Using Program Slicing to Estimate Software Robustness

Keith Gallagher[*]
Computer Science Department
Loyola College
Baltimore, MD 21228

Neale Fulton
Mathematical and Information Sciences
CSIRO
Canberra, Australia

Abstract

In most instances, the quantification of software reliability by empirical test is infeasible. Therefore, alternative measures and methods have been sought to gain assurance of the integrity of software operation. Statistically inferred measures of robustness, the extent to which a system can continue to operate correctly in the presence of faults, have been used for this purpose. Two problems arise in this approach: determination the location of "high-impact" points (ones where failure ripples widely); and exact nature of the error that could have a high impact. Program slicing provides a rigorous, automated method by which the dependency of a nominated variable on all other source statements in the program can be determined. This paper presents a method for automatically determining a "unique" fault injection point for any given variable of interest; that is, a point where a variable's value has the greatest impact on system output. This point is located using program slicing. Moreover, the location of the point assists an analyst in determining the kind of error that should be introduced. A rigorous assessment of the acceptability of output data subject to the susceptibility of failure of software due to corrupted or malicious incoming data is possible using this approach, and we provide such a measure. The benefits of this approach are the improved repeatability of the process, the rigorous mathematical basis, and greater objectivity in selection of fault injection points for a given test. These benefits provide a greater insight into the nature of failure within the software, and improved capacity to assess failure modes and robustness of software.

1 Introduction

A system is comprised of components integrated in a specified manner to fulfill a designated function or need. The analysis of a system may require many abstractions to fully describe its functionality and behavior. There is an obvious need for abstracting simple, yet rigorous system models. Specifically, system designers need to understand the likelihood of successful operation for any given use of the system and the limits which these abstractions and analyses place on conclusions.

A system may be considered as a network when considering how it may fail. In Reliability Engineering, a sub-discipline of Systems Engineering, success and failure are related through a method of analysis called the Reliability Block Diagram (RBD)[3, 11, 13]. The RBD is derived from an understanding of how the physical system is to work. The relationships among all the possible paths through the system, by which it may succeed, are considered. This diagram is then used to predict the overall reliability of a system, given the knowledge of the individual reliabilities of their sub-systems.

The RBD abstracts system complexity highlighting success and failure. RBD's vary markedly in complexity. For simple systems, the RBD may simply be a series of sequentially connected components, all of which are required for system success (e.g. links in a chain). More complex systems are represented as parallel combinations of components in which only one operating component out of several is required for success (e.g. one back-up power supply operating when the primary power supply has failed). In practice the RBD of a complex system will be represented by combinations of both series and parallel constructs. The RBD provides a logical model which can be interpreted as a series of success paths and subsystem failure combinations within the system.

Figure 1 shows a small RBD augmented with special start and end points, which are not part of the RBD. The nodes that `start` points to are the entry points and the nodes that point to `end` are the exit points. Note that node 2 is a possible exit.

A minimal tie-set is a group of edges through the

[*]Supported in part by CSIRO while on sabbatical leave.

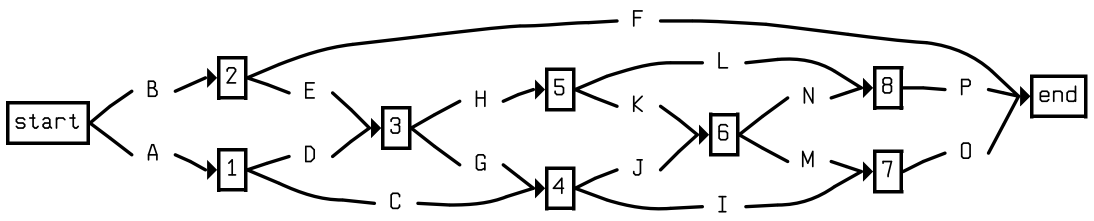

Figure 1: A small reliability block diagram

RBD which form a connection between the input and output when traversed in the direction of flow of the paths within the RBD and with no node encountered more than once. The minimal tie-set elaborates all possible paths which can produce a successful result for the system mission or nominated operational profile. In figure 1, {A,D,H,L,P} is a tie-set. Tie-sets can also be enumerated by the nodes that are traversed on the path; in the example the nodes would be {1,3,5,8}.

