A Language and Platform-Independent Approach for Reverse Engineering

Edward B. Duffy and Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634, USA
email: {eduffy, malloy}@cs.clemson.edu

Abstract

In this paper, we present an approach for reverse engineering a program to capture design and implementation artifacts such as metrics and UML class or sequence diagrams. We also describe an implementation of our approach, libthorin*. However, unlike previous reverse engineering tools,* libthorin *can be applied to a variety of programming languages including C, C++, Java, Fortran 90 and C#. Moreover,* libthorin *can provide both coarse-grained and fine-grained information about the program under development to enable re-engineering of information and high-level diagrams such as metrics and class diagrams, or low-level diagrams such as sequence diagrams, control flow graphs and other program representations and analysis tools for testing, debugging and profiling an application under development.*

1 Introduction

The deliverable produced by a quality development process is excellent software that satisfies the evolving needs of its users. An approach to the production of such software requires the construction of models to visualize and control the systems evolving architecture, requirements, structure and behavior. One visualization tool that has rapidly become the language of choice for developers who wish to visualize and model the system under development is the Unified Modeling Language, UML. The UML includes *use cases* to facilitate visualization of user requirements, *class diagrams* to visualize the design of the software and *sequence diagrams* to visualize the behavior of the objects in the system. These diagrams provide high-level information about the system under development. More detailed information can be captured by diagrams such as call graphs, control flow graphs and points-to escape graphs [1, 11].

Unfortunately, many systems are constructed without the use of modeling and visualization artifacts, due to constraints imposed by deadlines, or a shortage of manpower. Nevertheless, such systems might profit from the visualization provided by the UML diagrams and the detailed information provided by other kinds of diagrams; this information can facilitate all phases of the software life cycle. However, the inadequate infrastructure to support testing and other phases of the life cycle are well documented and there is an identified need for the creation of techniques and tools for use in estimating, predicting and performing analysis on evolving software systems [8, 12].

In this paper, we present a tool, *libthorin*, that enables a developer to reverse engineer a program to capture or recapture design and implementation artifacts including design metrics or a UML class or sequence diagram. However, unlike most previous reverse engineering tools, *libthorin* can be applied to a variety of programming languages including C, C++, Java, Fortran 90 and C#. Moreover, *libthorin* can provide both coarse-grained and fine-grained information about the program under development to enable construction of high-level diagrams, such as class diagrams, and low-level diagrams such as sequence diagrams or control flow graphs or other program representations and analysis tools that enable testing, debugging and profiling of an application under development.

In the next section, we provide background about the information format that we use, DWARF, and the language that we use, XMI, as a vehicle of information exchange between programs. In Section 3 we provide an overview of our tool and Section 4 we detail our implementation. In Section 6 we provide some results that indicate speed and space requirements of our tool. Finally, in Section 7 we draw conclusions.

Proceedings of The 3rd ACIS International Conference on Software Engineering Research, Management & Applications, August 2005

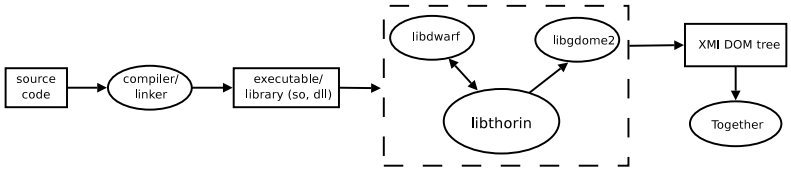

Figure 1: *System overview.* In this figure we list the important components in *libthorin*, where the items illustrated inside the dashed rectangle represent our contribution to the system, items to the left of the dashed rectangle represent components that process the application under study in preparation for input to our system and the item to the right of the dashed rectangle represent output of the system.

2 Background and Related Work

In this section we provide background information about the DWARF symbolic debugging format and compare it to some other popular formats. We also review The XML Metadata Interchange Specification (XMI) that facilitates easy exchange of data and metadata between models and tools. Finally, we provide background about the software metrics that we use to illustrate usage of libthorin.

