An Empirical Study of Static Call Graph Extractors*

Gail C. Murphy, David Notkin and Erica S.-C. Lan

Department of Computer Science & Engineering
University of Washington,
Box 352350
Seattle, WA 98195-2350 USA
{gmurphy, notkin, sclan}@cs.washington.edu

Abstract

Informally, a call graph represents calls between entities in a given program. The call graphs that compilers compute to determine the applicability of an optimization must typically be conservative: a call may be omitted only if it can never occur in any execution of the program. Numerous software engineering tools also extract call graphs with the expectation that they will help software engineers increase their understanding of a program. The requirements placed on software engineering tools when computing call graphs are typically more relaxed than for compilers. For example, some false negatives—calls that can in fact take place in some execution of the program, but which are omitted from the call graph—may be acceptable, depending on the understanding task at hand. In this paper we empirically show a consequence of this spectrum of requirements by comparing the C call graphs extracted from three software systems (mapmaker, mosaic, and gcc) by five extraction tools (cflow, CIA, Field, mkfunctmap, and rigiparse). A quantitative analysis of the call graphs extracted for each system shows considerable variation, a result that is counterintuitive to many experienced software engineers. A qualitative analysis of these results reveals a number of reasons for this variation: differing treatments of macros, function pointers, input formats, etc. In this paper, we describe and discuss the study, sketch the design space, and discuss the impact of our study on practitioners, tool developers, and researchers.

*This research was funded in part by National Science Foundation grant CCR-9506779, in part by a Canadian NSERC postgraduate scholarship, and in part by a University of Washington Department of Computer Science & Engineering Educator's Fellowship.

1 Introduction

Various software tools compute *call graphs*, which represent calls between entities in a given program. For example, compilers sometimes compute call graphs to determine whether specific optimizations can be applied. As another example, numerous software engineering tools extract call graphs with the expectation that they will help software engineers increase their understanding of a program.

A call graph is a binary relation over selected entities defined in a program: for example, over procedures, over modules, over files, etc.[1] Although we often discuss "the call graph" of a program, there are a number of reasonable interpretations of the actual relation a call graph represents.

Perhaps the most pleasing definition of a call graph is the relation describing exactly those calls made from one entity to another in any possible execution of the program. Unfortunately, computing this relation is undecidable. A less pleasing, but easily computed, relation is the cross product of all entities in the program, which simply assumes that any entity can call any other entity in the program. In practice, tools that compute call graphs that fall into different parts of this spectrum.

Compilers place one set of requirements on call graphs.[2] The most conspicuous requirement is that

[1] Call graphs are sometimes treated not as relations but as multigraphs [1, 2, 4, 6, 7, 12, 15].

[2] Call graphs have been used for interprocedural analysis and optimization for over two decades [1]. Early results constrained the programs for which call graphs could be computed. Ryder loosened some of these restrictions, considering limited forms of procedure parameters [17]. Callahan *et al.* extended Ryder's work to handle recursion [4], while Hall and Kennedy increased the ability to handle some assignments to procedure variables [8]. Lakhotia further extended this work to handle additional kinds of assignments to procedure parameters [12].

their call graphs must be *conservative*: a call can be omitted only if it can never occur in any execution of the program. Without this requirement, a compiler might apply an optimization that does not preserve the semantics of the source program.

Software engineering tools place a different—and in some ways more relaxed—set of requirements on call graphs, since the call graphs are most often consumed by humans for the purpose of understanding. As one example, a software engineering tool might not be required to compute conservative call graphs: some false negatives—calls that may take place in some execution of the program, but which are omitted from the relation—may be acceptable, depending on the understanding task at hand. As another example, although a compiler is likely to compute a call graph in terms of expanded macros, a software engineer might instead prefer a call graph computed in terms of unexpanded macros.

In this paper we empirically show a consequence of this variance in requirements: call graphs extracted by several broadly distributed software engineering tools vary—indeed, they vary significantly enough to surprise many experienced software engineers. Furthermore, these tools do not state the requirements they satisfy, leaving the engineers with little sense of the dimensions of approximation of the extracted call graph. Our study focuses solely on call graphs that are computed by statically analyzing the program itself, rather than by analyzing the program's run-time behavior.

This paper makes three contributions.