Conversely, a minimal cut-set is the set of system components such that if all components of the cut-set fail then the system will fail. However, if a member of the cut-set has not failed then the system can still operate and achieve mission success. That is, all components of the minimal cut-set must fail for the system to fail. In figure 1, {C,D,E,F} is a minimum cut-set. Cut-sets may also be described as sets of nodes; in this instance, {1,2} forms a cut-set of nodes.

We are interested in deriving an analogous representation of software operation which may be used to analyze successful operation and to identify specific failure mechanisms of the code. Program slicing provides a mathematically robust method of decomposing software into a graph whose edges represent success-paths and whose nodes represent software which may fail under given conditions. The assignment of reliability for each component, as is done for hardware systems, may be more appropriately computed for software by an empirical measure of robustness as reported by Voas, et al. [15]. This new approach provides a meaningful measure of performance and assuages some of the difficulties of measuring software reliability [4].

The relationship between program slices can be related to RBDs of hardware. This method permits an estimate of the lower bound of robustness of the software of a system to be established. The graph of certain slices can be used to build a logical representation of the software similar to an RBD. By considering single paths through the diagram / graph we obtain the collection of possible system executions. Examining the robustness of the software components in series provides estimates of the lower bound of the failure tolerance for a given operational profile. If components are in parallel then an upper bound may be inferred.

Information represented in this graph of slices also represents a logical sequence of relationships which must all be satisfied if the output is to be free from failure. Assigning reliability measures to software is not yet always meaningful; however, assigning failure tolerance measures can be. Thus the program slices represent a logical series construct from which we can make valid inferences on the estimated bound of robustness, i.e., failure tolerance, of the subject software.

In summary, methods exist for evaluating success and failure of systems at high levels of abstraction; similar methods have not yet been fully developed for software. In this paper we define a method for software analogous to the success-path / fail-set network for hardware.

2 Background

2.1 Terminology

We introduce our terminology[10], so that the reader is not confused by our usage. We believe that our usage is consistent with the framework proposed by Prasad, et al.[12].

Fail / Failure A system failure occurs when the delivered service deviates from the intended service. The user perceives that the software has failed to deliver an expected service. *Severity levels* may be associated with faults, depending on the impact to expected service.

Error is a discrepancy between the computed state value and the correct value. An error is the part of the system state which is liable to lead to a failure. Errors may be *latent* or *detected.*

Fault / "Bug" is the *cause* of

1. a failure
2. an internal error (discovered without a failure)

Defect A generic term for fault or failure.

Mistake A human action resulting in a fault. This term is used to avoid confusion with *error.*

Robust / Robustness the extent to which a system can continue to operate correctly in the presence of faults.

Tolerance The ability of a system to provide operations in the presence of unusual or abnormal conditions.

"...a software *failure* is an incorrect result with respect to the specification or an unexpected behavior perceived by the user at the boundary of the software system, while a software *fault* is the identified or hypothesized cause of the software failure."[10] *Robustness* characterizes the behavior of the system in the presence of failure.

Some authors use the term "tolerance" to note the *techniques* or *means* by which systems are constructed so as to continue operating in the presence of faults. Note that our usage is different from this; we are discussing an attribute of a system, not the technique by which it was constructed.

2.2 Software Robustness and Failure Tolerance

Voas, et al. [15] introduce a measure of a software statement/location internal error tolerance through a method called Extended Propagation Analysis (EPA). The EPA method seeds errors by injecting them into instrumented source code to corrupt the program's internal state; determines the effects of the propagation of the error; and then assesses the acceptability of the output data values. EPA is based on the following definition. Software is robust (i.e. failure tolerant in the terminology of Voas) if and only if:

1. the program can compute an acceptable result even if the program itself suffers from incorrect logic, and
2. the program, whether correct or incorrect, can compute an acceptable result, even if the program itself receives corrupted or malicious incoming data during execution. [15]

EPA generates a data-state error, known as an infection, which may result in more than one infected variable. Although largely automated, EPA requires some manual set-up which requires advising an analyst where in the software to perform fault injection, and what constitutes an unacceptable output. An acceptable result is simply an output result within the range of specified acceptable values for a given input.

The result of EPA analysis is that each program statement is assigned a measure of robustness: a number between 0 and 1. The number 1 indicates code which is highly tolerant of aberrant data and the number 0 indicates code which is intolerant of aberrant data. This is a measure of the attribute of tolerance.

