# SANDIA REPORT

SAND2000-1465
Unlimited Release
Printed June 2000

# A System Analysis Tool

Philip L. Campbell and Juan Espinoza, Jr.

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



**PATENT INTEREST**

A disclosure of invention relating to the subject of this publication has been filed with the U.S. Department of Energy.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

## DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

SAND2000-1465
Unlimited Release
Printed June 2000

# A System Analysis Tool

Philip L. Campbell
Secure Communications Systems Department

Juan Espinoza Jr.
Software Engineering Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87175-0449
{plcampb, jespino}@sandia.gov

## Abstract

In this paper we describe a tool for analyzing systems. The analysis is based on program slicing. It answers the following question for the software: if the value of a particular variable changes, what other variable values also change, and what is the path in between? Program slicing was developed based on intra-procedure control and data flow. It has been expanded commercially to inter-procedure flow. However, we extend slicing to collections of programs and non-program entities, which we term multi-domain systems. The value of our tool is that an analyst can model the entirety of a system, not just the software, and we believe that this makes for a significant increase in power. We are building a prototype system.

Keywords: program slicing, analysis, fault tree analysis, event tree analysis, software tools, software analysis, software assessment, software testing, analysis of programs, debugging, program understanding, reverse engineering, software maintenance.

# 1 Introduction

One of the ways we understand the world around us and of the increasingly complex objects we build is by "analysis." Analysis is the process of taking something apart, of cutting it up so that its constituents are distinct from each other. Analysis continues recursively until the pieces are simple enough to be understood on their own terms. From this understanding of the simple pieces, the analyst works to understand small groups of pieces that by their connections form a larger unit. This process continues recursively until the analyst arrives at an understanding of the entire set of objects. For the purpose of this paper we will refer to the focus of an analysis to be a "system," which we will define below.

There are two problems encountered again and again in analysis. The first question is, What are the borders of this system? That is, where does this system stop and the rest of the universe begin? The second question is, How should the system be divided? Another way of asking this second question is, What are the criteria that are to be used to divide the system into pieces? Recall that analysis can proceed only as the system is divided into simple enough pieces that they can be understood on their own terms. Analysis is a purposeful activity. The particular purpose that drives any given analysis effort drives both of these questions as well and thus is beyond the scope of this paper because it touches on the real world. However, what is not beyond the scope is the set of tools that an analyst wants.

One of the tools we believe that an analyst wants is one that shows cause and effect between components. If a particular component changes, what subsequent changes ripple to other components? Alternatively, what are the components that can effect change in a given component? The answers to these questions indicate how the system should be taken apart.

In this paper we present an evolving tool that has its roots in computer software. The tool is based on the notion of data and control dependence. We use the technique of "program slicing" to determine cause and effect.

We start with a review of data and control dependence, with a primer on program slicing. We then present the current level tool for program slicing. This first level is concerned with software that is can be considered one cohesive whole. Our tool resides at the next level. At the second level we consider software that consists of pieces that require files or unreliable links or even humans to bridge the gap between them. We then describe our prototype, discuss related work, and present conclusions.

# 2 Dependence, and Slicing

Program slicing uses the concept of "dependence" as the basis by which it extracts subsets of programs. Code in a program slice either contributes to a value at the end of the slice (a so-called "forward slice") or is affected by the value at the beginning of the slice (a so-called "backward slice").

## 2.1 Dependence

Dependence is a binary relation—prototypically, element A is dependent on element B. There are two kinds of dependence—data and control.

### 2.1.1 Data Dependence

Data dependence relies upon values, on data, hence the name. The notions of "definition" and "use" are central to the concept. When a statement assigns a value to a variable, it represents a definition (abbreviated "def") of the variable. When a statement uses the value of a variable, it represents a "use" of the variable. Since dependence is a binary relation and data dependence uses both "def" and "use," there are four possible data dependence relations: true, anti-, output, and a last one that is never used and thus unnamed. The four relations are shown in Table 1.

