An Empirical Study of Static Call Graph Extractors

GAIL C. MURPHY
University of British Columbia
DAVID NOTKIN
University of Washington
WILLIAM G. GRISWOLD
University of California—San Diego
and
ERICA S. LAN
Microsoft Corporation

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 that compute 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 article, 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 nine tools (cflow, cawk, CIA, Field, GCT, Imagix, LSME, Mawk, 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

This article is as revised and expanded version of a paper presented at the 18th International Conference on Software Engineering (ICSE-18), March 1996. This research was funded in part by National Science Foundation grants CCR-9506779 and CCR-9508745, a Canadian NSERC postgraduate scholarship and research grant, a University of Washington Department of Computer Science and Engineering Educator's Fellowship, and a grant from Fujitsu Laboratories, Ltd.
Authors' addresses: G. C. Murphy, Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, British Columbia Canada V6T 1Z4; email: murphy@cs.ubc.ca; D. Notkin, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195–2350; email: notkin@cs.washington.edu; W. G. Griswold, Department of Computer Science and Engineering, University of California—San Diego, San Diego, CA 92093–0114; email: wgg@cs.ucsd.edu; E. S. Lan, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399; email: erical@microsoft.com.

of reasons for this variation: differing treatments of macros, function pointers, input formats, etc. The fundamental problem is not that variances among the graphs extracted by different tools exist, but that software engineers have little sense of the dimensions of approximation in any particular call graph. In this article, we describe and discuss the study, sketch a design space for static call graph extractors, and discuss the impact of our study on practitioners, tool developers, and researchers. Although this article considers only one kind of information, call graphs, many of the observations also apply to static extractors of other kinds of information, such as inheritance structures, file dependences, and references to global variables.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Call graphs, design space, empirical study, software system analysis, static analysis

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.[2] 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. Unfortunately, this relation is seldom, if ever, useful. In practice, tools that compute call graphs fall into different parts of this spectrum.

Compilers place one set of requirements on call graphs.[3] The most conspicuous requirement is that their call graphs must be *conservative*: a

[1]Call graphs are sometimes treated not as relations but as multigraphs [Allen 1974; Banning 1979; Callahan et al. 1990; Cooper and Kennedy 1984; Cooper et al. 1986; Lakhotia 1993; Myers 1981].

[2]It is straightforward to reduce this problem to the halting problem.

[3]Call graphs have been used for interprocedural analysis and optimization for over two decades [Allen 1974]. Early results constrained the programs for which call graphs could be computed. Ryder [1979] loosened some of these restrictions, considering limited forms of procedure parameters. Callahan et al. [1990] extended Ryder's work to handle recursion, while Hall and Kennedy [1992] increased the ability to handle some assignments to procedure variables. Lakhotia [1993] further extended this work to handle additional kinds of assignments to procedure parameters.

call can be omitted only if it can never occur in any execution of the program. Without this requirement, a compiler might, for example, 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 program-understanding task at hand. As another example, although a compiler will compute a call graph on source with all macros expanded, a software engineer retargetting a software system might instead prefer a call graph computed on source without the macros expanded.

A consequence of the relaxation of requirements is that the call graphs extracted by different software engineering extraction tools will vary. Even though some variances are anticipated, we believe software engineers have two reasonable expectations about most extractors. One expectation is that extractors built from similar technology will extract similar call graphs. For instance, an engineer might reasonably expect that the call graphs extracted by tools that parse the source will be similar. A second expectation is that a tool will behave similarly on different software systems, extracting call graphs with the same kinds of characteristics across different systems. For example, if the call graph extracted by a tool for a particular system is conservative, it is reasonable to expect that the same tool will extract conservative call graphs on other target systems.

The use of several call graph extraction tools as part of our research in program understanding and transformation led us to question these implicit expectations. To investigate the variances occurring among different tools, we performed an empirical study that compared the results of applying nine different software engineering tools for extracting call graphs from C source code to three sample software systems. Our study showed that the call graphs extracted by these different software engineering tools vary (indeed, they vary in more dimensions than expected, surprising many experienced software engineers). The fundamental problem is not that the variances exist, but that engineers have little sense of the dimensions of approximation in any particular call graph. It is likely that this problem is not limited to C call graph extractors, but applies to extractors of other relations in other languages as well.

In this article, we describe the study (Section 2) and present a quantitative and qualitative analysis of the results of the study (Sections 3 and 4). Using the analysis, we sketch a design space for static call graph extractors (Section 5). We then discuss the ramifications of the study on practitioners, considering how a practitioner might use the design space information to help choose an appropriate extractor to aid a particular software engineering task (Section 6). We conclude the article (Section 7) with a discussion of

Table I. The Tools Studied

Tool	Description
cawk_{cg}	cawk_{cg} is an instantiation for C of the TAWK system [Griswold et al. 1996]. TAWK supports regular-expression-based querying of the abstract syntax tree of a program.
GCT	The Generic Coverage Tool [Marick 1994] can report on branch coverage, multicondition coverage, loop coverage, etc., for C programs.
Imagix	Imagix is a commercial software-understanding system intended to help software engineers reverse-engineer or document large programs.
Rigiparse	Rigiparse is a front-end to the Rigi reverse-engineering system [Müller and Klashinsky 1988]
Field	Field is a programming environment [Reiss 1995] tool. The tool within the Field environment used in this study is called xrefdb.
cflow	The cflow tool extracts external references for C programs. This tool is distributed with most Unix systems.
CIA	The C Information Abstraction System [Chen et al. 1996] extracts structural information information from C programs and supports the subsequent querying of that information.
LSME_{cg}	The lexical source model extractor tool [Murphy and Notkin 1996] permits engineers to extract information from source based on patterns consisting of hierarchical regular expressions.
Mawk_{cg}	This tool is a variant of awk [Aho et al. 1979] that supports patterns as data (see M. Brennan's Unix Manual Page, Dec. 1994, "Mawk—pattern scanning and text processing language").

a few of the different ways in which the problem raised by this study may be attacked by tool developers and researchers.

2. EMPIRICAL STUDY

The empirical study we performed focused solely on call graphs that are computed by statically analyzing the program itself, rather than by analyzing the program's run-time behavior.

2.1 Overview

To perform this study, we (1) gathered nine software engineering tools that can each extract a call graph from C source code, (2) applied the extractors to three target software systems, and (3) analyzed the results both quantitatively and qualitatively.

Table I summarizes the nine extractors used in the study. Some tools required configuration scripts to extract call information; the names of these tools are annotated with *cg*.

These nine tools represent an exhaustive list of the tools that were available, at reasonable cost, meeting two important criteria: the ability to produce a textual list of all calls extracted and the ability to run on the Sparc platform running the SunOS 4.1 Unix operating system.[4] Surpris-

[4]This platform was chosen because of the large number and kinds of call graph extractors available for it. The specific version of the operating system used was SunOS 4.1.3_U1.

Table II. The Software Systems Studied

Software System	Lines of Code	Lines of Nonblank, Noncomment Code
`mapmaker` (a molecular biology application)	31,349	24,541
`mosaic` (a World Wide Web browser)	69,492	48,236
`gcc` (the GNU C compiler)	287,133	201,488

ingly, many commercial tools, such as CenterLine Inc.'s CodeCenter, support the querying of an extracted call graph only by specific function names through a user interface. Since we required a textual list of the calls for comparison, we could not include these tools in our study.

This set of nine tools forms a suitable collection on which to base the study because they represent a spectrum of technologies and approaches available for building a call graph extractor. Five of the tools—cawk, xrefdb, GCT, Imagix, and Rigiparse—are syntactic extraction tools, extracting information based on a parse of the program. The CIA and cflow tools are also syntactic tools, but they extract a function-refers-to-function relation rather than a calls relation. If calls through function pointers are not considered, it is reasonable to expect that the function-refers-to-function relation is a superset of the call relation. The LSME and Mawk tools differ from the other seven in extracting information using a lexical scan of the source. Although these two tools are both lexical, they differ in the granularity at which regular expressions to match in the source are expressed: Mawk uses character-based regular expressions whereas LSME uses token-based regular expressions; this allows, for example, Mawk to decide on matches based in part on indenting conventions.

We chose three publicly available target software systems—`mapmaker`, `mosaic`, and `gcc`—to use as the source to input to the call graph extractors. The target systems, along with their sizes, are described in Table II. Applying the extractors to three different systems from a variety of application domains allows us to investigate whether and how the call graphs extracted by a particular extractor vary with the source analyzed.

Version information for the tools and for the analyzed target systems is provided in Appendix A.

2.2 Method

The procedure used in the study was as follows. Each target software system was compiled to determine the `include` paths and the `defines` necessary to produce a working executable. For each software system, we then applied each of the nine tools to extract a list of the calls between functions from the C 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.

2.2.1 *Input Preparation.* Applying 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 created 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, can be run on individual C files. Other tools, like Imagix, modify the makefile [Feldman 1978] for the target system, introducing new targets for analysis based on common makefile structuring conventions. For instance, the default Imagix analysis target assumes the symbol OBJECTS is defined to describe the objects required to link the executable for the system. The Field tool exemplifies a third category: Field works either from file information included in an executable (compiled and linked with the debug switch) or from the directory structure.