- It presents the results of an empirical study that compares static call graph extractors.
- Based on a qualitative analysis of the results, it sketches a design space for call graph extractors.
- It considers the impact of this design space on practitioners, tool developers, and researchers.

2 Empirical Study

We gathered five tools that each extract a call graph from C [11] source code, applied the extractors to three target software systems, and analyzed the results both quantitatively and qualitatively.

The five call graph extractors used in the study were:

- CIA, the C Information Abstraction System [5],
- rigiparse, a front-end to the Rigi reverse engineering system [14],
- the Field programming environment [16],
- cflow, a tool distributed with most Unix systems, and
- mkfunctmap, a Perl script developed by James Hoagland.

We chose these tools because they are readily available, because they can produce a textual list of all calls extracted from a body of C code, and because they all run on the same platform (a Sparc running SUNOS 4.1.3_U1). The first four tools are parser-based, while mkfunctmap is lexically-based.

We chose three publicly available target software systems—`mapmaker`, `mosaic`, and `gcc`—that represent a variety of application domains. The target systems, along with their sizes, are described in Table 1. Version information for the tools and for the analyzed target systems is provided in Appendix A.

2.1 Method

The procedure used in the study was as follows. Each software system was compiled to determine the include paths and defines necessary to produce a working executable. For each software system, we then applied each of the five tools to extract a list of the calls between C functions from the C, Yacc [10], and Lex [13] source code comprising the system. Scripts were run on the output produced by the tools to transform the extracted calls list to the form:

(function1;function2)

where function1 calls function2. For each pair of tools, A and B, the calls lists, $calls_A$ and $calls_B$, were then compared to determine the set intersection, $calls_A \cap calls_B$, and the set differences, $calls_A - calls_B$ and $calls_B - calls_A$, between the output produced by the two tools. These quantitative results were studied, and samples from the intersection and difference sets were qualitatively analyzed.

To apply the tools fairly required making decisions about the source from which calls would be extracted. This was complicated by two factors. First, many of the systems create header and source files as part of the compilation process; we had to make decisions about which of these files to include when extracting calls. Second, the tools present a range of choices for specifying the source files of interest. Some tools, like cflow, accept a set of file names. Other tools, like Field, work either on information included in an executable (compiled and linked with the debug switch)

Software System	Lines of Code	Lines of Non-blank, Non-comment code
`mapmaker`—a molecular biology application	46,153	22,929
`mosaic`—a world-wide web browser	66,562	46,578
`gcc`—the GNU C compiler	251,738	169,131

Table 1: The Software Systems Studied

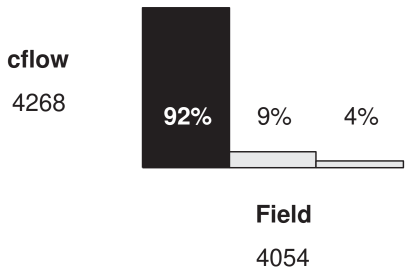
or on directories. To ensure the tools received the same input files, we had to rename or remove source files in the distribution that were not to be included in the extraction. For example, we removed the source for platforms other than the Sparc when extracting from the `gcc` source code.

Details on the data collection appear in Appendix B.

2.2 Quantitative Results

Figures 1, 2, and 3 summarize the quantitative results obtained from applying the call graph extractors to the `mapmaker`, `mosaic`, and `gcc` software systems, respectively. (The raw data on which the figures is based are available in Appendix C.) Each table presents a pairwise comparison between the call graphs extracted by the five tools. The number of calls extracted by each tool appears below the name of the tool in the column and row labels.

An entry for a pair of tools (A is the tool in the row and B is the tool in the column) consists of three bars. For example, the entry comparing the results of cflow (which found 4268 calls) and Field (which found 4054 calls) for `mosaic` is:



The first bar reports the percentage of the calls found by each of the two tools as given by:

$$\lceil (\#(calls_A \cap calls_B) \,/\, max(\#calls_A, \#calls_B)) \times 100 \rceil.$$

For this example, this means that of the calls extracted by cflow, at most 92% of them were also extracted by Field. The second bar reports the percentage of calls unique to the tool in the row as given by:

$$\lceil (\#(calls_A - calls_B) \,/\, \#calls_A) \times 100 \rceil.$$

For this example, this means that about 9% of the calls extracted by cflow were not extracted by Field. The third bar reports the percentage of calls unique to the tool in the column as given by:

$$\lceil (\#(calls_B - calls_A) \,/\, \#calls_B) \times 100 \rceil.$$

For this example, this means that about 4% of the calls extracted by Field were not extracted by cflow.