Table 1. The Four Data Dependence Relations

| name | description |
|---|---|
| true | def-use |
| anti- | use-def |
| output | def-def |
| (unnamed) | use-use |

For example, in the code in Figure 1

Figure 1. Sample Statements for Data Dependence

```
1: a = b + c;
2: b = a + d;
3: b = a + e;
```

Statement 2 is true dependent on statement 1, in variable a. Since statement 1 assigns a value to the variable a, and since statement 2 uses the value of variable a, statement 2 cannot execute before statement 1 completes. If statement 2 executes without waiting for statement 1 to complete, then there is no guarantee that the input/output behavior of the program will not change. (If the value of b is never subsequently used, then statement 2 can execute whenever it wants because it is superfluous; but we presume that an optimizing compiler would remove such dead code.)

Statement 2 is also anti-dependent on statement 1, but in variable b. Since statement 1 uses the value of variable b, and since statement 2 assigns a value to variable b, statement 2 cannot execute before statement 1 completes, or at least not until statement 1 has obtained the value that variable b had prior to the execution of either statement.

Statement 3 is output dependent on statement 2, in variable b. Both statements assign a value to variable b, so statement 3 must wait for statement 2 to complete before it executes. An optimizing compiler of sufficiently intelligence could easily conclude that the value of b set in statement

2 is never subsequently used, and thus statement 2 can be removed, without changing the input/output behavior of the program and speeding it up at the same time. However, in the code shown in Figure 2 a compiler would not be able to remove statement 2, and, to make matters more com-

Figure 2. Sample Statements, with a Print

```
1: a = b + c;
2: b = a + d;
2a: print (b);
3: b = a + e;
```

plicated, a compiler might or might not be able to remove statement 2 in the code shown in Figure 3.

Figure 3. Sample Statements, with a Function Call

```
1: a = b + c;
2: b = a + d;
2a: f(b);
3: b = a + e;
```

### 2.1.2 Control Dependence

Control dependence, as the name suggests, relies upon "control." If statement A is control dependent on statement B, then statement A determines whether or not control will reach statement A, whether or not statement A will execute. In traditional programming languages control statements are conditionals and loops.For example, in Figure 4, statements 2, 3, 6, and 7 are con-

Figure 4. Sample Statements for Control Dependence

```
0: read(x,y);
1: if (x<y) {
2:     a = ...;
3:     b = ...;
4: }
5: else {
6:     c = ...;
7:     d = ...;
8: }
9: while (x>0) {
10:    e = ...;
11:}
12: write(a,b,c,d,e);
```

trol dependent on statement 1. If the predicate x<y evaluates to true, then statements 2 and 3 execute; otherwise statements 6 and 7 execute. Statements 2 and 3 are control-true dependent on statement 1, and statements 6 and 7 are control-false dependent on statement 1. The set of statements that are control dependent on the same statement are said to be in the same control dependence region. These regions partition the program, since a statement is said to be control dependent only on the statement controlling its immediate region. Statement 12 is not control dependent on any of the statements shown here; we presume that if control reaches statement 1, then statement 12 will execute, so statement 12 is control dependent on the same statement as statement 1—they are in the same region. Note that knowing that two statements are in the same region is insufficient to determine their correct, relative order of execution. Note also that all of the statements in Figure 1, Figure 3, and Figure 3 are in the same control dependence region.

Loops add significant complexity. Consider the write for variable e in statement 12 of Figure 4. If $x \leq 0$, then the value to be printed is determined by the statement that defines e, wherever that is, but if $x > 0$, then the loop will execute at least once, perhaps many times. How do we express the fact that the value for e depends on the last execution of statement 10?

## 2.2 Slicing

Program slicing is based on both control and data dependence. The process can be described iteratively. We will describe it for a backward slice.

1. The analyst chooses a statement A. Statement A is included in the slice.
2. For each statement A in the slice, include the statements B such that A is dependent on B.
3. Repeat step 2 until no additional statements are included in the slice.

For a forward slice, step two is different: For each statement A in the slice, include the statements B such that B is dependent on A.

We can slice on control dependence, data dependence, or both.

The value of program slicing is that it identifies the statements in the program that effect or are effected by the first statement in the slice. All other statements in the program can be ignored.

Program slicing was first identified by Mark Weiser in his 1979 dissertation. [8] More (and current) information can be on-line obtained about slicing from Jens Krinke's program slicing page [1]; a good survey has been provided by Binkley & Gallagher [2].

# 3 Tools

Program slicing can be incorporated into a tool in a number of ways. The simplest way is to slice on one (control) region but the usual simplest way is to slice on one function or procedure. A more expansive way is to slice independent programs that comprise some part (possibly all) of a large system.

