PISCES: A Tool for Predicting Software Testability

Jeffrey M. Voas
Room 108
1121 Arlington Blvd.
RST Corp.
Arlington, VA 22209

Keith W. Miller
Dept. of Computer Science
101 Jones Hall
College of William & Mary
Williamsburg, VA 23185

Jeffery E. Payne
Room 108
1121 Arlington Blvd.
RST Corp.
Arlington VA 22209

Abstract

Before a program can fail, a software fault must be executed, that execution must alter the data state, and the incorrect data state must propagate to a state that results directly in an incorrect output. This paper describes a tool called PISCES (developed by Reliable Software Technologies Corporation) for predicting the probability that faults in a particular program location will accomplish all three of these steps causing program failure. PISCES is a tool that is used during software verification and validation to predict a program's testability.

1 Introduction

This paper presents a tool (*PISCES*) developed in C++ that implements the *PIE* technique defined in [8, 20, 6, 19, 22]. *PIE* (for Propagation, Infection, and Execution analysis) statistically estimates the probabilities of

1. a particular location in a program will be executed according to a particular input distribution (Execution),
2. a mutation of a location in a program will cause a change to the state of the program after it is executed (Infection), and
3. random values that are injected into a variable at a particular location in execution (meaning a change made to the state of the program) discernibly affect program output (Propagation).

The process for predicting the probability that a location is executed follows: the program is instrumented to print out when a particular location is executed. The instrumented program is then run some number of times with inputs selected at random according to some input distribution of the program. The proportion of inputs that cause the print command to be invoked in the instrumented program out of the total number of inputs on which the instrumented program is executed is an estimate of this probability. This probability estimate along with others for the software can then be used to predict the software's testability.

The process for predicting the probability that a fault in a location will affect the data state of the program is: a syntactic mutation is made to the location in question. The program with this mutated location is then run some number of times with inputs selected at random according to the program's input distribution. The proportion of times that the program with the mutated location produces a different data state (given that the mutated location is executed) than the original location out of the total number of times that the original location is executed is an estimate of this probability. For example, suppose that a program is executed 10 times, and during the 10 executions the original location is executed 1000 times, and 345 data states produced by the mutated program are different than what the original "unmutated" location produces, then our probability estimate is 0.345 with an associated confidence interval. It should be further mentioned that in general, many different syntactic mutants are made for a single location, each yielding a probability estimate in this manner. These probability estimates for this location along with those for other locations in the software can then be used to predict the software's testability.

The process for predicting the probability that a data state error will cause program failure given that a location creates a data state error follows: program execution is halted just after executing the location, a randomly generated data value is injected into some variable, and program execution is resumed. This process simulates the creation of a data state error during execution. We term this process "perturbing" a data state, since the value of a variable at some point

during execution represents a portion of a data state. The tool then observes any subsequent propagation of the perturbed data state to successor output states after execution is resumed. This process is repeated a fixed number of times, with each perturbed data state affecting the same variable at the same point in execution. For instance, assume that after performing this process on some variable 10 times the output is affected 3 of those times. Then the resulting probability estimate would be 0.3 with some confidence interval [3]. This process is performed using different variables as the recipients of the perturbed data states. Probability estimates found using the perturbed data states can be used to predict which regions of a program are likely and which regions are unlikely to propagate data state errors caused by genuine software faults. These probability estimates for this location along with those for other locations in the software can then be used to predict the software's testability.

2 Preliminary Definitions and Assumptions

Several terms require formal definition. A *data state* between two consecutive locations (where consecutive is determined dynamically) is a set of mappings between all declared and dynamically allocated variables and their values at that point in execution for a particular input. As a data state example, the data state {(a,5), (b,5), (pc,10)} records that variables a and b have the value 5, and that the program counter has the value 10.

A *location* is what Korel [5] terms a single instruction: an assignment, input statement, output statement, and the <condition> part of **if** or **while** statement. Currently, this tool only considers assignment statements and the <condition> part of an **if** or **while** statements. flow-of-control statements. Thus we restrict the application of the tool to these types of locations; other types of statements will eventually be included in our definition and analyzed by our tool. As our experience with the tool grows, we made add other types of statements.

The execution of a location is considered atomic, hence data states can only be viewed *between* locations. The definition of location does not include procedure calls; a procedure call is considered as having as many locations as the procedure itself.

A *data state error* is an incorrect variable/value pairing in a data state where correctness is determined by an assertion for that location. A data state error is frequently referred to as an *infection*, and these two terms are used interchangeably. If a data state error exists, the data state and variable with the incorrect value at that point are termed *infected*. A data state may have more than one infected variable.

If there exists at least one input from the program input distribution for which a program fails, then we say the program contains a *fault* with respect to the input distribution. Even though we may know that a fault exists in a program, we cannot in general identify a single location as the exclusive cause of the failure. For example, several locations may interact to cause the failure, or the program can be missing a required computation which could be inserted in many different places to correct the problem. However, if a program is annotated with assertions about the correct data state before and after a particular location l, and if there exists an input from the input distribution such that l's preceding data state is consistent with its assertion and the succeeding data state violates its assertion, then l contains a fault.

