Software Safety and Program Slicing

Keith Brian Gallagher
Loyola College in Maryland &
Computer Systems Laboratory
National Institute of Science and Technology
kb@loyola.edu

James R. Lyle
Computer Systems Laboratory
National Institute of Science and Technology
jimmy@swe.ncsl.nist.gov

Abstract

Software quality assurance auditors are faced with a myriad of difficulties, ranging from inadequate time to inadequate computer-aided software engineering (CASE) tools. One particular problem is the localization of safety critical code that may be interleaved throughout the entire system. Moreover, once this code is located, its effects throughout the system are difficult to ascertain. We present a method that uses program slicing to mitigate these difficulties in two ways. First, we show how program slicing can be used to locate all code that contributes to the value of variables that might be part of a safety critical component. Second, we show how slicing-based techniques can be used to validate functional diversity, i.e., that there are no interactions of one critical component with another critical component and that there are no interactions of non critical components with the safety critical components. We have begun to prototype this method as part of a research project at National Institute of Science and Technology.

KEYWORDS: Software Safety, Program Slicing, Software Quality Assurance, Functional Diversity, Validation and Verification

1 Introduction

Most systems that can possibly be hazardous to life, property or well-being have a safety system as a sub-component. Safety systems are used to detect and/or prevent unsafe operating conditions. Safety systems are also used to mitigate the consequences of accidents. Safety systems can be as complex as avionics control on advanced aircraft or as simple as a home smoke detector; avionics safety systems should both detect and prevent unsafe operating conditions while simple smoke detectors only detect hazards.

Many of the systems currently in use guard against hazards with safety systems using methods that do not rely on digital hardware and accompanying software. A large number of these safety systems are being replaced with new technology that uses programmed digital components. These components are proposed for both new construction and preventive maintenance on deteriorating, older systems. Modern digital systems have many features that suggest their use in safety systems: self-diagnostic aids, on-line testing, high accuracy, drift-free operation, signal multiplexing, and the use of fiber optics. While the hardware reliability and quality have increased, the use of software increases the risk of errors and failures. This is due to the complexity of software and the difficulty of validating it against any possible error. The nuclear power, avionics and medical device control industries face a common set of problems in development and assurance of high integrity software

Software quality assurance (SQA) auditors are faced with assessing the quality and integrity of software in the general system and the safety system. Guidance developed for software quality assurance provides some checklists that may be used during audits. However, standards and guidelines on the best practices for producing high integrity software are not widely available; the contents of such standards are not yet in agreement. The investigation of software safety is itself a new field; the discipline has not matured sufficiently to offer profound advice.

In this paper, we show a novel application of program slicing to two issues of software safety: functional diversity, and validation and verification of safety critical components. We start with the work of Leveson in software fault tree analysis and build on our own labors in program slicing to give SQA auditors more methods to assuage the inherent difficulties of their task. We have begun to prototype these methods as a research project at National Institute of Science and Technology (NIST).
2 The Problem

A comprehensive classification of error types would be a potent weapon in the auditor's arsenal; the auditor could simply validate against the list. Such a classification could also guide new methods of development so that a class of errors is not even injected into the product. The construction of this classification requires empirical error data from finished projects. To collect this data, software development organizations need a database that contains information on the cause and cure of individual errors. However, most software development organizations have insufficient capability to collect this data, let alone use it. Moreover, an investigation of software error analysis has shown that, in general, industry-wide error data (for any industry) is not sufficient to assist developers or to alert auditors to the types of problems that may exist in the safety system under audit[8]. Thus we are reduced to collections of ad-hoc methods for detection and assurance.

Auditors are faced with hard choices of how to use tools to assess software in safety systems. The functional capability provided by CASE tools is generic, rather than specific. Thus, the SQA auditor would need the same tool and all the products of the developer, and need to do additional work, to apply any tool's capability to the quality analysis. We propose in this work to provide a software tool and method that is specifically designed and constructed with the task of the auditor as the driving force.

A design error in hardware or software, or an implementation error in software may result in a Common Mode Failure of redundant equipment. A common mode failure is a failure as a result of a common cause, such as the failure of a system caused by the incorrect computation of an algorithm. For example, suppose that \textit{X} and \textit{Y} are distinct critical outputs and that \textit{X} measures a rate of increase while \textit{Y} measures a rate of decrease. If the computation of both of the rates depends on a call to a common numerical differentiator, then a failure in the differentiator can cause a common mode failure of both measures.