We discuss both ways in this Section. We have named these two ways Level 0 and Level 1. The capability provided by Level 1 is a superset of Level 0. We spend the most of the space discussing Level 1.

A tool for Level 0 has been commercialized. We are developing a tool for Level 1.

## 3.1 Level 0: Single Program

At the lowest level, what we call Level 0, we slice on a single program. Unfortunately the term "single program" is insufficient for our purposes but alternatives are unwieldy. We intend this to mean that all dependences in the system, which consists of the single program, can be automatically traced. This is equivalent to requiring that all of the data flows in the program, do so through named locations in memory, not through files, for example, or network connections, or floppy disks, or human operators (control regions typically do not span a function or procedure).

"CodeSurfer" from GrammaTech Inc. is a commercial tool, recently released, that performs backward and forward slicing on data and/or control dependence. [3] It provides for "chops" (slicing between two points in a program) as well as a variety of other features. Though the concept of slicing was articulated twenty years ago, it is just recently that a commercial tool has become available. Algorithms for computing interprocedural dependence are likewise not new. [4] The tool "unravel" from NIST is an experimental demonstration of program slicing that predates CodeSurfer, but for which there appear to be no future plans. [7]

The C program in Figure 5 prints the first 9 elements of the Fibonacci sequence. The program is contained in one function in one file. This exemplifies the simplest of arrangements.

Figure 5. Single Program, One File.

```
1 main() {
2     int i, a, b, temp;
3     a = b = 1;
4     printf("The Fibonnaci Number Sequence: \n");
5     for ( i = 1 ; i < 10 ; i++ ) {
6           printf("   %d:  %d\n", i, a);
7           temp = b;
8           b = a + b;
9           a = temp;
10    }
11 }
```

The next little step, from one function in one file, is to break up the program into multiple functions (or add new functions), but keep all the functions in the same file. This may not seem like a step at all, but it is. Dependence Graphs, upon which CodeSurfer is based, were developed for single functions. GrammaTech had to develop additional machinery to enable dependence graphs to work with multiple functions. For example, they developed "actual-in" and "actual-out"

vertices for the value passed in to and out of functions, as well as vertices for formal parameters, global parameters, return statements, and variable initialization (global or static), among others. GrammaTech refers to a set of these augmented dependence graphs as a "System Dependence Graph." Since all of the functions appear in the same file, CodeSurfer can analyze the program without requiring any additional information from the analyst.

The next little step is to put the multiple functions into multiple files, as shown in Figure 6. CodeSurfer can analyze the program in Figure 6, but, like the C compiler and any other slicer, CodeSurfer must be told the name of all the relevant files before it can resolve calls to functions whose declaration is in another file. Without the names of the other files, CodeSurfer is forced to treat the function calls like black boxes that CodeSurfer cannot penetrate to perform dependence analysis. However, when CodeSurfer has all of the functions together, it can perform dependence analysis.

Figure 6. Single Program, Multiple Files.

```
1 main() {
2     printf("The Fibonnaci Number Sequence: \n");
3     print_sub_sequence ( 1, 1, 10 );
4 }
```

```
1 print_sub_sequence( int a, int b, int length) {
2     int i,temp;
3     for ( i = 1 ; i < length; i++ ) {
4           printf("   %d:  %d\n", i, a);
5           temp = b;
6           b = a + b;
7           a = temp;
8     }
9 }
```

