Using the Sage^{++} Toolkit to Model Control Flow And Extend Cyclomatic Complexity

Brian A. Malloy, John D. McGregor and Sheryl A. Elliott
Department of Computer Science
Clemson University
Clemson, South Carolina, USA
{malloy,johnmc}@cs.clemson.edu

Abstract

The unavailability of comprehensive, user-friendly programming environments is one of the most imposing obstacles to the development and testing of software applications. The most critical advances needed to remove the obstacles are improved tools for program analysis, visualization, evaluation, and debugging. In this project, we use a tool, $Sage^{++}$, to model control flow using an internal representation of the source code called a program tree, a form similar to an abstract syntax tree. We use the xvcg tool to visualize the control flow graph. To facilitate program evaluation, the control flow graph representation is used to compute software complexity using McCabe's cyclomatic complexity measure. Since cyclomatic complexity does not include program exceptions in its consideration, we extend the complexity measure to include C^{++} exceptions, since they alter flow of control and thereby affect the cyclomatic complexity of the program.

Keywords: Control flow graph, program flow, cyclomatic complexity, exception handling, toolkit

1 Introduction

The unavailability of comprehensive, user-friendly programming environments is one of the most imposing obstacles to the development, testing and measure of software applications. The most critical advances needed to remove these obstacles are improved tools for program analysis, visualization, evaluation, and debugging[10]. Recent research has resulted in a proliferation of program representations for program analysis including the program dependence graph [4], the unified interprocedural graph [5], the object-oriented program dependence graph [9], the call graph [11] and the program summary graph [3]. However, the program representation used most frequently for optimizations is the control flow graph [1]. For example, in a recent conference on programming languages, virtually all of the program optimizations used the control flow graph or a variation of the control flow graph as the program representation.

One comprehensive programming environment that facilitates program optimizations is $Sage^{++}$, a toolkit and class library for Fortran and C^{++} [2]. $Sage^{++}$ is a powerful tool that provides an integrated set of parsers that transform the source program into an intermediate representation called a *program tree*, a form similar to an abstract syntax tree [1]. The $Sage^{++}$ toolkit includes a library of classes containing methods for manipulating and restructuring the program tree together with an *unparser* that generates new source code from the restructured program tree. However, there are several drawbacks to using the $Sage^{++}$ toolkit. First, the toolkit does not include methods for constructing the control flow graph (cfg) for C^{++} programs. Although cfg construction is straightforward when the program representation is assembly code, cfg construction is more involved when the program representation is source code, written in a high-level language, or a program tree. Furthermore, the $Sage^{++}$ toolkit does not include tools for visualization or evaluation of C^{++} programs.

In this work, we address the above drawbacks in the $Sage^{++}$ toolkit. We begin by incorporating methods into the $Sage^{++}$ toolkit to construct the control flow graph from the program tree representation of C^{++} source code. Using the xvcg graph tool [12], we add visualization to the toolkit by allowing the user to view the cfg representation of the program. Also, to facilitate program evaluation, we incorporate methods into the $Sage^{++}$ toolkit that use our cfg to compute McCabe's cyclomatic complexity measure [8]. Since McCabe's measure does not include program exceptions in its complexity considerations, we extend the measure to include C^{++} exceptions since they alter

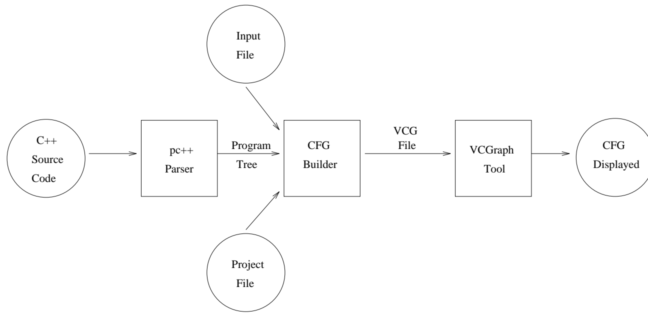

Figure 1: The process of building control flow graphs.

flow of control and thereby affect program complexity. Finally, we incorporate a menu of choices into the toolkit to make toolkit usage more user-friendly.

The remainder of this paper is organized as follows. In the next section, we describe how to model control flow using a program tree and Section 3 discusses how cyclomatic complexity is extended to include C^{++} exceptions. Section 4 presents experiments involving the extension to cyclomatic complexity. In Section 5, we draw conclusions about using $Sage^{++}$ and extending cyclomatic complexity.

2 Modeling Control Flow using a Program Tree

The cfg builder presented in this section constructs control flow graphs using the program tree. Each node in the program tree represents a statement in the source code. We use *node* and *statement* interchangeably throughout this section. To construct the cfg, basic blocks are limited to single statements, and the program tree is traversed one statement at a time with control passing to the next statement unless a control transfer statement is encountered. A control transfer statement for C^{++} is a *while, for, do while, if, switch, break, continue, return, exit, goto* or *loop end.* These statements require more consideration when determining predecessors and successors.