The figures point out a few notable results:

- No two tools extracted the same set of calls for any given system. This is evident given that there is no entry where either set difference (shown by the second and third bars) is zero.
- In a few cases, tools extracted a large set of calls, none of which overlapped with other tools. In the case of the `mapmaker` source code, the cflow and mkfunctmap tools each extracted hundreds of calls, yet the intersection of the two call sets was empty (as shown by the 0% first bar in the entry for the cflow and mkfunctmap tools in Figure 1).
- The tool that extracted the most calls varied with the source code considered. This can be seen from the differing numbers of calls extracted by each tool in the figures.

2.3 Qualitative Results

Studying the quantitative data alone can be deceptive. First, it would be misleading to believe that extracting more calls indicates better results, since additional calls may well represent false positives rather than true positives. Second, quantitatively reporting

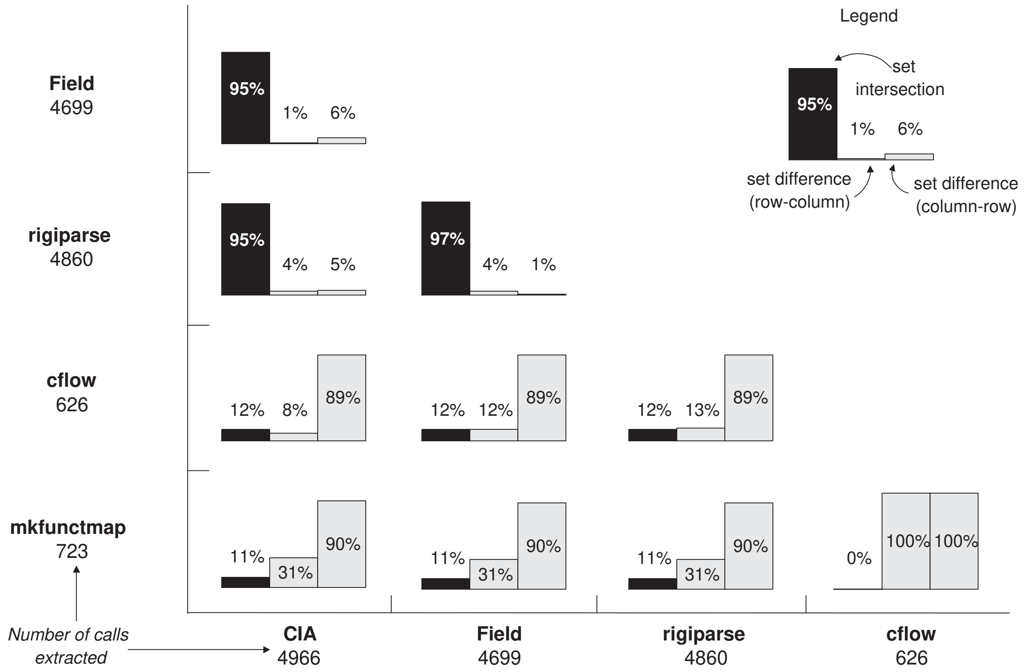


Figure 1: Quantitative results from `mapmaker`. The pairwise comparison of the call graphs extracted by the mkfunctmap and cflow tools (lower right) show that although each tool extracts hundreds of calls, none of the calls overlap.