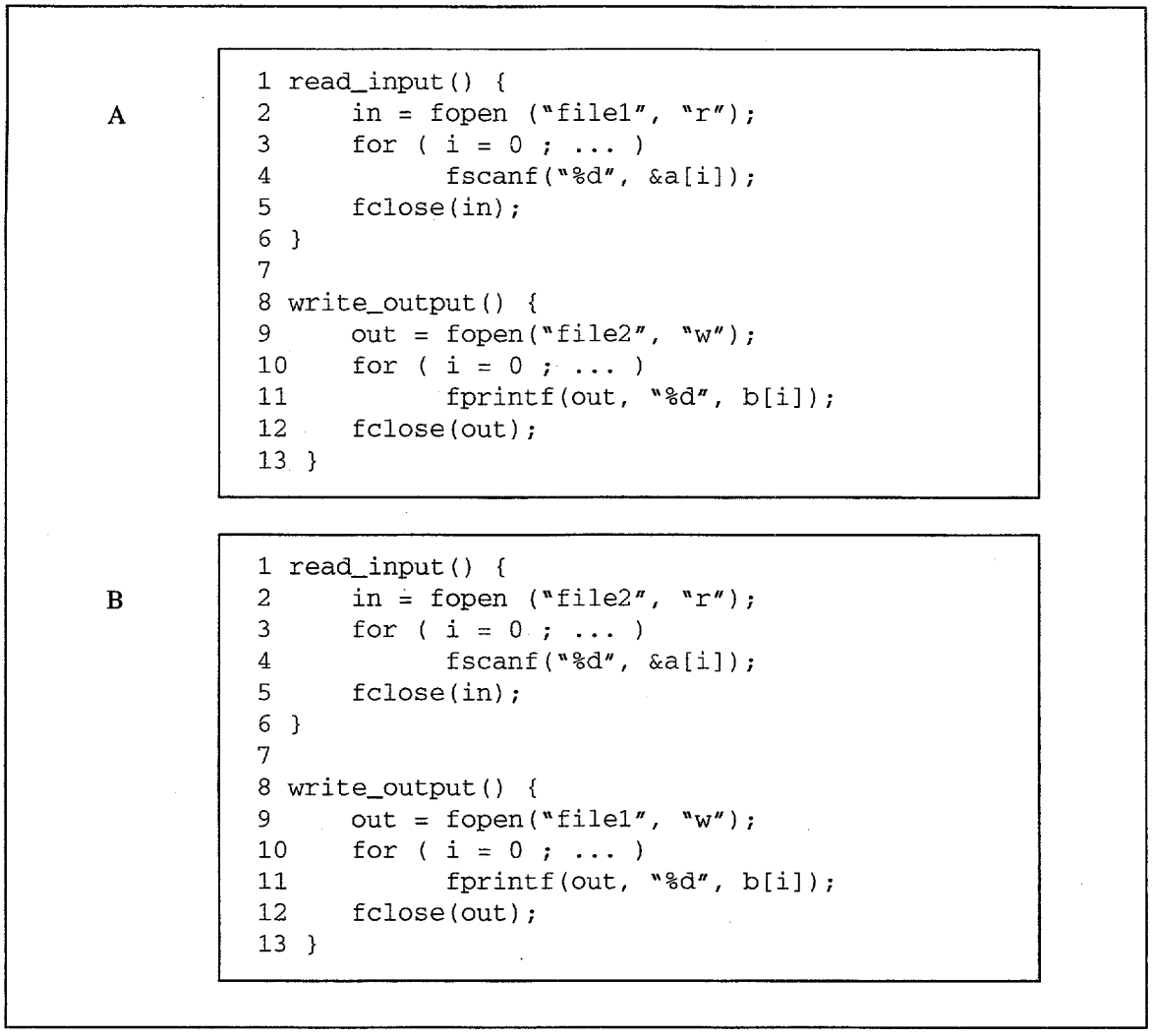
## 3.2 Level 1: Software System

The next step from a Level 0 capability is to be able to slice across independent programs. These systems have an explicit disconnect in data flow—such as one program writes to a file and another reads from the same file—so it is impossible to trace the flow of data automatically in the general case. The control dependence that exists between independent programs exists but is in none of the programs themselves; for example, if the output of program $P_0$ does not contain an error statement, then the output should be fed to program $P_1$.

Figure 7 shows two program files, named A and B, that communicate with files. File A reads from "file1" and writes to "file2;" and file B reads from "file2" and writes to "file1." Just the read

and write functions are shown in the Figure. We presume that other functions in each file, not shown in the Figure, protect against race conditions.

Figure 7. Multiple Communicating Programs.



