FAULT LOCALIZATION USING EXECUTION TRACES

Margaret Francel
Spencer Rugaber

College of Computing
and
Software Research Center
Georgia Institute of Technology

ABSTRACT

This paper describes an approach to the automatic localization of program faults using execution traces. The traces are used to construct a directed graph that models the propagation of values during execution. Knowledge of which output values are incorrect is then used to narrow the region that must be examined by the person debugging the program. Algorithms to construct and analyze the graph have been designed and implemented.

25 November 2013

FAULT LOCALIZATION USING EXECUTION TRACES

Margaret Francel
Spencer Rugaber

College of Computing
and
Software Research Center
Georgia Institute of Technology

1. INTRODUCTION

The process of detecting, locating and correcting problems in computer programs is called debugging and is a labor intensive and time consuming activity. Because of this, there is a need for debugging strategies that facilitate and tools that automate the debugging process. We are developing a strategy that applies to the fault location step of the debugging process. The strategy is implemented through a tool that automates part of the process. The purpose of this paper is to give an overview of the underlying theory and to describe the corresponding debugging tool and discuss its usefulness.

It is important to distinguish among the terms error, fault, failure, and bug. An *error* is a mistake in the process that lead to the construction of a program [8]. Evidence of errors come through program *failures*, typically incorrect output values, unexpected program termination, or non-terminating execution. It is often the case that a failure can be traced to a small region of a program. If so, that region is said to contain a *fault* or *bug*. For example, using the wrong termination condition for a loop is called an "off-by-one" bug. It is important to add, however, that sometimes program failures are indications of global problems such as mistaken assumptions or inappropriate architectural decisions. In such cases, it is inadequate to use the term "debugging" for the process of correcting the error.

Program failure is identified by an outside source called an *oracle*. After a failure is detected, the corresponding fault must be isolated. This may involve intensive work by the oracle.

The goal of our strategy is the development of a debugging tool that will simplify fault location by reducing the oracle's work load. The strategy is applied by tools intended to assist the oracle during the fault location step by helping with the organization and classification of needed data, the elimination of irrelevant data and the drawing of conclusions from the gathered data.

Many debugging methods identify program faults by examining or analyzing program code. One of our goals is to show that the debugging process can be improved by examining the program's execution history. The model developed and the methods presented do this while at the same time reducing the amount of work on the part of the oracle.

In order to concentrate fully on the fault localization problem we examine only a limited subset of all program faults and only in a restricted environment. The limitations and restrictions we place on the programming environment and the behavior of the faults are given by the following assumptions:

- The programming language used consists of assignment, input, output, and **if** statements along with **while** loops.
- Only terminating programs are considered.
- Programming faults are not self correcting. Once a fault occurs, it propagates through the code causing a failure during program execution.
- All information provided by the oracle is accurate.

In section 2 we describe several projects that use ideas related to ours. In section 3 we show how we build our model from an execution history and the mechanism we use for propagating information through the model. In section 4 we describe what the debugging tool can do to help the oracle.

2. RELATED WORK

Many people have worked on tools for debugging with the common goal of reducing the oracle's work. Among the more ambitious is the work of Shapiro on algorithmic program debugging [16]. Shapiro developed efficient algorithms that lead to a semiautomatic debugger. The debugger works by guiding the oracle through a series of questions whose answers are used to reduce the region that may contain a program fault. The Shapiro debugger was developed for use with logic language programs such as those written in Prolog. Because of this, Shapiro chose as the basic unit of action the procedure call (rule invocation) and decided that a major indication of program faults is the situation where some procedure call produces an incorrect result. Not only does the Shapiro debugger lead the oracle to the fault location but it does so with a minimal amount of work on the oracle's part. Here work is measured by the number of questions that the oracle has to answer. Shapiro showed that the number of oracle queries was proportional to the logarithm of the number of procedure calls in the faulty computation.

Renner built a debugging tool that adapts the algorithms of Shapiro to Pascal programs. In [14], Renner discusses ways to deal with Pascal procedure calls with side effects and the more natural basic action of the Pascal program, the assignment statement. However, in his debugging tool he continues to use the procedure call as the basic program action.

Another project of worth is the STAD system as presented in [10]. The goal of the debugger part of this project was to guide the oracle in locating program faults. The system makes use of both the program structure and a history of a particular execution to build a "dependence network". It then guides the oracle from "bad" output back through the network until the oracle is able to identify the program fault.