The overall process of building control flow graphs using the program tree is depicted in Figure 1. C^{++} source code is parsed using the pc^{++} parser to produce the program tree, which is stored in a file with the extension *.dep*. The project file, *cfg.proj*, contains the names of the *.dep* files that are input to the `CFGBuilder` algorithm. There is also another input file to the `CFGBuilder` algorithm that contains the names of the files and functions to be graphed. The `CFGBuilder` algorithm performs routines to compute control flow information and format that information for the VCG tool. The output of the builder is

```
algorithm   ComputeComplexity
input       .dep file – program tree
            cfg.input – input file
output      complexity – software complexity measure
variable    nodes = 0 – node count
variable    edges = 0 – edge count
variable    succs = 0 – number of successors for the block

begin ComputeComplexity
   for each function in cfg.input loop
        set last to the last node of the function
        set stmt to the first node of the function
        while stmt <= last loop
              controlFlow(stmt, func, succs)
              nodes++
              edges = edges + succs
              stmt = stmt->lexNext()
        end while
        complexity = edges − nodes + 2
   end for
end ComputeComplexity
```

Figure 3: Algorithm to compute software complexity.

a VCG specification with extension *.vcg* for each function graphed. The VCG tool is used to read a *.vcg* file and visualize the graph. The VCG tool parses the graph specification, assigns positions to each node, assigns positions to edges so they do not overlap nodes, and draws the resulting picture in a window [12].

3 Computing Software Complexity

Using a cfg, G, complexity, V, is defined as $V(G) = e - n + 2p$, where e is the number of edges, n is the number of nodes, and p is the number of connected components in G. In the next section, we describe our technique to compute cyclomatic complexity using a cfg produced by our builder. In Section 3.2, we extend cyclomatic complexity to include C^{++} exceptions.

3.1 Algorithm to Compute Cyclomatic Complexity

We use algorithm `ComputeComplexity` illustrated in Figure 3 to compute cyclomatic complexity during construction of the cfg; this algorithm is an extension of the one that we use tp build the cfg. Input to `ComputeComplexity` is the program tree, contained in the .dep file, and the file *cfg.input* that contains the number of files to be graphed, the names of the files that contain functions to be graphed, and the names of the functions to be graphed. `ComputeComplexity` finds the complexity of each function in `cfg.input`. The algorithm begins by setting `stmt` to the first node in the function to be considered, and continues until the last node is encountered. As a basic block is added to the cfg by algorithm `controlFlow`,

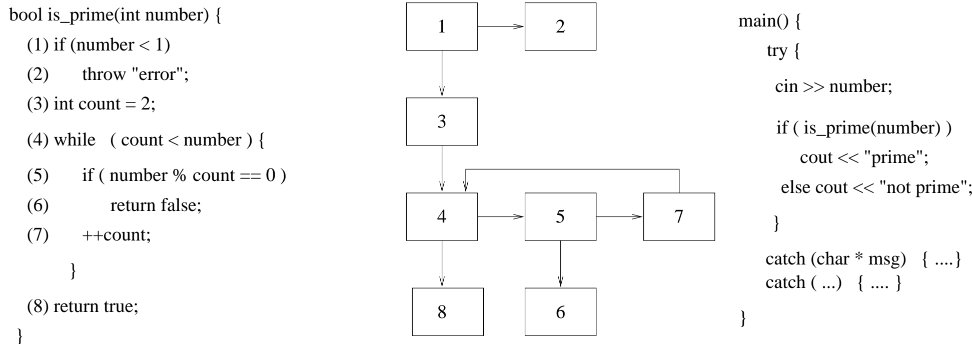

Figure 2: *Exception example.* This figure exploits exceptions to handle negative input to the sample function, *is_prime.* The cfg for *is_prime* is included in the figure, together with a corresponding main program.

`ComputeComplexity` increments the node count by one and adds the number of successors for the block to the edge count. After `edges` is incremented, `stmt` is updated to the next statement in the function. After the cfg is built for this function, cyclomatic complexity is computed by subtracting the node count from the edge count and adding two [8]. `ComputeComplexity` continues by computing cyclomatic complexity for the next function in `cfg.input`

3.2 Extending Cyclomatic Complexity to include C^{++} Exceptions

Cyclomatic complexity was first measured using Fortran as the source language. As a result, the control structures that were defined to be predicates were those structures found in Fortran and other languages available at that time. Since C^{++} was developed in the early 1980s, the constructs of C^{++} were not explicitly considered by McCabe's measure. In this section, we consider how a specific construct of C^{++}, the exception, relates to McCabe's cyclomatic complexity measure.