A

```
1 read_input() {
2     in = fopen ("file1", "r");
3     for ( i = 0 ; ... )
4           fscanf("%d", &a[i]);
5     fclose(in);
6 }
7
8 write_output() {
9     out = fopen("file2", "w");
10    for ( i = 0 ; ... )
11          fprintf(out, "%d", b[i]);
12    fclose(out);
13 }
```

B

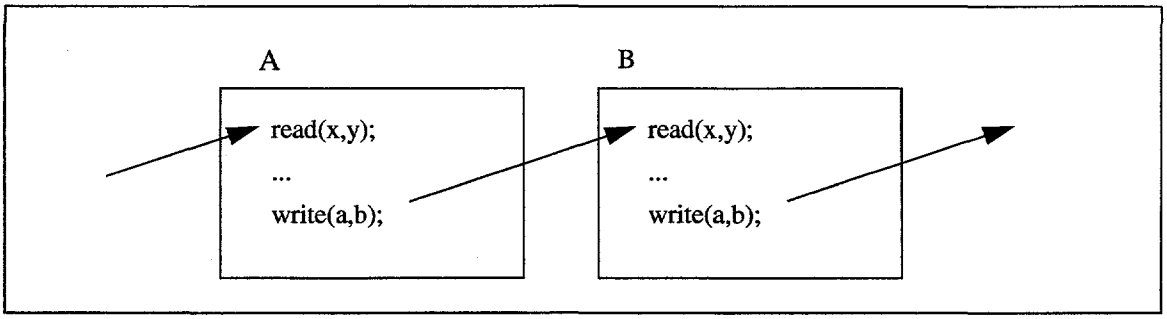
```
1 read_input() {
2     in = fopen ("file2", "r");
3     for ( i = 0 ; ... )
4           fscanf("%d", &a[i]);
5     fclose(in);
6 }
7
8 write_output() {
9     out = fopen("file1", "w");
10    for ( i = 0 ; ... )
11          fprintf(out, "%d", b[i]);
12    fclose(out);
13 }
```

Analyzing dependence for each of the files in Figure 7, by themselves, can be handled by a Level 0 technique. However, a Level 0 technique is unable to analyze dependence for the collection of two files. There is no information that a slicer could use in the general case that would link file1 in File A with file2 in File B, nor are there any ready-at-hand techniques to communicate this information to an analyzer.

We can represent the essentials of Figure 7 in a simpler way, as shown in Figure 8. Note that we have included arrows in the Figure, indicating that the write in File A is the generator for the read in File B. We do not know, given the Figure, from where File A reads parameters x and y, and we do not know who reads File B's output of a and b. However, with the information represented

by the arrow from File A to File B in hand, an analyzer can proceed with dependence analysis and can automate slicing between the two independent programs, as though they were one program.

Figure 8. Communicating Independent Programs.




What we have introduced is the ability to slice between independent programs, where independent means that the data connection they share is outside the memory space of any of the programs. A file, for example, is outside the memory space of a program. The program can read and/or write to a file, but it does so via the operating system.

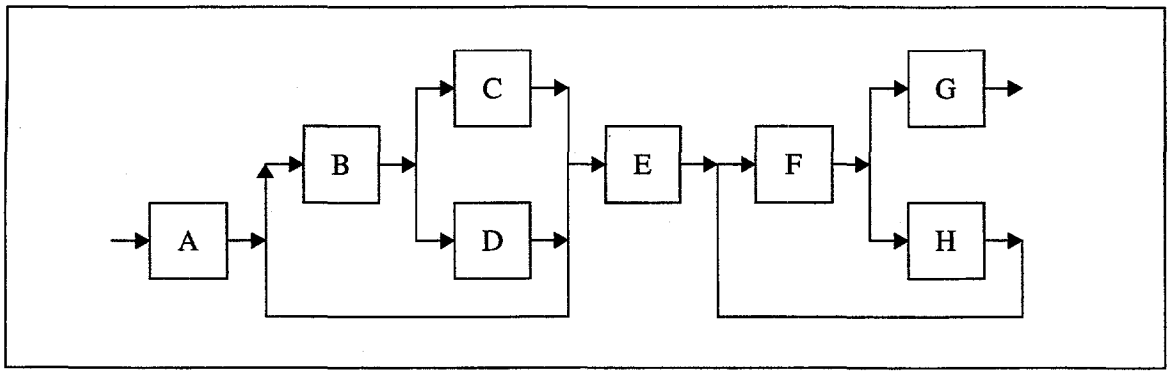
This small step is a significant one. It enables slicing on an arbitrary collection of single programs that together act as a single system. (We presume that the slicer can operate on each of the individual programs.) Without this tool, the analyst's only avenue for slicing, besides slicing by hand (hardly an alternative, some would say), is to combine the individual, separate programs into a new and larger single program, changing each program as needed and writing whatever additional code is necessary to make the data connections between the reads and writes. Not only is this a nuisance and error-prone, it is a waste of time. The effort required to combine one set of programs does not help decrease the effort to combine another set. To make matters worse, if the programs are written in different languages, say, the task may not even be possible.