In any interactive system the number of responses expected from the oracle must be taken into consideration when measuring work. Any mechanism that can help to lessen the number of oracle responses will benefit the system. *Slicing* is one such mechanism. A program slice, as defined in [17], is a complete program obtained by taking the subset of the statements from another program that relate to a given behavior. The slice is "guaranteed to faithfully represent the original program within the domain of the specified subset of behavior". Program slicing has been successfully used in reducing the number of statements that need to be examined when doing program debugging.

In [15] program slicing is used along with Shapiro's algorithmic debugging methods to present a fault localization technique that can be used with imperative language programs with side-effects. In this work, as was true in [16] and [14], the basic program action is assumed to be the procedure and the fundamental thread between actions is the procedure call.

A variation of slicing called *dynamic program slicing* was introduced in [9], also see [3]. Here the program slice is taken from the history of some execution of the program. Korel and Laski mention in [9] that they are investigating the use of dynamic program slicing as a means of extending the debugging capabilities of STAD. In [2], the Agrawal/Horgan definition of dynamic slicing is presented as part of a prototype debugging tool.

3. THE MODEL

As stated above, we are interested in determining the location of faults in a program through the examination of the history of the program's execution along with the output produced by the program. In this section we present a model for organizing this information that provides fast retrieval of the stored data as well as a structure that lends itself to easy analysis of interrelationships between execution actions. The basis of our model is a program dependency graph built from an execution history of the program.*

* Several different definitions for program dependency graph can be found in the literature, see for example

A program is a sequence of statements that access and combine named program variables. An example of a program containing a fault is the following.

PROGRAM SWAP
1) GET(x)
2) GET(y)
3) x := y
4) y := x
5) PUT(x)
6) PUT(y)

Figure 3.1

Once written a program can be executed any number of times. Each *program execution* is a map from a vector of data, called the input, to another vector of data, called the output.

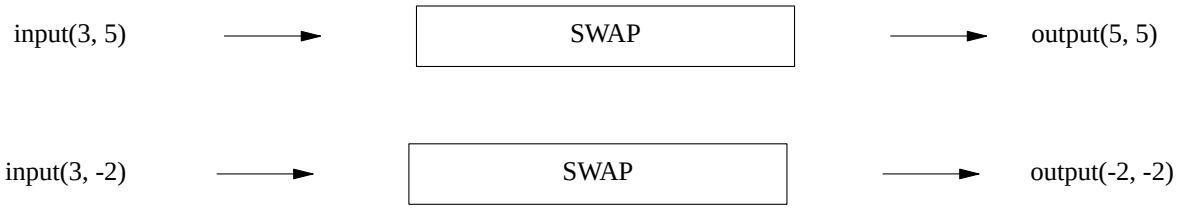

Figure 3.2

An execution's actions can be summarized in an execution trace. For each execution action, the trace records the statement executed along with information concerning the variables referenced and values computed. The execution trace for the second execution shown in Figure 3.2 is shown in Figure 3.3.

PROGRAM SWAP EXECUTION TRACE input(3, -2) output(-2, -2)			
program statement executed	action taken	variable references	variable assignments
1	GET(x)		x/3
2	GET(y)		y/-2
3	x := y	y/-2	x/-2
4	y := x	x/-2	y/-2
5	PUT(x)	x/-2	
6	PUT(y)	y/-2	

Figure 3.3

Our description of an execution history assumes that value assignment and expression evaluation are being used as the basic program actions. This seems natural for a procedural language and comparable to Shapiro's use of the procedure call as the basic action of the logic program. Just as procedures are related by procedure calls, value assignments and evaluation are related through value action.

We model the execution trace and the relationship between trace actions by letting each action of the trace be represented by a node in a graph and joining these nodes by directed arcs. An arc exists between two nodes, **n1** and **n2**, if and only if during program execution action **n2** references a variable whose last

[5][7][12]. We use the term to mean a graph that shows the relationship of value assignment and usage during program execution.

value assignment occurred during action **n1**.

