor, using a 5400 RPM hard drive formatted with Extended 3 file system, and running Ubuntu version 6.06 operating system. The version of gcc used for compiling test programs for g4re is version 4.0.3. The version of gcc used for the parse2xml tool is gcc version 4.0.0. We use perl version 5.8.7 to run our scripts. We use sclc to compute the number of lines of code in each system.

Section 5.1 will describe the various test cases used in our study. Section 5.2 will breakdown the amount of class nodes found for each test case for each methodology. Section 5.3 will investigate the number of edges found for all test cases in each methodology.

5.1 Test Suite

We use three test cases in our study. Fluxbox is a window manager system. We test fluxbox version 1.13. Jikes is an open source JAVA to byte-code processor, in our test we use jikes version 1.22. Doxygen is a fuzzy parser used to aid the documentation of source code, we use doxygen version 1.5.2 in our test. Our final test case is Hippodraw. Hippodraw is a data analysis system, that provides graphical analysis tools for C++ programs; we use version 1.15.8 in our tests. Table 1 depicts the various test cases used along with the number of uncommented lines of code that are present in the system. This metric is important because it only accounts for the number of source code lines that are present in the system.

5.2 Number of Class Nodes

Each tool under study generates a gxl encoding of the class diagram of a system. These graphs all adhere to the same schema, and can thus be easily compared using the same

Test Case	Doxygen	FluxBox	Jikes	Hippodraw
Version	1.5.1	0.1.13	1.22	1.15.82
NCLOC (K)	161	20	70	55
Number header files	105	42	33	243

Table 5.1: Testsuite. *This table lists the four test cases that we use in our study, together with version number and statistics about the test cases.*

Test Case	Doxygen	FluxBox	Jikes	Hippodraw
Number of Classes	238	95	284	21
Number of Edges	457	249	967	26
Inheritance Edges	188	47	170	10
Composition Edges	23	91	51	6
Association Edges	246	111	746	10

Table 5.2: Results for the Doxygen Test Case. *This table lists the four test cases and the metrics computed on Doxygen's class diagram output.*

metrics.

5.2.1 Doxygen Class Nodes

Doxygen extracts class nodes from input files and generates an XML file for each one. Our tool interprets these files and builds a class diagram. Doxygen does not generate files for templated classes or for forward declarations of classes such as those declaring classes from standard library files. Therefore, large systems such as the source for hippodraw are not properly represented. Doxygen finds 238 class nodes for its own source code. The fluxbox test case results in ninety-five class nodes. This may seem like a large number given the relatively small number of source files in the system. However, since fluxbox is a window manager, it should make significant use of standard library and system header files to achieve its goals. Doxygen documented 284 classes for the jikes system. This test case contains the fewest amount of header files of all under test. But, the system has a higher number of lines of code than fluxbox, which had a comparable number of header files. Finally, doxygen locates twenty-one classes for the hippodraw test case. The system's source folder contains nearly three hundred header files, though. This suggests that hippodraw has several templated classes that are not included in doxygen's search.

Test Case	Doxygen	FluxBox	Jikes	Hippodraw
Number of Classes	358	1124	404	253
Number of Edges	635	1830	1805	382
Inheritance Edges	17	92	50	180
Composition Edges	511	1702	1733	29
Association Edges	107	36	22	173

Table 5.3: Results for the g4re Test Case. *This table lists the four test cases and the metrics computed on g4re's class diagram output.*

Test Case	Doxygen	FluxBox	Jikes	Hippodraw
Number of Classes	304	369	432	584
Number of Edges	1265	714	1236	3451
Inheritance Edges	213	138	258	419
Composition Edges	444	259	810	1925
Association Edges	608	317	168	1107

Table 5.4: Results for the parse2xml Test Case. *This table lists the four test cases and the metrics computed on parse2xml's class diagram output.*

5.2.2 g4re Class Nodes

The g4re system presents class diagrams that result from traversing the ASG of a system. This means that g4re class diagrams should contain more class nodes than the other class diagrams in our study. This is because the ASG will contain information about the types used in instantiating templated classes. With this expectation, the g4re class diagrams for the systems under test yield appropriate results. The doxygen source code yielded 562 classes. The fluxbox test case results in over eleven hundred class nodes. This implies that the system heavily relies on templates, that are not recognized by the doxygen tool. The jikes test case has few more class nodes using g4re, yielding a class diagram with 404 class nodes. The g4re tool found 253 class nodes with the Hippodraw test suite.