This small step, once achieved, enables us to consider programs that know nothing of each other, that are written in different languages, that are run on different machines and could be in execution at different times.

But there is an even more significant capability that this step opens up. The analyst can reduce the individual programs to icons and thus consider the system as a whole. For example, Figure 9 shows what we will call a "software system." It consists of eight, independent programs. We do not know what language these programs are written in or where they run—and we do not

need to know this. All that we need to know, at this level, is how they communicate, which is shown for us via the arrows in the Figure.

Figure 9. A Software System.

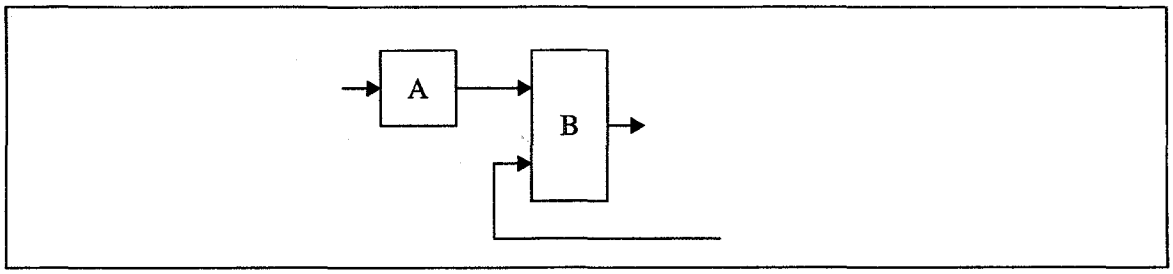



This Figure gives us our first view of the system as a system. We believe that anyone familiar with this system, when requested to explain it, would begin by reproducing Figure 9, probably by hand since there would probably be no corresponding diagram on-line, precisely because the "system" exists only in the mind of the people familiar it. However, the diagram in Figure 9 can represent the input for a slicer. As far as the slicer is concerned, the diagram is the system.

To build Figure 9, the analyst would need a graph editor of some kind. The analyst would create boxes, identify them in some way, indicate what software program was associated with each box, and finally indicate the flow of data between the boxes. This last step would require associating a read or write in one box with a corresponding write or read in some other box. The editor would have to be able to open the associated program and allow the analyst to select the appropriate input/output statement.

Note that in Figure 9 we have chosen not to distinguish multiple inputs. For example, the box labelled "B" could have been rendered as shown in Figure 10, and we presume that a sufficiently powerful editor would allow us to switch to this view whenever we pleased.

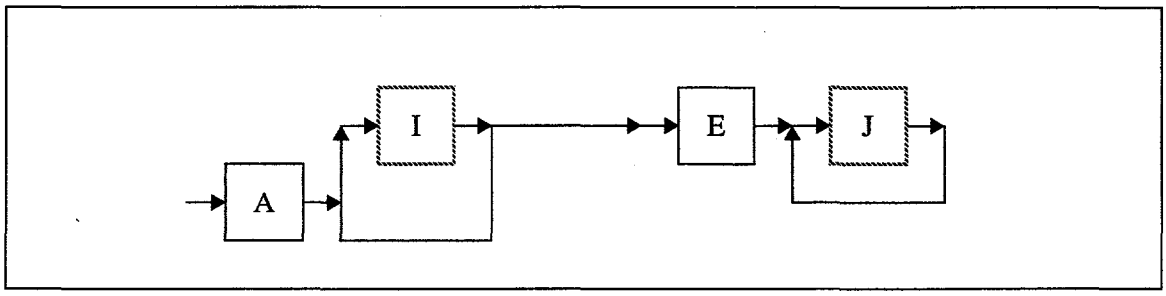
Figure 10. Another View of Box B from Figure 9.