2.2.1 EPA Model

A program may be completely specified by a triple: a finite set of states; a finite set of inputs; and a set of mappings determining the possible transitions from each and every state to each and every other state. This model is general although accounting for all the states in the program may quickly become a practical impossibility as the set of program states increases in number. Never-the-less for a practically small set of data inputs and program states this model is tractable. Software tools for program slicing can accommodate program sizes of order 100,000 lines of code, whilst robustness measures may be required on critical software of order 10 to 1000 lines of code. This model is tractable for these latter practically small sets of data inputs and program states.

The EPA analysis model assumes a program P which has a set A_P of all possible program states. We refine the set A_P by considering only those states A_{Pn} which result from the execution of the program P at statement n. This set of states is uniquely determined by the possible initial states of the program immediately before execution of statement n; the data input to the program for the execution of statement n (that is the data that statement n acts upon); and on the logic and algorithms embedded in statement n code. Further refinement leads to only identifying those states A_{nPx} which can be reached by execution of statement n and which were created by an input x. The program can only be in one state at a time so if the nature of the program is a loop, for example, then we identify the unique state of the program, due to an input x, at the i^{th} iteration through n as A_{nPix}.

Thus for a specific statement n of a program P, when subjected to a specific data input x the resultant set of program states over I_{xn} iterations of statement n is given by:

$$A_{nPx} = \{A_{nPix} \mid 0 \leq i \leq I_{xn}\}$$

Since the resulting state is dependent on the input data x, and since that data can take on a range of values the EPA model denotes the set of all possible data values as Δ. A single value of the input data is x, such that $x \in \Delta$. The manner in which data is used over time, is called an operational profile. This unique use of the input data identifies a certain probability distribution denoted Q.

The requirements on a program will typically be expressed in terms of both the functions the program is to implement and, in the case of safety critical systems, functions which a program must not implement. The purpose of a program is then to generate an output either in terms of a state-variable change, or an output event. From the knowledge of the requirements and the nature of the variable under question, a predicate, s, can be formed to define the acceptable ranges of data for the output. In the case of a program state, a given range of data with respect to an ideal value may be acceptable. However, in the case of an output variable only one value may be acceptable. When the programs are regarded as partial functions over the input domain, the output(s) will be uniquely determined by the function that the program computes. If such a function is known, it may be used for an oracle.

Definition 1 (Failure Tolerance of v at n) $\Psi(v, n, s, P, \Delta, Q)$
Let s be a predicate describing acceptable states. Then the failure tolerance *of v at statement n, denoted $\Psi(v, n, s, P, \Delta, Q)$, is the percentage of the successes of s with respect to variable v at statement n of program P with inputs selected from Δ using distribution Q.*

Thus, high values for $\Psi(v, n, s, P, \Delta, Q)$ indicate highly tolerant computations and low values indicate intolerant computations. This notation is slightly different than that used by Voas, et al. [15]. We have added two parameters: the predicate s and Δ, the input set. We have also moved the parameters into a list, rather than denoting them with subscripts.

```
1 main()
2 {
3   scanf ("%d", &i);
4  }
```

Figure 2: A simple program, with large differences in EPA values

The $\Psi(v, n, s, P, \Delta, Q)$ values are dependent both on a value and a position (statement) in the source. For instance, when s is a predicate that evaluates whether the value of `i` is the same as the value that was actually input, the program of figure 2 has high $\Psi(i, k, s, P, \Delta, Q)$ for `i` for $k \leq 3$, (before statement 3), but low values for $k > 3$ (after statement 3). We would like provide a systematic method for finding such locations.

We propose to refine the EPA estimation process by locating the the "most likely" point in the the program at which the perturbation will have the largest impact and thereby automate this process of the analyst. We use program slicing to find such points.

2.3 Program Slicing

Program slicing provides an automated method which analyzes the dependency of a nominated variable on all other source statements in the program. Program slicing provides a method to rigorously identify all possible sources of infection of a given variable.

A program slice, $\mathbf{SBS}_P(v, n)$, of program P on variable v, or set of variables, at statement n yields the portions of the program that contributed to the value of v just before statement n is executed [16]. The pair (v, n) is called a *slicing criteria.* Another way to think of this to answer the question "What statements may have contributed to the value of the variable v at the statement n?"

This formulation is now called *static backward* slices, hence the **SB** in the definition of the function. *Static* because they are computed as the solution to a static analysis problem (*i.e.*, without considering the program's input). And *backwards* because they computed from the statement in question back to the beginning of the program.