We took two steps to try to ensure that all tools received the same input. First, we captured the output of a successful compilation run using `make` on each system. If a call graph extraction tool accepted individual C files as input, we created a script to run the tool based on the output of the compilation process. Only two tools could not be run this way: Imagix and Field. For Imagix, we edited the system makefiles it modified to ensure the appropriate list of source files was provided as input to the Imagix extractor. In the case of Field, we compiled each system with the debug switch and provided the executables to Field as input. Second, we applied each tool to preprocessed C source.[5] For tools that allowed the preprocessor to be specified, we used the `cpp` tool distributed with SunOS to eliminate differences that might arise among the preprocessors distributed with various tools.

2.2.2 *Selecting a Baseline for Comparison.* In an earlier study of static call graph extractors [Murphy et al. 1996], we performed a pairwise comparison of the call graphs extracted by various tools. A pairwise comparison was used because we were interested in investigating the behavior of the various extractors as they might normally be used. For instance, an engineer applying a call graph extractor as part of a reverse-engineering task would likely apply a tool, such as cflow, on all C source files in the directories comprising the software system. Since the inputs varied among tools, it was not reasonable to pick one extractor to form a baseline.

In this extended study, we were interested in investigating the behaviors of the extractors when they were provided as similar input as possible. A common basis of input that could be provided to all extractors was preprocessed input as described above. Given these study parameters, it is reasonable to choose a definition of a call graph to use as a baseline for the

[5]In this article, we use the term "preprocessed source" to refer to the source resulting from running the original source through the C preprocessor.

comparison of all call graphs extracted. The definition we chose as a baseline is the call graph consisting of all calls found during a static parse of the C source code where the parse is sufficiently complete to permit compilation of the program.

To collect the baseline call graphs for each target system, we used the output of the GCT test coverage tool. GCT is built on top of the GNU C compiler distribution and provides, among other functionality, the ability to instrument the call sites in a system before passing the instrumented code to the compiler. As part of the instrumentation process, the GCT tool produces a map file that describes the call sites in a system. It is straightforward to write an awk script to convert the call site information in the map file into the call graph format used in our study. GCT can thus be used to produce a call graph, for a compilable system, based on information statically extracted from the system's source.

The call graphs extracted by GCT form a reasonable baseline for comparison in this study because

(1) they are derived using a parser designed to compile the code and

(2) they include information about call sites involving function pointers.

These points help increase the confidence that all call sites recognized by the compiler are recognized by the extractor.

2.2.3 *Data Gathering*. We collected the data for the study by iteratively applying the method described above. Iterations occurred for three reasons. First, comparisons of extracted call graphs were used to tailor the scripts specified to recognize calls by the lexical tools, LSME_{cg} and Mawk_{cg}. A numerical subscript is used when reporting the results from the LSME and Mawk tools to indicate when different scripts were used with these tools.[6] Second, multiple passes were needed to resolve differences in the files scanned by various tools. Finally, the method was applied iteratively to allow for upgrades to the tools being studied that fixed data reporting bugs. For example, in gathering data using the Imagix tool, we discovered a bug in the database-loading procedure used by Imagix; the calls were extracted by the Imagix parser and stored in the intermediate files, but the calls were not being included in the textual output from the tool. We allowed the vendor to correct this data output problem.

As a final note, the data analyses appearing in the next two sections also reflect a simple treatment of the data to remove from consideration the majority of the calls through function pointers. As part of the iterative application of the method, it was determined that two tools report calls

[6]Six different LSME scripts were used: one script for each of the unprocessed and preprocessed sources for each of the three systems. The slight variations in the scripts result from a need to recognize differences in the use of macros and embedded calls in the three systems. Two different Mawk scripts were used: a separate script was needed for the unprocessed source of `mosaic` because of a use of macros in the definition of functions. Further details on the scripts are provided in Appendix B.

through function pointers: GCT and cawk$_{cg}$. The filtering on the call graphs reported by these two tools was performed because the tools report the sites involving calls through function pointers in different ways: GCT often simplifies the reporting by using the notation `*<...>`, while cawk$_{cg}$ reports the expression involved (e.g., `*((resolveImageProc)hw->html-.resolveImage)`). This reporting difference means a simple lexical comparison of call graphs containing the calls through function pointers can be misleading. The filtering we performed consisted of removing calls involving a function with an asterisk in its name from the extracted call graph. These call sites were considered as part of the qualitative analysis and are reported on in Section 4. Based on the qualitative analysis, this simple filtering process caught all but approximately four differences in the reporting of calls through function pointers.

Details on the data collection appear in Appendix B.

2.2.4 *Comparing the Call Graphs.* For each target system, we compared the call graphs extracted by each of the software engineering tools to the call graph extracted using GCT. The call graphs were compared by computing the set intersection, $calls_{GCT} \cap calls_{Tool}$, and set differences, $calls_{GCT} - calls_{Tool}$ and $calls_{Tool} - calls_{GCT}$, where $Tool$ refers to one of the eight extractors studied other than GCT. These computed sets were then studied both quantitatively and qualitatively.

3. QUANTITATIVE RESULTS

Two quantitative analyses were performed on the extracted call graphs. First, we present a quantitative comparison of each of the graphs extracted by the eight software engineering tools to the baseline graphs (Section 3.1). Then, to provide a sense of the similarity and differences among the extracted call graphs, we report on the frequency distribution of the number of calls reported by different combinations of tools (Section 3.2).

3.1 Quantitative Comparison to the Baseline

Figure 1 presents a quantitative comparison of the extracted call graphs to the baseline call graphs for each of the three target systems: `mapmaker`, `mosaic`, and `gcc`. (The raw data on which the graphs are based are provided in Appendix C.) Each bar in each figure is divided into three parts:

—the black portion of the bar represents the intersection between the call graph extracted by the software engineering tool and the baseline graph;

—the white portion represents the calls in the baseline graph that were not reported by the software engineering tool; and

—the gray portion represents the calls reported by the software engineering tool that were not in the baseline graph.

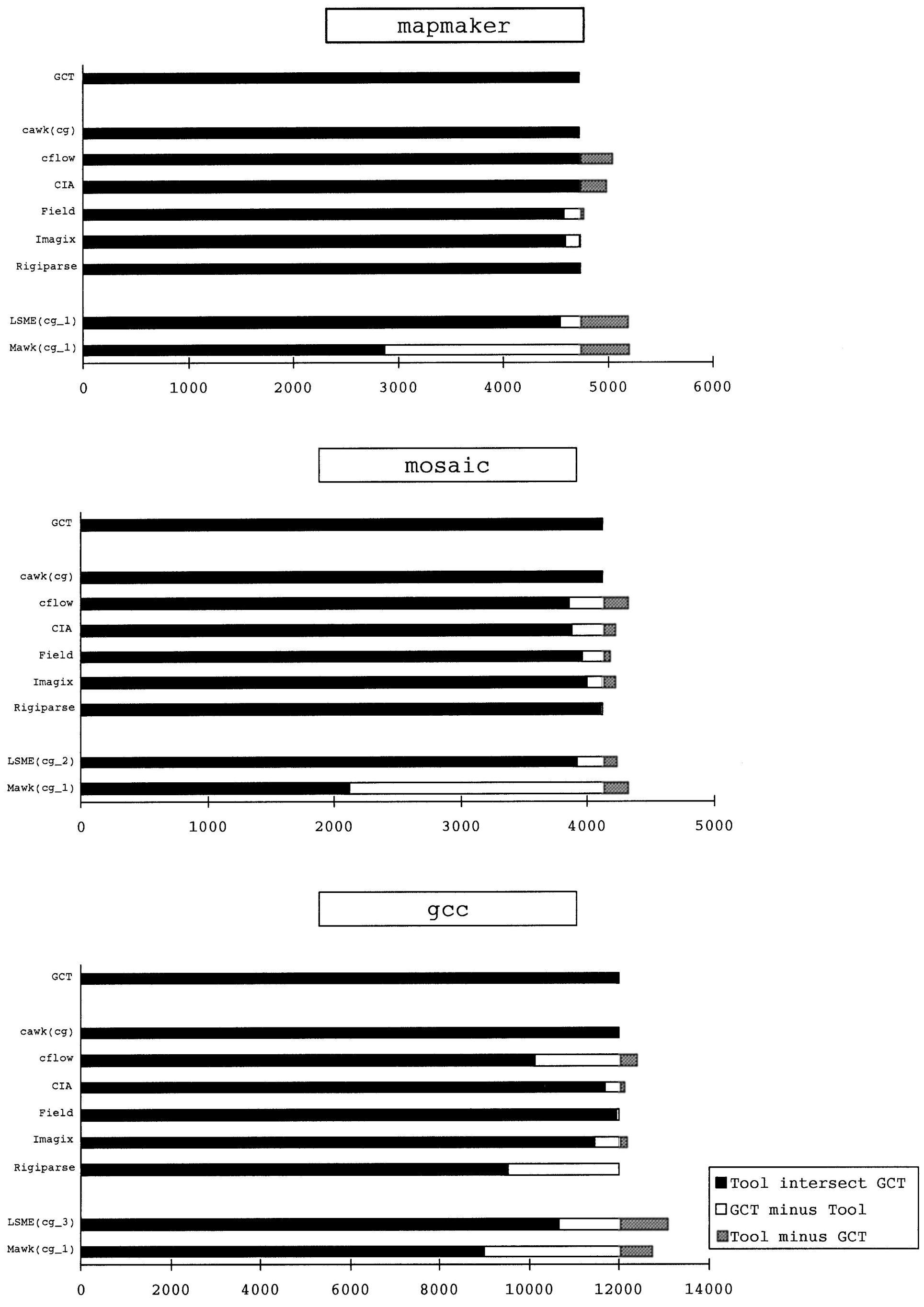


Fig. 1. Quantitative comparison of extracted call graphs to the baseline GCT graph.