Propagation of a data state error occurs when a data state error affects the output. In this definition of propagation, we assume a data state error occurs after a location that either affects a variable or the program counter. *Cancellation* of a data state error occurs when infection occurs but propagation does not.

In order to perform *PIE* using *PISCES*, we expect that an input distribution exists for the program, and that the input distribution is the same one that will be used during testing of the program. Since we hope to make predictions concerning where faults are actually occurring, we must use the same input distribution that is used during testing to uncover the existence of faults.

3 The Fault/Failure Model

PIE simulates the conditions of the three part fault/failure model [10, 9, 17]. The fault/failure model is simply the following three conditions that are both necessary and sufficient for software failure to occur. Also, the conditions must occur in the following sequence.

1. The input must cause the fault to be *executed*.
2. Once the fault is executed, the succeeding data state must contain a *data state error*.
3. Once the data state error is created, the data state error must *propagate* to an output state.

```
{specification: output 1 if a^2 + b^2 + c^2 < 900000
else output 0}
{1} read(a);
{2} read(b);
{3} read(c);
{4} if (c = 50) then
{5}    d:= sqr(a) * a
  else
{6}    d := sqr(a);
{7} e := sqr(b);
{8} f := sqr(c);
{9} if ((d + e + f) < 900000) then
{10}     writeln("1")
  else
{11}     writeln("0");
```

Figure 1: Program P.

Software has high quality if it produces correct outputs for each possible input. When software produces an incorrect input, it has failed. The fault/failure model relates program inputs, faults, data state errors, and failures. Since faults and data state errors can lead to failures, this model is tightly tied to software quality. any theory of software quality must take account of these three steps. §3 demonstrates the model by placing a fault in a small toy program—this demonstration shows the four types of execution scenarios that can occur for a given (input, fault) pair. The model is further described in [10, 9, 17].

The program P in Figure 1 is intended to display the function shown in the braces. P has been coded incorrectly. Although there are many ways of correcting this program we will consider only one here: if the fault in location 5, `* a`, were removed from location 5, P would be correct.

Each execution of P falls into one of four scenarios:

1. The fault is not executed;
2. The fault is executed, but no data state error is created;
3. The fault is executed, some data state error or data state errors are created, but the program output is correct anyway; or
4. The fault is executed, a data state error occurs, and the data state error causes incorrect output.

Only executions of the final scenario type make the fault *visible* during random testing. Propagation analysis studies the probabilities of scenarios 3 and 4 occurring. For a fixed fault and a given input distribution,

Location	a	b	c	d	e	f	Output
1	0	↑	↑	↑	↑	↑	
5	0	0	49	0	↑	↑	
7	0	0	49	0	0	↑	
10	0	0	49	0	0	2401	1

Table 1: $(0, 0, 49)$ as input to P.

Location	a	b	c	d	e	f	Output
1	0	↑	↑	↑	↑	↑	
5	0	5	50	0	↑	↑	
7	0	5	50	0	25	↑	
10	0	5	50	0	25	2500	1

Table 2: $(0, 5, 50)$ as input to P.

Pr[Scenario (3) occurring] = 1−Pr[Scenario (4) occurring].

We next briefly present an example of each type of scenario. §5 explains the tool.

1. Scenario 1: The execution for the input $(a, b, c) = (0, 0, 49)$ is displayed in Table 1. (In the following tables, ↑ represents an undefined value.) The value of $c = 49$ causes the selection of a path that does not include the fault. Clearly any such execution will not fail.
2. Scenario 2: The execution for input (0,5,50) is shown in Table 2. The fault is reached but the computation proceeds just as if there were no fault present, because $a = 0$ prevents the fault from impacting the execution. No data state error has been created.
3. Scenario 3: For input (5,1,50) the fault creates a data state error in the succeeding data state, producing $d = 125$ instead of $d = 25$ (See Table 3). This data state error then *propagates* to location 7 where it is *cancelled* by the predicate, because the predicate is true with or without the fault.
4. Scenario 4: Executing the program with input (700,0,50) executes the fault which then creates a data state error in the succeeding data state in such a way that the data state error propagates to the output (See Table 4).

Scenario (1) demonstrates that an execution of P can only reveal information about the portion of P that is executed. Scenarios (2) and (3) provide a false sense of security to a tester because the fault is executed but no visible failure results. Scenario (4) illustrates the three necessary and sufficient conditions for a fault to produce a failure.

Location	a	b	c	d	e	f	Output
1	5	↑	↑	↑	↑	↑	
5	5	1	50	125	↑	↑	
7	5	1	50	125	1	↑	
10	5	1	50	125	1	2500	1

Table 3: $(5, 1, 50)$ as input to P.

Location	a	b	c	d	e	f	Output
1	700	↑	↑	↑	↑	↑	
5	700	0	50	343000000	↑	↑	
7	700	0	50	343000000	0	↑	
11	700	0	50	343000000	0	2500	0

Table 4: $(700, 0, 50)$ as input to P.

The fault/failure model and the example above illustrate the mechanisms by which a fault may or may not propagate an error. Ideally, a static analysis would be devised that would predict, for a given input distribution, the propagation characteristics of a given program location. Unfortunately, no such static analysis exists, nor is there likely to be any such static analysis in the near future. Instead, the remainder of the paper describes a dynamic technique that estimates testability.