In Section 2.1 we review the DWARF format and in Section 2.2 we review XMI. In Section 2.3 we review the metrics.

2.1 DWARF

The DWARF debugging information specification is a portable debugging information format. DWARF was originally designed for the Intel 32-bit architecture but has been extended for other platforms including MIPS, SPARC and PPC and for 64-bit architectures. The debugging format is designed to meet the symbolic, source-level debugging needs of different languages in a unified manner. Individual language needs, such as virtual functions for C++ or common blocks for Fortran, are accommodated by creating attributes that are used only for those languages. The DWARF format also facilitates vendor extensibility issues, for example the GNU C++ compiler adds DWARF support for mangled names.

The UNIX International Programming Languages Special Interest Group, SIG, has designed the DWARF version 2.0 format to sufficiently cover the debugging information required for C, C++, FORTRAN77, Fortran 90, Modula 2 and Pascal. However, the version 2.0 format was specified in 1995, prior to the 1998 ratification of the ISO C++ standard [9]. Thus, version 2.0 does not include specification for some important C++ language constructs such as namespace or mutable class attributes. The DWARF version 3.0 format, which is under review, addresses the version 2.0 language deficiancies and provides a format to sufficiently cover the debugging information required for more recently developed languages such as Java [4].

There are two major sections in each DWARF format specification: the encoding and representation of the information in an object file, and the information content of the debugging entires. The encoding and representation information resides in the `.debug_abbrev` section, and the information content resides in the `.debug_info` section of the object file.

Other symbolic debugging formats include *stabs*, the default format for GCC, and *COFF* (Common Object File Format), the default for Visual Studio. Neither format provides the detail that is included in the DWARF format.

2.2 XMI

The XML Metadata Interchange Specification, XMI, facilitates the easy interchange of data and metadata between modeling tools and data repositories in distributed heterogeneous environments. The XMI integrates three industry standards: (1) the extensible markup language, XML, (2) the unified modeling language, UML, and (3) the meta object facility, MOF.

XML is a W3C standard protocol for managing and containing information, formatting documents and filtering data. The UML is an object oriented modeling language that supports a wide range of design tools to support specification of all phases of the software life cycle. The MOF standard defines an extensible framework for defining models for metadata and providing tools with interfaces to store and access metadata in a repository.

The XMI specification supports the interchange of any kind of metadata that can be expressed using the MOF specification. The specification supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for

interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

2.3 Object-Oriented Metrics

Software measurement and evaluation has become an essential part of good software engineering [6]. Many developers measure characteristics of the application to obtain some sense of whether the requirements are consistent and complete, whether the design is high quality, and whether the code is ready to be tested.

To illustrate the benefit of libthorin, we use it to measure some characteristics of the applications in our testsuite of five programs. These characteristics capture some implementation artifacts in the form of two metrics: *Depth of Inheritance Tree* (DIT), and *Number of Children* (NOC) [3].

In an object-oriented system, an application is modeled as a hierarchy of classes. This hierarchy can be represented as a tree, referred to as an inheritance tree. The nodes in the tree represent classes and, for each class, the DIT metric is the length of the longest path from the node to the root of the tree. This measure captures the notion of scope of properties. The DIT metric is a measure of how many ancestor classes there are in the system that can potentially affect this class [6].

The NOC metric is computed for each class in the system and is the number of immediate successors of the class. The NOC metric is the inverse of the DIT metric in that it measures the number of classes that might be effected by this class.

3 Overview of libthorin

In this section we review our approach toward construction of a reverse engineering tool, *libthorin*[1], that accepts an executable or library, compiled with DWARF information, and produces XMI DOM tree for the UML class diagram. We begin with an overview of the *libthorin* system followed by an example execution of the tool.

Figure 1 lists the important components in *libthorin*, where the items illustrated inside the dashed rectangle represent our contribution to the system, items to the left of the dashed rectangle represent components that process the application under study in preparation for input to our system and the item to the right of the dashed rectangle represent output of the system.

[1]Thorin is a dwarf that appears in *The Hobbit*.