Since the bars in Figure 1 explicitly represent the calls in the baseline graph that were not reported by the software engineering tool (i.e., the white portion of the bar), the total length of a bar represents the number of calls in the union of the baseline graph and the graph for the particular tool. The data in Appendix C include the number of calls reported by each tool. The three graphs in Figure 1 highlight a few notable results:

—Only the cawk$_{cg}$ tool extracted essentially the same call graph as the baseline graph on all systems. (We discuss the few discrepancies between the call graphs extracted by these two tools in Section 4.)

—The characteristics of the call graphs extracted by five of the syntactically based tools—cflow, CIA, Field, Imagix, and Rigiparse—varied considerably between the three systems. For instance, Rigiparse, which reported 0 false positives[7] and 13 false negatives for the `mosaic` system, reported 2 false positives and 2,449 false negatives for the `gcc` system. The Field system, which reported 7 false positives and 59 false negatives for the `gcc` system, reported 52 false positives and 165 false negatives for the much smaller `mosaic` system.

—As described above, the CIA and cflow tools both extract the "references between functions" relation rather than the "calls between functions" relation. Since the references relation includes tuples for functions being passed as parameters, e.g., to callbacks, it was expected that the CIA- and cflow-extracted call graphs would contain false positives, but not false negatives. This expected result held only for the `mapmaker` system.

—The LSME$_{cg}$ and Mawk$_{cg}$ tools both attempt to recognize syntactic constructs in source based on lexical patterns. It was thus expected that the call graphs they extracted would contain false positives where the patterns match unexpected constructs and false negatives where the patterns miss pertinent constructs. The data in Figure 1 show that this expected result held. For at least one system, `mosaic`, the LSME$_{cg}$ tool was able to perform similarly to one of the syntactic tools: LSME$_{cg}$ extracted a call graph containing 95 false positives and 206 false negatives compared to the CIA tool which extracted a call graph containing 90 false positives and 243 false negatives.

As described in Section 2.2, to ensure a fair comparison we made a number of decisions about the input provided to the tools. Some of the decisions we made, such as ensuring the use of a particular preprocessor, the SunOS `cpp` preprocessor, may not be representative of how the tools are typically used. To provide a sense of how extracted call graphs may vary in more typical circumstances, Figure 2 presents a comparison of some call graphs that result when the input to the extractors is not tightly constrained. The figure compares call graphs extracted from unprocessed source and from source preprocessed with a tool other than the SunOS

[7]In all discussions of the results, the terms "false positive" and "false negative" represent differences in a call graph with respect to the baseline call graph and should not necessarily be construed to be errors.

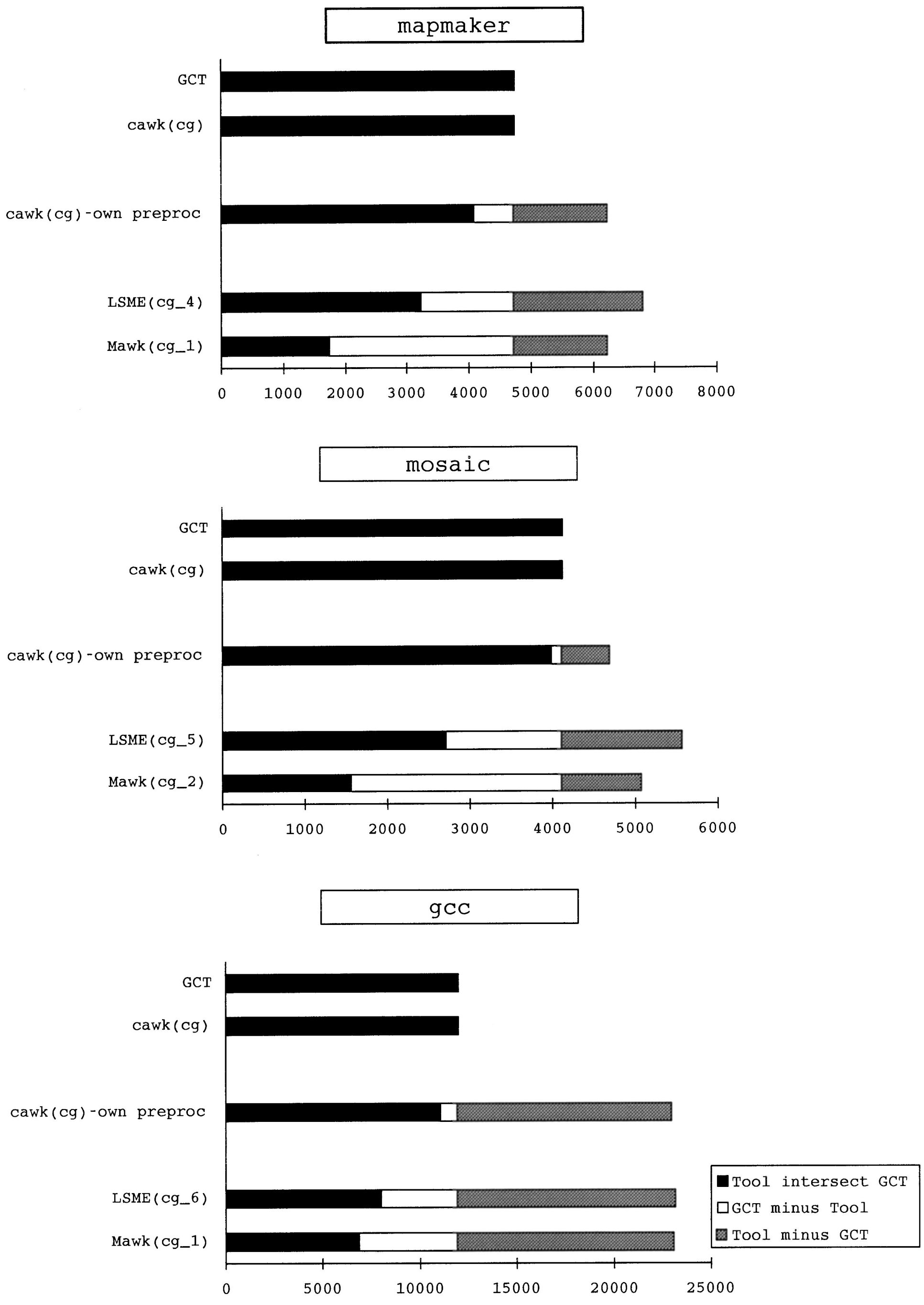


Fig. 2. A comparison of graphs extracted with different input restrictions.

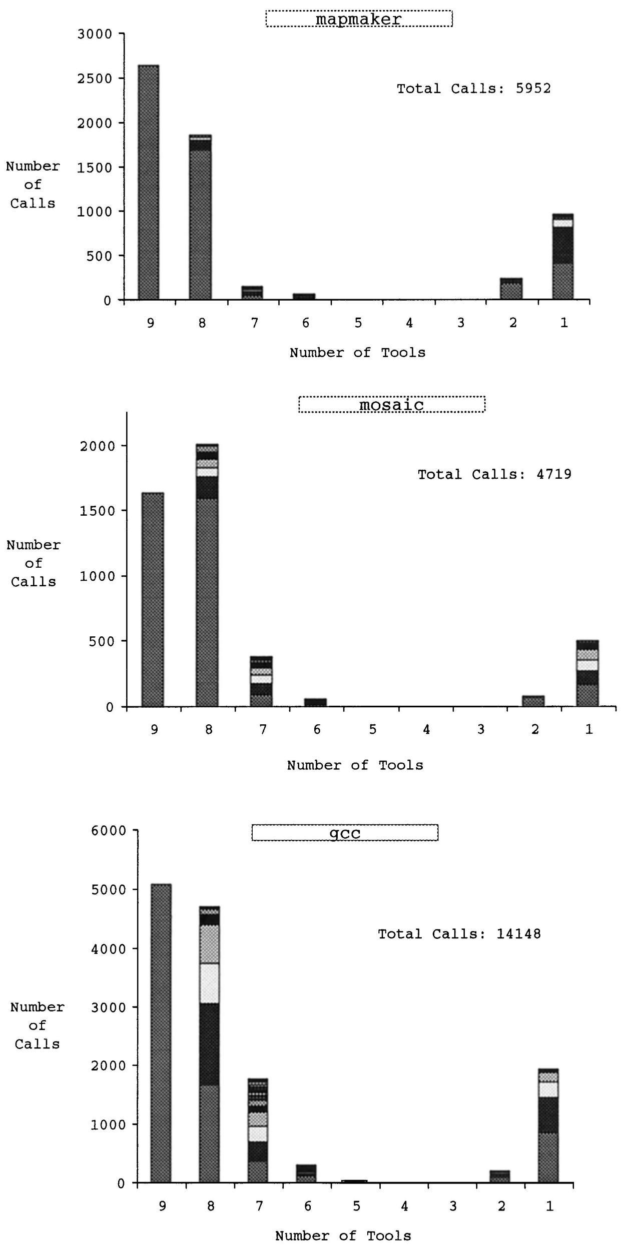
preprocessor, cpp, to some of the graphs described earlier. The two lexical tools—LSME$_{cg}$ and Mawk$_{cg}$—were used to extract graphs from unprocessed source. The cawk$_{cg}$ tool with its own preprocessor was used to represent the extraction of a graph from source preprocessed with a tool other than the SunOS preprocessor. For comparison, the baseline graphs and the call graphs extracted using cawk$_{cg}$ with the SunOS preprocessor from Figure 1 are also included.