4 The Propagation Analysis Algorithm

Propagation analysis estimates the probability that an altered data state will *propagate* it's effect to the program output. To make this estimate, a data state is *perturbed* (or altered) by changing the value of one live variable in the data state. A variable is considered *live* if the variable has *any* potential of affecting an output computation. The set of variables that are live is determined statically using data-flow analysis [23]. After perturbing a data state, program execution resumes until termination (assuming no infinite loop occurs); the following algorithm tells how to handle the suspected occurrence of an infinite loop caused by perturbed data states.

This process of first perturbing a live variable's value and then resuming execution is repeated many times. This repeated process yields the number of instances k out of n attempts where perturbing a live variable a at some program location l made a discernible effect in the program's output. The value $\frac{k}{n}$ is a rough estimate of the probability that an altered value to variable a will affect the program output.

The probability that live variable a, whose value is perturbed at program location l, affects program output is denoted by $\psi_{l,a}$. The estimate of this probability is termed a *propagation estimate*; $\hat{\psi}_{l,a}$ denotes this probability estimate and is computed according to the following algorithm:

1. Set variable **count_prop** to 0.

2. Randomly select a data state from the space of data states that occur immediately after program location l.

3. Perturb the value of live variable a in this data state, and execute the code that succeeds program location l on both the perturbed and original data states. There are issues that must be addressed concerning how to perturb a value in a data state, but they are out of the scope of this paper. Those issues are addressed in [19, 6].

4. If a different result occurs in the program output between the perturbed data state and the original data state, increment **count_prop**; also, set a time limit for termination of the program executing on the perturbed data state, and if execution is not finished in that time interval, increment **count_prop**.

5. Repeat algorithm steps 2-4 n times.

6. Divide **count_prop** by n yielding $\hat{\psi}_{l,a}$.

(More specific details of the formal propagation analysis algorithm can be found in [6, 19].)

A propagation estimate is a function of the program location, the live variable, the input distribution, the space of data states occurring after the program location, the method used to perturb a value, and the code that is potentially executed after the program location. Propagation analysis finds a propagation estimate for each member of the set of live variables at each program location. This produces a large set of propagation estimates—one propagation estimate per live variable per program location.

A propagation estimate can be thought of as an estimate of the effect that a live variable at a program location has on the program's output. For instance, if some live variable's value can be changed at some program location l without producing any evidence of this change in the program's output (i.e., a tiny propagation estimate), then we have evidence that this variable at this program location has little effect on the program's computation. It may be that this variable has a greater effect on the program's behavior at some other program location.

4.1 Practical Significance of Propagation Analysis

We anticipate that propagation analysis will be immediately useful in two major tasks during software development:

1. Locating sections of code that do not propagate data state errors.
2. Investigating output behavior when data states errors are created.

Both of these tasks are critical to improving software quality. Statements and group of statements that do not propagate data state errors are likely to hide faults from testing. Locating these statements is critical to both programmers and testers. Programmers should be wary of code regions where data state error propagation is unlikely to occur, and should only allow such code when it is part of the required functionality. (Some specifications do include inherently low propagation functions. See [21].) When these code sections are required despite their low propagation, the programmer must focus analysis other than random testing on these sections, since random testing is unlikely to uncover problems there. A tester or quality control analyst should also approach low propagation code differently than other code; random testing cannot be relied on to assure the quality of that code.

The process of propagation analysis can also be used to explore the code's behavior during debugging or maintenance. Traditionally, programmers use different inputs to a program to debug programs. More sophisticated recent debugging environments allow a programmer to halt execution, change data values, and resume execution. Propagation analysis suggests a further refinement of debugging technology: automatically perturbing the data state of a variable (or variables) and observing the effect on the output. As with debugging environments, a facility for propagation analysis can help confirm or reject hypotheses about failure producing code, and can explore the possible ramifications of proposed changes during code maintenance. As propagation analysis improves the process of debugging, it improves the quality of the resulting code.

By hand, propagation analysis is time consuming and tedious. Fortunately, almost all the analysis can be automated. There are certain undecidable problems that can occur during propagation analysis, but all of these problems can be inelegantly avoided in the automated system. For example, whether a program that has had an injected data state error will halt is not decidable. We avoid this problem by placing a time limit on each execution; if termination has not occurred within that time limit, we assume that termination will never occur and we end the computation, labeling the execution as a propagated error. Since the time limit is arbitrary, the tool may label a long but eventual correct computation as a propagated error. Although inelegant, the time limit solution is both feasible and practical.

Propagation estimates are both practical and relevant for software development; to make such estimates more conveniently, *PISCES* allows a user different ways to exploit propagation analysis in an interactive environment.

4.2 *PISCES* and Propagation Analysis

Propagation analysis is fundamentally a three stage process:

1. Obtain a data state at a location in the code.
2. Perturb the data state.
3. Execute to completion and examine the resulting output to see if the perturbed data state has changed the output.