5.2.3 Parse2xml Class Nodes

The parse2gxl tool produces class diagrams based on traversing the parse tree of a system. Because of this, the tool should present more classes than a source trace, as is done in the doxygen tool, but less than diagrams with semantic information, as is generated with g4re. This is certainly true with the fluxbox test case. Our tool recognizes 369 classes in the fluxbox system, which shows that some standard classes, as well as in-lined classes are recognized with our tool. The jikes system is shown to have 432 class nodes, a number

higher than the doxygen tool's results. However, this is higher than the number found by g4re. Hippodraw is shown to have 548 classes, which is higher than the number of classes output by the g4re tool. The explanation for this inconsistency comes from the high number of forward declarations of classes in the Hippodraw source. The g4re tool chain version that we used does not handle these language constructs well. Finally, the doxygen test case contained 304 classes.

5.3 Number of Edges

For the same reasons that the number of class nodes increased from tool to tool in our study, we expect to observe increasing counts of edge nodes. Not only will the added class nodes have their own member variables and inheritances that will add to the count, some classes will contain template class edges. These were not counted by the tools that ignore semantic information.

5.3.1 Doxygen Edge Nodes

Our doxygen tool converts doxygen's class diagrams into the GXL schema that we use. However, several problems arise when templates are introduced in a test case. As stated earlier, doxygen only generates XML files for declared classes, not templated classes. Therefore, when our tool encounters a class with an edge to a template class, there is no mapping in our system. Therefore, all edges to template classes are ignored by our system, and give alerts when found. The doxygen source contains several templated classes, and thus creates several errors with the system. The number of all edges would increase for this system, if the doxygen tool could extract all templated classes. Other systems do not generate the alerts, and are assumed to be based on normal class declarations.

5.3.2 g4re Edge Nodes

The anticipated results of metrics computation for g4re class diagrams would be that more edges will be present. Due to the fact that g4re incorporates semantic information into the formation of its graphs, instantiated template classes, and member variables will increase

the total number of edges. For the jikes and fluxbox test cases, this idea holds. Both systems are shown to have over 1800 edges. The large majority of these edges come from composition edges, which implies that each class contains member variables that template classes. The doxygen test case had 1317 edges in its g4re class diagrams. Finally, the g4re tool generated a diagram with 382 edge nodes.

5.3.3 Parse2xml Edge Nodes

The parse2xml tool should return results that are between those of the doxygen and g4re tools. This is because the g4re tool uses semantic information, and the doxygen does not parse to the depth that parse2xml does. Once again, the fluxbox and jikes test cases present consistent results with this hypothesis. The doxygen test case follows the number of edges consistenty. The Hippodraw test case exhibits unusual results for the number of edges, this is due to the same explanation as the inconsistent number of classes mentioned above.

Chapter 6

Conclusions and Future Work

There is a large number of tools available to developers that can reverse engineer a class diagram for large scale applications. These tools use information extracted from different stages of compilation of the application to generate the class diagrams. Few studies have compared these class diagram generation tools using a common framework to define the semantics of nodes and edges and using a standard set of metrics. In this paper, we have tested three tools that each extract information for class diagrams from different approaches or phases of compilation: a fuzzy parser, an abstract syntax tree, and an abstract syntax graph. We compared the class diagrams obtained from these tools when run on a test suite of applications. These diagrams, where each diagram was validated against a common schema, were then compared. Our study indicates that each of the tools produces markedly different results. Moreover, our results support our conjecture that class diagrams constructed from a parse tree have more nodes and edges than class diagrams constructed using a fuzzy parser. Furthermore, class diagrams constructed from an abstract semantic graph (ASG) have more nodes and edges than class diagrams constructed from a parse tree.