The rectangle on the far left side of Figure 1 represents the *source code*, written in any language that can be compiled into the DWARF format; there are compilers that produce DWARF binaries for many languages including *C*, *C++*, *Fortran*, *Java* and *C#* [17]. The ellipse to the right of the *source code* rectangle represents the *compiler/linker* and the rectangle to the right of the the *compiler/linker* is either an executable binary, a shared object library, *so*, for Unix or Linux platform or a dynamically linked library, *dll*, for the Windows platform.

The rectangle on the right side of the figure represents the output of *libthorin*, expressed as an XMI DOM tree. The ellipse shown below the XMI DOM tree rectangle represents a tool, *Together*® [18], that we use to illustrate a class diagram built using information gathered by *libthorin*.

The dashed rectangle in the middle of Figure 1 illustrates the important components in *libthorin*, represented by the ellipse at the bottom of the dashed rectangle. *Libthorin* exploits libdwarf library, which was developed by Anderson [5]. We use libdwarf to query the binary file under investigation and to obtain detailed information to build the XMI DOM. The information extracted by libdwarf includes DWARF debugging information about records, global names, static function names, and type information. The ellipse in the upper right corner of the dashed rectangle represents the gdome2 library, a level 2 DOM engine created for the GNOME project [2]. A DOM tree is an in-memory representation of an XML file.

4 Implementation of libthorin

In this section we provide details about *libthorin*, including a description of the layout of DWARF debugging information and our approach for translating the information into an XMI file. We use the sample code in Figure 2 as an example to explicate some of these details; this example threads the discussions in this paper.

4.1 DWARF layout

A DWARF layout is composed of two major sections: the `.debug_abbrev` section and the `.debug_info` section. The `.debug_abbrev` section describes the encoding and representation of the DWARF content, which is contained in the `.debug_info` section.

4.1.1 The `.debug_abbrev` section

The `.debug_abbrev` section, *abbrev*, describes the layout for each Debugging Information entry (DIE) in the

```
struct Base {
   vritual int foo() = 0;
};

struct Derived : Base {
   int foo() { }
   void bar(Base &b);
private:
   float x;
};
```

Figure 2: **Sample**. This figure contains C++ code that we use to illustrate our approach to reverse engineering. The class diagram on the right of the figure is rendered in *Together*®, using XMI output generated from our tool, *libthorin*.

`.debug_info` section. An *abbrev* entry begins with an unsigned integer, a lookup code that the DIE uses to refer back to the *abbrev*; each lookup code is unique to the *abbrev* in a given compile unit. Following the lookup code is a *tag* that specifies the type to which the DIE entry belongs. The DWARF2 specification includes forty-seven possible DIE tags. Following the DIE tag, is a flag, called the *child flag*, which is set to `DW_CHILDREN_yes` if the DIE contains nested DIEs; nested DIEs are used to represent nested constructs, such as a member function in a class, or a class in a compilation unit. Finally, the *abbrev* entry contains a variable length list of *attribute-form* pairs. The *attribute* defines specific characteristics of the DIE, providing more detailed information, and the form contains information about the representation of that attribute in the `.debug_info` section. The DWARF2 specification lists fifty-nine possible attributes, and six classes of forms. The attribute-form list terminates when both the attribute and the form entries are zero.

Figure 3 contains a listing of information that a `.debug_abbrev` might contain; our listing is based on the code shown in Figure 2. For brevity, we only include high level information about the Derived struct. There are, altogether, six `.debug_abbrev` entries in the figure, where the final entry on line 28 is a null entry indicating the end of entries for this particular compile unit.