A dynamic backward slice, $\mathbf{DBS}_{P_x}(v, n)$, of program P with input x on variable v at statement n is the portions of the program that contribute to the value of v at statement n with input x. A dynamic slice answers the question "What statements actually contributed to the value of a variable v at statement n for a particular input, x?"

Forward slices may also be computed. $\mathbf{SFS}_P(v,n)$ is the static formulation; $\mathbf{DFS}_{P_x}(v,n)$ is the dynamic. A forward slice answers the question "What statements are affected by the value of v at statement n?" The propagation of the data state error may be traced, by application of forward program slicing, to all potentially infected variables.

We have discussed backward/forward, static/dynamic dimensions of a "slicing space;" there is a third dimension, executable/non-executable, that determines whether or not the output of a slicer is a compilable source. For the purposes of our discussion, we will require that that the program slices be executable.

The backward/forward, static/dynamic and executable/non-executable attributes of program slices defines a framework for analysis of slicing methods. The static/dynamic character of program slices may be parameterized [6] to obtain constrained slices. The parameter/constraint refers to the amount of input available. A fully constrained slice is a dynamic slice; a completely unconstrained slice is a static slice. Recent surveys of program slicing may be found in [2, 14]. The interested reader is referred to those.

In all of our definitions, we include the statement at which the slice is taken in the slice.

2.3.1 Decomposition Slicing

A decomposition slice [8] does not depend on statement numbers. It is the union of a collection of slices, which is still a program slice [16]. A decomposition slice captures all relevant computations involving a given variable and is defined as follows:

Definition 2 (Decomposition Slice) $\mathcal{DS}(v,p)$
Let

1. *Out(p,v) be the set of statements in program p that output variable v,*
2. last[1] *be the last statement of p,*
3. *N = Out(p,v)* $\cup$ {last}.

The statements in $\mathcal{DS}(v,p) = \bigcup_{n\in N} \mathbf{SBS}_p(v,n)$ *form the* decomposition slice on v.

We take the decomposition slice for each variable in the program and form a graph,[2] using the partial ordering induced by proper subset inclusion. Larger sets will be toward the top; smaller sets will be lower; edges point downward. An edge between A and B means $B \subset A$ and there is no C, such that $B \subset C \subset A$.

A decomposition slice is *maximal* if it is not contained in any other decomposition slice. We abuse the notation and identify the decomposition slice by its variable name. It is usually, but not always the case, that the maximal variables are the outputs of the program.

The graph of decomposition slices was originally intended to give software maintainers a method for visual impact analysis [7, 9]. The decomposition slice graph provides information to a software engineer about dependencies that exist between variables in a system. The graph shows what slices for variables are included in the slices for other variables in a system. This information is useful to a software engineer in trying to gain an understanding of a system as it can be used to identify the data that impacts on a particular variable (those variables on which it depends) and the impact of a variable (those variables which depend on it).

The graph can be used to assist the analyst in determining the fault injection point. Each node in the graph represents the largest possible slice on the given variable. We would like to insert the fault immediately "after" the code represented in the nodes of the graph. In effect, we injecting the fault on all the edges of the graph that point to a given node. Another way to think of this is that the computation represented by the node in the the graph is replaced by a random, perturbed, value.

3 Determining the Fault Injection Point

Clearly, there is more than one point in a program that will generate this decomposition slice. However, we want to be able to speak about *the* fault injection point.

Definition 3 (Fault Injection Point) $\mathcal{FIP}(v)$
Let $\mathcal{DS}(v,p)$ *be the decomposition slice for v. The* fault injection point *for v is the last definition of v in its decomposition slice.*

Now we can talk about THE fault injection point of a variable. We use the "uniqueness" of this point define a new kind of slicing.

Definition 4 (Software Robustness Slice) $\mathcal{SOROS}(v,p)$
Let $\mathbf{SBS}_P(v,\mathcal{FIP}(v))$ *be the static backward slice at*

[1]This assumes single-exit functions, but can be extended to multiple-exit functions.

[2]The term "lattice" was used in [8].

