Empirical Software Engineering Journal 3(2), March, 1998, pp. 203-211

Empirical Studies of Control Dependence Graph Size for C Programs

Mary Jean Harrold
Computer and Information Science
Ohio State University
395 Dreese Lab
Columbus, OH 43210
harrold@cis.ohio-state.edu

James A. Jones
Computer and Information Science
Ohio State University
395 Dreese Lab
Columbus, OH 43210
jjones@cis.ohio-state.edu

Gregg Rothermel
Computer Science
Oregon State University
307A Dearborn Hall
Corvallis, OR 97331
grother@cs.orst.edu

Abstract

Many tools and techniques for performing software engineering tasks require control dependence information, represented in the form of control dependence graphs. Worst-case analysis of these graphs has shown that their size may be quadratic in the number of statements in the procedure that they represent. Despite this result, two empirical studies suggest that in practice, the relationship between control dependence graph size and program size is linear. These studies, however, were performed on a relatively small number of Fortran procedures, all of which were derived from numerical methods programs. To further investigate control dependence size, we implemented tools for constructing the two most popular types of control dependence graphs, and ran our tools on over 3000 C functions extracted from a wide range of source programs. Our results support the earlier conclusions about control dependence graph size.

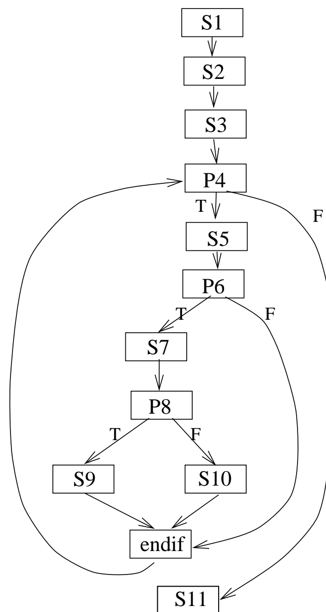
1 Introduction

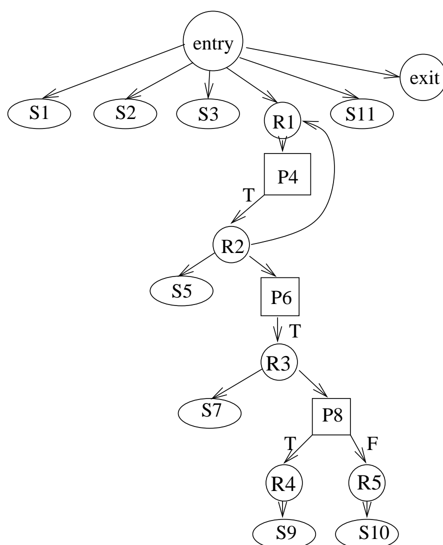
Many software engineering techniques and tools rely on control-dependence information. Techniques for selecting test data and determining test set adequacy [14], extending data-flow testing approaches [8], generating reduced test sets for programs [11], determining the retesting required after program modifications [2, 4, 5, 10, 15, 16], integrating different versions of programs [13], and performing static and dynamic slicing [1, 3] all use such information. To represent control dependence information, these techniques typically use control dependence graphs. Attempts to scale these techniques to handle large programs are constrained by the memory and disk space required to construct and store the control dependence graphs for these programs.

A worst-case analysis by Cytron et al. [6] shows that the size of the control-dependence graph for a procedure P can be quadratic in the size of the control-flow graph for P, and hence, quadratic in the number of statements in P. However, in a study using the 61 Fortran routines in Eispack [17] (a set of matrix Eigensystem routines), Cytron et al. found that, in practice, the relationship between control-dependence graph size and program size is linear. A larger study by Cytron, Ferrante, and Sarkar [7] on over 400 Fortran procedures from several popular numerical analysis programs also showed a linear relationship between the size of the control-dependence graph and the number of program statements.

To further study the relationship between control-dependence graph size and program size, we performed studies using a variety of C programs as subjects. For each function in these programs, we recorded the number of executable statements in the function, and the size of control-dependence graphs for the function.

```
procedure Sums
  S1.   read i
  S2.   sum = 0
  S3.   done = 0
  P4.   while i <= 5 and !done do
  S5.         read j
  P6.         if j >= 0 then
  S7.               sum = sum + j
  P8.               if sum > 100 then
  S9.                     done = 1
                    else
  S10.                    i = i + 1
                    endif
              endif
        endwhile
  S11.  print sum
end Sums
```