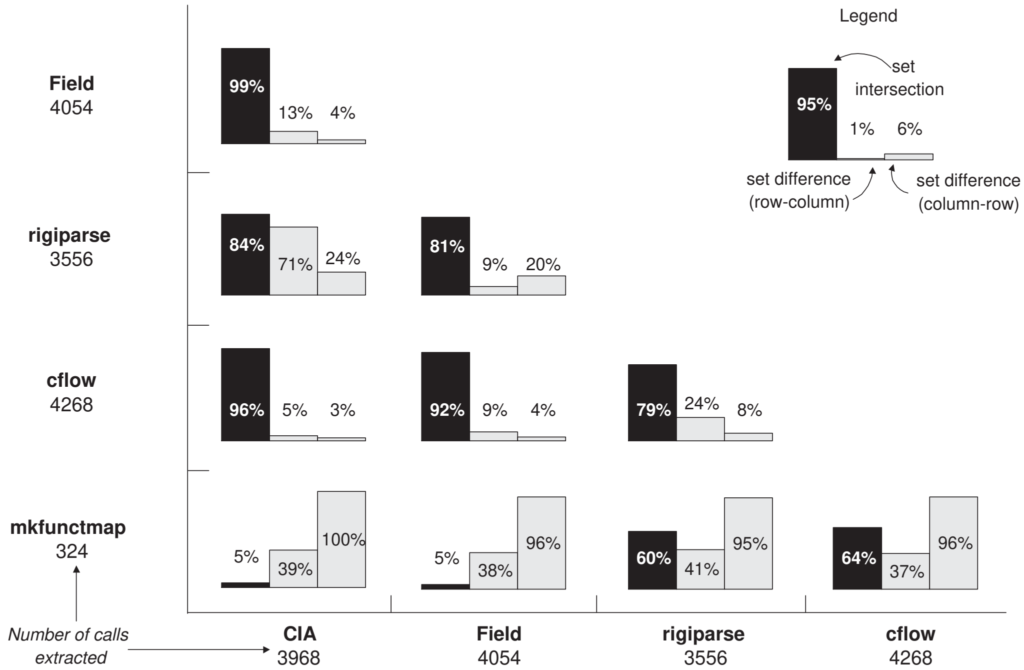


Figure 2: Quantitative results from `mosaic`. As in Figure 1, these data indicate that no two tools extract the same set of calls (i.e., no set difference is 0%).

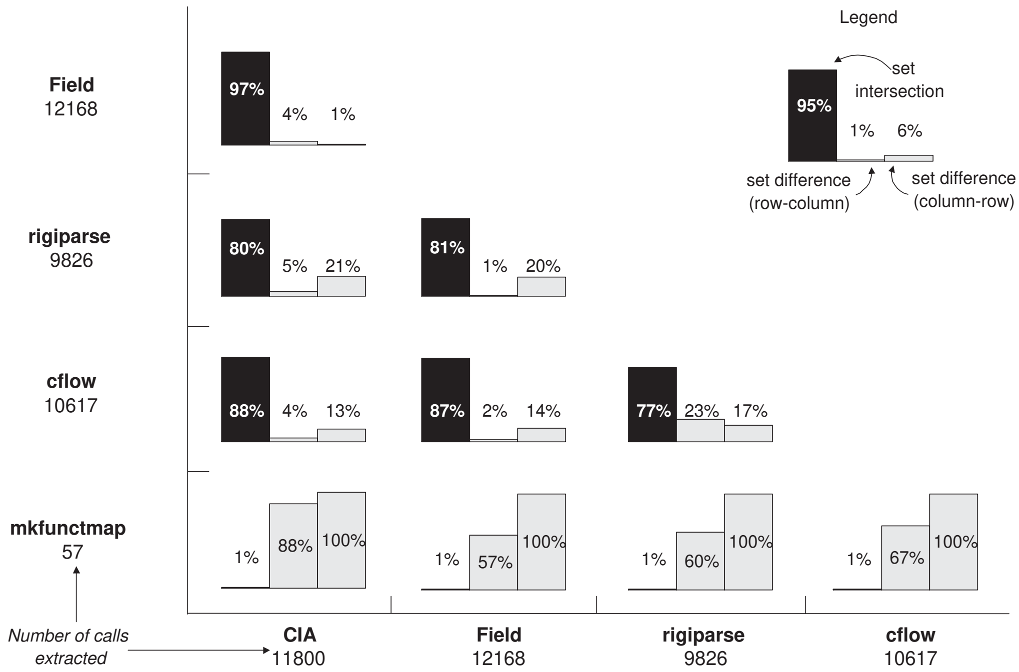


Figure 3: Quantitative results from `gcc`. Comparing these data to Figures 1 and 2 indicates that the percentage of calls extracted by various tools depends on the characteristics of the systems analyzed. For example the rigiparse tool found 95% of the calls extracted by the CIA tool for the `mapmaker` system in Figure 1, but for the `gcc` system shown here, the rigiparse tool extracted only 80% of the calls extracted by CIA.

the exact number of false positives and false negatives produced by each tool is impossible since determining the "true" call graph is undecidable.