Not surprisingly, the two lexical tools report more false positives and false negatives when given unprocessed source. The false positives are often the result of misinterpreting a macro as a call when the macro actually defines statements not containing calls, such as statements accessing data. If the macro does include a call statement, a false negative will also result. The cawk$_{cg}$ tool also reports more false positives and false negatives when using its own preprocessor rather than the SunOS `cpp` preprocessor. The cawk$_{cg}$ tool uses an extended C syntax that includes the `#define` preprocessor directive and a modified preprocessor that leaves C-like macro directives and invocations in the source code. Thus, it will create false positives and negatives in situations similar to the lexical tools, except that macros that are not C-like will be expanded.

3.2 Quantitative Comparison of Overlap among the Graphs

Comparing the call graphs extracted by each tool to the baseline graph provides some insight into the nature of each graph, but it does not tell us much about the similarities and differences among all the extracted graphs. To provide insight into these similarities and differences, we determined, for each call returned by any tool (including both false positives and false negatives), the number of other tools also reporting that call. Figure 3 summarizes the results of this computation. Each bar in the charts in Figure 3 reports on a combination of a specific number of tools. The height of each bar shows the number of calls reported by a particular number of tools. For instance, the leftmost bar of the top chart shows that over 2,500 calls extracted from the `mapmaker` system were reported by all nine extractors. The rightmost bar of the same chart shows that approximately 1,000 calls were reported by only one tool.

The height of each bar in Figure 3 tells only part of the story about the overlap between the extracted graphs. It is also important to ask whether it is always the same collection of a particular number of tools reporting overlap. For example, do the bars representing the calls reported by two tools always correspond to the lexical tools? To investigate this aspect of overlap, we also determined how many calls were reported by particular combinations of tools. The stratification of the bars in Figure 3 shows these data: each gray-scale value represents a different combination of tools. The tables on the right side of each bar chart clarify the number of different combinations of tools shown in the bar charts. For example, the table for the `mapmaker` system shows that five combinations of eight tools reported some overlap; the bar chart shows that one combination of the tools dominates with an overlap of over 1,500 calls. As another example, the table for the `gcc` system shows that a call reported by seven of the tools



# Tools Reporting a Call	Combinations of x Tools
9	1
8	5
7	8
6	5
5	1
4	1
3	2
2	5
1	6

# Tools Reporting a Call	Combinations of x Tools
9	1
8	7
7	11
6	9
5	2
4	1
3	0
2	3
1	6

# Tools Reporting a Call	Combinations of x Tools
9	1
8	7
7	17
6	15
5	5
4	0
3	1
2	6
1	9

Fig. 3. Quantitative comparison of all extracted call graphs for mapmaker, mosaic, and gcc.

could have been reported by any of 17 different combinations of seven tools. The graphs in Figure 3 highlight a few notable results.

—For all three target systems, most calls are found by almost all the tools or very few of the tools. Somewhat surprisingly, there is no obvious trend in the groupings of the tools based on the extracted calls. For instance, although most of the calls reported for the mosaic system are found by either seven to nine of the tools or only one or two of the tools, there are 11 combinations of seven tools reporting some overlap, and seven combinations of eight tools reporting call graphs with some overlap.

—Only in the gcc case is there quantitative evidence of all nine tools reporting unique calls. However, upon inspection, it was determined that the cawk$_{cg}$ and baseline call graphs do indeed overlap completely. The number of tools reporting unique calls for gcc should, based upon the qualitative investigation reported on in Section 4, be seven. If this qualitative information was accounted for in Figure 3, then, the right-most bar in the chart for gcc would show seven tool combinations rather than nine.

In the case of the other two target systems, the call graph extracted by at least one other tool completely overlapped with the graphs extracted by cawk$_{cg}$ and GCT; the tool with the complete overlap differed in each case.

—For the gcc system, there are seven combinations of eight tools reporting overlap with four of these combinations reporting hundreds of calls in the overlap.

4. QUALITATIVE RESULTS

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. Moreover, the false positive and false negative identification was performed with respect to a baseline that has not been definitively shown to be conservative, since determining the “true” call graph is undecidable. Consequently, to better understand the differences indicated by the quantitative comparison, we also analyzed the extracted call graphs qualitatively.

In Section 4.1, we present the results of a qualitative comparison of the extracted call graphs to the baseline graphs. In Section 4.2, we assess the stability of the various extractor tools across the three target systems through a qualitative categorization of the extracted graphs.

4.1 Qualitative Comparison to the Baseline

To investigate the false negatives, we first found tuples that appeared in the set difference between the baseline graph and the graph extracted by another tool, say cflow. We then inspected the source code to determine whether the tuples represented false negatives in the graph reported by cflow, or false positives in the baseline graph extracted with GCT. In all cases sampled, the calls represented false negatives in the graph reported by a non-GCT tool. Some of the sampled false negatives raise some interesting points.

—CIA's extraction of the calls from gcc missed calls, such as (yyparse; build_function_call), from a yyparse function in the c-parse.c file that were found by seven of the tools, including the Mawk$_{cg}$ tool (which found several thousand fewer calls than CIA).

—The call graphs reported by a number of the syntactic tools include false negatives even when the tool does not report a syntax error in processing the C source files. For instance, the CIA tool did not report a syntax error when processing the c-parse.c file. As another example, the cflow tool often missed function definitions when processing the mosaic source, leading to false negatives when call sites occurring after the missed function definition were incorrectly associated with the wrong caller.

—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.

—It was only in the case of the gcc system that the cawk$_{cg}$ tool reported false negatives. The 21 false negatives were the result of differences in the format of reporting calls through function pointers and, interestingly, the result of a bug in the SunOS cc compiler that dropped the first argument after any undefine flag.

False positives were investigated analogously to false negatives, except the examined tuples were sampled from the set difference between a graph extracted by a particular tool and the baseline graph. Before sampling the false positives for qualitative analysis, we first adjusted for the difference between the function-refers-to-function relation extracted by the CIA and cflow tools and the desired calls between the functions relation. After filtering the false positives reported by these two tools for tuples belonging only to the function-refers-to-function relation, we found that CIA does not report any false positives for mosaic or gcc, and reports 1 false positive for mapmaker, whereas cflow reports 89, 108, and 308 false positives for mapmaker, mosaic, and gcc, respectively. We then proceeded with the analysis of the false positives. Some of the notable false positives include the following:

—Field generated a number of false positives such as (CBChangeRadio; XtVaGetValues) from mosaic, apparently by ignoring or mishandling a preprocessor directive to conditionally exclude code.[8]

[8]The Field tool, xrefdb, used to extract a call graph does not permit the specification of the C preprocessor to use. Field was thus selecting the preprocessor to apply, which may have been different than the SunOS cpp tool.

—In `mapmaker`, LSME$_{cg}$ mistook forward declarations placed in the middle of a file for call sites, generating false positives such as (insert_unsorted-_map;kosa_add).

—LSME$_{cg}$ also reports a number of false positives for `mosaic` which result from the use of printf debug statements of the form: `printf("CCI-_free()");`.

—CIA apparently judged that a parameter (chrom) was a reference to a function when it reported the tuple (attach_this;chrom) from `mapmaker`. Although there is a function chrom in the `mapmaker` system, at this call site, chrom appears to be an integer variable.

—When processing the source for `mosaic`, Imagix apparently does not expand the `include` files correctly, resulting in false positives to an X Window System symbol, XtDisplay, which is defined as both a macro and as a function call. For instance, the call, (BulletRefresh,XtDisplay), reported by Imagix is a false positive, since in this case XtDisplay refers to a macro.

—Rigiparse reported false positives, such as (actual_hazard_this instance; function_units), when extracting from `gcc`, apparently as a result of the parser continuing after encountering syntax errors while processing the file `sched.c`.

—Similar to the case of false negatives, the cawk$_{cg}$ tool produced false positives only in the case of the `gcc` system. The three false positives were due to differences in the reporting of calls through function pointers not caught by the simple filtering described in Section 2.