A software tool for propagation analysis can offer an enormous savings in the manual effort required in each of these stages. It also offers flexibility by providing additional features that are impractical in a manual technique. Several of these additional features will be mentioned in this section.

In the following sections, we refer to *PISCES* meaning the tool in its current design. Not all the functions described have been completely implemented.

Obtaining a data state at a location can be accomplished by executing data from the start of the program, or can be set arbitrarily by the user. The data state can be perturbed manually or can be done automatically using a perturbation function and the current data state. "Completion" might be defined as some internal point within the program other than the end of execution as well, e.g., an intermediate data state, such as used in *firm mutation testing*[1]. After all the aspects are specified, *PISCES* performs the calculations and delivers the results in a convenient form.

When *PISCES* is started, it prompts the interactive user to specify several different aspects required to define the desired propagation analysis. Each subsection below describes one required aspect. After these selections are made (or defaults taken), *PISCES* performs all executions necessary for the analysis that is

specified by the user, and produces the requested results. This cycle of specify-execute-report is termed a *session*. The tool can cycle through multiple sessions without being restarted, by allowing the user to either perform additional analyses on the program that is currently loaded into *PISCES*, or by allowing the user to bring in a different program for analysis.

4.2.1 Selecting the Granularity of Analysis

Propagation analysis can be performed for a single location, a group of locations, or for an entire program. Although the analysis does not require an oracle (as does testing), the analysis can consume considerable computer resources (the algorithm is quadratic in the number of locations executed [18]). Therefore the user should be allowed to specify exactly what code is to be analyzed in order to not perform a complete analysis if that is not desired. Currently, the tool allows the user to specify:

1. The entire program.
2. One subprogram (procedure or function) at a time.
3. A section of contiguous location within a program or procedure or function.
4. One location at a time.

If option 1 is selected, *PISCES* identifies all locations of the program that are either assignment statements, or <condition> parts of **if** and **while** statements. If option 2 is selected, the tool identifies all locations of a module (that the user must then specify) that are either assignment statements, or <condition> parts of **if** and **while** statements. If option 3 is selected, *PISCES* identifies all locations of a code segment (that the user must then specify) that are either assignment statements, or <condition> parts of **if** and **while** statements. And if option 4 is selected, the user must specify the exact location where the analysis is to be performed. Analysis will subsequently be performed after these and the options below are specified.

4.2.2 Completing an Execution

Propagation analysis involves multiple executions for each location. For each such execution, the user must specify a criterion which identifies completion of an execution. By default, program completion is the criterion as shown in the algorithm, but *PISCES* also allows the user to specify completion as reaching some successor location of the location under analysis in the code. The notion of a successor location is made dynamically, not statically. Note that there are some locations that are both successor and predecessor locations of other locations. (An alternative to program completion is required when, for example, the program is designed to not terminate.)

If the analyst specifies a completion location, that location should be executed for all the executions possible from the location being exercised. If the completion point is not reached before program completion, program completion will successfully end the analysis. However, if a program does *not* complete and does not execute the specified completion location, the session will halt only after the arbitrary time limit described in the previous section.

Theoretically, a different stopping criterion could be specified for each location the user wants exercised in a single tool session. In its current design, however, *PISCES* requires a single completion criterion for each session.

4.2.3 Selecting Variables to be Perturbed

At a given location, more than one variable may be involved in the computation. The user of the propagation tool may want one, all, or only some of these variables perturbed. In the example above, only variable a was perturbed; however, *PISCES* allows the user to specify combinations of variables to be perturbed. The tool gives the following choices for selecting the variables to be perturbed during the analysis:

1. If an entire program or subprogram is to be analyzed, two choices are available:
   (a) Perturb all statically declared and dynamically allocated variables. Note in the algorithm, only 1 variable is perturbed repeated times to produce a single propagation estimate. However in *PISCES*, we allow the user, if they wish, to perturb combinations of variables, i.e., not necessarily one variable at a time.
   (b) All variables that are live. These are variables that do have the potential of affecting the output. If the first option is used, then potentially variables will be perturbed in a section of the code where they truly have no impact. This option is an optimization of the first option.
2. If a section of code or a single statement is to be analyzed, two choices are available:

(a) All statically declared and dynamically allocated variables perturbed.

(b) User selected variables perturbed throughout the section.

4.2.4 Specifying the Number of Sampled Data States for a Single Propagation Estimate

Propagation analysis is an empirical technique that relies on program executions to predict future program behavior. As such, more executions should lead to generally better predictions, because of the decrease in the variance associated with an estimate. In more technical terms, the more often we perform propagation analysis on a particular variable at a particular location, the lesser the width of the resulting confidence interval that is associated with the propagation estimate. The improved results (meaning the lesser the width) must be balanced against the increased computational time required to complete the analysis.

The propagation algorithm calls for a constant as the default number of data states that will be sampled for each variable selected for perturbation at a given location. This is the n that is required in Step 5 of the algorithm. The site-specific default can be overridden by the *PISCES* user. For a given session, the tool requires a single number of sampled data states for all the variables to be perturbed.