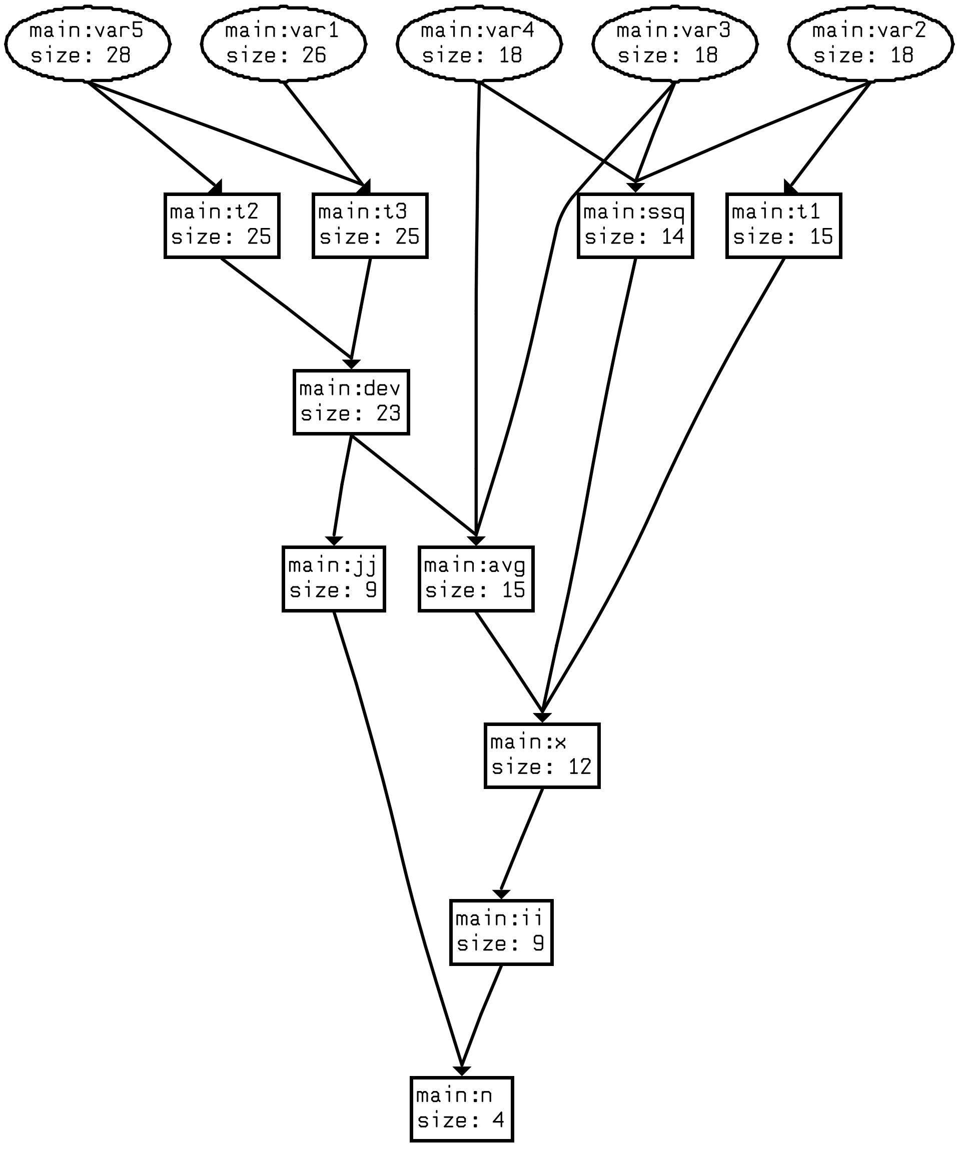

Figure 3: The decomposition slice graph of a program. The source for this program is shown in figure 4.

on v, $\mathcal{FIP}(v)$ and $\mathbf{SFS}_P(v, \mathcal{FIP}(v))$, be the forward static slice, from $\mathcal{FIP}(v)$ on v. The union of these 2 slices is $\mathcal{SOROS}(v,p)$, the static software robustness slice on v.

There are static and dynamic formulations of $\mathcal{SOROS}(v,p)$. The dynamic version will, of course, attend to input x and permit an abuse of notation in the definition: $\mathcal{FIP}(v) == i$ is taken to mean the i^{th} iteration of statement $\mathcal{FIP}(v)$. The idea behind these formulations is to slice backward from $\mathcal{FIP}(v)$ and forward from it.