The above defined collection of nodes and arcs forms an acyclic directed graph. The nodes of the graph are topologically ordered by the natural order given in the execution trace. Since the graph is acyclic and has a topological order we were able to develop an algorithm which constructs such a graph from an execution trace. We refer to a graph constructed in the above described manner as a *trace graph*. In [6] we show that given an execution history, the corresponding trace graph (V,E) can be constructed in O(|E|+|V|) time. The graph for the trace illustrated in Figure 3.3 is given by:

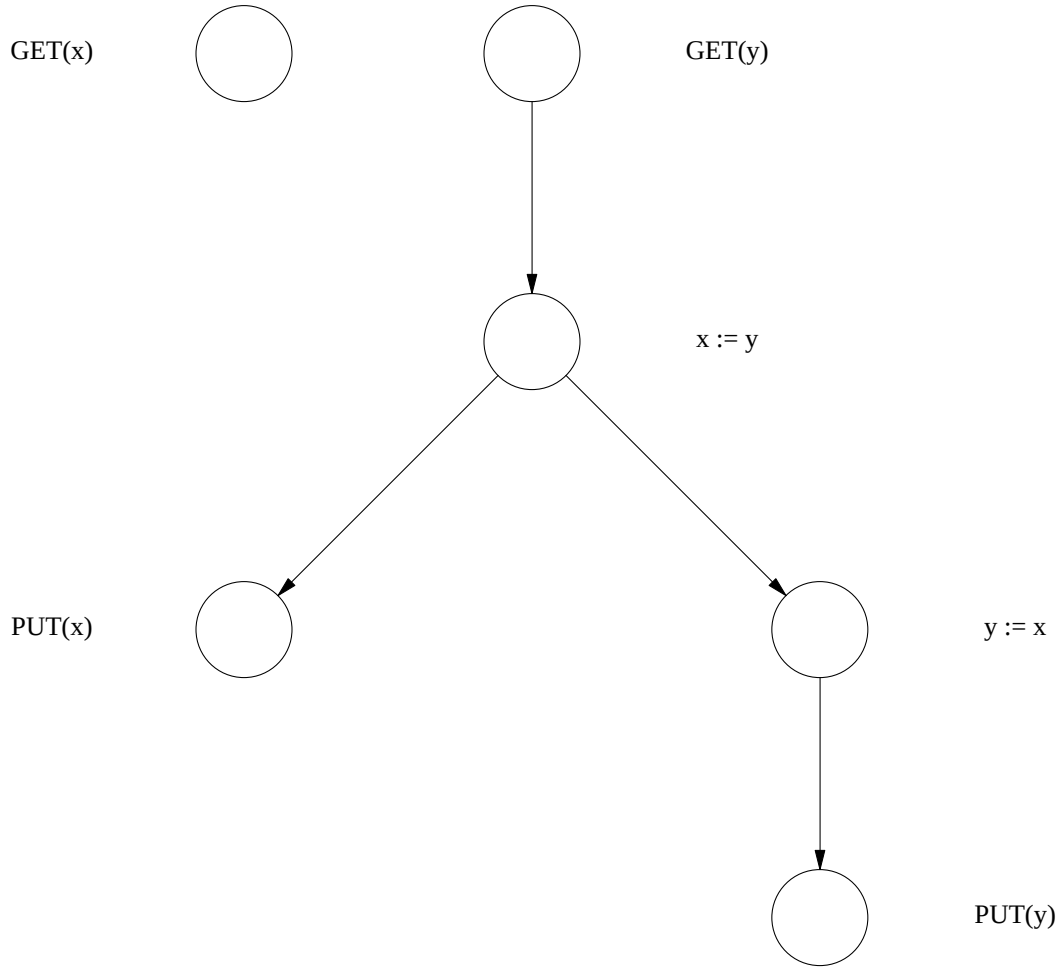

Figure 3.4

We add to our dependency graph knowledge determined by the correctness or falsity of the execution output. For a given program execution the oracle can provide information that allows us to divide the output values of the execution into two sets, those that are correct and those that are incorrect. We project this information onto the trace graph by labeling correct output nodes with a "1" and labeling incorrect output nodes with a "0". This correctness information about the output nodes can then be propagated through the trace graph to give information about other actions. Given the assumptions stated in section 1, correctness propagates upward in a graph, while incorrectness propagates downward. To complete our labeling, we label all input nodes with a "1" to indicate that we are assuming that all input is correct. The above trace graph labeling is called the *0/1 labeling* of the trace graph. Note a "0" at a node implies the value assigned or output at that node is incorrect, while a "1’ at a node implies the value assigned or output is correct. Many nodes are left unlabeled by the above propagation procedure. Some of these nodes will be labeled

later through information generated by answers to oracle questions, some will be labeled later by information generated by other executions and some will never get labeled. In [6] we show that given a trace graph the corresponding labeled trace graph (V,E) can be constructed in O(|E|+|V|) time.