The first five rows of the listing in Figure 3 represent a `.debug_abbrev` entry that describes the compile unit in which the Derived struct resides. The value 1 in the first row of the figure is the lookup code for this *abbrev* entry, and its DIE tag, `DW_TAG_compile_unit`, marks the beginning of a new compile unit. line 2 contains a flag, the DIE's child flag; in this case the flag is *yes*, so this DIE will contain nested DIEs. lines 3 and 4 contain the attributes for this compilation unit. line 3 indicates that the name for this DIE is represented as a string, and line 4 indicates that the programming language used to create the compilation unit is represented by a 1-byte constant. The pair of zeros on line 5 indicates the end of the attribute-form pairs for this DIE. line 6 starts a new DIE, this time with a lookup code of 2 and the tag `DW_TAG_structure_type`. line 7 contains the child flag. The DIE that this *abbrev* entry describes contains two attributes, the name of the structure, represented as a string, and the size of the structure, represented by a 1-byte constant, as illustrated on lines 8 and 9. The next *abbrev* entry, on lines 11 through 15, describes an inheritance DIE, as shown by its DIE tag, `DW_TAG_inheritance` on line 11. An inheritance DIE represents an inheritance relationship between the parent DIE of the inheritance DIE and the DIE referenced by the `DW_AT_type` attribute

```
1  1 DW_TAG_compile_unit
2  DW_CHILDREN_yes
3  DW_AT_name              DW_FORM_string
4  DW_AT_language          DW_FORM_data1
5  0                       0
6  2 DW_TAG_structure_type
7  DW_CHILDREN_yes
8  DW_AT_name              DW_FORM_string
9  DW_AT_byte_size         DW_FORM_data1
10 0                       0
11 3 DW_TAG_inheritance
12 DW_CHILDREN_no
13 DW_AT_type              DW_FORM_ref4
14 DW_AT_accessibility     DW_AT_data1
15 0                       0
16 4 DW_TAG_subprogram
17 DW_CHILDREN_no
18 DW_AT_name              DW_FORM_string
19 DW_AT_type              DW_FORM_ref4
20 DW_AT_virtuality        DW_FORM_data1
21 0                       0
22 5 DW_TAG_member
23 DW_CHILDREN_no
24 DW_AT_name              DW_FORM_string
25 DW_AT_type              DW_FORM_ref4
26 DW_AT_accessibility     DW_FORM_data1
27 0                       0
28 0
```

Figure 3: **Partial listing of the `.debug_abbrev` section for the Derived struct listed in Figure 2**

```
1  length
2  2
3  abbrev offset
4  4
5  1
6  "sample.cpp"
7  DW_LANG_C_plus_plus
8  2
9  "Derived"
10 4
11 3
12 DIE offset of Base
13 DW_ACCESS_public
14 4
15 "foo"
16 DIE offset of int
17 DW_VIRTUALITY_virtual
14 5
15 "x"
16 DIE offset of float
17 DW_ACCESS_private
18 0
19 0
```

Figure 4: **Partial listing of the `.debug_info` section for the code listed in Figure 2**

(line 13). In C++an inheritance relationship may be qualified by one of the accessor keywords, public, private, or protected, and is represented in DWARF via the `DW_AT_accessibility` attribute which is encoded as a 1-byte constant on line 14. For brevity, we elide explanation of the last two `.debug_abbrev` entries describing a member function and a class data member.

4.1.2 The `.debug_info` section

Figure 3 shows the contents of the `.debug_abbrev` section, which describes the encoding of the DIEs in the `.debug_info` section. Figure 4 lists the contents of the `.debug_info` section described by the `.debug_abbrev` section from Figure 3. A DWARF `.debug_abbrev` consists of a series of compilation units; a compilation unit is represented by a compilation unit header followed by a series of debugging information entries. The compilation unit header consists of four pieces of information. The first is an unsigned integer representing the length of the entire compilation unit, not including the space required to represent the length. The next field is two bytes that represents the version of the DWARF information used to describe this compilation unit. The next field is an offset into the `.debug_abbrev` section, which points to the beginning of the *abbrev* entry that describes this compilation unit. The final field is the address size in bytes of the target architecture. Following the compilation unit header is variable length list of DIEs. The first field of a DIE is its lookup code, which is used to reference the *abbrev* entry that describes the encoding for this particular DIE. Following the lookup code is the data described by the *abbrev* entry.