3.1 The Connection

There is a natural association between $\mathcal{FIP}(v)$, $\mathcal{SOSRS}(v)$ and the location for an EPA evaluation. The points of interest for EPA evaluations are reasonable candidates for slicing statements and vice-versa. The engineer selects the variables of interest in both cases. In fact, from the robustness point of view, program slices were invented to answer the question "why isn't our program robust (*i.e.*, working correctly) at this location?"

A decomposition slice is generated for a specified program variable. Since each program slice subgraph is associated with a unique variable, and contains all statements relating to that variable, then EPA failure tolerance results can be associated with each program statement within the sub-graph. This association together with the graphical presentation of the program slice provides a powerful, automated analysis tool for identifying potential failure modes of the software under examination.

3.2 Robustness and Decomposition Slice Graphs

The graph of slices shows set inclusion i.e., computational dependence; the RBD shows subsystem dependence. To construct a robustness indication from the slice graph we need to map the idea of a decomposition slice into system component. We do this by noting that each decomposition slice is a computation – a complete compilable program.

Some of these slices may not have outputs, per se, but they do define a state transition, and in this sense, can be regarded as having "outputs." Each above and adjacent node in the slice graph determines an "extension" by which the computation is extended.

For instance, in figure 3, `main:x` represents the computation that fills and array `x` with values. The node `main:ssq` represents the *extension* of the computation of `x` by one that also computes the sum of the squares of the components of `x`. Similarly, `main:avg` extends `x`, but in a different manner than `main:ssq`.

A node/computation/decomposition-slice may have more than one extension. To remedy this, we will place 2 parameters in the robustness indicator: *from* and *to*.

Definition 5 (Robustness of a Component) $\mathcal{R}(v,w)$
Let

1. $\mathcal{DS}(v,p) \subset \mathcal{DS}(w,p)$
2. *$s(w)$ be a predicate describing the acceptable states of w.*
3. *$\mathcal{SOROS}(v)$ be the software robustness slice on v,*

Then
$\mathcal{R}(v,w) = 1 - \Psi(v, \mathcal{FIP}(v), \mathrm{s(w)}, \mathcal{SOROS}(v), \Delta, Q)$
is the robustness of component v, with respect to component w.

As $\mathcal{FIP}(v)$ is obtainable from v, we have a location independent, single parameter version of robustness. By taking the union of w such that $\mathcal{DS}(v,p) \subset \mathcal{DS}(w,p)$, we can speak of the robustness of variable v, $\mathcal{R}(v)$. Thus, the maximal slices taken in parallel, give an upper bound on system tolerance.Lower bounds may be inferred from RBD's by considering the components (nodes) in series.

Thus, each node in the decomposition slice graph can be regarded as a "component" that must successfully perform if the entire system is to succeed. In this interpretation, tie-sets mean that a computation has been "successfully" extended to its (next) additional functionality. Cut-sets may be interpreted as failure of the extension.

3.3 Analyzing the Components

For instance, a calculation that sums the elements of an array could be one "component." The computation of the average would be an extension to (the state produced by) this computation, and would be considered a system component. A failure in the computation "average" while the computation "total" was correct, would indicate that all decomposition slices that contain "average" would be *suspect.* The reason for suspicion and not assurance is that subsequent components may mask, compensate, or tolerate the error. Moreover, we are attempting to obtain a lower *bound* on the tolerance of the system, not an exact measure; the system may still have reliable output in some instances.

```
#include <stdio.h>
#define MAX 1024
main()
{
    float x[MAX];
    float var1, var2, var3, var4, var5 ;
    float ssq, avg, dev;
    float t1, t2, t3;
    int  ii, jj , n;
       t1 = 0.0 ;
       t2 = 0.0 ;
       t3 = 0.0 ;
       ssq = 0.0 ;
       scanf ("%d", &n);
       for ( ii = 0 ; ii < n ; ii = ii + 1)
       {
         scanf ("%f", &x[ii]);
         t1 = t1 + x[ii];
         ssq = ssq + x[ii] * x[ii];
        }
       avg = t1 / n;
       var3 = (ssq  - n * avg * avg) / (n - 1);
       var4 = (ssq  - t1 * avg) / (n - 1);
       t1 = t1 * t1 / n;
       var2 = (ssq  -  t1 ) / (n - 1);
       for ( jj = 0 ; jj < n ; jj = jj + 1)
       {
         dev = x[jj] - avg ;
         t2 = t2 + dev ;
         t3 = t3 + dev * dev ;
       }
       var1 = t3 / (n - 1);
       var5 = (t3 - t2 * t2 / n ) / (n - 1);

      printf("variance 1 (two pass): %f \n",var1);
      printf("variance 2 (one pass, using square of sum): %f \n",var2);
      printf("variance 3 (one pass, using average): %f \n",var3);
      printf("variance 4 (one pass, using average, sum): %f \n",var4);
      printf("variance 5 (two pass, corrected): %f \n",var5);
}
```

