ASSISTING PROGRAM ANALYSES WITH LIBRARY SYNOPSES

Darren C. Atkinson
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566
atkinson@engr.scu.edu

Abstract

Understanding a program based on its source code is tedious and error-prone. Unfortunately, such a task is often necessary due to lack of adequate documentation. To assist software engineers in this task, automated analysis tools are often used. Such tools analyze the program source, computing information, and present that information to the tool user in a useful way. To compute correct information, the entire program may need to be analyzed; however, most programs written today use libraries extensively, for which source code may not be available. Therefore, tools must somehow model the libraries to summarize their effects. However, such models are hard to construct by hand and are often tool-specific, limiting reuse.

We present a general, automatable technique for generating models of libraries given their source. The resulting models are themselves pieces of source code called synopses. Synopses are more accurate than hand-generated models and are reusable across tools. Furthermore, the use of synopses can increase precision of subsequent analyses since they allow functions to be expanded inline, thereby gaining one level of precision with respect to calling context. Finally, synopses can be extended to entire software layers, improving the efficiency of analysis tools.

Keywords: Software Tools, Software Re-engineering, Program Development, Program Libraries.

1 Introduction

1.1 Motivation

Many software engineering tasks require that the engineer have a thorough understanding of the program. For example, correctly making a modification to a function's signature requires knowledge of which functions call that function; anticipating the effect of a proposed change requires knowledge of the control-flow and data-flow properties of the program. Ideally, software engineers would have adequate documentation to assist them in these tasks. Requirements, specification, and design documents present information about the program in ways that are easier for the engineer to understand, as compared with examination of the source code. However, this documentation is often not available or is out of date. This lack of adequate documentation most often results from market-driven software development practices that force developers to rush their products to market. Consequently, the best alternative left to the engineer is often to examine the program's source.

However, understanding a system by examining its source code is both tedious and error-prone. Market-driven development often results in code that is poorly designed and structured. Legacy systems are especially hard to understand. Many of these systems were not implemented using modern programming techniques that can help reduce the complexity of a system, such as information hiding [1]. Furthermore, as a system evolves, its complexity may grow exponentially with the number of changes made [2]. Therefore, the software engineer needs some form of help in understanding the program.

Automated tools have been proposed as a solution to the problem of understanding source code by hand. These tools analyze the system, computing and extracting information, and present the information in a way that is most useful to the tool user. For example, given a code location and name of a variable, a program slicing tool computes and presents the set of all statements that may have had an effect on the value of that variable [3, 4]. A program slicer can be useful during debugging. As another example, a call graph extractor determines the calling relationships of the functions in the program, and presents a graphical view of which functions call which other functions [5, 6]. Such a view is useful in trying to anticipate how a change to one function may affect the rest of the system. Recent work has focused on the efficiency [7, 8, 9] and precision [10] of such tools.

In order to compute correct information, an analysis tool often needs to analyze the entire program. However, the source code for libraries is often not available to the analysis tool. Today's programs make extensive use of library functions. These functions provide mechanisms for storing objects, sorting, searching, string manipulation, file access, and accessing resources provided by the operating system. A program understanding tool must somehow account for the effects of calling a library function without having access to its source code. Simply ignoring the library function calls leads to an incorrect analysis. Therefore, analysis tools typically *model* the effects of library functions. Such a model often summarizes the effects of

the function such as which variables are used, which are defined, and the relationship between the return value and the parameters. These models are often programmed by hand directly into the analysis tool itself. Although initially straightforward, this approach suffers from several problems:

- Creating the models is time-consuming and error prone. Often, the tool user has only a vague description of the library function from which to work. As discussed, this sort of documentation is itself often out of date, resulting in an inaccurate model being constructed. Furthermore, hundreds of library functions may need to be modeled in order to analyze even a relatively small program.
- The models are not reusable across multiple tools. Since the models are programmed to work with a certain tool, they often make use of the tool's internal data structures. Although this approach may ease implementation or improve efficiency, it makes reusing the models across tools difficult, thereby increasing cost.
- The analysis tool must be constantly updated. If a newer version of the library is developed, the models need to be updated so they correspond with the newer version. Since the models are part of the tool itself, the tool itself must therefore be changed.

1.2 Approach

Our approach to overcoming these problems is to automatically generate library models from their source and to have the models themselves take the form of source code. By automatically generating the models, the tool user is free to concentrate on the tool design alone. Furthermore, such models will be accurate with respect to the source code used to generate them. Since the models are themselves pieces of source code, they can be used by any tool simply by specifying them as additional source files to be analyzed. Finally, by separating the models from the tool, only the models need be (automatically) updated when the library is updated. The tool itself need not be changed.