The comparisons of the call graphs extracted by the eight software engineering tools to the baseline graphs show that a software engineer cannot rely on information about the underlying technology of an extractor as a predictor of the behavior of the extractor. The five syntactic extractors, for example, produce call graphs with varying numbers and kinds of false positives and false negatives. The expectation that extractors built from similar technology will extract similar call graphs thus does not hold.

4.2 Qualitative Categorization of the Graphs

A qualitative analysis of samples from the graphs provides insights into the reasons for the differences among the graphs. However, it does not provide any indication of the kind of reasoning a software engineer may perform based on the call graph. If, for instance, a software engineer knows that a call graph does not contain any false negatives, the engineer may be more confident in using the information as the basis to perform a testing task on the system. As another example, when performance tuning a system, an engineer may desire a call graph that does not contain any false positives. But, how does a software engineer know if a call graph extracted with a

Table III. Tool Categories

	No False Positives	False Positives
No False Negatives	Precise	Conservative
False Negatives	Optimistic	Approximate

particular tool has the appropriate characteristics? Do tools extract call graphs with similar characteristics across different systems?

To investigate this question, we categorized the tools for each system based on whether or not the graphs extracted by the tools contain false positives and false negatives. We then assessed stability by comparing the categorization of each tool across the three target systems. As before, false positives and false negatives are defined with respect to the baseline call graph.

In this comparison, the baseline call graphs could be categorized as precise because no false positives or false negatives were identified when the baseline graphs were compared to the graphs extracted by the other eight tools.[9] This evidence, while strong, is not enough to characterize these graphs as precise for several reasons. One, there may well be false negatives. Two, we have no evidence of what calls are actually made in some execution stream at run-time. And three, we have only analyzed three target systems. As a result, we designate the baseline call graphs as conservative. To simplify the discussion, we assign names to each of the possible categories as shown in Table III.

Four tools—Field, Imagix, LSME$_{cg}$, and Mawk$_{cg}$—produce an approximate call graph for each system. The remaining four tools produce different kinds of call graphs, depending on the system analyzed:

—The cawk$_{cg}$ tool produces conservative call graphs for the `mapmaker` and `mosaic` systems and an approximate call graph for the `gcc` system. However, as discussed earlier, the false positives and the false negatives reported were the result of differences in reporting calls through function pointers and a bug in the SunOS `cc` compiler. From a qualitative perspective, the cawk$_{cg}$ tool could be considered to produce a conservative call graph for each system analyzed.

—The cflow tool produces a conservative call graph for the `mapmaker` system and approximate call graphs when analyzing the `mosaic` and `gcc` systems.

—The CIA tool produces a conservative call graph for `mapmaker` and an optimistic call graph for both the `mosaic` and `gcc` systems.

[9]The categorizations discussed in this section use a definition of a call graph in which calls through function pointers are identified, but the target of the call is not resolved.

—The Rigiparse tool produces approximate call graphs for each of the `mapmaker` and `gcc` systems and produces an optimistic call graph for the `mosaic` system.

This categorization of tools must be interpreted in light of several points. First, as mentioned previously, different kinds of call graphs are likely useful for different software engineering tasks. For instance, a software engineer trying to assess the cost of a change to the system may find an optimistic call graph suitable as a basis for performing the task. Thus, it is not necessary for every tool to produce a conservative call graph. Second, the conservative category in this discussion does not account for calls through function pointers and therefore should not be equated with the meaning of the term in the compiler community. Finally, the categorization is shown for a small sample of target systems; for example, it is not necessarily the case that GCT would produce a conservative call graph for all systems or that Imagix would always produce an approximate call graph.

The categorization demonstrates that the second of our initial expectations for the study was not upheld. We had expected, as many software engineers might, that a call graph extraction tool would behave similarly on the source code for different software systems, extracting call graphs with similar characteristics for the different systems. There is more than one tool, however, that extracted call graphs that fall into different categories. These tools could cause problems for software engineers expecting a call graph with particular characteristics. For instance, a software engineer may miss a change in a maintenance scenario if a call graph unexpectedly contains false negatives.

5. DESIGN DECISIONS

Although these empirical results report on only a few selected extractors, they provide insight into the collection of design decisions that the developer of any static call graph extraction tool must make. One way to characterize these decisions is to consider call graph extraction as a function that maps source into a set of calls. The tool developer must make trade-offs 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 call graph (the behavior of the function). The tool developer must also face a number of engineering considerations. How fast must the tool execute? How much customization by the user must be supported? The choices a developer makes about the domain and behavior of the function and the way the tool is packaged affect the kind of call graph—conservative, optimistic, or approximate—a developed tool will produce.

5.1 Domain

One key decision that a developer of a static call graph extractor must make is the form of the input to the tool. Four choices of input format are available for the developer of an extractor for a system implemented in C: unprocessed source, preprocessed source code, object code with symbol table information, and executable code with symbol table information. Our study focused on tools that could accept unprocessed and preprocessed source code. In all cases we studied, the call graphs extracted from unprocessed source code were approximate with respect to the baseline definition we chose. The set of call graphs extracted from preprocessed source code varied considerably, with no dominant kind of call graph produced. Investigating call graphs extracted from object or executable code with a symbol table was not within the scope of our study.

A second decision facing the tool developer is the set of constraints to place on the input accepted by the extractor. The most obvious constraint is whether the source forms a compilable or compiled system. If, for example, the C source was preprocessed with a set of consistent `defines`, producing source capable of passing both the syntactic and semantic checks of a compiler, then the tool developer may be able to use various static analysis techniques to handle indirect calls. In this case, the tool may be capable of producing a call graph that is conservative in the compiler sense. On the other hand, a tool that accepts source stripped of conditional definitions, representing the source across every configuration, may be useful for extracting call graphs to help understand a system prior to a reengineering activity. However, it is sometimes the case that expanding all the `defines` results in input containing multiple variants of statements such as definitions of functions; for instance, `defines` are sometimes used to provide one definition of a function for ANSI C and another definition for K&R-style C. In these cases, it may only be feasible to use a lexically based tool, increasing the probability of extracting an approximate call graph.

Another class of constraints includes static restrictions on the source text. For example, some versions of cflow may not produce output unless the input passes the first phase of lint, a C static program checker.[10] For tools extracting calls from source code, another constraint that must be considered is the set of language variants the tool accepts; for C, for example, does the tool extract from K&R C, ANSI C, the C subset of C++, or C languages for various PCs? A related issue is how the tool handles code generated by other programs. For example, Lex [Lesk 1975] and Yacc [Johnson 1975] files almost always contain C code. Are calls extracted from these files and reported in terms of the source before or after generation? Only the lexical tools in our study can report the calls in terms of the source before generation.[11] If the appropriate extensions are used, cflow will automatically generate and extract from the resulting files. In general, the stricter the constraints placed on the input, the more guarantees a tool developer can place on the characteristics of the extracted graph. Stricter

[10]Manual page distributed with the Sun Release 4.1, 1988.

[11]Since these are both programmable tools, the engineer can determine the appropriate caller to use in reporting calls prior to generation.

input constraints, however, can make it more difficult to apply a tool on a particular system to a specific software engineering task.

A third decision is the interface for providing input to the tool. The tools we studied allowed software engineers to specify input either by invoking the tool on individual files, on a list of files, on a directory structure, or else on an executable file. For the majority of the tools studied, input can be specified on an individual file basis. This approach provides fine-grained control to the tool user because supplementary information, such as the `defines` or `include` paths, can be set on a per-file basis similar to the compilation process. In contrast, the Imagix tool accepted a list of files, requiring a set of possibly conflicting `defines` and `include` paths to be specified across the files. Some of the false positives in the call graphs extracted by Imagix may be a result of this interface decision. Of the tools studied, only Field supports the specification of input by directory structure and by an executable file (compiled and linked with the debugging switch).[12]

Another interface factor that affects the kind of call graph extracted from a collection of C source code is the set of parameters accepted by a tool. If the tool accepts the same parameters accepted by most C compilers, such as the -I and -D parameters for `includes` and `defines`, respectively, it is possible the tool can be substituted for the C compiler in the configuration management tool for building the system. The GCT tool included in the study, for instance, not only accepted the same parameters as `gcc`, but also, after performing the desired instrumentation, runs the C compiler on the resultant code. This feature further eases the substitution of the call graph extraction tool for the compiler, particularly for systems such as `gcc` which produce and use a number of temporary programs. Other tools, such as Rigiparse, require variations on the parameters accepted by the C compiler; for instance, the `includes` parameters must be included in double quotations. This requirement complicates the substitution of Rigiparse for the C compiler and may result, in some cases, in false negatives in an extracted call graph because of difficulties in determining the system files that should be processed.