Figure 4: The variance program

4 Example

We use a very simple example to illustrate. The program of figure 4 computes the variance of a set of values using 5 different techniques. The first value is the number of values to be read, the subsequent input are the values whose variance is to be computed.

We propose to demonstrate how to use the graph on the variable avg, the average. First, we obtain $\mathcal{FIP}(avg)$, which is statement 21. (Recall that we include this statement in the slice.) The program passes through the injection point only once.

We then perturb avg. In the static formulation, we compute the forward slice $\mathbf{SFS}_P(avg, 21)$. This has statements 22, 23, 26-33. These statements are represented by the shaded areas in Figure 5.

We can observe from Figure 5 that var2 is not impacted by the computation of avg; *i.e.*, $R(avg, var2) = 1$. This fact makes it a suitable candidate for an output oracle (in this instance). Outputs var1, var2, var4 and var5 are impacted by the computation of avg, so we compute $R(avg, var1)$, $R(avg, var3)$, $R(avg, var4)$ and $R(avg, var5)$. (Other sensible values are $R(avg, dev)$, $R(avg, t2)$ and $R(avg, t3)$. We omit these from the discussion.)

To simulate the infection, we assign a random value to avg. For this infection we determined that var1, (two pass), is always wrong; var3, (one pass, using average), is always wrong; and var4 (one pass, using average, sum), is always wrong; *i.e.*, $R(avg, var1) = R(avg, var3) = R(avg, var4) = 0$. However, for var5 (two pass, corrected), $R(avg, var5) = 0.043$ *i.e.*, var5 was still correctly computed for 4.3% values of avg that were randomly perturbed[3].

5 Conclusion

In this paper we have shown how program slicing can be used to further automate the determination of robustness measures for software operation. One of the keys to determining robustness is the selection of a fault injection point which is used to test the software with respect to aberrant input data behavior. Program slicing can identify which variables within a program may influence the value of a specified variable, or alternatively which variables will be subsequently influenced by the value of a specified variable. The mathematical rigor associated with program slicing allows a unique fault injection point to be identified. Automation of program slicing techniques means that higher levels of process assurance can be achieved than with processes which use manual heuristic means and guesswork to identify injection points. Program slicing enables the scope of influence of a selected variable to be consistently and completely determined. This is significant in the maintenance of software where changes are considered and the impact of change needs to be assessed; or where existing software is exhibiting aberrant behavior and the scope of influence needs to be determined. Thus the ability to automate the slice process means that the determination of scope of influence can be determined repeatedly and consistently.

Robustness of software performance is crucial to safety critical and other high integrity systems. The method proposed by this paper enables a higher level of assurance to be claimed for those critical elements of code, and enhances productivity of test in establishing the practical limits of test for those elements. In practical terms the completeness of robustness assessment may limited if the slicing techniques presented in this paper are not available to the developer or maintainer in versions of program slicers available on the market. Secondly it is still the case that each variable within the scope of influence of a specified variable, will require manual inspection of the computed robustness measure. Whether the robustness measure is satisfactory for any one variable of this set can only be determined by a criticality assessment of the specific functionality under consideration.

For high integrity systems software robustness measures based on program slicing provide an essential verification tool set required to ensure design assurance.

[3] It was ultimately uncovered as a floating point truncation error in the evaluation of the sub-expression t3 - t2 * t2 of line 33.

References

[1] M. Aboelfoh and C. Colburn. Series parallel bounds for the two-terminal reliability problem. *ORSA Journal of Computing*, 1(4), 1989.

[2] D. Binkley and K. Gallagher. A survey of program slicing. In M. Zelkowitz, editor, *Advances in Computers*. Academic Press, 1996.

[3] Benjamin S. Blanchard. *System Engineering Management*. John Wiley & Sons Inc., 1991.

[4] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical systems.