The labeled trace graph of program SWAP is shown below. Solid circles represent nodes labeled with a "1", and circles containing "x's" represent nodes labeled with a "0".

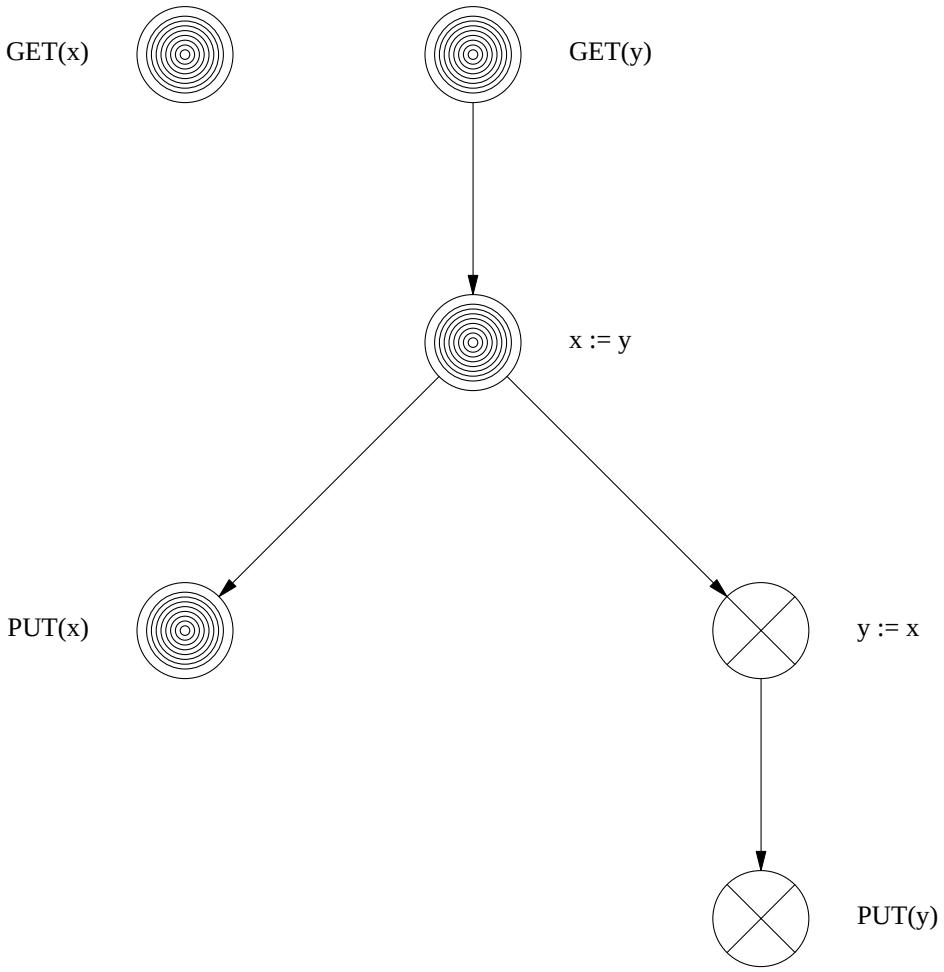

Figure 3.5

Most modern compilers identify uninitialized variables and dead statements. This task is also easily done by our debugger during model construction. Adding these tasks to graph construction does not change the complexity of the construction algorithm, see [6].

4. MODEL USAGE

Once the trace graph has been built and labeled, the debugger is ready for the oracle to tell it what direction to take next. Several approaches are open to the oracle. The debugger can lead the oracle in finding the program faults, the debugger can accept another execution history from the oracle or the debugger can suggest another test that the oracle might run. We present below a short discussion of several of the major issues underlying these approaches.