The compilation unit header is shown on the first four lines of Figure 4. On line 1 is the number of bytes for the entire compilation unit, represented by *length*. line 2 contains the DWARF version, in this case we are using DWARF version 2.0, so a 2 appears in this location. line 3 holds the byte-offset into the `.debug_abbrev` section that describes this compilation unit. The final line of the compilation unit header, line

Algorithm 1 *numberOfChildren* (*root*, *id*)

```
1: noc ← 0
2: nodes ← root.getElementsByTagnameNS(
   UML_NS, "Generalization.parent")
3: for all Element e ∈ nodes do
4:    idref ← e.firstChild.getAttribute("xmi.idref")
5:    if idref = id then
6:       noc ← noc + 1
7: return noc
```

Algorithm 2 *depthOfInheritance* (*root*, *id*)

```
1: rv1, rv2 ← 0
2: nodes ← root.getElementsByTagnameNS(
   UML_NS, "Generalization")
3: for all Element e ∈ nodes do
4:    parent ← e.getElementsByTagNameNS(
      UML_NS, "Generalization.parent")
5:    child ← e.getElementsByTagNameNS(
      UML_NS, "Generalization.child")
6:    pid ← parent.item(0).firstChild.getAttribute(
      "xmi.idref")
7:    cid ← child.item(0).firstChild.getAttribute(
      "xmi.idref")
8:    if id = cid then
9:       depth ← depthOfInheritance(root, pid)
10:      return depth + 1
11: rv1 ← depthOfInheritance(root.nextSibling, id)
12: rv2 ← depthOfInheritance(root.firstChild, id)
13: return max(rv1, rv2)
```

4, represents the address size of the target architecture; a 4 appears in this location, indicating a 32-bit architecture.

The first debugging information entry of the compilation unit is listed on lines 5 through 7 of Figure 4. line 5 contains the lookup code for the *abbrev* entry that describes this DIE. For the first DIE, the lookup code is a 1; therefore, the *abbrev* entry listed on lines 1 through 5 in Figure 3 describe this DIE. line 6 contains the string `"sample.cpp"` representing the filename of the source code that was used to create the compilation unit, and line 7 contains the constant `DW_LANG_C_plus_plus`, indicating that C++was the programming language used to create the compilation unit.

The next DIE, on lines 8 through 10, represent the DIE for a structure type (a C `struct`). Since the child flag of the previous DIE was set to *yes*, this DIE is a child of the previous compilation unit DIE. Its *name* attribute is represented by the string `"Derived"` on line 9, and the integer 4 on line 10 represents the size in bytes.

The next DIE, on line 11 through 13, represent an inheritance relationship. Since this DIE is a child DIE of the previously described struct Derived, Derived is the child of the inheritance relationship. The parent of the relationship is referenced by the `DW_AT_type` attribute on line 12, which is the struct Base, not represented in DWARF by Figures 3 or 4, but is shown in source code in Figure 2. This DIE has another attribute `DW_AT_accessibility`, which is set to `DW_ACCESS_public` denoting that is public inheritance.

4.2 Generation of XMI

Our approach to XMI code generation requires an examination of each of the DIE entries in the `.debug_info` section of the file that is input to the system. We analyze the relevant attributes in each DIE entry to facilitate generation of XMI.

5 Using libthorin for Metric Evaluation

The listing of Algorithm 1 represents an implementation of the NOC metric using operations defined by the DOM2 specification from the OMG. To find the number of immediate children of a class in an inheritance hierarcy, given the parent class's identification string (*id*), we first create a list of all nodes with tags "UML:Generalization.parent" [line 2]; this tag indicates a node that is a base class and therefore may be involved in an inheritance relationship. The first, and only, child of this tag is a "UML:GeneralizableElement" tag [line 4]; if this tag has the *idref* attribute that references *id*, then we have found an instance of an inheritance relationship where *id* is the parent [line 5 and 6].