We presume that the analyst could arbitrarily group the boxes into meta-boxes, such as those shown in Figure 11. We call this a "zoom" capability. The editor could suggest different high-level groupings. In the general case, determining the simplest set of meta-boxes is computationally difficult, so we should not hope for a great deal of help from an editor here. However, an edi-

tor could show us the results of our own choices, making it easier for us to explore the alternatives that would provide a simpler view for ourselves and others. Used iteratively this capability provides the obverse of the only tool we have to deal with complexity: divide-and-conquer. Perhaps we could call this capability "group-and-simplify" or "combine-and-understand." It enables the analyst to explore different ways to partition the system, thereby increasing understanding and control.

Figure 11. A Software System, a meta-view (boxes with perforated borders represent groups of boxes).




Note that the diagram in Figure 9 is itself a dependence graph. The nodes of this graph represent possibly larger units of code than the nodes in Figure 9.

With a Level 0 capability, the analyst depends upon the slicer (or, more accurately, a pre-processor to the slicer) to develop the dependence graph, given a program represented by a text file. After that operation is complete, slices (and chops) can be performed by traversing the dependence graph. However, with Level 1 capability, the analyst is the one who creates the dependence graph, not the slicer. Given this difference, what value does a slicer provide?

The slicer's value is in its handling of details. The slicer can accept an arbitrarily large or complex dependence graph, a graph within which the analyst is lost, and the slicer can automatically determine slices. Computers enable so many tasks which would be just too much work and too error-prone if they had to be done by hand. The value is quantitative, but so much so that it becomes qualitative.

With this Level 1 capability we would like to be able to define and use the relationship between input values and output values. We would like, using Figure 9, to be able to specify an input value for box A, and then be able to see the output value for box A. We would like to see how the input value propagates through the system. We would like to be able to do that for any subset of the system. For example, we would like to be able to see what output value is generated by box E for a given value input at box B. A more powerful capability would be to see the relationship between a set of input values and a set of output values. We call this "set value propagation." Both capabilities require that the slicer be able to execute code represented by each node of the dependence graph.

One way to implement this capability would be for the slicer to execute the code represented by each node. Unfortunately this presents a host of problems. For example, what does the slicer do when presented with a problem that does not halt? Even if such problems never crop up, the

task is intractable for set values, since in the general case the number of executions that would need to be run grows exponentially with the size of the input set. A safe way to proceed would be to require that the graph developer provide a set value propagation function, independent of the code describing the graph node, that provides a range of output values when given a range of input values.

An alternative approach to the graph editor shown in Figure 9 through Figure 11 above, would be to develop a text-based programming language that could express connections between programs. just as a programming language such as C expresses connections between operations (such as addition and subtraction). This way the analyst could describe the system in text symbols instead of graph symbols. The associated slicer would analyze a program written in that language, and would produce a dependence graph in a diagram such as Figure 9 in response. However, this seems like a step backwards to us. We believe that the graphical editor is the superior path.

A last additional thought concerns the capability that SAT provides. We have presented SAT as a "system" analysis tool, meaning software systems. We have emphasized its capability of enabling the analyst to analyze independent software programs. This is certainly of value, but as with all tools, after the tool is understood, it can be applied to new problems. The same is true here. With a different viewpoint a significantly more important capability is available. SAT enables the analyst to consider as part of the system pieces that are not software.

For example, let us take an experience one of the authors was involved with a number of years ago. A computer served as the central router for an internal network. The computer would occasionally (and inexplicably) crash. The problem was isolated to the board within the computer that handled the communications. Probes were attached to the board; the software running on the board was analyzed. The board was replaced, multiple times. The software was re-installed, multiple times. But the computer continued to crash, raising the frustration level of those whose work depended on the machine. The situation pulled in the manager, and then his manager. The second level manager wanted reports on the hour and to be notified each time the machine crashed. Finally, someone did something that was "outside" the system: they pulled out an "unnecessary" board, and the crashes stopped. It was discovered that the two boards had been on the same internal circuit, and had thus pushed the circuit very close to its breaking point. Occasional but small spikes would cause the power to fail. SAT enables an analyst to bring under analysis some piece, such as that unnecessary board, that is "outside" the system.