The most straightforward situation exists if the debugger is able to determine that a single program action is responsible for causing all of the incorrect output values. Such a situation is illustrated in Figure 4.1 where the single error assumption indicates that node **n2** is the responsible action.

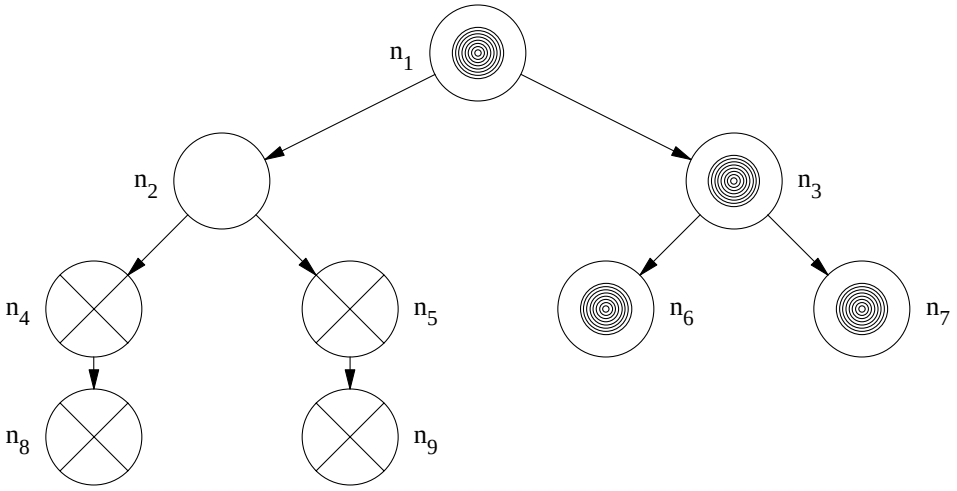

Figure 4.1

In situations such as this, it may be possible for the debugger to do all of the work itself. Usually, however, there comes a point where an oracle is needed. A "good" system makes the oracle's work as easy as possible. One way of doing this is by repeatedly weighing the questions that must be asked and always asking the question that will produce the greatest amount of information with the least work. There is no best way to do this. If we (for the time being) assume that the oracle can answer questions about all program statements with the same amount of effort, then a good weight function is one that weighs the number of additional graph nodes that can be labeled once an answer to the question is given. When the trace graph is a tree this weight function reduces to the one used by Shapiro in [16]. Some other assumptions that lead to weight functions suitable for use with an acyclic graph are the following.

- Statements close to input statements are easier for an oracle to answer questions about.
- Faults are most likely to occur close to output statements This produces an ordering like that illustrated in [10].
- The more complex the expression the greater the chance of error.

The Competent Programmer Hypothesis of Mutation Testing theory [1] states that incorrect program code is likely to differ by only a few small errors from correct program code. If you are willing to accept this hypothesis, the acyclic graph structure of our model provides a wealth of information about the location of program errors. For example, the system without the use of the oracle can determine if a single fault could not be responsible for causing all of the incorrect output, see Figure 4.2. In the case where a single fault is causing all the incorrect output, the system can indicate for the oracle possible locations of the fault. Coupling the single error assumption together with questioning by weight can reduce an oracle's work significantly.