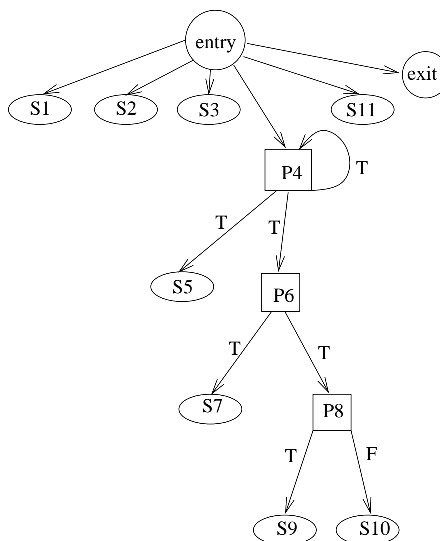


Figure 1: Procedure `Sums`, its control-flow graph in the upper right, its control-dependence graph with regions in the lower left, and its control-dependence graph without regions in the lower right.

We then performed linear regression analysis to fit a line to the data, and computed the correlation coefficient to determine how well the regression line fit the data. Our results support the earlier conclusion that, in practice, control-dependence graph size is linear in the number of program statements, and suggest that this relationship holds for general-purpose, as well as scientific, programs. In addition, our studies utilized two popular varieties of control dependence graphs; our results show that the linear relationship persists over both varieties of graphs, and that neither variety offers significant space savings over the other.

2 Control Dependence Graphs

A control-dependence graph encodes the control dependencies in a procedure or monolithic program. Control dependence is defined in terms of control-flow graphs and the postdominance relation [9]. Given a control-flow graph augmented with unique *entry* and *exit* nodes, and nodes W and V in that graph, W is *postdominated* by V if every directed path from W to the exit (not including W or exit) contains V.[1] For example, Figure 1 depicts procedure `Sums` in the upper left of the figure and its control-flow graph in the upper right of the figure; labels in the control-flow graph correspond to statement numbers in the program. In the control-flow graph, S11 postdominates all other nodes, P8 postdominates S7, and P6 postdominates S5, P4 postdominates S1, S2, S3, S7, P8, S9, and S10.

For statements (nodes) X and Y in a control flow graph, Y is *control dependent* on X if and only if (1) there exists a directed path P from X to Y with all Z in P (excluding X and Y) postdominated by Y and (2) X is not postdominated by Y. A control-dependence graph encodes the control dependencies for a program. The nodes in a control-dependence graph represent single statements. The control-dependence graph for `Sums` is shown in the lower right of Figure 1. A control-dependence graph contains several types of nodes. Statement nodes, shown as ellipses, represent simple statements. Predicate nodes, depicted as squares, correspond to statements from which two edges may originate. Node numbers labeling statement and predicate nodes correspond to statement numbers in the program.

Ferrante, Ottenstein, and Warren [9] give a method for inserting *region* nodes that summarize nodes with identical control dependencies. The control-dependence graph for `Sums` with region nodes inserted is shown in the lower left of Figure 1. Region nodes, shown as circles in the figure, group nodes that share the same control dependencies. For example, nodes S7 and P8 are each control dependent on P6-true; region node R3 represents this control dependency.

3 Empirical Results

The objective of our study was to investigate the relationship between program size and control dependence graph size for two popular varieties of control dependence graphs, over a sample population of C functions of various sizes, drawn from a variety of programs. Our hypothesis was that, for these functions, we could demonstrate a linear relationship between program and control dependence graph size, irrespective of the variety of programs involved. Evaluation of the hypothesis required tools for constructing control dependence graphs, and a sample population of C functions.

To construct control dependence graphs, and gather data about those graphs and about our subjects, we used the `Aristotle` analysis system [12]. `Aristotle` processes C source code, and computes various types of information for use in code-based analysis tools. One of these code-based analysis tools uses a function's control-flow graph to construct a control-dependence graph with region nodes for the function, following the algorithm given in [9]. We also implemented a tool that takes a control-dependence graph with region nodes as input, and creates the corresponding control-dependence graph without region nodes.

[1] This definition of postdominance does not include the initial node on the path; thus, a node never postdominates itself.

Program Name	Number of Functions Analyzed	Avg. Number of Exec. Stmts	Description
agrep	62	44.26	Fast grep Unix utility
aristotle	149	33.46	Aristotle analysis system
bash	112	13.93	Unix bourne again shell
finger	65	23.15	Unix finger utility
gcc	404	24.60	GNU C compiler
gdb	413	17.82	GNU C debugger
make	146	28.63	Unix make utility
miscellaneous	170	15.32	Collection of functions
player	582	29.70	Player module from Empire internet game
mcc	182	18.16	Modular and extensible C compiler
tcsh	130	23.45	Unix C shell
xvcg	732	37.08	Compiler graph visualizer

As a sample population of subjects for our studies, we collected 3147 functions from a variety of C programs. Table 3 summarizes information about these subjects.[2] For each program, the table lists the program name, the number of functions in that program that we analyzed, the average number of lines of code in the functions analyzed, and a brief description of the program. To calculate the size of functions, we counted executable source code statements only: we did not count blank lines, comments, preprocessor directives, or data declarations.