Together, these decisions on the form, specification process, and any other constraints on the input 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 prior to the expansion of macros. Furthermore, as an aid in the porting process, the tool user may find an approximate call graph to be sufficient. As another

[12]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, neither extracted call graph was a subset of the other. The directory-based call graph included 303 calls not included in the executable-based call graph; the executable-based call graph included 24 calls not included in the directory-based graph. The 24 calls missed in the directory-based approach resulted from a file that exists in the directory structure not being scanned. We were unable to determine why this file was not included in the scan. This kind of anomalous behavior is by no means limited to Field.

example, a tool that extracts from an executable system may be more useful to an engineer writing test plans. To help with this task, the tool user may require a conservative call graph.

5.2 Behavior

The behavior of an extraction tool is defined primarily by the analysis algorithms it uses. Determining the algorithms used based on documentation is often difficult, since the documented information is seldom complete. Many of the tools we studied, for example, do not explicitly state how they handle calls through function pointers. Determining the algorithms used by inspecting the source of a tool is also extremely difficult, since the algorithmic code is neither small nor isolated within the source. A search of the literature is not sufficient either [Callahan et al. 1990, p.484]:

> While many interprocedural data-flow problems are based on a call multigraph [Allen 1974; Banning 1979; Cooper and Kennedy 1984; Cooper et al. 1986; Myers 1981] or a relation between pairs of procedures indicating possible call sequences [Barth 1978], few authors discuss how it is constructed from the source program.

Inferring some algorithmic aspects, however, is feasible. Simple tests can be used, for instance, to determine whether or not tools try to report calls through function pointers. Of the tools we studied, GCT and cawk report the sites of calls through function pointers. None of the tools we studied attempted to perform any static analysis to report the actual calls that might result at these call sites.

A secondary aspect of behavior is how tools handle source that does not satisfy the input constraints. The Rigiparse tool, for instance, reports syntax errors while processing the `sched.c` file that is part of the `gcc` system; a false positive is subsequently reported when an access to an array is misinterpreted as a call site. In contrast, the CIA tool separates information extracted when a syntax error is reported. We chose not to include this information in the extracted call graph, thus introducing false negatives into the extracted call graph and moving the tool from reporting a conservative call graph to reporting an optimistic call graph. The decision not to include partial information for a source file, though, might also reduce the kinds of false positives resulting from the misidentification of syntactic constructs after a parse error that arose in the Rigiparse case.

Another aspect of behavior is whether the call graph extraction algorithm is local, reporting call information simply as function-calls-function, or whether it is global, resolving the location of called functions to files. All of the tools we considered could report local call information. Only two of the tools—Rigiparse and Mawk$_{cg}$—could not easily produce global information as well. It may be necessary to take into account global call information to achieve certain characteristics in the extracted information for particular definitions of call graphs.

5.3 Engineering Considerations

Tool developers must also make a number of engineering decisions when building a call graph tool. In particular, the tool developer must consider how programmable the tool needs to be, and the performance requirements for the tool.

Typically, a tool developer can introduce programmable aspects into a tool at many different levels and in many different ways. One choice the developer may make is to not provide any programmable features. For instance, although cflow may be executed with many different options, it does not provide any significant programmable features to the software engineer. At the other end of the spectrum are tools that require configuration scripts to report or extract particular information such as call graphs; the two lexical tools and cawk are representative of this approach. Somewhere in between on the spectrum of programmability are tools that allow the engineer to specify some parameters to the extraction process. Rigiparse, for example, permits the engineer to specify the preprocessor to use. Programmability of the tool is desirable in that it may help an engineer extract a call graph with the desired characteristics for a task. On the other hand, the use of programmable features places an onus on the software engineer to assess the characteristics of extracted graphs to determine the kind of information being gathered.

Determining and delivering appropriate performance for software systems is never easy. As with most properties, developers must balance speed with other desired features. For call graph extractors, tool developers must typically balance speed with programmability. Here, we use the term "speed" to refer to both the time required to execute the extractor and produce a call graph and the time required to appropriately program and configure the extractor. A software engineer may be willing to trade some execution speed for enhanced programmability, but, in general, the tool must still be fast enough to permit reasonable iterative use so that a suitable call graph can be produced. For instance, a software engineer would not likely be willing to spend the time to write a suitable call graph extraction script for LSME if it required several hours to run the tool over a system that was tens of thousands of lines of code.[13]

Tool developers must also factor the intended use of the extractor with execution speed. It is unlikely given the large design space for call graph extractors that one extractor will be suitable for all tasks. In some cases, software engineers may be willing to trade speed of execution for an assurance of the kind of call graph produced by a tool. For instance, a software engineer may be willing to run a tool over a period of a few hours if it will produce a conservative—but not too conservative—call graph for the purposes of reengineering.[14] For other tasks, such as helping an

[13]LSME has been used to extract call graphs from tens of thousands of lines of code, requiring on the order of tens of minutes of CPU time [Murphy and Notkin 1996].

[14]As described earlier, a call graph that is too conservative might consist of the cross product of the names of all defined functions.

engineer become familiar with a new code base, approximation in the call graph may be an acceptable trade for speed.

6. CHOOSING AND USING THE RIGHT EXTRACTOR

Call graphs extracted by software engineering tools are intended to aid engineers in performing various engineering tasks. How does an engineer choose a suitable extractor for a particular task? How does an engineer ensure that a call graph with the appropriate characteristics is available for the task? We consider each of these questions in turn.

6.1 Choosing a Suitable Extractor

Engineers choose a particular tool or tools for a task, based on a number of technical and managerial considerations. In this discussion, we focus on two of the technical properties influencing selection of a call graph extractor: the constraints placed on the use of a tool and the communication of a description of the behavior of a tool from the developer to the user.

6.1.1 *Constraints.* Does the extractor have constraints that are 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 tools that input-pass a lint check may be too strong for an engineer to use when modifying a large legacy system, since many realistic systems do not pass lint's strenuous checks. As another example, syntactic tools that require the presence of all `include` files may be too strong when trying to port a system onto a new platform.

6.1.2 *Design.* Are the design decisions made clear to the engineer? Ideally, the design decisions made by the tool developer should be made explicit in the documentation of the tool. In practice, however, most tools make explicit only a few of the decisions. 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 the (CCI_free, CCI_free) false positive reported by LSME_{cg} results from the lexical approach of the tool not distinguishing call syntax used inside a string in a print debug statement. 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, Imagix's inconsistent handling of

XtDisplay may confuse the engineer, perhaps decreasing the engineer's 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.

6.2 Extracting an Appropriate Call Graph

Sometimes an engineer will not be able to find an extractor that produces a call graph with the appropriate characteristics for a specific task. An engineer has several options in this case. The engineer might become a tool developer, perhaps by using the programmable features of an existing tool, and create the appropriate extractor. However, it can be difficult sometimes for an engineer to produce an appropriate extractor within the time and cost constraints placed on the software engineering task. This study, in part, shows how complex an engineering activity it can be to produce an appropriate extractor.

Another approach available to the engineer is to combine the output of two or more existing static call graph extraction tools. For instance, an engineer could decrease the false negatives in a call graph for the `mosaic` system by combining the graphs produced by the CIA and Rigiparse tools. This combined call graph would still be an optimistic graph in that it would contain no false positives. A difficulty facing the software engineer in this scenario is the determination of the characteristics of each of the extracted graphs.

An engineer might also combine the output of a static call graph extractor, such as GCT, with the output produced from a dynamic extractor, perhaps based on a profiling tool such as gprof [Graham et al. 1982], to supplement the statically produced graph with information about the actual calls through function pointers. Using gprof, we collected a dynamic call graph for `mosaic`, based on a simple test case that covered approximately 23% of the call sites reported by GCT. In comparison to the GCT-extracted graph, the dynamically gathered call graph reported on 1,741 calls: 950 of those calls overlapped the statically extracted graph; 791 calls did not appear in the statically extracted graph; and 3,169 calls from the statically extracted graph did not appear in the dynamically extracted graph. A large number of the calls reported that differ from the GCT-reported calls were due to the reporting by the profiling tool of calls between system routines. Calls through function pointers accounted for 50 of the calls; these calls were filtered in a straightforward manner from the profiling output and could be used to supplement a statically extracted call graph for tasks requiring more complete information.

7. CONCLUSION

Abstractly, call graph extractors seem fairly straightforward to develop. The design and 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.[15]

It is not that these tools are inherently flawed. For the most part, the differences represent differing choices made by the developers of these tools in a design space for call graph extractors; however, the volume of the design space raises several problems.

One problem is that we know little about which points in the design space 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 to determine 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.

Even given an appropriate selection of tools and an understanding of the kinds of call graphs suitable for different tasks, the software engineer is faced with the problem of selecting an extractor for a particular task. As our study showed, an engineer cannot make assumptions about the kind of call graph produced based on simple information such as the technology used to create an extractor. Nor can the engineer assume that if a tool extracts a particular kind of call graph—say a conservative call graph on one system—that it will produce a conservative call graph for all target systems.

There are several possible ways to attack this problem. It may be possible to engineer tools that guarantee certain behavioral properties, such as always returning an optimistic call graph. Or, it may be sufficient 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. The aspects of the design space described in this article provide a sketch of a basis for the documentation required. Another possible approach is to develop new tools and techniques for helping an engineer assess the kind of call graph extracted. Advances are likely needed in all of these directions to adequately support the engineer.