4.2.5 Specifying the Source of Data States

For a given location, the data states that the algorithm needs can either come from executing a program input until that location is reached, thus producing a complete data state, or the user can specify the values that they want to use as the data state. Also, the potential exists for storing data states in a file from previous executions from which data states can be sampled. Storage is still a feature that we are studying, and have not yet implemented due to the feasibility problems of storing and retrieving such a large amount of information.

When an entire program or subprogram is being analyzed, *PISCES* requires that input to the program be that of some input distribution, preferably the operational distribution as described in §2 (more on this option in the next section). If a single location is specified for a session, the tool allows three options for obtaining data states:

1. Input data can be executed in order to create one or more data states at that location.

2. The user can manually specify an initial data state for the location.

3. The user can identify a file with data states for the location that they desire are used.

For each perturbation of a variable at a location, a new data state could theoretically be used. However, *PISCES* defaults to using a single data state several times before using a new data state. The number of times each data state is used can be adjusted by the user. If the user defines data states manually or with a file, these data states are used over if more data states are required by the session than the defined data states. As an example, suppose that data state $\mathcal{S}$ occurs at a location where we wish to perturb several different variables at different times during the analysis. Then $\mathcal{S}$ is used once for each of these variables.

4.2.6 Specifying a Source of Program Input Data

Program input data can be specified in two ways: a file of possible values or input distributions. Currently the input distributions consist of a range of values and a named standard definition (e.g., uniform or normal) with parameters when appropriate. If the file specified by the user does not contain enough data for the session, the data are reused. Inputs are needed to create the data states described in §6.5.

4.2.7 Specifying Perturbation Functions

When a session perturbs a variable, *PISCES* uses a *perturbation function*. A perturbation function is the mechanism for perturbing a data state. This mechanism can perform its duties at various levels, depending on the way in which *PISCES* is implemented at a particular installation. For instance, one method of perturbing a data state is to actually inject source code into the source program statically. This can then be compiled and executed. Another method is to actually inject object code into the object program statically. This is a bit more difficult, but can be accomplished. Yet a third way is to monitor an executing program externally and dynamically halt the program and alter some memory location that it currently has allocated to it. *PISCES*, in its first implementation, performs the first of these.

The default perturbation function is a uniform distribution over the range $0.5x \ldots 1.5x$ where x is the current value of the variable in the data state. However, *PISCES* allows the user to define a different

range and a different standard distribution as the default. The user can also define a specific perturbation function for any variable instance in the program. To insure that results are repeatable, the perturbations are implemented with a Lehmer random number generator with a fixed initial seed [7].

4.2.8 Specifying on Which Iterations to Perturb a Variable

Another concern that the user will be asked to decide is upon which iterations of the location do they wish to have the perturbation invoked. For instance, suppose that a location is in a loop, and on a given execution it is usually executed 1000 times. Then the user will need to specify on which of those 1000 iterations to apply a perturbation function. As it stands now, *PISCES* can perturb on

1. All iterations of a location.
2. Or the user can manually specify each individual iteration of the location on which they want the perturbation function applied.

Most of our experimental evidence suggests that the first option of these is the better. The conjecture that this suggestion is based on can be simply stated:

Conjecture 1 *In general, most faults create data state errors on each iteration for which they are executed.*

This conjecture is still under review.

4.2.9 Displaying Results

By default, all results are written to a file and a summary can displayed on the screen. Since the results of propagation analysis on all locations will produce huge amounts of information as well as take a long period of time for a program of any size, the user will undoubtedly opt to send that information to a file. If the user is only finding a single propagation estimate, then the wait will be less severe, and displaying the information on the screen will become a practical alternative. The user can optionally browse through the results in the file and ask for a more detailed summary without leaving *PISCES*. Statistics from multiple sessions concerning a single program can be accumulated. However, the user is cautioned against accumulating data from different versions of a program, since any changes in source code make the results from previous sessions incompatible with the new source code results.

5 The Infection Analysis Algorithm

Infection analysis estimates the probability that a syntactic mutant injected into a program location will will cause a discernible difference in the data state that results when the mutant is executed. In other words, will a particular syntactic mutant of a program location produce a value in the resulting data state that is visibly different than the value that is produced by the original program location?

Infection analysis is a technique that is similar to *weak mutation testing* [2]; what is different in these techniques is the information collected. In infection analysis, a set of syntactic mutants is created for each program location. A restriction is placed on each syntactic mutant—the syntactic mutant must also be semantically different for at least one input to the program. This means that there must be at least one program input such that when the syntactic mutant is executed, the resulting data state *is* different.

Without this requirement, a syntactic mutant could be used during infection analysis that is not discernible when the resulting data states are compared. In this situation, such a syntactic mutant is functionally equivalent to the original program location for all inputs given a particular input domain. For the revealing ability metric, such a situation is unacceptable, because it suggests that the program location has a greater ability to protect faults from detection, which cannot be justified from such a mutant. For example, $\mathbf{x = x + 1}$ would not qualify as a syntactic mutant of $\mathbf{x = 1 + x}$.

