AccessViz: A Tool generating Access Graphs from C/C++ Source

R & D Project Report

Course CS691
May 2006

Master of Technology

by

Kanika Nema
Roll No: 05305401

under the guidance of

Prof. R. K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgments

I am thankful to my guide Prof. R. K. Joshi for his constant encouragement and invaluable guidance. I would also like to thank my senior Nishit Desai for his support and help at all times, without which I would have been unable to this project.

Contents

1 Introduction **2**

2 Overall Approach **3**
2.1 Linking C, C++ object files 4

3 AccessViz - Extending CodeViz **5**
3.1 Tools 6
3.1.1 What is CodeViz? 6
3.2 Details of AST 7
3.2.1 Abstract Syntax Tree 8
3.3 Implementation Details of AccessViz 9
3.4 Format 10
3.5 Using AccessViz 11

4 Conclusions **12**
4.1 Current Status 12
4.2 Future Work 12

Chapter 1

Introduction

This report covers some of the designs and tools developed for the Project Ox. These tools are required for objectification of large sized procedural code. For this purpose the Linux kernel has been taken as an example, though none of the techniques are kernel specific. The design approach and the tools can be applied any kind of procedural code.

1. What is Project Ox?
   Object oriented programming has many advantages over conventional procedural programming for constructing systems with highly modular and extensible abstractions. The aim of the project is to automate the process of objectification of procedural code including very large-sized code eg: the Linux kernel. Currently, the project aims at converting C code to C++ code. The compiler being used is GCC.
   Project Ox is an on-going project at Indian Institute of technology, Bombay under the guidance of Prof. R. K. Joshi. This project is the Mtech Thesis Project of a senior graduate student Nishit Desai.

2. What is required for implementing objectification?
   (a) Resolving Linking issues between C and C++.
   These issues are discussed in 2.1
   (b) Finding Candidate objects by analyzing the C code i.e studying functions, the data structures used by them (parameters and return types) and their call hierarchies. For this purpose an Access Graph is built. This report gives the details about AccessViz - the tool built to provide the input to an Access Graph generator.
   (c) Once good Candidate objects have been identified using the Access Graph, then the actual source code needs to be modified. The classes encompassing their data and functions need to be written.
   (d) Once we have an working C++ file, refactoring can be applied to further improve the code design.

3. What is the approach taken?
   A top-down approach for objectification of large-sized code is difficult to follow. Therefore a step by step (bottom-up) processing of files is chosen. The details are discussed in chapter 2. The Linux kernel has been used for testing the bottom-up approach, since it falls in the category of large-sized code. In phase I of implementation, a few files were manually objectified. The kernel was then compiled, linked and was found to be booting successfully. The Ox project is now in phase II, where the aim is to automate the process of identifying candidate objects.

The first chapter covers the design approach and some of the issues in making C code compilable and linkable using a C++ compiler. Chapter 3 describes a tool required for building access graphs. Access Graphs are required for automatically generating objects by studying the data structures used and function calls made by the C code.

Chapter 2

Overall Approach

Applying objectification to existing applications like Linux kernel requires a different approach owing to its size, implementation complexity, and process manageability issues. An **inside-out** objectification technique is proposed. In this the application (procedural code) remains compilable and bootable at all times in presence of partial objectification. The inside-out approach is essentially a bottom-up approach, in which, the deepest structures are objectified before the outer layers get objectified. The approach is completely defined in [3].

Five levels of objectification are defined. The files move from level 0 to level 4 as they mature in terms of degree of objectification. These levels of objectification are used to indicate the extent to which a particular file in the application has been objectified. At all levels, all files are required to be compilable by appropriate (gcc or g++) compiler to produce a working binary. In case of the Linux kernel, at all times the kernel must be bootable.

Besides guiding the objectification process, the five levels described below for the kernel can also be used to track the progress of the objectification process.

1. **level 0 (Procedural)** A file at level 0 is the conventional C file in the existing application.

2. **level 1 (g++ Compatible)** A file at level 1 is made g++ compatible by solving all compatibility issues. A file at level 1 also links with the rest of the application. The file need not contain any object oriented features, but once brought to this level, it is capable of objectifying itself. It can be noted that all header files included in a file that has reached level 1, are also at level 1.