One limitation of our study is that we considered static 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 extractors that work on 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

[15]Unexpectedly inconsistent computations from long-lived, widely available, and broadly used programs have also been documented in seismic data processing [Hatton and Roberts 1994].

developer must determine how to handle dynamic binding of method invocations to method definitions.

Although this article only considers one kind of information, call graphs, the observations about design decisions also apply to static extractors of other kinds of information, such as inheritance structures, file dependences, and references to global variables.

APPENDIX

A. VERSION INFORMATION

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

—The version of cawk used was 0.6.

—The version of cflow used was the executable distributed with Sparc SUN OS 4.1.3_U1. The only available documentation is a Unix man page. Although cflow is distributed as part of the operating system on many Unix platforms, it performs quite differently on each platform.

—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.

—Version 2.5 of Imagix was used. Imagix is copyright by Imagix Corporation.[16]

—The version of Field used was downloaded in September, 1996, from the semianonymous FTP site at wilma.cs.brown.edu.

—The version of LSME used was Version 1.6 (Build 1).

—Version 1.2.2 of Mawk was used. Mawk is available by anonymous FTP at several sites including tsx-11.mit.edu.

—The version of Rigiparse—or more specifically, cparse—used was 5.4.1d. Rigiparse is available by anonymous FTP from tara.uvic.ca.

The version information for the systems we analyzed is

—MAPMAKER/EXP 3.0 and MAPMAKER/QTL 1.1,

—`mosaic` Version 2.5 (for the X Window System), and

—`gcc` version 2.7.2.

B. RUNNING THE TOOLS

The following describe characteristics of running the software engineering call graph extraction tools and the GCT tool:

[16]http://www.imagix.com

```
[(expression:FUNCTION:$fcall identifier:$name *)] {
   AstNode dclId = Declaration($name);
   int declType;
   AstNode node;

   /*
    * If the dclId is NULL or the dclId is a function ID, then the call is
    *  a normal function call.
    */
   if (dclId == NULL || IsFunctionId(dclId))
     fprintf(stdout,"%s;%s\n", FunctionName($fcall), IdentifierName($name));
   else {
     fprintf(stdout,"%s;*%s\n", FunctionName($fcall), IdentifierName($name));
     }
   }

[(expression:FUNCTION:$fcall expression:$expr *)] {
     fprintf(stdout,"%s;", FunctionName($fcall));
     CPrint($expr);
     }
```

Fig. 4. Script used to configure the cawk tool.

cawk$_{cg}$. The cawk tool was run with the options `-file -no-macro -N -inclLoc -I<cawk directory> -f call.scr` in addition to the flags derived for each file through the compilation process. When running cawk with its own preprocessor, the `-no-macro` was dropped, and one to two dozen -E switches and zero to a few -T switches were used, depending on the program being processed. The -E and -T switches give information to cawk's `cpp` about what macros must be inlined (for those macros that cawk's `cpp` cannot figure out for itself) and what identifiers in macros are forward references to typedefs. When compiling `gcc`, the `-U_GNUC_` switch was added so that GNU C language extensions would not be used in the source code.

The script used to configure the cawk tool, call.scr, is shown in Figure 4. The script causes the tool to traverse ASTs for function call expressions. When matches are found, the associated action code is executed.

cflow. cflow was run over the C source code files comprising a target system with the `-r -i_` option as well as any necessary `includes` or `defines`. The `includes` and `defines` were derived on a per C source file basis from the compilation process.

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 necessary `includes` or `defines` were derived and applied, on a per C source file basis, from the compilation process. The resultant `.A` files were linked using CIA, and the database was queried using `cref -u func - func -`.

Field. The xrefdb tool of Field was used to generate a program database. The `INCLUDE_PATH` environment variable was set to any necessary

```
comment /* */

function
  %
  [ <type> ] <fn> @
    if keywordq(fn) | operatorq(fn) then
       fail
  @ \( [ { <arg> }+ ] \) [ { { ( <type> | { \( <foo> \) \( \) } ) }+ ; }+ ] \{
  %

  function.call
    %
    <cf> @
       if keywordq(cf) | (not letterq(cf)) | compareopq(cf) | endinequalq(cf) then
         fail
     @
       \( [ { ( <arg> | { \( [ { <arg> }+ ] \) } ) }+ ] \)  @
        writeCall(fn, cf)
        fail
    @
    %
```

Fig. 5. LSME call graph script used for the mosaic system (LSME$_{cg2}$).

`include` directories as derived from the output of the compile process. The values of the `defines` necessary for compilation were passed to the xrefdb tool using a series of -F flags. Since the xrefdb tool was run per executable, any necessary `defines` or `includes` required for one C source file were applied to the analysis of all C source files.

The xrefdb tool was run once against each executable produced for a given system, sometimes producing more than one database per system. Each Field database was then queried with `(C.from, C.call);` to produce a textual list of the calls between functions. The lists of calls between functions from each query were combined to form a call graph for the system.

GCT. GCT accepts a superset of the command-line parameters accepted by `gcc`. After instrumenting the code, GCT passes the file to a C compiler for compilation. This design allows GCT to be substituted for the compiler used to build the target systems. The control file used in the invocations of GCT was as follows:

```
(coverage routine call)
(options instrument macros instrument-included-files)
```

To produce a call graph, a simple awk script was used to extract the relevant information from the map file produced as part of the instrumentation process.

Imagix. The Imagix environment was used to add targets to the makefile(s) for a system being analyzed. The makefile targets inserted by Imagix to run the Imagix analyzer depend on makefile conventions, such as having

Table IV. LSME Script Differences Compared to the Script Shown in Figure 5

Script	Use	Number of Lines	Difference from LSME$_{cg2}$
LSME$_{cg1}$	Preprocessed `mapmaker` source	22	Permits declaration of pointers to function calls in function definitions, and alters callee name identification to permit operators attached to the callee name.
LSME$_{cg3}$	Preprocessed `gcc` source	22	Permits use of multiple parentheses in call site to accomodate macro expansions.
LSME$_{cg4}$	Unprocessed `mapmaker` source	27	Permits declaration of pointers to function calls in function definitions, and alters callee name identification to permit operators attached to the callee name. Permits more liberal function definition than LSME$_{cg1}$ as based on LSME$_{cg5}$.
LSME$_{cg5}$	Unprocessed `mosaic` source	31	Permits presence of macro between the name of a function being defined and the definition of the function's arguments; allows multiple parentheses in call site to accomodate macros; and recognizes commas in function definitions.
LSME$_{cg6}$	Unprocessed `gcc` source	21	Does not recognize embedded calls.

symbols defined for the C files to be compiled. As necessary, these targets were modified to refer to the C source files reported as being used in the compilation process. The `defines` and `includes` derived from the compilation process were added to the definition of the IMAGIX_FLAGS symbol. Since `mapmaker` and `gcc` had only one makefile, any `defines` or `includes` necessary for one C source file were applied by Imagix to the analysis of all C source files for that system. The `mosaic` source, on the other hand, includes one makefile per subdirectory. In this case, the `includes` and `defines` were set on a per-directory basis.

The Imagix database was created by executing the command `make imagix` for the appropriate makefiles comprising a given system. The necessary directories were added to a project for a target system using the Imagix environment's `Add Data` command. The call graph for a target system was produced by loading the database and executing a special print script provided by Imagix Corporation.

LSME$_{cg}$. The LSME tool is a generator. The tool accepts a description of a set of hierarchical regular-expression-based patterns to search for in a set of input files and generates an appropriate extractor. Generating an extractor for calls between functions in C code requires the writing of a short script. Six similar but slightly different scripts were used. The scripts differ in such aspects as the handling of arguments passed to a function, the handling of embedded calls, and the handling of macros (for unprocessed source).

The script used to report calls from the preprocessed `mosaic` source code is shown in Figure 5. The first pattern in the script, named "function,"

```
BEGIN {
    WS = "[ \t\n]*"
    ID = "[a-zA-Z0-9_]+"
    IDCC = "[^a-zA-Z0-9_]"  # added for KEYWORDIC
    CALL = ID WS "\("
    DEFN = ID WS "\([^{]+{"
    KEYWORD = "for|while|do|switch|if|typedef"
    KEYWORDIC = IDCC "(for|while|do|switch|if|typedef)" IDCC  # match in context
    OUTSIDE = 1
    RS="\n\n+"
}

!OUTSIDE {
    s = $0;
    while (s != "") {
      if ((start = match (s, CALL))) {
        match (substr (s, start), ID)
        len = RLENGTH
        if (!match (substr (s, start, len), KEYWORD))
          print fdecl, " ", substr (s, start, len)
          s = substr (s, start + len)
        } else
          break
    }
}

OUTSIDE {
    # use of KEYWORDIC avoids matching things like "double"
    if ((start = match ($0, DEFN)) && !match ($0, KEYWORDIC)) {
      match (substr ($0, start), ID)
      fdecl = substr ($0, start, RLENGTH)
      OUTSIDE = 0
    }
}

/\n}/ { OUTSIDE = 1 }
```

Fig. 6. Mawk call graph script.

describes the structure of a function definition. The code between the @ symbols is action code that is executed when a potential function definition is found in the source being scanned. In this case, the action code will reject the match if the function name is a keyword or an operator. Once a function definition is matched, scanning continues looking for one or more instances of the function call pattern, named "function.call," and subsequent instances of the function definition pattern. The function call pattern permits the matching of calls with one level of embedded calls. Murphy and Notkin [1996] provide further details on the meaning of LSME scripts.