If the particular aspect of that outside piece and its function can be represented in software, then SAT can include it in the system and the analyst can consider it as the analysis proceeds. Alternatively, if the piece can be represented in some binary form, and if a slicer can be written for that binary form, then the pieces do not have to be represented in software; all that is needed is that its representation be sliceable. This small step opens the door for the analyst to include anything sliceable in the representation of the system, and this is a significant step indeed. It eliminates artificial boundaries in the analysis. It is always around these boundaries that problems linger, precisely because the tool is incapable or at least awkward there. However, with SAT, there are no artificial boundaries: the analyst is free to set the boundaries between system and non-system wherever it makes sense to do so.

# 4 Current Work

We are currently in the process of developing a Level 1 prototype. We are using CodeSurfer as our slicer back-end and the graph editor from Tom Sawyer Software as our graph editor front-end. Our tool, which can be simply described, provides the association between graph edges and program statements, enabling a Level 1 slicer.

# 5 Related Work

There are three areas of related work: program slicing, graph editors, and modeling languages such as UML.

Program slicing is more than "related work;" it is the basis for the work presented in this paper, as noted in earlier sections. The reader is referred to the "program slicing page on the Internet for more information. [1]

A "graph" in the most general sense is a mathematical object consisting of nodes and edges. A graph editors is to a graph what a text editor is to a text document. The graph editor enables the user to create a graph, but the real power is in the ability to modify a graph—to add and remove both nodes and edges. It is straightforward to imagine higher-level functions that are attractive to graph editor users, such as the ability to nest graphs to arbitrary depth, so that a given node can represent a nested graph which can contain nodes themselves represent nested graphs. There are a number of graph editors available, including commercial ones such as the one from Tom Sawyer Software. [5]

A third area that is related to this research is UML. UML is a language designed for describing models of a system. One view is that UML is the language of systems.

> The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system function, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable of software components. ([6], page xv)

The difference between UML and the tool we have presented in this paper is that we are closer to the system itself, as opposed to a model of it. In the lower level tools we present, the tool does not operate on a model of the system but on the system itself. The software that constitutes the system is the focus of the tool. In the higher level tools, parts of the system can be represented in software. As more and more parts of the system are represented in software, the entity that the tool focuses on becomes increasingly a model of the system.

# 6 Conclusions

Program slicing provides a way to understand programs. At its most basic level program slicing operates on a single procedure. At the next level inter-procedural slicing is provided, enabling slicing over an entire program. This is what we call Level 0. At the next level, Level 1, inter-program slicing is provided, enabling slicing over a system consisting of separate programs.

We believe that the advantage of a Level 1 slicer is the ability to model a real system. Note that different pieces can be included in the model of the system as long as a functional description of those pieces can be written and the functional description is sliceable. This provide the analyst with the opportunity of moving the boundaries of the system as seems best, enabling better modeling of the system.

# References


[1] Program Slicing page: http://brahms.fmi.uni-passau.de/st/staff/krinke/slicing/.

[2] David W. Binkley, Keith Brian Gallagher, "Program Slicing." Advances in Computers, volume 43, 1996, pp. 1-50, edited by Marvin V. Zelkowitz.

[3] GrammaTech's Home page: http://www.grammatech.com/grammatech.html.

[4] Mary Jean Harrold, Gregg Rothernel, Saurabh Sinha, "Computation of Interprocedural Control Dependence." Software Engineering Notes, Vol. 23, No. 2, 1998, pp. 11-20.

[5] Tom Sawyer Software's Home page: http://www.tomsawyer.com.

[6] Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide." Addison-Wesley, Reading, Mass. 1999. ISBN 0-201-57168-4.

[7] "unravel" page: http://hissa.ncsl.nist.gov/publications/nistir5691/

[8] Mark Weiser, "Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method." Ph.D. thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

## Distribution List

| | | | | | |
|---|---|---|---|---|---|
| 2 | MS | 0449 | P. L. Campbell | Org. | 6236 |
| 2 | | 0455 | J. Espinoza | | 6238 |
| 1 | | 0449 | R. L. Hutchinson | | 6236 |
| 1 | | 0455 | M. A. Tebo | | 6238 |
| 1 | | 0717 | R. L. Craft | | 6234 |
| 1 | | 0451 | R. E. Trellue | | 6202 |
| 1 | | 0612 | Review & Approval Desk For DOE/OSTI | | 9612 |
| 1 | | 0741 | S. G. Varnado | | 6200 |
| 2 | | 0899 | Technical Library | | 9616 |
| 1 | | 9018 | Central Technical Files | | 8940-2 |