Figure 2 exploits exception handling including the primitives *try*, *catch* and *throw*. The left side of the figure illustrates the typical use of the *throw* primitive: coupled with a decision statement. Thus, any influence that the *throw* primitive might have on cyclomatic complexity is automatically included in the computation. The right side of the figure illustrates the use of the *try* and *catch* primitives; the *try* and *catch* statements that are used to handle exceptions are not included in complexity computation. The *try* primitive simply marks the code to be included in exception handling; however, the *catch* primitive acts as a conditional statement. Thus, we handle the *catch* primitive as a predicate when building the cfg. *Catch* primitives are handled as predicates because there is a decision to be made about the flow of control based on whether or not the exception type in the *catch* statement matches the type of the exception thrown. For example, on the right side of Figure 2 there are two *catch* statements following the *try* block in the main program. If the thrown exception is of type *char **, it is handled by the first *catch* statement; otherwise it is handle by the statement *catch(...)* that matches any thrown exception.

In summary, *catch* statements are predicates and therefore increase the number of paths through a program. The increased complexity introduced by *catch* statements is not covered by McCabe's original definition of cyclomatic complexity. If exceptions are modeled with *catch* statements as predicates, the algorithm in Figure 3 correctly computes complexity for code including exceptions.

4 Experiments

To investigate the change in complexity when exceptions are used, we computed McCabe's overall cyclomatic complexity for a test suite of six programs. The statistics computed by our experiments are illustrated in Table 1 where column one lists the program name, column two lists the number of lines of code in each program, column three lists the number of classes in each program, column four lists the number of functions and the final column lists the cyclomatic complexity for each program. This final column shows the overall complexity of each program as the sum of the McCabe numbers for each function in the program[8].

The first row of Table 1 lists statistics for our program that builds the cfg and computes cyclomatic complexity and the second row lists statistics for a program that simulates a personal communications system for cellular phones (PCS). The last eight rows

Program	No. of Lines	No. of Classes	No. of fns	Comp plex
our $Sage^{++}$	993	0	9	99
PCS	1880	13	87	144
DecToBin, Ret	95	1	7	14
DecToBin, Exc	94	1	7	14
Paldrome, Ret	162	1	8	21
Paldrome, Exc	166	1	8	21
FileSystm, Ret	75	1	4	7
FileSystm, Exc	102	3	6	7
NASSim, Ret	174	3	12	25
NASSim, Exc	175	3	12	25

Table 1: *This table overviews the test programs and the computed results obtained by our cfg builder and cyclomatic complexity computation.*

of the table list statistics for four programs: *DecToBin*, *Paldrome*, *FileSystm*, and *NASSim*. DecToBin is a decimal to binary number converter. In Paldrome, the user enters words that are checked for palindromes. FileSystm is a C^{++} version of a file system class from the *Ada 95 Reference Manual*[6]. NASSim is a simulator for the National Airspace System (NAS)[7].

For each of the four test program, we developed a version that used return values and a version that used exceptions. We then computed software complexity for each version of the test program. For example, rows 3 and 4 indicate that the cyclomatic complexity of each version of *DecToBin* is 14. The use of return values is an alternative to exceptions whereby the programmer returns a number whose value is used, upon return, to indicate if an exception occurred in the called routine. The final column of Table 1 shows that the complexity is the same for programs that use exceptions and for programs that use return values to indicate errors in called routines.

5 Concluding Remarks

We have addressed several drawbacks in the $Sage^{++}$ toolkit for C^{++} programs. First, we augment the tool to permit construction of the control flow graph (cfg), a valuable representation for program analysis and optimization. Our `CFGBuilder` is incorporated into the $Sage^{++}$ toolkit and uses the program tree together with methods in the class library to facilitate cfg construction. Second, using the xvcg graph tool [12], we add visualization to the toolkit by allowing the user to view the cfg representation of the C^{++} source program. Third, to facilitate program evaluation, we incorporate methods into the $Sage^{++}$ toolkit that use our cfg to compute the cyclomatic complexity of the program [8]. Finally, we extend cyclomatic complexity to include C^{++} exceptions, since they alter flow of control and thereby affect program complexity.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. $sage^{++}$: An object-oriented tookkit and class library for building fortran and c^{++} restructuring tools. Technical report, University of Rennes and Indiana University, November 1993.

[3] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of SIGPLAN'88 Conf. Programming Language Design and Imple mentation*, pages 47–56, 1988.

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

[5] M. J. Harrold, B. A. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 160–170, June 1993.

[6] International Standards, ANSI/ISO/IEC-8652:1995. *Ada 95 Reference Manual*, January 1995.

[7] B. A. Malloy, D. E. Bushey, and S. Yang. Using jet routes to model path re-routing in the national airspace system. *Proceedings of the 13th European Simulation Multiconference*, June 1999.

[8] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec 1976.

[9] J. D. McGregor, B. A. Malloy, and R. L. Siegmund. A comprehensive program representation for object-oriented software. *Annals of Software Engineering*, 2, 1996.

[10] P. Messina and T. Sterling, editors. *System Software and Tools for High Performance Computing Environments.* Society for Industrial and Applied Mathematics, April 1993.

[11] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–225, May 1979.

[12] G. Sander. Vcg - visualization of compiler graphs, user documentation v. 1.30. Technical report, Universitat des Saarlandes, February 1995.