Table IV summarizes the differences among the five other LSME scripts used in the study and the script shown in Figure 5.

In the case of preprocessed source, the program generated from the LSME patterns was provided as input, the result of passing the target system source through the SunOS preprocessor. The necessary `includes` and `defines` were passed to the preprocessor on a per-C-source-file basis as determined from the compilaiton process. In the case of unprocessed

Table V. Raw Data for mapmaker System

Tool	Number of Reported Calls	Intersection with GCT	GCT Minus Tool	Tool Minus GCT
GCT	4,732	–	–	–
cawk$_{cg}$	4,732	4,732	0	0
cflow	5,032	4,732	0	300
CIA	4,969	4,732	0	237
Field	4,594	4,580	152	14
Imagix	4,608	4,596	136	12
Rigiparse	4,721	4,710	22	11
LSME$_{cg1}$	4,994	4,551	181	443
Mawk$_{cg1}$	3,326	2,867	1,865	459
cawk$_{cg}$ (*own preproc*)	5,607	4,077	655	1,530
LSME$_{cg4}$ (*no preproc*)	5,320	3,224	1,508	2,096
Mawk$_{cg1}$ (*no preproc*)	3,259	1,738	2,994	1,521

Table VI. Raw Data for mosaic System

Tool	Number of Reported Calls	Intersection with GCT	GCT Minus Tool	Tool Minus GCT
GCT	4,119	–	–	–
cawk$_{cg}$	4,119	4,119	0	0
cflow	4,040	3,583	266	187
CIA	3,966	3,876	243	90
Field	4,006	3,954	165	52
Imagix	4,077	3,994	125	83
Rigiparse	4,106	4,106	13	0
LSME$_{cg2}$	4,008	3,913	206	95
Mawk$_{cg1}$	2,310	2,123	1,996	187
cawk$_{cg}$ (*own preproc*)	4,575	3,990	129	585
LSME$_{cg5}$ (*no preproc*)	4,179	2,707	1,412	1,472
Mawk$_{cg2}$ *no preproc*	2,521	1,547	2,572	974

source, the generated program was provided the original source of the target systems.

Mawk$_{cg}$. Mawk is a version of awk that supports patterns as data. Similar to LSME, Mawk requires a script with patterns and action code to define a call gaph extractor. The script used to extract calls between functions with Mawk is shown in Figure 6. This script takes advantage of GNU formatting conventions to represent function definitions. The meaning of a similar script to extract calls with Mawk is described by Griswold et al. [1996].

In the case of preprocessed source, the Mawk tool, configured by the call graph extraction script, was invoked on each C source file after it was preprocessed using the SunOS preprocessor. The necessary `includes` and `defines` were passed to the preprocessor on a per-C-source-file basis as determined from the compilation process. In the case of unprocessed source, the original script was provided to the Mawk tool configured by the appropriate call graph extraction script.

Table VII. Raw Data for gcc System

Tool	Number of Reported Calls	Intersection with GCT	GCT Minus Tool	Tool Minus GCT
GCT	11,980	–	–	–
cawk$_{cg}$	11,962	11,959	21	16
cflow	10,494	10,128	1,852	366
CIA	11,746	11,664	316	82
Field	11,928	11,921	59	7
Imagix	11,622	11,459	521	163
Rigiparse	9,533	9,531	2,449	2
LSME$_{cg3}$	11,716	10,659	1,321	1057
Mawk$_{cg1}$	9,696	8,980	3,000	716
cawk$_{cg}$ (*own preproc*)	22,107	11,067	913	11,040
LSME$_{cg6}$ (*no preproc*)	19,255	8,014	3,966	11,241
Mawk$_{cg1}$ (*no preproc*)	18,042	6,841	5,139	11,201

Rigiparse. The RIGICPP environment variable was set to use the C preprocessor distributed with SunOS. The Rigiparse tool was invoked on each C source file with any necessaary includes and defines passed on the command-line in quotations. The Rigiparse tool was run with the -f option for nonhierarchical output. The necessary includes and defines were derived from the compilation process.

The one exception to this process was the use of Rigiparse to extract a call graph from the gcc system. In this case, the preprocessor was invoked separately due to the large number of parameters, and the output was sent directly to the cparse executable which is normally invoked by rigiparse.

Source Code. The following describes the source code considered when producing call graphs for each system:

—*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).

—*mosaic*: The mosaic software considered as part of this study is structured into four subdirectories: libXmx, libhtmlw, libwww2, and src. This software, when compiled, produces one executable: Mosaic. The version of Motif used was 1.1.

—*gcc*: The gcc software considered as part of this study was the C source files used to produce a compiler for the C language. This software, when compiled, produces several executables, some of which are used to produce intermediate files during the compilation process.

C. RAW DATA

Tables V–VII contain the raw data collected from running the call graph extraction tools. The tables include data showing the number of calls reported by each tool (not including calls through function pointers), the number of calls that were also reported by the baseline tool GCT, the number of calls reported by GCT that were not reported by a particular tool, and the number of calls reported by a particular tool that were not reported by GCT.

ACKNOWLEDGMENTS

We wish to thank the authors and companies of all the call graph extractor tools used in the study for making their tools available to us. We also thank Robert Bowdidge, Michael VanHilst, Michael Ernst, and the anonymous reviewers for providing valuable comments on drafts of this article.

REFERENCES

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1979. Awk: A pattern scanning and processing language. *Softw. Pract. Exper. 9*, 4, 267–280.

Allen, F. 1974. Interprocedural data flow anlaysis. In *Proceedings of Information Processing 74 (Software)*. North-Holland Publishing Co., Amsterdam, The Netherlands, 398–402.

Banning, J. P. 1979. 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*. ACM, New York, NY, 29–41.

Barth, J. M. 1978. A practical interprocedural data flow analysis algorithm. *Commun. ACM 21*, 9 (Sept.), 29–41.

Callahan, D., Carle, A., Hall, M. W., and Kennedy, K. 1990. Constructing the procedure call multigraph. *IEEE Trans. Softw. Eng. 16*, 4 (Apr.), 483–487.

Chen, Y.-F., Nishimoto, M. Y., and Ramamoorthy, C. V. 1990. The C information abstraction system. *IEEE Trans. Softw. Eng. 16*, 3 (Mar.), 325–334.

Cooper, K. and Kennedy, K. 1984. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*. ACM Press, New York, NY, 247–258.

Cooper, K. D., Kennedy, K., and Torczon, L. 1986. Interprocedural optimization: Eliminating unnecessary recompilation. *SIGPLAN Not. 21*, 7 (July), 58–67.

Feldman, S. I. 1978. Make: A program for maintaining computer programs. Tech. Rep. 57. AT&T Bell Laboratories, Inc., Murray Hill, NJ.

Graham, S. L., Kessler, P. B., and McKusick, M. K. 1982. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*. ACM, New York, NY, 120–126.

Griswold, W. G., Atkinson, D., and McCurdy, C. 1996. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE 1996 4th Workshop on Program Comprehension (WPC '96)* (Berlin, Germany). IEEE Press, Piscataway, NJ, 144–153.

Hall, M. W. and Kennedy, K. 1992. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst. 1*, 3 (Sept.), 227–242.

Hatton, L. and Roberts, A. 1994. How accurate is scientific software?. *IEEE Trans. Softw. Eng. 20*, 10 (Oct.), 785–797.

Johnson, S. C. 1975. Yacc: Yet another compiler compiler. Tech. Rep. 32. AT&T Bell Laboratories, Inc., Murray Hill, NJ.

Lakhotia, A. 1993. Constructing call multigraphs using dependence graphs. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, SC, Jan. 10–13, 1993). ACM Press, New York, NY, 273–284.

Lesk, M. 1975. Lex—A lexical analyzer generator. Tech. Rep. 39. AT&T Bell Laboratories, Inc., Murray Hill, NJ.

Marick, B. 1994. *Craft of Software Testing*. Prentice-Hall, Inc., Upper Saddle River, NJ.

MÜLLER, H. A. AND KLASHINSKY, K. 1988. Rigi—A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, April 11-15, 1988). IEEE Computer Society Press, Los Alamitos, CA, 80–86.

MURPHY, G. C. AND NOTKIN, D. 1996. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol. 5*, 3 (July), 262–292.

MURPHY, G., NOTKIN, D., AND LAN, E.-C. 1996. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, Mar. 25-29). IEEE Computer Society Press, Los Alamitos, CA, 90–99.

MYERS, E. 1981. A precise inter-procedural data flow algorithm. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 219–230.

REISS, S. 1995. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, Hingham, MA.

RYDER, B. G. 1979. Constructing the call graph of a program. *IEEE Trans. Softw. Eng. SE-5*, 3 (May), 216–226.

Received: February 1997; revised: May 1997; accepted: October 1997