3. **level 1.x (Client of Objects)** A file at level 1.x has reached level 1 already, and hence outgoing procedural calls can be replaced by member function calls on objects located in other files that have reached levels 3 or 4.

4. **level 2 (Local Objectification)** A file at level 2 has only local objectification, i.e., the external visibility is still procedural. To achieve level 2 maturity, wrapper functions are used.

5. **level 3 (Dual Use)** At this level, some of the dependent code of objectified file reaches level 1.x or above, while the rest of the dependent code may still be at level 0. Thus it is required to maintain two interfaces to the objects located in this file.

6. **level 4 (Full Objectification)** A file at level 4 is used only by files that are at level 1.x or above. Wrapper functions of files at level 4 are dissolved as they are not needed. Further object oriented refactoring such as structural adjustments can be applied on a group of files that have reached level 4.

A proof-of-concept implementation was done by manually changing only the *IPC* related files in the Linux kernel. The kernel was found to run successfully.

2.1 Linking C, C++ object files

There are many issues involved in linking C and C++ files together. The Linux kernel 2.6.10 and Gcc 3.4.x. was used for this purpose.

The issues are discussed in detail in [3]. Some of the issues that arose in compiling and linking the IPC files - msg.c msgutil.c shm.c are

1. Name Mangling - Handled using the *extern C* construct.
2. Type Casting - C compiler does implicit typecast. In C++, explicit type casting is required. It might be able to automate this by parsing the compiler errors generated.
3. Non-trivial Initializers - labeled initializations of structure fields in C++ can be done only when all fields are initialized and in order.
4. Keywords being used as variable names - Some of them were part of structures used in most of the kernel. For example the *struct_page* structure contains a field called *private*. To rename this field would require a change to about half the kernel. Thus a quicker way to handle this was to use a macro as

   ```
   #define private private_XXX_changed_
   ```

5. Building objects out of structures - A simple way to implement this is to inherit classes from structures. This ensures that other files referencing the structure need not be objectified.

A change in the kernel's top level Makefile was required. The final linking should be done by g++ and not gcc.

A technical report [8] describing this approach has been written.

Chapter 3

AccessViz - Extending CodeViz

For automating the process of objectification, it is required that the objects to be created are suggested automatically. The technique uses an **access graph** to analyze the C code to suggest the data structures and functions that should be made into a class. The method is a combination of the Global based [1] and Type based [2] object identification techniques. Metrics for measuring Class Cohesion is used before the final classes are suggested. The details about Access Graphs are discussed in [4]. Access graph G = (V,E) is generated for each function using function signature and function body. V is the set of vertex which contains all functions, all global variables and parameters and return type of all functions. E is the set of edges. There is an edge from node v1 to v2, when one of the following conditions

1. v1 is a function and v2 is a global variable access by v1.
2. v1 is a function and v2 is a parameter of v1.
3. v1 is a function and v2 is the return variable of v1.
4. v1, v2 are functions and v1 calls v2.

The type of the edge can be a call, read, write, read-write or a return-type. In case the return type is different from the types of the parameters, it is called a generate edge. An edge joining a function node and a variable is can also have the tag - *type*, if the variable is a parameter or its return type. It is tagged *global* if the variable is a global variable. The figure 3.1 shows the access graph corresponding to the sample input shown in table 3.1.

The AccessViz is a tool that outputs information that can be used to build access graphs. The AccessViz tool can be applied to any sized procedural code.

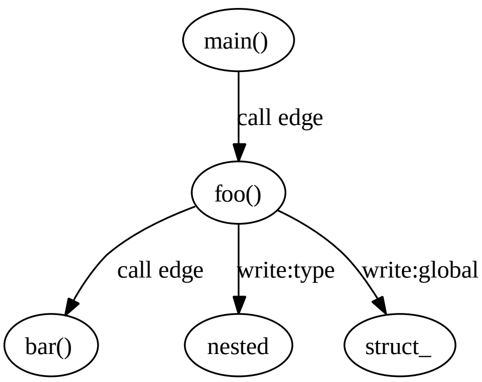

Figure 3.1: Access Graph

3.1 Tools

The tool required to build the access graph should be able to provide the following information

1. Call graph (call edges)
2. Argument types to a function
3. For each argument, the information if the function reads and/or writes the argument.
4. Return type of the function and if it is different from the argument types.
5. Details of Structures used to detect part-type relationships and used to decide that data members of the class to be created.