In our approach, a model for a library function is itself a function that consists of a small number of source lines that correctly capture the effects of the function. We call such a model a *library synopsis* since it is a compact but accurate description of the most important aspects of the function, namely its effects as seen by *any* calling function. Since the function may use pointers to data structures provided by the caller, but unknown at the time the synopsis is constructed, the synopsis itself will also make use of pointers. To correctly account for the effects of a library function, its set of *observables* must first be computed. An observable is any program component that may have an effect on the calling function. For example, changes to global variables and return values are observable by the calling function, but changes to local variables are not. Once the sets of observables have been computed, the dependencies among the observables are then derived, yielding a dependency graph. Finally, a set of program statements is constructed from the graph. This final set of statements has the same dependency graph as the original library function.

```
int cmp (int *p, int *q) {
   return *p - *q;
}


int main ( ) {
   /* initialize array */
   p = bsearch (k, a, n, sizeof (int), &cmp);
   /* use result pointer */
}
```

Figure 1. Proper determination of the call graph requires an accurate model for `bsearch`.

The remainder of this paper is structured as follows. In Section 2, we present examples showing the necessity of modeling library functions and how simple errors in models can yield inaccurate results from subsequent program analyses. In Section 3, we present our approach in detail and show how to construct the set of observables for a library function, along with its associated dependency graph. Section 4 discusses how synopses can be used to increase the precision of the subsequent analysis through the use of function inlining. We then discuss how the concept of synopses can be extended to software architectures, reducing analysis time. Finally, we conclude with some thoughts on future work in the area of automatic generation of synopses.

2 Modeling Library Functions

Construction of models for library functions is essential for an accurate analysis. For example, an engineer may wish to use a call graph extractor to determine which functions in the system are called and therefore "live" and which functions are never called and therefore "dead". Consider the problem of extracting a call graph for the program given in Figure 1. The function `cmp` is not called directly from anywhere in the given program text. However, it is called indirectly by `bsearch`, a standard C library function. If we ignore the call to `bsearch` or model it incorrectly, `cmp` will be absent from the call graph, which may lead an engineer to conclude that it is never called (i.e., "dead code"), and therefore can be safely removed.

As another example, the engineer may need to know how a proposed change will affect the rest of the system. The engineer might use an analysis tool such as a program slicing tool to compute the dependencies between modules. At first glance, no dependency exists in Figure 2 between function `f` and the error module. However, `atan2` will set the value of `errno` if both arguments are zero, thus creat-

```
int f (int x, int y) {
   /* do something with x and y */
   return atan2 (x, y);
}


int main ( ) {
   f ( );

   if (errno)
      /* call error module */
}
```

Figure 2. The call to `atan2` may set the value of `errno`, causing a dependency between function `f` and the error module.

ing a dependency. However, if we ignore the call to `atan2` or model it incorrectly, the engineer might erroneously believe that a change to function `f` (such as replacing `atan2` with `atan`) will have no effect on the error module.

As these examples have illustrated, failure to model a library function, or modeling it incorrectly, can lead to an inaccurate program analysis. As a result, the engineer using an analysis tool may come to incorrect conclusions about the program's behavior. Such errors are likely to cause an engineer not to use a program analysis tool at all, and instead try to understand the program by analyzing its source code by hand.

3 Constructing Synopses

Errors in modeling library functions are easily made if the models are constructed by hand. In this section, we present our approach to automatically constructing synopses from existing code. Since a synopsis must be correct for any call to the function, we first compute the set of objects that may have an effect on the caller or that may have an effect on the called function. The set of observable objects, or just observables, is defined as follows:

- The formal parameters are the only IN observables: changes made to a formal parameter are unseen by the caller; however, a formal parameter does have an effect on the called function.
- The return value of the function is the only OUT observable: this value is undefined until specified by the function and is returned by the function and therefore has a visible effect.
- Pointer dereferences and global variables are IN-OUT observables: the function may consume the values of these observables and any changes to their values may be seen by the caller.
- Nothing else is an observable: local variables and compiler generated temporaries are private to the function and therefore do not have an effect on the caller nor do they provide any information to the called function.

```
char *strcpy (char *s, char *t) {
   char *p = s;

   while (*t != '\0')
      *p ++ = *t ++;

   return s;
}


int strlen (char *s) {
   int n;

   for (n = 0; *s != '\0'; n ++)
      s ++;

   return n;
}
```

Figure 3. Example implementations of two common C functions for manipulating strings.