The Depth of Inheritance Tree metric (DIT), is defined as the longest path from a class to its top-most ancestor class. The listing in Algorithm 2 represents one possible implementation of this metric using libthorin. To calculate the DIT metric for a given class, identified by *id*, a list of all generalization tags is constructed [line 2]. Each inheritance relationship in a program has a parent class, identified by *pid* [lines 4 and 6], and a child class, identified by *cid* [lines 5 and 7]. If *cid* identifies the same class as *id* [line 8], then the DIT of *id* is one more than the DIT of its parent class [lines 9 and 10]. Because of languages, such as C++, that support multiple inheritance, there may be multiple top-most ancestor classes for a given class. Therefore, all possible paths must be evaluated [lines 11 and 12] and the maximum DIT for a path is returned [line 13]. The DIT value is computed recur-

Test	Time (s)	XMI (MB)	Memory (MB)
Sample	0.02	6.9KB	1.04
Necko	97.4	41.5	232.6
XPCOM	64.1	29.6	166.6
Fluxbox	34.7	21.9	117.8
Gtk+	73.6	25.5	94.0
Keystone	83.6	48.0	261.8

Figure 5: **Time and space results.** In this figure, we provide some results about the time and space requirements of *libthorin*. These results were obtained using our suite of five public domain applications.

	DIT		NOC	
Test	Mean	Max	Mean	Max
Necko	–	–	4.89	126
XPCOM	2.86	7	3.05	56
Fluxbox	2.25	5	3.66	22
Gtk+	1	1	0	0
Keystone	2.56	5	10.77	98

Figure 6: **Results for metrics.** This figure illustrates results for the two metrics applied to the five test programs. The results were obtained using the *libthorin* reverse engineering tool.

sively, with calls on lines 9, 11 and 12.

To improve performance, libthorin generates three hash tables whose keys are frequently occuring attributes in the XMI UML specification. These hash tables improve symbol table lookup from linear to constant time.

6 Results

In this section, we present results using several programs and libraries as test cases. We first present results that measure execution speed and memory usage for *libthorin*, followed by results for the two metrics described in Sections 2 and 5. We have also used *libthorin* to reverse engineer UML class diagrams, as illustrated in Figure 2.

All test cases are executed on a Dell Precision 530 with two 2 GHz Intel Xeon^{TM}processors, 1 GB of RAM, and 2 GB of swap space, running the Ubuntu Linux 5.04 operating system with the 2.6.10 Linux kernel. The actual input to *libthorin* is generated by the GCC 3.3.5 compiler, where each test case is compiled with the `-gdwarf-2` flag to ensure that the required debugging information is included in the object files. We created a small program that uses *libthorin* to gather the statistics about the library. The testing program generated an XMI tree, printed the requisite statistics, and serialized the DOM tree to a file.

6.1 Time and Space Efficiency of libthorin

Figure 5 shows our results with the six test cases listed in the first column of the figure. The first test case is the sample program listed in Figure 2; we use this test case as a baseline: a small example whose memory usage and execution time are close to a lower bound. The next two test cases are modules from *Mozilla 1.5*, the open source, cross-platform web and e-mail application suite [15]. *Necko* is Mozilla's networking kernel, providing the Mozilla suite with a cross-platform API for handling all the necessary networking tasks [14]. *XPCOM* is Mozilla's cross-platform component architecture [16], a framework for writing portable, modular applications similar to CORBA and other COM architectures. Although XPCOM consists of several libraries, we only test the core library. The next test case is *Fluxbox 0.9.6*, a window manager for X11 [7]. The test case listed on row five of the table is *Gtk+ 2.2.4*, a widely-known multiplatform toolkit for creating graphical user interfaces [13]. The final test case is *Keystone*, a parser and front-end for ISO C++ [10].

Columns two through four of Figure 5 show the statistics for each of the test cases. The first column shows the total time taken to generate the XMI tree only, this does not include the time require to write the tree to disk. This value was calculated using the `gettimeofday` POSIX system function, which may fluctuate if the system is heavily loaded. The second column lists the total size of the serialized XMI file on the disk. This value comes from the `stat` POSIX system call. In the last column we report the memory usage of *libthorin* in the analysis of each test case. The memory usage value is extracted from the `/proc` pseudofilesystem; this value is the `rss` field in the process specific `stat` file.