Such a tool will typically will need to parse the C code under observation. We would be reinventing the wheel if we tried to try our own parser for the entire C grammar. Thus we tried to use already available parsers and tools for producing access graph information. The tool developed is based on a call-graph generator known as CodeViz.

3.1.1 What is CodeViz?

There are a lot of call graph generators available like Codesurfer[7], Ariadne[**?**], Fjalar[5] and so on. Since some of these are not C++ based and some are not open-source free products, hence were not used. CodeViz [6] developed by *Mel Gorman*, was found to be most appropriate for our purpose. CodeViz provides the ability to generate call graphs to aid the task of understanding code. It uses a highly modular set of collection methods and can be adapted to support any language although only C and C++ are currently supported. CodeViz uses GCC structures to output useful information. This is an efficient way to do as GCC's C, C++ parser is most complete. Also, this is an open source product and thus is easily available for use. Another advantage of CodeViz is that, it was built keeping Linux kernel Code in mind. This implies that it works for large-sized code.
Show in table 3.2 is output of CodeViz for the input shown in table 3.1. CodeViz generates a file *.cdepn* whose format is shown below.

```
Stmt -> F {Fn (Alist)} {File:Ln} |
        C {Fn (Alist)} {File:Ln} {Fn (Alist)}

Fn    -> FunctionName
Alist -> ArgumentList
File  -> Filename
Ln    -> LineNum
```

This *.cdepn* file is processed by Perl scripts to generate a *.dot* format file. The *dot* command can be used to generate the call graphs. AccessViz has extended the format of CodeViz and the details are discussed in the next section.

```
struct nested {
    int field;
};

typedef struct struct_
{
    int (*funcp) (int,int);
    union {
        struct nested check;
        char dummy_char;
    }u_union;
    int a ;
}struct1;

struct1 global_var;

void bar()
{
    return;
}

int foo (int count, struct nested *tmp)
{
    global_var.a = count * 20;
    tmp->field = 10;
    bar();
    return 0;
}

int main (int argc)
{
    struct nested temp;
    int i = foo (argc, &temp);
    return 0;
}
```

Table 3.1: Sample C++ Input

```
F {void bar()} {sample.cpp:15}
F {int foo(int, nested*)} {sample.cpp:19}
C {int foo(int, nested*)} {sample.cpp:22} {call:void bar()}
F {int main(int)} {sample.cpp:26}
C {int main(int)} {sample.cpp:28} {call:int foo(int, nested*)}
```

Table 3.2: CodeViz output .cdepn file

3.2 Details of AST

GCC parses the input files and builds an abstract syntax tree (AST). This is an in-memory structure built by gcc during compilation. All front-end code is converted into this format. All information is stored in terms of nodes and child pointers. Every node is of a particular kind and accordingly its children nodes are defined. The tree nodes and their fields are described in the tree.def file in GCC.

By, correctly following the node pointers, all information related to function arguments, their types, return types, globals and program statements can be recovered from the AST. There are two possible options available for using the AST

1. Debug files outputted by gcc
   Using this method has the advantage that the tool will not require a re-compilation of *gcc*. The user can just download the tool and run it on the files outputted by *gcc*. The biggest disadvantage of this method is that the format of the debug files outputted by various versions of gcc vary greatly. Though the AST format has remained mostly same over the GCC 3.3, 3.4.x and 4.0.x, the file formats outputted by gcc using the -fdump-tree-xxx flag differ considerably. In GCC 3.3 both data and control structure of the program was outputted in the same file. In GCC 3.4.x this was split into two files - the .tu and the .original files. From 4.0.x onwards the file outputs almost in a C - like syntax. Another difficulty is the fact that, there is some amount of loss of information while re-building the AST from the files. This is because, not all of a tree node's field are dumped in the file. For a specific gcc version using file formats requires

   (a) Parsing the files. This involves re-doing a lot of work, that the compiler has done during the compilation.