Infection analysis creates a set of syntactic mutants for each program location. After creating a program location's mutant set, each syntactic mutant and original location is executed with a data state that is selected at random from the space of data states that occurs before the location. This process is repeated; thus many data states are selected. This repeated process yields the number of instances k out of n attempts where injected syntactic mutant p discernibly affects the data state in a manner that is different than the original location. The value $\frac{k}{n}$ is a rough estimate of the probability that injected syntactic mutant p into program location l discernibly affects the data state. The goal is to determine the effect, $\frac{k}{n}$, for each of these syntactic mutants.

The probability that syntactic mutant p affects the data state differently than the original location does is denoted by $\lambda_{l,p}$. An estimate of this probability is termed an infection estimate and is denoted by $\hat{\lambda}_{l,p}$; $\hat{\lambda}_{l,p}$ is computed according to the following algorithm:

1. Set variable **count_inf** to 0.
2. Create a syntactic mutant for program location l denoted as p.
3. Present the original program location l and the syntactic mutant p with a randomly selected data state from the space of data states that occur immediately prior to program location l, and execute both program locations in parallel (the space of data states used here is expected to be a function of the same program input distribution that is used in §4.2.6).
4. Compare the resulting data states and increment **count_inf** when the result computed by p does not equal the result computed by l for this particular data state.
5. Repeat algorithm steps 3 and 4 n times.
6. Divide **count_inf** by n yielding $\hat{\lambda}_{l,p}$.

An infection estimate is a function of the program location, the syntactic mutant used, and the space of data states that occur immediately before the program location.

The mutants that have been used in this research have been limited to mutants of arithmetic expressions and predicates. For arithmetic expressions, the mutants considered in our research are limited to single changes to a location—this is similar to the mutations used in mutation testing [15, 2, 11, 12]. Our assignment statement mutants include: (1) a wrong variable substitution, (2) a variable substituted for a constant, (3) a constant substituted for a variable, (3) expression omission, (4) a variable that should have been replaced by a polynomial of degree k, and (5) a wrong operator. For boolean predicates, the mutants have included: (1) substituting a wrong variable, (2) exchanging **and** and **or**, and (3) substituting a wrong equality/inequality operator. We have purposely limited the syntactic changes to single changes to avoid the explosion that occurs in the number of combinatorial changes that could be made at each location.

In summary, infection analysis is a strengthened weak mutation testing. Infection analysis extends weak mutation testing to reveal information about the data state. All of the problems associated with generating mutants in mutation testing exist here as well. In this initial stage of our research, we have closely followed the mutation techniques developed by [15, 2, 11, 12]; as our experience with *PIE* increases, we expect to gain insight into the strengths and weaknesses of different mutation techniques.

5.1 Practical Significance of Infection Analysis

We anticipate that infection analysis will be useful in locating sections of code that do not create data state errors even when faults exist in these code sections during software development. Statements or groups of statements that do not create data state errors are likely to hide faults during testing. We admit however, that just as infection analysis is the "weakest" part of the *PIE* model, any tool that implements infection analysis inherits its weaknesses. This weakness comes from reliance on syntactic mutations, which have been argued against for years with fault-seeding techniques.

5.2 *PISCES* and Infection Analysis

Infection analysis is essentially a three stage process:

1. Obtain a data state immediately before a location in the code.
2. Mutate the location.
3. Execute the mutated location on the data state and see if the resulting data state is different than the data state the the original "un-mutated" location would have produced.

Like propagation analysis, a software tool for performing infection analysis can save enormous effort. This section describes some of the more important features that this tool provides.

5.2.1 Selecting the Granularity of Analysis

The granularity at which infection analysis is currently designed to be performed is at the location level. This includes all assignment statements and <condition> parts of **if** and **while** statements. The options currently available for infection analysis are more limited than for propagation analysis. We can, however, tell *PISCES* to perform infection analysis at each location within a block of locations. In this case, the user allows *PISCES* all control as to which mutants are made at these locations. If the user wishes to individually determine which mutants are used, see §5.2.3.

5.2.2 Infection Analysis Preprocessor

Before infection analysis is performed, a LALR(1) parser creates the abstract syntax tree (AST) of the location. Once the AST of the location is created, the

various rules for how to mutate can be applied to the various nodes in the tree. Thus this preprocessor is executed before any mutants are actually generated, once the user has determined which location the analysis is to be performed at.

5.2.3 Selecting the Mutants to be Used

Our tool allows the user to select either all possible mutants that can be created for the location (according to a set of predefined mutations defined above), or allows the user to enter manually a single mutation that is desired. If the user selects the "all mutants" option, then a infection estimate is produced for each mutant that is created. If the user enters a mutant manually, then only a single infection estimate is produced.

5.2.4 Specifying the Number of Sampled Data States for a Single Infection Estimate

Similar process as defined in §4.2.4.

5.2.5 Specifying the Source of Data State

Similar process as defined in §4.2.5.

5.2.6 Specifying the Source of Program Input Data

Similar process as defined in §4.2.6.

5.2.7 Specifying on Which Iterations to Sample the Data States

For locations that are potentially repeated by a single input, there is the possibility of more than one data state being available that is created by that input. We denote this set of data states by $\mathcal{B}$. In this case, we allow the user to either select a specific data state, for instance, the 4th data state that is created (if a 4th data state is created). Otherwise, *PISCES* uniformly selects a data state within $\mathcal{B}$.