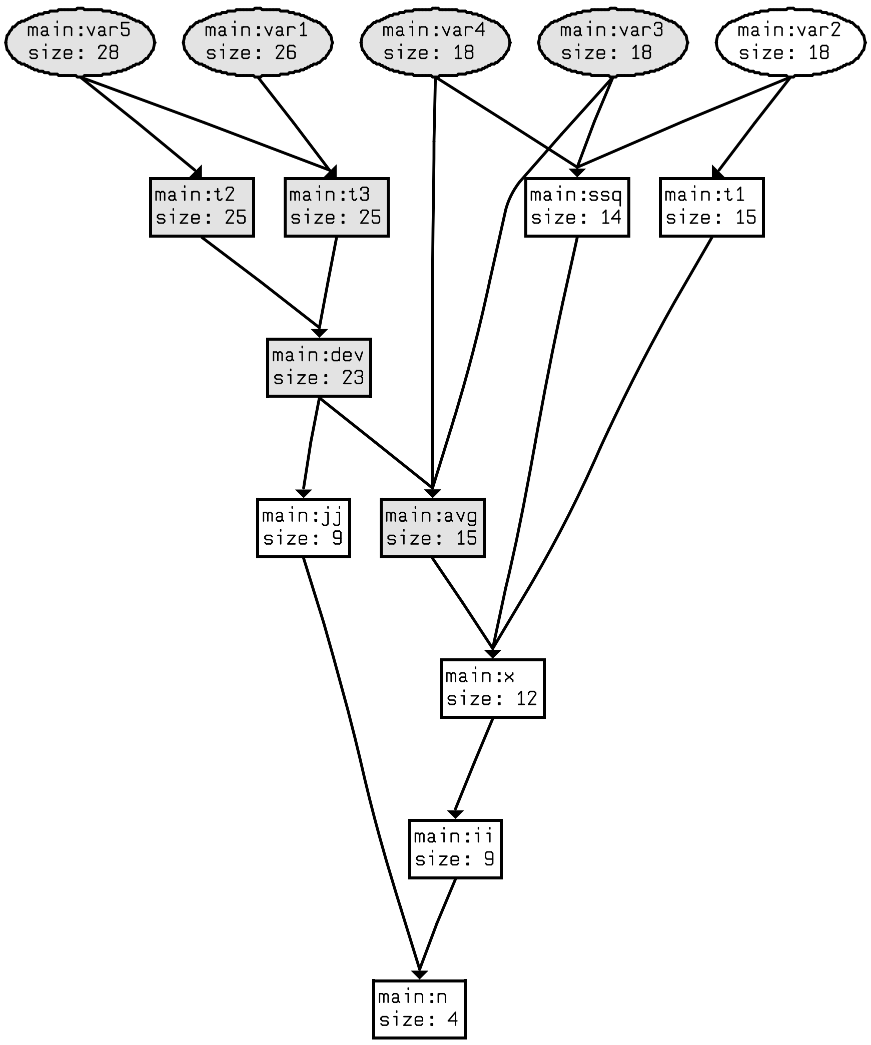

Figure 5: Impact of `avg`.

In *Proceedings of the ACM SIGSOFT Conference on Software for Critical Systems*, volume 16. ACM, December 1991. Software Engineering Notes.

[5] C. Colburn and L. Neel. Using and abusing bounds for network reliability. *IEEE global Telecommunications Conference*, December 1990.

[6] J. Field, G . Ramalingam, and F. Tip. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 379–392, 1995.

[7] K. B. Gallagher. Visual impact analysis. In *Proceedings of the Conference on Software Maintenance - 1996*, 1996.

[8] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[9] K. B. Gallagher and L. O'Brien. Reducing visualization complexity using decomposition slices. In *Proceedings of the 1997 Software Visualization Workshop, SoftVis97*, number ISBN 0725806303, Dec 1997.

[10] M. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[11] M. Moderres. *What Every Engineer Should Know About Reliability and Risk Analysis.* Morcel Dekker, Inc, 1993.

[12] D. Prasad, J. McDermidt, and I. Wand. Dependability terminology: Similarities and differences. In *AES Systems*. IEEE, January 1996.

[13] R. Ramakumar. *Engineering Reliability, Fundamentals and applications.* Prentice Hall International, 1993.

[14] Frank Tip. A survey of programming slicing techniques. *Journal Of Programming Languages*, 13(3):121–189, 1995.

[15] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" software can behave. *IEEE Software*, August 1997.

[16] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.