Consequently, to better understand the differences indicated by the quantitative comparison, we sampled the comparisons of the extracted call sets and inspected the source code related to the calls. From this sampling, we determined that all of the tools sometimes generate false positives, and all of the tools sometimes generate false negatives.

To identify false negatives, we first found tuples that appeared in the extraction of another tool, and then inspected the source code to determine whether the tuples represented false negatives in one tool or false positives in the other tool. Some of the sampled false negatives raise some interesting points.

- CIA's extraction of the calls from `mosaic` missed the call (EVJotMove;AddStroke) that was found by mkfunctmap. This is noteworthy since mkfunctmap found only about 8% (324 vs. 3968) of the calls in `mosaic` that CIA found.

- Field, when run on `mapmaker`, missed the call (main;banner), apparently because main is not defined with a return type. Editing the definition of main to add a return type allows Field to extract the call. Not all functions, however, need be declared with a return type for Field to extract calls. For example, a small test case of defining a function without a return type did not result in a false negative.

- Most tools—with the exception of CIA and cflow—do not attempt to handle calls through function pointers, leading to false negatives. All of the target software systems in the study have at least some calls through function pointers.

False positives were identified analogously to false negatives. Some of the notable false positives include:

- CIA apparently judged that a parameter (chrom) was a function pointer when it extracted the tuple (attach_this;chrom) from `mapmaker`.

- The call (check_format_info;isdigit), reported by cflow when run on `gcc`, apparently occurs because of inconsistent treatment of macro expansion.

- Field generated a number of false positives such as (CBChangeRadio;XtVaGetValues) from `mosaic`, apparently by ignoring a define statement.

- In `mapmaker`, mkfunctmap mistook forward declarations for definitions, generating false positives such as (apportion;poission_add).

- For `mapmaker`, rigiparse sometimes returned calls to macros and sometimes returned calls to expanded macros (e.g., the macro UNMETA was expanded in the file `readline.c`, but not in the file `vi_mode.c`, which does not include the appropriate header file).

3 Design Decisions

Although these empirical results report on a few selected extractors, they provide insight into the collection of design decisions that the developer of any call graph extraction tool must make. One way to characterize these decisions is to consider call graph extraction as a function $extract\_cg(s) = cg$ that maps source (s) into a set of calls (cg). The tool developer must make tradeoffs concerning the kind and condition of the source that the tool will accept as input (the domain of the function), and the algorithm for extracting and computing the calls graph (the behavior of the function).

3.1 Domain

One key decision that a developer of an extractor must make is the form of the input to the tool. The tools we studied allowed engineers to provide either a list of file or directory names (containing C source, object code, etc.) or else an executable file (compiled and linked with the debugging switch). Only Field supports both approaches.[3] An alternate approach, not taken by any of our selected extractors, would be to use a makefile to specify the input.

A second decision is how to handle macro preprocessing. Of the tools we studied, only the lexically-oriented mkfunctmap tool extracts call between entities in the original source, rather than between entities in the postprocessed code. A related issue is how the tool handles code generated by other programs. For example, Lex and Yacc files almost always contain C code. Are the resulting calls reported in terms of the source before or after generation? Only mkfunctmap reports the calls in terms of the source before generation. If the appropriate extensions are used, cflow will automatically generate and extract from the resulting files.

A third decision is to determine what, if any, additional constraints are placed on the input. The most obvious constraint is the language variant: for C, for example, does the tool extract from K&R C, ANSI C, or C's for various PC's? Another class of constraints includes static restrictions on the source text. For example, cflow produces no output unless all input files pass the first phase of the C static program checker, lint.[4]

Together, these decisions on how to specify input and on any additional constraints on the source code affect the tool users as well as the tool developers. For example, a tool may be more useful to an engineer during a port of a system to another platform if extraction is done in terms of unpreprocessed source code. As another example, a tool that extracts from an executable system may be more useful to an engineer writing test plans.

3.2 Behavior

The behavior of an extraction tool is primarily defined by the analysis algorithms it uses. Determining the algorithms used by inspecting the source of the tools is extremely difficult, since the algorithmic code is neither small nor isolated within the source. A search of the literature is not sufficient either:

> While many interprocedural data-flow problems are based on a call multigraph [1, 2, 6, 7, 15] or a relation between pairs of procedures indicating possible call sequences [3], few authors discuss how it is constructed from the source program [4, p. 484].