   (b) Re-writing the tree functions already available in gcc to traverse and extract information from the AST re-built. This program will be a large overhead.

2. Modifying GCC and using the AST when built by gcc during compilation.
   This has the disadvantage that user will need to re-compile the compiler before using the tool. Also, the patch built will be version specific and will have to be ported for each version of GCC, till it becomes a part of GCC. The advantage of the method is that, it is independent of the file formats and can use all the GCC tree functions available to extract information from the AST. No unnecessary parsing is required. Since the infomation extracted is a side-effect of compilation, hence lower overhead. Also, as mentioned above the AST format has remained mostly same over the various versions of GCC, maintaining the patches will not be very difficult. For a specific version modifying GCC requires

   (a) A careful study of the GCC code to find out where to insert this code

   (b) Understanding the various tree related functions/macros available in GCC.

AccessViz uses the second method of implementation.

3.2.1 Abstract Syntax Tree

This section explains the AST generated by gcc in brief. The overall structure of the AST is shown in the figure 3.2. Each node in the tree is of a particular kind accessed using the TREE_CODE function. Depending on the type of node, it points to other nodes in the tree. When a file is compiled using gcc with the -fdump-tree-original switch, the .original file is created. A snippet of the file has been show in table 3.3.

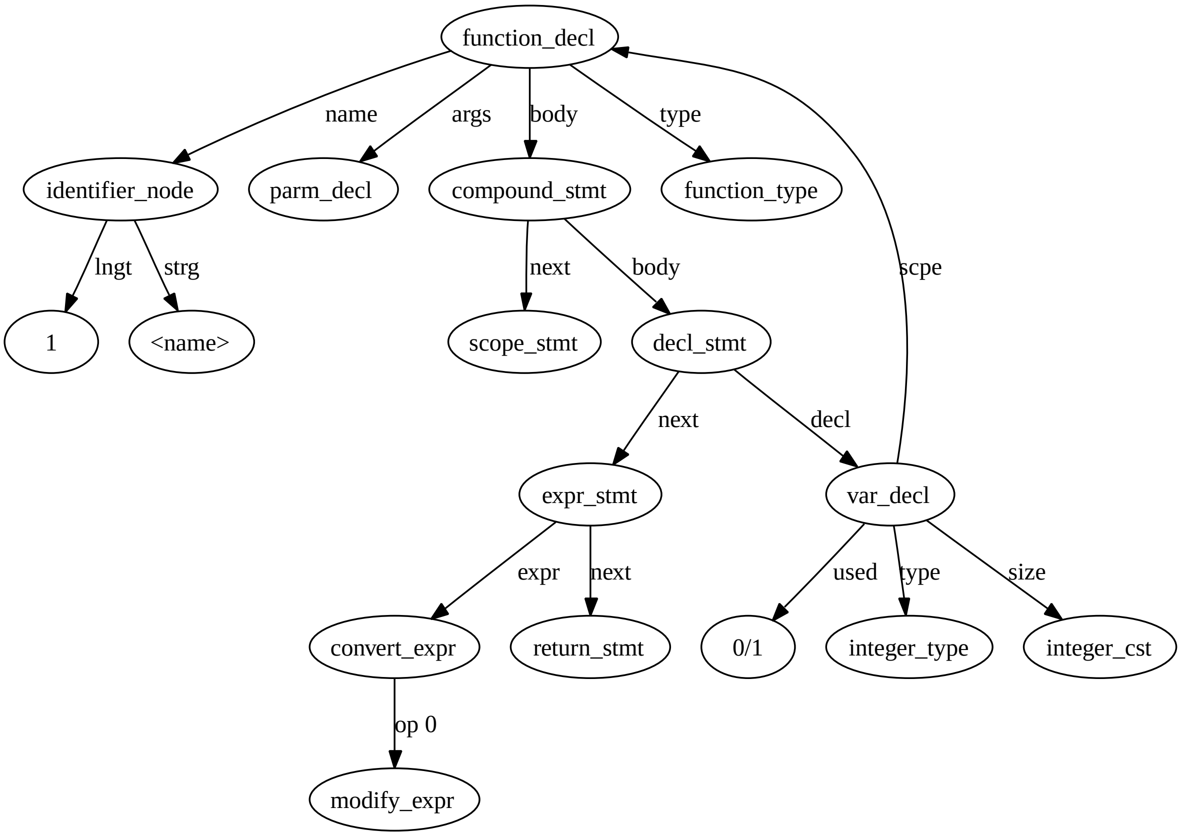

Figure 3.2: Overall structure of the AST

@1	function_decl	name: @2	mngl: @3	type: @4
		srcp: sample.cpp:19		args: @5
		extern	body: @6	
@2	identifier_node	strg: foo	lngt: 3	
@5	parm_decl	name: @10	type: @8	scpe: @1
		srcp: sample.cpp:19		argt: @8
		size: @11	algn: 32	used: 1
@6	compound_stmt	line: 24	body: @12	
@12	scope_stmt	line: 19	begn	clnp
		next: @18		
@28	record_type	name: @31	size: @27	algn: 32
		struct	flds: @32	fncs: @33
		binf: @34		
@36	convert_expr	type: @29	op 0: @45	
@45	modify_expr	type: @8	op 0: @53	op 1: @54

Table 3.3: .original file snippet

3.3 Implementation Details of AccessViz

This code was added in the gcc/cgraphunit.c file. CodeViz's code to output the call graph is in the same file. The output is written in the same format as expected by the CodeViz scripts. This is so that the same scripts can be used to generate human readable access graphs.

The Algorithm can be briefly defined as