Figure 3 shows example implementations of two common C functions for manipulating strings. Using these rules, the sets of observables are therefore:

	strcpy	strlen
IN :	{s, t}	{s}
OUT :	{s*}	{n}
IN-OUT :	{p*, t*}	{s*}

To construct a program synopsis, we must determine how the observables relate to one another. Naively, we can state that the set of OUT and IN-OUT observables depends upon the set of IN and IN-OUT observables, as suggested by [11]. However, such a dependence relation is too conservative to be used in practice. For example, the return value of `strcpy` depends only on the formal `s` and not on the other observables. To overcome such a limitation, we must construct the dependency graph for the function. The dependency graph explicitly shows the dependencies between observables. Before the graph can be constructed, each function is translated to a simpler form known as three-address code [12], as shown in Figure 4. This form exposes any subtle dependencies caused by the operators of the language. Once the three-address code has been built, the dependency graph is then constructed. An edge in the dependency graph links each operand to those operands on which it depends. The graph can be built during a single pass over the three-address code.[1]

[1]The resulting graph is *flow-insensitive*. A more accurate analysis can be obtained by building a *flow-sensitive* graph, in which each *occurrence* of an operand becomes a node. For simplicity, the flow-insensitive graph is discussed in this paper.

```
strcpy: p := s
L1:     t0 := *t
        if t0 = 0 goto L2
        t := t + 1
        *p := t0
        p := p + 1
        goto L1
L2:     t1 := s
        return t1


strlen: n := 0
L1:     t0 := *s
        if t0 = 0 goto L2
        s := s + 1
        n := n + 1
        goto L1
L2:     t1 := n
        return n
```

Figure 4. Three-address code for the functions in Figure 3.

The graphs for both `strcpy` and `strlen` are shown in Figure 5. A dependency graph can contain three different types of edges:

- *Data dependence edges* link uses of objects to definitions that consume their values.
- *Control dependence edges* link uses of objects to definitions that are conditionally executed based on their values.
- *Dereference edges* link pointer variables to a dereference of the pointer.

Once the dependency graph for a function has been constructed, the synopsis can be derived. To derive a synopsis, we find a backward path from each OUT or IN-OUT observable to IN or IN-OUT observables. For example, the return value of `strcpy`, t_1, has a path to `s`, and therefore the return value in the synopsis of `strcpy` is simply `s` itself. The return value of `strlen` is more interesting. The return value t_1 depends upon `*s` through first a control dependency and then a data dependency. Therefore, the return value in the synopsis of `strlen` is `*s`.

Continuing with the IN-OUT observables of function `strlen`, we find that `*s` has no data dependencies, but does have a dereference dependency with `s`. Following the edges that lead to `s`, we are led back to `*s` itself. Since `*s` is the only IN-OUT observable, the synopsis of `strlen` is therefore complete, as shown in Figure 6. Continuing with the IN-OUT observables of function `strcpy`, we find that `*t` leads back to itself and that `*p` leads to `*t`. Therefore, we can claim that `*p` depends upon `*t`. However, we need to express this fact in terms of formal parameters and global variables in order to construct a synopsis. The variable `p` is a local variable. Following the dereference edge from `*p` to `p`, we find that `p` depends on `s`. Therefore, we can express the body of `strcpy` as the fact that `*s` depends upon `*t`, as shown in Figure 6.

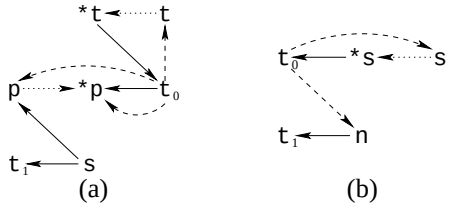

Figure 5. Dependency graphs for (a) `strcpy` and (b) `strlen`. Solid edges are data dependencies, dashed edges are control dependencies, and dotted edges are dereference dependencies. Loops are not shown for clarity.

In summary, to construct a synopsis for a library function we first translate the function into three-address code to expose the dependencies. We then construct the dependency graph from the three-address code and we compute the set of observables. Finally, we find all paths in the graph from OUT or IN-OUT observables to IN or IN-OUT observables. Each path becomes a statement in the synopsis, with each IN-OUT observable expressed in terms of global variables or formal parameters. The resulting synopsis has the same effect on the caller as the original function.

4 Using Synopses in Practice

Synopses can be constructed automatically from existing library implementations. Implementations of the standard C library are available through [13] and as `glibc`, available via the Free Software Foundation. By constructing synopses from such an implementation, a working model of the standard C library is generated as source code such as in Figure 6. This code can then be specified as another source file to a program analysis tool such as a call graph extractor or a program slicing tool. In practice, the use of synopses over hand-generated models has two additional benefits.

Improved precision: Most data-flow analyses used in program understanding are interprocedural. An interprocedural analysis must descend into and analyze a called function in order to account for its effects. If the function is called multiple times, the analysis may elect to either (1) analyze the function once, reusing information from previous calls, or (2) analyze the function each time it is called. The first approach is *context-insensitive* since the calling context is not taken into account, whereas the second approach is *context-sensitive*. Context-insensitive analyses are, in general, more efficient, but suffer from poorer precision as compared to a context-sensitive approach. The loss of precision occurs when data-flow information from all callers is shared (i.e., incoming information is merged)

```
char *strcpy (char *s, char *t) {
   *s = *t;
   return s;
}


int strlen (char *s) {
   return *s;
}
```

Figure 6. Synopses for the two functions in Figure 3.

in the called function. In contrast, context-sensitive analyses are more precise, but are, in general, less efficient since a function may be analyzed repeatedly. A context-sensitive analysis can be viewed as expanding all function bodies inline, resulting in a possible exponential expansion in code size.