5.2.8 Displaying Results

Similar process as defined in §4.2.9.

6 The Execution Analysis Algorithm

Execution analysis is the most straightforward and least computationally expensive of the three techniques. *Execution analysis* requires a program input distribution. Execution analysis executes the code with randomly selected inputs consistent with the input distribution and records which program locations are executed by each input. It is preferred that the input distribution used during execution analysis will be the *operational distribution*, since this is the distribution that is expected to be used during program testing. It is required that the input distribution used during execution analysis will be the same distribution that is used to create the internal data state spaces used by infection analysis and propagation analysis.

The probability that a particular program location l is executed by a randomly chosen input from this input distribution is denoted by ε_l. Execution analysis produces estimates of ε_l. This probability estimate is denoted by $\hat{\varepsilon}_l$ and is termed an *execution estimate*; the method for computing $\hat{\varepsilon}_l$ is given in the following algorithm:

1. Set array **count_exec** to zeroes, where the size of **count_exec** is the number of program locations.
2. Instrument the program with **write** statements at each program location that print the program location number when the program location is executed. Make sure that if a program location is repeated more than once for some input, the **write** statement for that program location is only executed once for that input.
3. Execute n randomly selected program inputs on the instrumented program, producing n strings of program location numbers. It is assumed that the program halts for each of these n inputs, however if it is suspected that this will not be true, even with the close to being semantically and syntactically correct assumption, then a process must be inserted to ignore inputs on which execution analysis *appears* to not be terminating and hence select new inputs. This process will be performed until the program is executed to termination on n inputs.
4. For
   each program location number l in each string, increment the corresponding **count_exec**[l]. If it happens that some program location k is executed on each of the n inputs, then when Step 4 is completed, **count_exec**[k] would equal n.
5. Divide each **count_exec**[l] by n yielding an execution estimate $\hat{\varepsilon}_l$.

An execution estimate is a function of the program, the program location, and a particular input distribution. If the input distribution is changed, then the

estimates of execution analysis, as well as the probability estimates from infection analysis and propagation analysis, will in all likelihood change.

6.1 Practical Significance of Execution Analysis

We anticipate that execution analysis will be immediately useful in locating sections of code that are not frequently executed. Statements or groups of statements that are not frequently executed are likely to hide faults during testing. Execution analysis is both the "cheapest" in terms of computer time to execute and implement of the three *PIE* analyses.

6.2 *PISCES* and Execution Analysis

Execution analysis is essentially a three stage process:

1. Instrument the code to reveal when a location is executed.

2. Execute the instrumented code many times.

3. Find the frequency with which a particular location is executed.

Like the other two analyses, a software tool for performing execution analysis can save enormous effort. This section describes some of the more important features that this tool provides.

6.2.1 Execution Analysis Preprocessor

Before *PISCES* can perform execution analysis, a preprocessor must run in order to uniquely identify all syntactic structures satisfying the definition for a location as defined in §2. This preprocessor will insert "comments" into the program that uniquely numbers locations, e.g., when the preprocessor parses the statement **x = x * 5**, it will modify the source program to look something like **/* locationnumber */ x = x * 5**. However for the syntax **main()**, this preprocessor will not assign a unique location number, since this does not satisfy the definition as a location. The modified source code that results after the preprocessor executes is the source code that infection analysis and propagation analysis reads in. This execution analysis preprocessor is the first subprocess of *PISCES* that is executed.

6.2.2 Selecting the Granularity of Analysis

The granularity at which execution analysis is currently performed is the location level (as described in §6.2.1). This includes all assignment statements and <condition> parts of **if** and **while** statements. The granularity options that are currently available for execution analysis are more limited than for propagation analysis.

6.3 Specifying the Number of Inputs for a Single Execution Estimate

The greater the number of inputs selected, the smaller the width of the accompanying confidence intervals of the execution estimates. Each execution estimate has an associated confidence interval, given a particular level of confidence and the value of n used in the algorithm. The computational resources available when execution analysis is performed will determine n in the algorithm. For example, for 95% confidence, the confidence interval is approximately $p \pm 2\sqrt{p(1-p)/n}$, where p is the $\frac{\text{number of occurrences of some event A}}{\text{total number of attempts of event A}}$ (p is the sample mean) [13, 3]. Since the n used in the algorithm are expected to be large, $2\sqrt{p(1-p)/n}$ will likely be insignificant. Unless p is close to 0 or 1, for $n = 10^4$, $2\sqrt{p(1-p)/n}$ is approximately 0.01; for $n = 4 \times 10^4$, $2\sqrt{p(1-p)/n}$ is approximately 0.005 [13].

6.3.1 Specifying the Source of Program Input Data

Similar process as defined in §4.2.6.

6.3.2 Displaying Results

Similar process as defined in §4.2.9.

7 Performing *PISCES* in its Entirety

Although *PISCES* can be forced to perform an single type of analysis, most users will will want to perform all three analyses on their software. Thus the question arises concerning which analysis should be performed when.