$\forall f \in$ functions in the input code
do
traverse the AST pointed to by the *body* node of f
$\forall\ m \in modify\_expr$
do
$write\_var(operand(0, m))$
$read\_var(operand(1, m))$
$\forall\ c \in$ callee of f
do $dump\_callee(c)$

The functions $write\_var$ and $read_var$ are recursive procedures that traverse the tree till the end of a function (detected from scope statement nodes) and outputs if a parameter or a global variable is being read or written. Also, if the variable under consideration is a structure then, the complete structure is also written to the output file. This information is used to discover the part-type relations between structures. If structure A contains structure B and both of them are paramters to a function f, then only structure A is considered.

Some of the functions used to access the various fields of the tree node are shown below.

1. TREE_CODE - The type of the tree
2. DECL_CONTEXT - Returns the scope of the declaration (variable)
3. TREE_OPERAND - Typically, the first child pointer.
4. TREE_CHAIN - The next link pointer
5. EXPR_STMT_EXPR - To access the *expr* field on an *EXPR_STMT*
6. COMPOUND_BODY - The body (next node) in a *compound_stmt*

3.4 Format

AccessViz has extended the CodeViz format, shown in section 3.1.1. The non-terminal Arg represents a single argument i.e the name of the argument along with its type. Similarly all the Name rules produce valid name strings. For example, SName produces the structure name and FName the field name. Type produces the type of the field. Table 3.4 shows the output of AccessViz for the sample input shown in table 3.1

```
Stmt -> F {Fn (Alist)} {File:Ln} |
        C {Fn (Alist)} {File:Ln} {call:Fn (Alist)} |
        C {Fn (Alist)} {File:Ln} {Kind:Access:Arg} |
        S {Type SName:: FName}

Fn     -> FunctionName
Alist  -> ArgumentList | Arg
File   -> Filename
Ln     -> LineNum
Kind   -> global | param
Access -> R|W
```

```
F {void bar()} {sample.cpp:15}
F {int foo(int, nested*)} {sample.cpp:19}
C {int foo(int, nested*)} {sample.cpp:1} {param:R:int count}
C {int foo(int, nested*)} {sample.cpp:1} {global:W:struct_ global_var}
S {int (*struct_::funcp)(int, int)}
S {struct_::<anonymous union> struct_::u_union}
S {nested struct_::<anonymous union>::check}
S {char struct_::<anonymous union>::dummy_char}
S {int struct_::a}
C {int foo(int, nested*)} {sample.cpp:1} {param:W:nested*tmp}
S {int nested::field}
C {int foo(int, nested*)} {sample.cpp:22} {call:void bar()}
F {int main(int)} {sample.cpp:26}
C {int main(int)} {sample.cpp:28} {call:int foo(int, nested*)}
```