Inferring some algorithmic aspects, however, is feasible. Simple tests can be used, for instance, to determine whether or not tools try to handle function pointers. Of the tools we studied, only CIA and cflow attempt to determine the results of some uses of function pointers in the code.

Another aspect of behavior is whether the call graph extraction algorithm is local, reporting call information on a file-by-file basis, or whether it is global, resolving call information between files. All of the

[3] We performed a comparison of Field's two approaches on `mosaic`. We expected that the executable-based extracted call graph should be a subset of the directory-based extracted call graph. But, unexpectedly, if the executable-based extracted call graph is considered as the column entry, and the directory-based extracted call graph is considered as the row entry, the leftmost bar would show 85%, the center bar would show 16%, and the rightmost bar would show 7%.

[4] This constraint varies from version to version of cflow; for example, the DEC OSF/1 V3.2 version of cflow uses a modified lint pass that relaxes some of the input constraints.

tools we considered could report local call information. Only two of the tools, rigiparse and mkfunctmap, could not produce global information as well.

4 Discussion

Call graphs extracted by software engineering tools are intended to aid engineers in performing various engineering tasks. The degree to which a given extractor succeeds in aiding an engineer depends on a number of properties.

Does the extractor have constraints acceptable to the engineer? If the constraints are too strong then the engineer will have to either modify the input or else use another tool with weaker constraints. For example, the requirement of the cflow tool that input pass a lint check may be too strong for an engineer to use when modifying many large legacy systems.

Are the design decisions made clear to the engineer? Ideally, the design decisions should be made explicit in the documentation of the tool. In practice, however, most tools make explicit few, if any, decisions, with the engineer generally being responsible for inferring specific properties. Engineers develop inferences primarily based on comparing the actual output to the expected output. Gaps between the actual and expected output affect an engineer's confidence in the tool.

Looking at various kinds of false positives clarifies this point. The engineer can quite easily identify some tuples as false positives, while simultaneously understanding why the tool might have generated them. For example, by looking at the source, an engineer may easily determine that CIA was incorrect in judging chrom to be a function pointer. Such false positives tend to help the engineer infer how the tool works, without necessarily decreasing confidence in the extractor.

In contrast, false positives that arise from inconsistent interpretations of syntactic constructs tend to decrease the engineer's understanding and confidence in the extractor. For instance, rigiparse and cflow may treat macros differently through the source if the preprocessor is unable to complete without warning or error. This may in turn confuse the engineer, perhaps decreasing the engineer's understanding of and confidence in the extractor.

Several of the tools produce erroneous results by mistaking syntactic constructs. An engineer who sees a large number of erroneous results may have decreased confidence in the tool, and will, in any case, have a hard time inferring what is supposed to be extracted.

5 Conclusion

Abstractly, call graph extractors seem fairly straightforward to develop. The engineering aspects, however, are quite complex. As confirmed by our empirical study, this leads to tools that, in practice, return significantly different results when passed the same source programs as input.[5]

The choices made by the developers of these tools represent different points in a design space for call graph extractors. None of these choices are inherently "bad." Even mkfunctmap, which quantitatively looks like a poor tool, provides specific design decisions not handled by the other tools—in particular, computation in terms of the unexpanded macros.

Little is known, however, about the points in the design space of call graph extractors that are useful to practitioners. Joint collaborations between practitioners and researchers are needed to identify the software engineering tasks that could benefit from call graph information and the appropriate design space choices for tools to support the extraction of that information. With this information, the necessary research and development can be done to provide the needed tools. In addition, tool developers need to more effectively communicate the design decisions that specific extractors have made so that practitioners can select an appropriate extractor based on the needs imposed by the particular task they are performing.

One limitation of our study is that we considered call graph extractors for only one language, C. It may be that determining call graph extraction for C is uncommonly difficult compared to many other languages. For example, the separation of macro expansion from the C language per se places pressures on C call graph extractors that are less likely to arise for languages such as Ada, Modula-2, and many others. Similarly, there may be more variants of C than for many other languages, which adds additional pressures. On the other hand, object-oriented languages require that the tool developer make design decisions not considered for C call graph extractors. For example, a tool developer must determine how to handle dynamic binding.