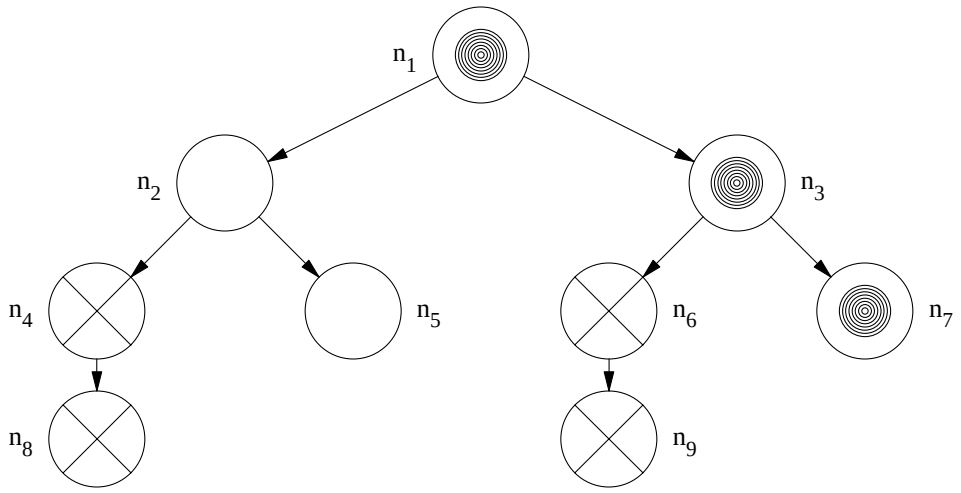

Figure 4.2

Figure 4.3 shows that two execution histories can produce the same trace graph with different labelings. The program is supposed to add the values of **x** and **y**; instead, it multiplies them. The difference in the output results (correct vs. incorrect) is due to the fact that the given code and "correct" code have a set of input values for which they generate the same output values. We will refer to this type of situation (for some input data, correct output being produced with incorrect code) as the *fixed point* phenomenon. We use this term since an easy proof that this phenomenon exists in "almost all" code is dependent on a well known theorem from mathematical analysis called the fixed point theorem[4]. Since you can never assume that the fixed point phenomenon does not exist in code being debugged, the phenomenon must be addressed by a debugging process.

Input(2,2)

Output(4)

Input(1,2)

Output(2)

Figure 4.3

The fixed point phenomenon can provide useful knowledge to debugging tools. Our tools demonstrate that a single execution history with a knowledge of the correctness of the output variables is enough information to automatically reduce the code space in which program faults are known to exist from the original program space to a smaller subspace. What happens when a second execution history is added? In a straight line program any two execution histories generate the same unlabeled graph. Since we are assuming the faults are never self-correcting, the fixed point phenomenon is the only cause for two labelings of a straight line graph to be different. Thus from two different labelings of one straight line program

graph we can produce a third "better" labeling. To do this we bitwise-and the labels together. This labeling gives us a more accurate view of the code in that there is less chance that the fixed point phenomenon occurs.

For a program having a conditional or a loop, two execution graphs that differ only in labels give us new information that we can not generate from any other source. In this case the same graph with two different labels can indicate the fixed point phenomenon, a branching fault, or a loop entry or exit fault.

As implied above once the system has received one execution history of a program, a second random execution of the same program may not provide a lot of new information. For this reason it might be better to let the system generate input data for a second program test. There are several ways a system can do this. A favorite of ours is the following one. Each action in an execution history can be written as a function of the input variables, see Figure 4.4 for example. Using this view of the actions any collection of conditionals becomes a system of inequalities and a solution to a system of inequalities becomes a set of test data. For a description of a system that implements a similar approach and a bibliography on related work, see Offutt's thesis [11].

Original Actions
get(n)
avg = 0
i = 1
i<=n
get(x)
avg = avg+x
i = i + 1
i<=n
get(x)
avg = avg+x
i = i + 1
i<n
avg = avg/i
put(avg)

Actions as a Function of Input
get(n1)
avg = 0
i = 1
(1) <= (n1)
get(x2)
avg = (0) + (x2)
i = (1) + 1
((1) + 1) <= (n1)
get(x3)
avg = ((0) + (x2)) + (x3)
i = ((1) + 1) + 1
(((1) + 1) + 1)) < (n1)
avg = (((0)+(x2))+(x3))/(((1)+1)+1)
put((((0)+(x2))+(x3))/(((1)+1)+1))

Figure 4.4

Writing the actions of an execution history in terms of the input is useful in more than the above situation. For example, this information can be used to advantage in forming oracle questions and to facilitate reevaluation of actions.

5. IMPLEMENTATION STATUS

The fault location tool can be viewed as three independent subtools. The first subtool is used to convert a program into an equivalent program that records, during program execution, a history of the original program actions. The second subtool is used to build the labeled trace graph. The third subtool is used to interact with the oracle.

The first and second subtools are complete as described in this paper. The first subtool is dependent on NewYacc [13]. The NewYacc tool allows one to associate rewrite rules with grammer productions so that high-level transformations of a source file can be easily generated. The third subtool is still under production. Currently it can lead an oracle to a fault location using the Competent Programmer Hypothesis or it can accept a second execution history from the user. A weight function similar to Shapiro's is used. When completed the third subtool will also be dependent on NewYacc. Here the NewYacc tool is used to rewrite execution actions as functions in terms of the input variables. A tool that solves systems of inequalities will also be needed for this subtool.