We offer the following additional descriptions of our subjects. `agrep` is a fast regular expression parser. `aristotle` is a program analysis system – the same system we used in these studies – that provides static and dynamic analyses of C programs; it consists of a user interface, parsers, code instrumenters, and various analysis tools. `bash` is version 1.12 of the bourne again Unix shell. `finger` is version 1.37 of a tool that provides user information. `gcc` is version 2.3.3 of the GNU C compiler; included in `gcc` are functions that perform tasks such as register allocation and target code generation. `gdb` is version 4.16 of the GNU C debugger. `make` is version 3.68 of a utility that assists in the maintenance, modification, and regeneration of programs. `miscellaneous` summarizes a number of small C programs, such as euclid.c, that have been used as subjects in studies of testing techniques. `player` is one module of the Internet game `Empire`, that we have used in studies of regression test selection techniques. `mcc` [18] is an experimental C compiler written to facilitate evaluation of machine dependent code optimizations. `tcsh` is version 6.04 of a Unix C shell. `xvcg` is version 1.30 of a compiler graph visualizer.

Given the foregoing tools and C functions, we conducted two studies. The first study compared the number of executable statements in each C function to the size of the control-dependence graph, without region nodes, for that function. The second study compared the number of executable statements in each C function to the size of the control-dependence graph, with region nodes, for that function. Sizes of control dependence graphs were measured in terms of the count of total nodes and edges. Each study yielded a set of 3147 ordered pairs. In these pairs, the first coordinate represents the number of executable statements in a function, and the second coordinate represents the size of the control-dependence graph for that function. We plotted these ordered pairs on a scatter graph – one for each study. We used Microsoft Excel to fit regression curves to the data and to calculate the square of the Pearson product moment correlation coefficient, or r^2. r^2 is a dimensionless index that ranges from zero to 1.0, inclusive, and reflects the extent of the linearity between two data sets. An r^2 value of 1.0 indicates a perfectly linear relationship with no noise, whereas an

[2] All data that we used for our study is available on request.

r^2 value of zero indicates a random relationship.

Figure 2 contains a scatter graph that shows, for each C function, the number of executable statements versus the size of the control-dependence graph without region nodes. The computed regression line can be represented by the equation $y = 2.0452x - 0.452$, where x is the number of executable statements, and y is the size of the control-dependence graph without region nodes. The square of the Pearson product moment correlation coefficient for the linear regression trendline is 0.9935.

Figure 3 contains a scatter graph that shows, for each C function, the number of executable statements versus the size of the control-dependence graph with region nodes. The computed regression line can be represented by the equation $y = 2.7787x + 6.505$, where x is the number of executable statements, and y is the size of the control-dependence graph with region nodes. The square of the Pearson product moment correlation coefficient for the linear regression trendline is 0.9593.

In drawing conclusions from the above data, although the set of functions analyzed in our studies was inclusive of many types of functions, our results support predictions only for functions whose types lie within our categorical bounds. In particular, our results are supported with confidence only for C functions of at most 200 executable statements; our sample population did not contain a sufficient number of functions above this size to support conclusions. However, the fact that we located only a small number of functions with size above 200 indicates that such functions may be rare in practice.

Given these considerations, our results clearly demonstrate a linear relationship between the number of executable statements in a function and the size of its control-dependence graph. The high square of the Pearson product moment correlation coefficient reflects the high linearity of this relationship. Moreover, these results hold both for graphs that contain region nodes and for graphs that do not. These results support our hypothesis that, in practice, control-dependence graph size is linear in the number of executable statements in a program.

Acknowledgments

This work was partially supported by grants from Microsoft, Inc. and by NSF under Grant CCR-9357811 to Ohio State University. Michael Gray gathered some of the data for this study.

References

[1] H. Agrawal, R. DeMillo, and E. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the Symposium on Testing, Analysis and Verification*, pages 60–73, October 1991.

[2] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 348–357, September 1993.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–56, June 1990.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[5] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50, November 1992.