However, since a program synopsis is much smaller than the original function it summarizes, it can be expanded inline without causing this rapid growth in code size. In our approach, we wish to accommodate inline expansion in a portable (i.e., tool-independent) manner in order to maintain the flexibility of using synopses. Fortunately, the C programming language supports inline expansion of code through the use of macros.[2] Rather than generating a function body for a synopsis, a macro definition can be generated instead, as shown in Figure 7. The macro definitions are placed in a header file (e.g., `<string.h>`) and a directive to the preprocessor is used to instruct the analysis tool to use the generated header file rather than the system header file. By expressing the synopses as macros, they will be expanded inline, thereby gaining one additional level of context-sensitivity for the subsequent program analysis.

Improving efficiency: As software grows in size and functionality, a software architecture [14] is typically employed to manage complexity. Layered architectures are particularly common, with lower layers providing functionality to higher layers, which are at a greater level of abstraction. Program libraries are an example of layering. However, beyond system-provided libraries, many software systems devise and implement their own libraries. Although these application-provided libraries are often specific to an application or set of applications and are therefore not generally reusable, they help reduce the complexity of the system. For example, many programs developed as part of the GNU project make use of libraries that encapsulate and circumvent the idiosyncrasies of many UNIX system calls and library functions. Often, such libraries are quite old and well-tested, existing in an organization for many years. As such, they are generally not of particular interest to the software engineer. For example, when searching for the source of an error, an engineer is more likely to suspect code written one month ago than code located in a library written two years ago. Since the library code is not of interest, rather than analyzing it in its entirely, the code can be replaced with its synopses. By extending the notion of synopses to software layers such as application-provided libraries and modules, we can improve the efficiency of subsequent program analysis since less code needs to be analyzed.

[2] In general, a macro may evaluate an argument more than once, which could lead to incorrect code should the argument have side-effects. A solution to this problem is to first assign the arguments to previously declared global variables or to variables declared locally in an expression block "`({ ... })`", a language extension supported by GCC.

```
# define strcpy(s,t) (*(s) = *(t), (s))

# define strlen(s) (*(s))
```

Figure 7. The synopses of Figure 6 expressed as macros for future inline expansion.

5 Conclusions and Future Work

Automated tools are necessary for helping software engineers understand systems based on their source code. Since these tools need to analyze a system in its entirety, they must somehow account for the effects of calls to library functions. A library synopsis is a concise description of the effect a library function has on its caller and have several advantages over hard-coding library models into an analysis tool. Synopses are less likely to have errors compared to hand-coded models since they can be automatically generated from an existing library implementation. Since synopses are themselves pieces of source code, they are instantly reusable across multiple tools. Furthermore, the use of synopses can improve the subsequent program analysis in two ways. First, additional precision can be obtained by expressing synopses as macros to be expanded inline. The inline expansion provides an additional level of context-sensitivity without negatively impacting performance since a synopsis is much smaller than its corresponding original library function. Second, by creating synopses for entire software layers, portions of the system that are uninteresting to the tool user can be reduced in size, speeding subsequent analyses.

The techniques presented in this paper are automatable. Work is currently being done to produce a tool to automate the generation of synopses from existing code. The resulting synopses will be compared with the actual library models used by two existing program slicing tools [15, 16] to verify their accuracy. The synopses will also be used in experiments involving both points-to analysis [17] and program slicing to determine the benefits of macro expansion on precision. Finally, application-provided libraries such as those used in the GNU project, will be synopsized to evaluate the performance gain achieved by using synopses.

References

[1] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[2] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press, Orlando, FL, 1985.

[3] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the 6th ACM International Symposium on the Foundations of Software Engineering*, pages 46–55, Lake Buena Vista, FL, November 1998.

[4] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[5] D. C. Atkinson. Call graph extraction in the presence of function pointers. In *Post-Conference Proceedings of the 2002 International Conference on Software Engineering Research and Practice*, Las Vegas, NV, June 2002. (To appear).

[6] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, Berlin, Germany, March 1996.

[7] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 52–61, Florence, Italy, November 2001.

[8] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, CA, January 1995.

[9] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, Kyoto, Japan, April 1998.

[10] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[11] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

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

[13] P. J. Plauger. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[14] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–39. World Scientific, Singapore, 1993.

[15] GrammaTech, Inc. Codesurfer user guide and reference manual.

[16] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*, Charleston, SC, November 2002. (To appear).

[17] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.