6. IMPLICATIONS

Our debugging tool shows that a single program execution history together with knowledge about the execution output values provides sufficient information for building an efficient semiautomatic environment in which an oracle can locate program faults. Our method of propagating "correctness" information throughout the graph generates information not available in any other execution history based debugging model. Our debugging tool also shows that weight functions applied to ordering oracle questions can reduce the amount of work the oracle must do by limiting the number of questions the oracle must answer.

Our current research is directed toward completing the implementation of the debugging tool. This will include a relaxation of the assumption that the oracle must provide consistent information along with a search algorithm for multiple errors and a test data generator.

Future research will include 1) expanding the tool to work on some standard procedural language, 2) expanding the model to include nonterminating executions, 3) experiments to determine which weight functions are preferred by oracles, and 4) a study of models from "real" programs to determine if any usable special graph structures occur in trace graphs.

REFERENCES

References

1. Acree, A. T., Budd, T. A., DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Mutation Analysis," GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA (September 1979).
2. Agrawal, H., Demillo, R., and Spafford, E., "A Process State Model to Relate Testing and Debugging," SERC-TR-27-P, Software Engineering Research Center, Purdue University (September 1988).
3. Agrawal, Hiralel and Horgan, Joseph R., "Dynamic Program Slicing," *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York (June 20-22, 1990). http://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/slicing.pdf.
4. Border, Kim C., *Fixed Point Theorems with Applications to Economics and Game Theory*, Cambridge University Press, New York, New York (1985).
5. Cytron, Ron, Ferrante, Jeanne, Rosen, Barry K., Wegman, Mark N., and Zadeck, F. Kenneth, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *Transactions on Programming Languages and Systems*, 13, 4, pp. 451-490 (October 1991).
6. Francel, Margaret Ann and Rugaber, Spencer, "The Value of Slicing while Debugging," *Science of Computer Programming*, 40, 2-3, pp. 151-169 (July, 2001).

7. Horwitz, S., Reps, T., and Binkly, D., “Interprocedural Slicing Using Dependence Graphs,” *Proceedings of the SIGPLAN’88 Conference on Programming Language Design and Implementation*, ACM Press, Atlanta, Georgia (June 22-22, 1988).

8. IEEE, “Standard Glossary of Software Engineering Terminology,” Standard 729-1983, New York, New York (1983).

9. Korel, B. and Laski, J., “Dynamic Program Slicing,” *Information Processing Letters*, 29, 20, pp. 352-357 (July 1984).

10. Korel, B. and Laski, J., “STAD - A System for Testing And Debugging,” *IEEE 2nd Workshop on Software Testing, Verification and Analysis*, pp. 13-20, Banff, Alberta, Canada (July 19-21, 1988).

11. Offutt, A. J., “Automatic Test Data Generation,” SERC-TR-25-P, Software Engineering Research Center, Purdue University.

12. Ottenstein, Karl J. and Ottenstein, Linda M., “The Program Dependence Graph in a Software Development Environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Pittsburgh, Pennsylvania (April 23-25, 1984). SigPlan Notices 19(5), May 1995.

13. Purtilo, James M. and Callahan, John R., “Parse Tree Annotations,” *Communications of the ACM*, 32, 12, pp. 1467-1477 (December 1989).

14. Renner, Scott A., “Diagnosis of Logical Errors in Pascal Programs,” UIUC-DCS-F-84-915, University of Illinois at Urbana (April 1984).

15. Shahmehri, Nahid, Kamkar, Mariam, and Fritzson, Peter, “Semi-automatic Bug Localization in Software Maintenance,” *Proceedings Conference on Software Maintenance*, pp. 30-36, San Diego, California (November 26-29, 1990).

16. Shapiro, Daniel G., “Sniffer: a System that Understands Bugs,” AI Memo No. 638, p. debugging, Massachusetts Institute of Technology (June 1981). Masters Thesis.

17. Weiser, Mark, “Program Slicing,” *Proceedings of the 5th International Conference on Software Engineering*, pp. 439-449, IEEE Computer Society, San Diego, California (March 9-12, 1981). http://dl.acm.org.prx.library.gatech.edu/ft_gateway.cfm?id=802557&ftid=66597&dwn=1&CFID=142594006&CFTOKEN=79681482.