The importance of our study is, firstly, to inform the software and reverse engineering communities that class diagram generation tools can produce markedly different diagrams, including different numbers of nodes and edges. Secondly, our study shows that the numbers of classes and edges indicated for the system under study can be larger, depending on the phase of compilation used to extract information about the application. And finally, the results of our study can be used to guide developers in choosing an appropriate class diagram generation tool. For example, for program comprehension and visualization, class diagrams generated using a fuzzy parse of the program may be more efficient than class diagrams generated using a full parse or semantic analysis of the application under study. However, for debugging purposes, class diagrams generated using an ASG may provide more accurate information to guide fault localization. Our ongoing work features the

design of an evaluation study to characterize the type of analysis needed to address the requirements at each phase of the software life cycle. The study will include the construction of an assessment metric to evaluate the usefulness and appropriateness of class diagram generation tools that use information gathered from different phases of compilation of the application under study.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, techniques, and tools*, Addison-Wesley, 1986.

[2] AT&T Labs, *Graphviz*, http://www.research.att.com/sw/tools/graphviz/, June 2005.

[3] B. A. Malloy and J. F. Power, *ast2xml: A reverse engineering infrastructure for c++*, Available at www.brianmalloy.com.

[4] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, *Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools*, The second annual object-oriented numerics conference (OON-SKI) (Sunriver, Oregon, USA), 1994, pp. 122–136.

[5] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide*, Object Technology Series, Addison-Wesley, 1999.

[6] Holger Eichelberger, *Nice class diagrams admit good design?*, SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization (New York, NY, USA), ACM Press, 2003, pp. 159–ff.

[7] M. Fowler, *Uml distilled*, third ed., Addison Wesley, 2004.

[8] D. Jackson and A. Waingold, *Lightweight construction of object models from bytecode*, IEEE Transactions on Software Engineering (2001).

[9] P. Klint, R. Lämmel, and C. Verhoef, *Towards an engineering discipline for grammarware*, Draft, Submitted for journal publication; Online since July 2003, 47 pages, February22 2005.

[10] G. Knapen, B. Lague, M. Dagenais, and E. Merlo, *Parsing C++ despite missing declarations*, 7th International Workshop on Program Comprehension (Pittsburgh, PA, USA), May 5-7 1999.

[11] Rainer Koppler, *A systematic approach to fuzzy parsing*, Softw. Pract. Exper. **27** (1997), no. 6, 637–649.

[12] N. A. Kraft, E. L. Lloyd, B. A. Malloy, and P. J. Clarke, *The implementation of an extensible system for comparison and visualization of class ordering methodologies*, Journal of Systems and Software **49** (2007).

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

[14] John Lilley, *PCCTS-based LL(1) C++ parser: Design and theory of operation*, Version 1.5, February 1997.

[15] B. A. Malloy, T. H. Gibbs, and J. F. Power, *Decorating tokens to facilitate recognition of ambiguous language constructs*, Software, Practice & Experience **33** (2003), no. 1, 19–39.

[16] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan, *Reveal: A tool to reverse engineer class diagrams*, Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, ACM, Feb 2002, pp. 13–21.

[17] S. McPeak, *Elkhound: A Fast, Practical GLR Parser Generator*, Tech. Report UCB/CSD-2-1214, Berkeley University, April2 2005, Technical Report.

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

[19] J. F. Power and B. A. Malloy, *Symbol table construction and name lookup in ISO C++*, 37th International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS Pacific 2000) (Sydney, Australia), November 2000, pp. 57–68.

[20] S.P. Reiss and T. Davis, *Experiences writing object-oriented compiler front ends*, Tech. report, Brown University, January 1995.

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

[22] Rational Software, *Rational rose*, Rational Corporation, 1999.

[23] TogetherSoft, *Together Control Center 5.5*, http://www.togethersoft.com/ (2001).

[24] D. van Heesch, *Doxygen version 1.4.4*, http://stack.nl/ dimitri/doxygen.

[25] Tong Yi, Fangjun Wu, and Chengzhi Gan, *A comparison of metrics for uml class diagrams*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 5, 1–6.