Although this paper only considers one kind of information, call graphs, many of the observations about

[5] Unexpectedly inconsistent computations from long-lived, widely available, and broadly used programs have been documented in seismic data processing as well [9].

design decisions apply to extractors of other kinds of information—such as inheritance structures, file dependences, references to global variables, etc.—as well.

Acknowledgments

We wish to thank the authors of all the call graph extractor tools used in the study for making their tools available to us. Robert Bowdidge and Michael Van-Hilst provided valuable comments on drafts of this paper, as did several of the anonymous reviewers. Alan Borning suggested the use of bar charts in the presentation of the quantitative data.

References

[1] F.E. Allen. Interprocedural Data Flow Analysis. In *Proceedings Information Processing 74 (Software)*, pages 398–402. North-Holland, August 1974.

[2] J.P. Banning. An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pages 29–41. ACM Press, January 1979.

[3] J.M. Barth. A Practical Interprocedural Data Flow Analysis Algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[4] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the Procedure Call Multigraph. *IEEE Transactions on Software Engineering*, SE-16(4):483–487, April 1990.

[5] Y.-F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, SE-16(3):325–334, March 1990.

[6] K. Cooper and K. Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 247–258. ACM Press, June 1984.

[7] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 58–67. ACM Press, June 1986.

[8] M.W. Hall and K. Kennedy. Efficient Call Graph Analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[9] L. Hatton and A. Roberts. How Accurate is Scientific Software? *IEEE Transactions on Software Engineering*, SE-20(10):785–797, October 1994.

[10] S.C. Johnson. Yacc—Yet Another Compiler Compiler. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[11] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[12] A. Lakhotia. Constructing Call Multigraphs using Dependence Graphs. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages*, pages 273–284. ACM Press, January 1993.

[13] M.E. Lesk. Lex—A Lexical Analyzer Generator. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[14] H.A. Müller and K. Klashinsky. Rigi—A System for Programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Computer Society Press, April 1989.

[15] E. Myers. A Precise Inter-procedural Data Flow Algorithm. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, pages 219–230. ACM Press, January 1981.

[16] S.P. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, 1995.

[17] B.G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

A Version Information

The following summarizes the version (and availability) information for the tools used:

- The version of CIA used was "CIA 10/1/94", as reported by executing `cia -h`. CIA is available from http://www.research.att.com:80/orgs/ssr/book/reuse/. The version used in the study was downloaded on June 26, 1995.
- The version of rigiparse used was obtained in December 1994 from the authors of rigi. For information on obtaining rigiparse, contact hausi@csr.uvic.edu.
- The version of Field used was downloaded in April, 1995 from the semi-anonymous ftp site at wilma.cs.brown.edu.
- The version of cflow used was the executable distributed with the Unix distribution for the Sparc SUN OS 4.1.3_U1 (GENERIC) #2. The only available documentation is through Unix man pages. Note also that cflow performs quite differently on different platforms.

- The version of mkfunctmap used was a slight modification of Version 1.0, changed to output a normalized calls format. mkfunctmap is available from J. Hoagland at hoagland@cs.ucdavis.edu.

The version information for the systems we analyzed is:

- MAPMAKER/EXP 3.0 and MAPMAKER/QTL 1.1,
- mosaic Version 2.4 (for the X Window System), and
- gcc-2.7.0.

B Running the Tools

The following describes characteristics of running the tools common across the software systems studied:

- Field: The xrefdb tool of Field was used to generate a program database. The `INCLUDE_PATH` environment variable was set to any necessary include directories (see the description for each tool). The `XRDB_FLAGS` environment variable was set to the value of any necessary defines (see the description for each tool). The created database was then queried with `(C.from, C.call);` to produce a textual list of the calls between functions.
- rigiparse: The `RIGICPP` environment variable was set to use the C preprocessor distributed with the SPARCCompiler Version 2.0.1. The environment variable was also set to contain any necessary includes and defines.
- cflow: cflow was run with the `-r` option over C source code files as well as any necessary defines and includes.
- CIA: CIA was run with the `-c` option and any necessary includes and defines to create a `.A` file for each C source code file. The `.A` files were then linked using CIA and the database queried using `cref func - func -`.
- mkfunctmap: The input C source code files were provided as input to the mkfunctmap script.