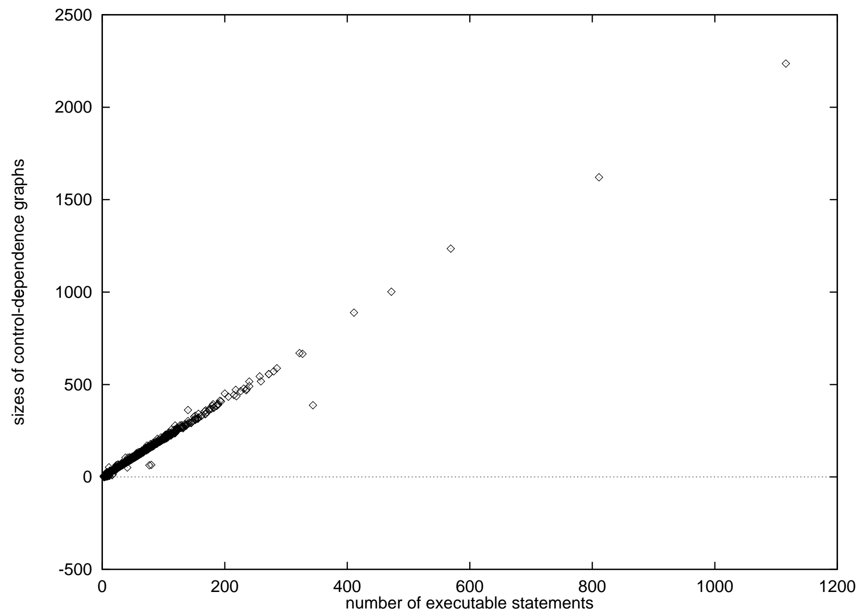


Figure 2: Scatter graph containing a point for each C function; the horizontal axis represents numbers of executable statements, and the vertical axis represents the sizes of control-dependence graph without region nodes.

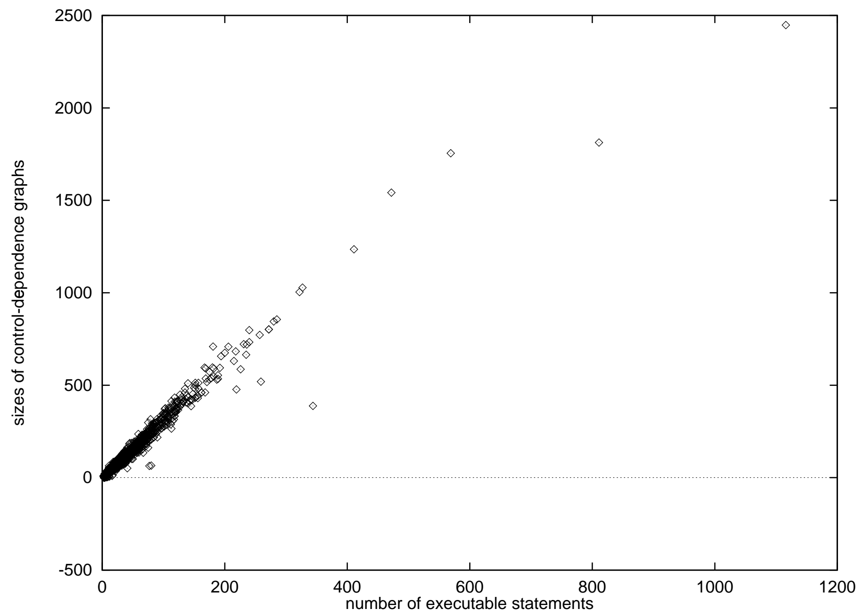


Figure 3: Scatter graph containing a point for each C function; the horizontal axis represents numbers of executable statements, and the vertical axis represents sizes of control-dependence graph with region nodes.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K . Zadeck. An efficient method of computing static single assignment form. In *POPL89*, pages 25–35, January 1989.

[7] R. Cytron, J Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, June 1990.

[8] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Second Irvine Software Symposium* , pages 131–145, March 1992.

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

[10] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 299–308, November 1992.

[11] R. Gupta and M. L. Soffa. Employing static information in the generation of test cases. *Journal of Software Testing, Verification and Reliability*, 3(1):29–48, December 1993.

[12] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: A system for the development of program-analysis-based tools. In *Proceedings of the 33rd Annual Southeast Conference*, pages 110–119, March 1995.

[13] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[14] B. Korel. The program dependence graph in static program testing. *Information Processing Letters*, 24:103–108, January 1987.

[15] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 358–367, September 1993.

[16] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

[17] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – Eispack Guide.* Springer-Verlag, 1976.

[18] J. M. Smith. MCC: A modular and extensible C compiler. Master's thesis, Clemson University, August 1995.