In *PISCES*, execution analysis is performed first. Execution analysis can indicate to the user whether the needed data states will be available for infection and propagation analysis. Also, the execution analysis algorithm can be used as a "barometer" for determining how long much computer time the other analyses

will require to finish. A feature can easily be added to *PISCES* that will estimate for the user how long the other two analyses will require given the length of time execution analysis required. Which analysis is performed after execution analysis does not matter. This is decided by the user.

8 *PISCES* Testability Postprocessor

We have described in elaborate detail how this tool performs the three main analyses. We have yet to describe how a prediction of testability is achieved from the results of these analyses.

Assuming *PISCES* is performed in its entirety, we have:

1. The estimate of the probability that a location is executed;
2. The estimates of the probabilities, one estimate for each syntactic mutant at the location, that given the location is executed, the mutant will adversely affect the program state; and
3. The estimates of the probabilities, one estimate for each live variable at the location, that given that the variable in the program state following the location is infected, the output will be changed.

A *PISCES* testability postprocessor inputs all probability estimates and allows the user several choices (based on granularity) of how the testability predictions will be displayed: either for a location, module, or for the entire program. The equations governing how these testability predictions are calculated are found in [6]. This postprocessor uses the equations from the cited reference and displays testability predictions. If the user solely wants the probability estimates and not the testability predictions, this postprocessor can be turned off.

9 Summary

This paper has described a tool for studying the likelihood that faults will be revealed during random testing. This tool differs from previous fault-based testing tools. For example, this tool is not solely based on syntactic mutations such as the MOTHRA tool [4, 16, 14].

The value of this tool to software quality is twofold: improved testing, and improved debugging. The tool aids in testing by predicting locations likely to hide faults. This has enormous importance for software that is classified as critical, meaning software where even the most unlikely failure can result in a loss-of-life. For example, if we were to predict that faults are likely to result in failures, and after testing we observed no failures, then we are somewhat more confident that faults are not hiding in our program. However if we predict that faults are unlikely to result in failures, then we are less confident that faults are not hiding. Furthermore, when a program is identified as not likely to produce failures when faults exist, *PISCES* identifies those locations in the code most likely to hide the faults. In this manner, this tool provides us with information about the program's testability and focuses attention on locations where testability is particularly low.

Acknowledgements

This work has been funded by a National Research Council NASA-Langley Resident Research Associateship, NASA Grant NAG-1-884, and RST Corp. Since collaborating on this paper at NASA-Langley Research Center, Voas has accepted a position at Reliable Software Technologies Corporation.

References

[1] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.

[2] Willam E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[3] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.

[4] R. DeMillo, D. Guindi, W. McCraken, A. Offutt, and K. King. An Extended Overview of the Mothra Software Testing Environment. *Proceedings of Second Workshop on Software Testing Verification and Analysis*, July 1988.

[5] Bodgan Korel. PELAS-Program Error-Locating Assistant System. *IEEE Transactions*

on Software Engineering, SE-14(9), September 1988.

[6] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[7] Stephen K. Park and Keith W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[8] J. Voas and L. J. Morell. Applying Sensitivity Analysis Estimates to a Minimum Failure Probability for Software Testing. In *Proc. of the 8th Pacific Northwest Software Quality Conf.*, pages 362–371, Portland, OR, October 1990. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.

[9] L. J. Morell. Theoretical Insights into Fault-Based Testing. *Second Workshop on Software Testing, Validation, and Analysis*, pages 45–62, July 1988.

[10] Larry Joe Morell. A Theory of Error-based Testing. Technical Report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[11] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[12] A. J. Offutt. The Coupling Effect: Fact or Fiction. *Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification*, December 1989. Key West, FL.

[13] S. K. Park. Lecture notes on simulation, version 3.0. Department of Computer Science, College of William and Mary in Virginia, 1990.

[14] ByoungJu Choi, Aditya P. Mathur, and Brian Pattison. P^{M}othra: Scheduling Mutants For Execution on a Hypercube. *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, pages 58–65, December 1989.

[15] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[16] Software Engineering Research Center. *The Mothra Software Testing Environment*, Report SERC-TR-4-P edition. Purdue University, 1987.

[17] D. Richardson and M. Thomas. The RELAY Model of Error Detection and its Application. *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, July 1988. Banff, Canada.

[18] J. Voas. *A Dynamic Failure Model for Performing Propagation and Infection Analysis on Computer Programs.* PhD thesis, College of William and Mary in Virginia, March 1990.

[19] J. Voas. A Dynamic Failure Model for Estimating the Impact that a Program Location has on the Program. In *Lecture Notes in Computer Science: Proc. of the 3rd European Software Engineering Conf.*, volume 550, pages 308–331, Milan, Italy, October 1991. Springer-Verlag.

[20] J. Voas. A Testing Metric On The Ability Of A Program To Hide Faults During Random Black Box Testing. In *Proc. of the 3rd Oregon Workshop on Software Metrics*, Silver Falls, OR, March 1991. Oregon Center for Advanced Technology Education.

[21] J. Voas. Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, pages 235–247, Portland, OR, October 1991. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.

[22] J. Voas. Sensitivity Analysis. In *Proc. of the 8th Int. Conf. on Testing Computer Software*, pages 165–174, Washington, D.C., June 1991. International Test and Evaluation Association.

[23] Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.