Our results indicate that construction of a DOM tree can be achieved by *libthorin* in reasonable time. Our choice of the DOM tree representation provides a well-recognized, platform- and language-neutral document interface. Columns three and four of Figure 5 show that the memory consumed to read the DWARF format and produce the DOM tree is larger than the actual XMI representation in memory. For example, 100.5 MB of memory is required for the entire construction process of the *Gtk+* test case, shown on row four of the figure; however, only 25.6 MB of actual

disk space is required for the XMI representation.

We have shown that we can compute analysis information using the DOM tree interface in a reasonable time. We chose a DOM tree as our data structure because its a platform- and language-neutral interface that is used by researchers, industry and the Open Source community. We use OMG's XMI, an XML language, for representing UML diagrams. Also, we use W3C's document object model interface (DOM) for representation of our symbol table as a UML diagram, which permits programs and scripts to dynamically access and update the content, structure and style of documents.

6.2 Computing Metric information with libthorin

Figure 6 lists metric results, computed with libthorin, for the five test programs. The rows of the table list the results for the five test cases. The first column lists the name of the test case, the second and third columns list results for the Depth of Inheritance Tree metric (DIT) and the fourth and fifth columns list results for the Number of Children metric (NOC).

For example, consider the DIT results for *Fluxbox* and *Gtk+*, illustrated in the second and third columns of the third and fourth rows of the table in Figure 6. For the DIT metric, the mean depth for *Fluxbox* is 2.25 and the longest depth is 5; however, the mean depth for *Gtk+* is 1 and the longest depth is 1. Thus, there is no inheritance in the *Gtk+*, a C program that facilitates construction of graphical user interfaces.

To further illustrate the metrics, consider the NOC results for *Fluxbox* and *Gtk+*, illustrated in the fourth and fifth columns of the third and fourth rows of the table in Figure 6. For the NOC metric, the mean number of children for *Fluxbox* is 3.66 and the maximum number of children is 126. This rather large number of children for *Fluxbox*, 126, results from the fact that iterators are created using inheritance and *Fluxbox* makes extensive use of iterators. For the NOC metric, the mean number of children for *Gtk+* is 0 and the maximum number is also 0; since this program does not use inheritance there are no children.

7 Concluding Remarks

We have described our approach toward construction of a tool, *libthorin*, for reverse engineering applications. Our tool is language and platform independent, provided that a compiler is available to build a DWARF binary representation of the application. We have presented implementation details of our tool and some results that indicate that the time and space requirements are reasonable when applied to the medium-sized applications in our test suite of five programs. These applications are public domain and used by the open source community. We have also used libthorin to compute metrics for the five programs.

We use a DOM tree as our principle data structure because it provides both a platform and language neutral interface used by researchers, industry and the open source community. We have also used libthorin to reverse engineer UML class diagrams, using OMG's XMI, an XML language, for representation of these diagrams.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] P. Casarini and L. Padovani. The Gnome DOM Engine. *Appeared in the Proceedings of the Conference on Extreme Markup Languages*, March 2001.

[3] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–498, 1994.

[4] D. Anderson. DWARF3: Better than DWARF2. http://reality.sgi.com/davea/, December 2001.

[5] D. Anderson. A Consumer Library Interface to DWARF. http://reality.sgi.com/davea/, November 2003.

[6] N. E. Fenton and S. L. Pfleeger. *Software Metrics A Rigorous & Practical Approach*. PWS Publishing Company, second edition edition, 1997.

[7] H. Kinnunen. Fluxbox. *http://fluxbox.sourceforge.net*.

[8] M. J. Harrold. Testing: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, Dec. 2000. ACM.

[9] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

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

[11] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan-Kaufman, 1997.

[12] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report, May 2002.

[13] The GTK+ Team. Gtk+. *http://www.gtk.org*.

[14] The Mozilla Organization. Necko. *http://www.mozilla.org/projects/netlib*.

[15] The Mozilla Organization. The Mozilla Application Suite. *http://www.mozilla.org*.

[16] The Mozilla Organization. XPCOM. *http://www.mozilla.org/projects/xpcom*.

[17] TIS Committee. DWARF Debugging Information Format Specification (Version 2.0). May 1995.

[18] TogetherSoft. Together Control Center 5.5. *http://www.togethersoft.com/*, November 2001.