Table 3.4: AccessViz Output

3.5 Using AccessViz

This describes how to install and use the AccessViz.

```
Installing the AccessViz

Ox has extended the already available CodeViz tool, to generate information
required for automation of objectification of procedural code.

1. Download

a. GCC 3.4.1  http://ftp.gnu.org/gnu/gcc/
b. CodeViz www.csn.ul.ie/~mel/projects/codeviz/
b. patches i.  CodeViz patch www.csn.ul.ie/~mel/projects/codeviz/
           ii. Ox patch www.torana.cse.iitb.ac.in/~ox/

2. Patching

tar zxvf gcc-3.4.1
cd gcc-3.4.1
patch -p1 < ../gcc-3.4.1-cdepn.diff
patch -p1 < ../Ox.patch

3. Installing

Configure and compile
../gcc-3.4.1/configure --prefix=$INSTALL_PATH --enable-languages=c,c++
make bootstrap
make install

4. Using CodeViz
genfull etc. to generate dot files.  Please refer to
www.csn.ul.ie/~mel/projects/codeviz/
for further details
```

Chapter 4

Conclusions

4.1 Current Status

All the project tools and documentation is available at http://torana.cse.iitb.ac.in/ ox/. This site can be followed to learn about the approach and to use the available tools.

1. Getting the the message queue code to level 5. This was done manually as proof of concept. The shared memory code shm.c was also moved at level 1 along with other several files. An attempt to use ANTLR [9] (ANother Tool for Language Recognition) has been studied by [3]. It is useful to get files automatically at level 1. On large files like in the kernel code, the time taken by ANTLR is very large. But, ANTLR can be used for smaller codes.

2. The tool to output information required by Access Graphs is available as a patch to GCC. Currently only 3.4.1 is supported. This patch can be applied only after the CodeViz patch is applied.

3. The user manual 3.5 and format3.4 for file outputted by gcc/g++, to be used for Access graphs.

4. Parsing the output of AccessViz output to build an in-memory access graph, for further processing.

4.2 Future Work

The following have been identified as the future work for the project

1. Making AccessViz available for GCC version 4.0.2

2. All the information to build the access graph is outputted by the AccessViz. The format in which it is outputted can be modified slightly so that it is easier for the program/script reading it.

3. Analyzing the generated access graphs to suggest good classes.

4. Detecting the relationships between the objects created.

References

[1] *Identifying objects in a conventional procedural language: An example of data design recovery*, 1990.

[2] Gerardo Canfora, Aniello Cimitile, Malcolm Munro, and C. J. Taylor. Extracting abstract data types from c programs: A case study. In *ICSM*, pages 200–209, 1993.

[3] Nishit Desai. Towards object orienting linux kernel structures. Mtech Thesis Second Stage Report, Department of Computer Science Engineering, Indian Institute of Technology Bombay, 2006.

[4] Nishit Desai. Towards re-engineering of linux kernel from procedural to ob ject-oriented. Mtech Thesis First Stage Report, Department of Computer Science Engineering, Indian Institute of Technology Bombay, 2005.

[5] Fjalar http://pag.csail.mit.edu/fjalar/.

[6] CodeViz http://www.csn.ul.ie/ mel/projects/codeviz.

[7] CodeSurfer http://www.grammatech.com/products/codesurfer/index.html.

[8] Rushikesh Joshi, Nishit Desai, Kanika Nema, and Anil Gracias. Towards object orienting linux kernel structures. Technical report, Department of Computer Science and Engineering Indian Institute of Technology, Bombay Mumbai, 2006.

[9] Terence Parr. Antlr reference manual. Technical report, University of San Francisco, 2005.