The following describes the source code considered when producing call graphs for each system, and details the defines and include paths used:

- `mapmaker`: The `mapmaker` software is structured into four subdirectories: lib, mapm, quant and (the GNU) readline. This software, when compiled, produces two executables: `mapmaker` and `qtl`. The extractor tools were run after make had been called in the `mapmaker` directory (resulting in the creation of a makehelp.c file in the lib subdirectory). The include path used was `mapmaker:mapmaker/lib`. The defines used were `_SYS_SUNOS`.
- `mosaic`: The `mosaic` software considered as part of this study is structured into four subdirectories: libXmx, libhtmlw, libwww2 and src. The include paths used were: `<path to motif 1.2 directory>:mosaic:mosaic/libXmx:mosaic/libwww2`. The defines used were: `MOTIF` and `XMOSAIC`.
- `gcc`: The `gcc` software considered as part of this study were the C and Yacc source files in the main `gcc` directory (as distributed), and the C source files in the config/sparc directory. The include pathrs used were: the `gcc` directory and the config subdirectory. The defines used were: `IN_GCC` and `USE_COLLECT2`.

C Raw Data

Tables 2, 3, and 4 contain the raw data collected from running the call graph extraction tools. Each entry contains three values. The value prefixed by $\cap$ is the number of calls in the intersection of the tools; the value prefixed by $R - C$ is the number of calls resulting from subtracting the calls found by the tool in the column from the calls found by the tool in the row; the value prefixed by $C - R$ is the number of calls resulting from subtracting the calls found by the tool in the row from the calls found by the tool in the column.

	CIA	Field	rigiparse	cflow
mkfunctmap	$\cap = 504$ $R-C = 219$ $C-R = 4456$	$\cap = 505$ $R-C = 218$ $C-R = 4194$	$\cap = 502$ $R-C = 221$ $C-R = 4358$	$\cap = 0$ $R-C = 723$ $C-R = 626$
cflow	$\cap = 578$ $R-C = 48$ $C-R = 4382$	$\cap = 552$ $R-C = 74$ $C-R = 4147$	$\cap = 550$ $R-C = 76$ $C-R = 4310$	
rigiparse	$\cap = 4709$ $R-C = 151$ $C-R = 251$	$\cap = 4670$ $R-C = 190$ $C-R = 29$		
Field	$\cap = 4688$ $R-C = 11$ $C-R = 272$			

Table 2: Raw Data for `mapmaker` system.

	CIA	Field	rigiparse	cflow
mkfunctmap	$\cap = 198$ $R-C = 126$ $C-R = 3956$	$\cap = 201$ $R-C = 123$ $C-R = 3853$	$\cap = 192$ $R-C = 132$ $C-R = 3364$	$\cap = 206$ $R-C = 118$ $C-R = 4062$
cflow	$\cap = 4056$ $R-C = 212$ $C-R = 98$	$\cap = 3917$ $R-C = 351$ $C-R = 137$	$\cap = 3279$ $R-C = 989$ $C-R = 277$	
rigiparse	$\cap = 3305$ $R-C = 251$ $C-R = 849$	$\cap = 3250$ $R-C = 306$ $C-R = 804$		
Field	$\cap = 4003$ $R-C = 51$ $C-R = 151$			

Table 3: Raw Data for `mosaic` system.

	CIA	Field	rigiparse	cflow
mkfunctmap	$\cap = 7$ $R-C = 50$ $C-R = 11793$	$\cap = 25$ $R-C = 32$ $C-R = 12143$	$\cap = 23$ $R-C = 34$ $C-R = 9803$	$\cap = 19$ $R-C = 38$ $C-R = 10598$
cflow	$\cap = 10279$ $R-C = 338$ $C-R = 1521$	$\cap = 10491$ $R-C = 126$ $C-R = 1677$	$\cap = 8181$ $R-C = 2436$ $C-R = 1645$	
rigiparse	$\cap = 9376$ $R-C = 450$ $C-R = 2424$	$\cap = 9799$ $R-C = 27$ $C-R = 2369$		
Field	$\cap = 11711$ $R-C = 457$ $C-R = 89$			

Table 4: Raw Data for `gcc` system.