One technique to defending against common mode failures uses functional diversity. Functional diversity in design is a method of addressing the common mode failure problem in software that uses multiple algorithms on independent inputs. Functional diversity allows the same function to be executed along two or more independent paths. For example, a function may be computed manually and with automation. Alternatively, versions of the same program may be written by different teams [3]. Within a software system, the same function may be programmed in more than one way, but eventually a voting mechanism within the software decides what output to accept. The value of functional diversity for software is debated in the technical literature and some experiments have been conducted to find a valid approach[3]. For software, functional diversity may be used to ensure that no two critical functions use the same paths from input to output. Thus, an error in one critical function can never impact the output of another critical function.

3 Fault Tree Analysis

3.1 System Fault Tree Analysis

Once the system hazards have been identified, the objective is to mitigate the risk that they will occur. One approach to achieving this objective is to use system fault tree analysis. Under the assumption that there are relatively few unacceptable system states and that each of these hazards has been determined, the analysis procedure is as follows. The auditor assumes that a hazard has occurred and constructs a tree with the hazardous condition as the root. The next level of the tree is an enumeration of all the necessary preconditions for the hazard to occur. These conditions are combined with logical and or as appropriate. Then each new node is expanded "similarly until all leaves have calculable probability or cannot be expanded for some reason"[5].

The system fault tree analysis gives the auditor the sub-components of the system that must be carefully examined. Part of this examination is the validation that there are no interactions with non-critical functions. The determination of the specific components that will be examined is up to the auditor. This information should be obtainable from the design documentation. The task of locating the components is outside the scope of this paper.

3.2 Software Fault Tree Analysis

The results of system hazard analysis must be examined for their impact on software. The level of detail available at the software requirements or software design phases may not be sufficient to fully understand potential hazards and some critical information may be overlooked in the development of the design and code. Once code is available, the equivalent of system fault tree analysis can be applied to the software.

"Software fault-tree analysis works backward from the critical control faults ... through the program code or the design to the software inputs. In other words, it starts from the hazardous outputs (or lack of them) and traces backward to find paths through the code from
particular inputs to these outputs or to demonstrate that such paths do not exist"[5]. This is the particular feature of the auditor's job that is suited to program slicing.

4 Program Slicing

Program slicing is a family of program decomposition techniques based on extracting statements relevant to a computation in a program. Program slicing as originally defined by Weiser [12] produced a smaller program that would reproduce a subset of the original program's behavior. This is advantageous since the code for a given calculation may be scattered throughout a program. Because the slice is smaller than the original program and the slice collects an algorithm without intervening irrelevant statements, it should be easier for anyone (developer, maintainer or auditor) interested in a subset of the program's behavior to understand the slice that produces the behavior of interest rather than to deal with the entire program. This idea has applications in program debugging, program testing, parallel program execution and software maintenance[1, 2, 6, 7]. The utility and power of program slicing comes from the potential automation of tedious and error prone tasks.

Weiser's original definition of a program slice is now called static program slicing, since the slice is found by static data flow analysis on a graphical representation of the program.

Definition:

A slicing criterion for a program slice is a tuple \(<i,v>\) where \(i\) is a statement in the program and \(v\) is a subset of the program variables.

A slice of a program is computed on \(v\), at statement \(i\).

Definition:

A (static) program slice of a program \(P\) for a given slicing criterion, \(<i,v>\), is an executable program obtained by deleting zero or more statements from \(P\) such that the values of the variables in \(v\) are the same when execution reaches statement \(i\) for both \(P\) and the slice on \(P\).

We may view a program slice as a trace of the states of the computation through which the program passes. (The state of the computation is the set of defined variables and their respective values.) With this trace in hand, all program inputs have been captured for the given slicing criterion. Thus, all the program inputs can be (conceptually) replaced by assignment statements, with the input variable acting as the left hand side of the assignment and the value of the input variable acting as the right hand side of the assignment. As statements are eliminated from the original program to construct the slice, if an input (i.e., conceptual assignment) is eliminated, the corresponding input value is also deleted from the input sequence. So, to get the same value in a given variable at a given point, we may need to slice the input, too.

Dynamic program slicing was introduced by Korel as a method to significantly reduce the size of static slices[4]. Dynamic slices are useful in debugging and testing. Dynamic slices are computed from a program trajectory for some particular, fixed input. Several programming language features that are handled in a general way by static slicing can be handled effectively by dynamic slicing. These features include arrays, structures and dynamic objects (pointers), which can only be handled in a general way by static slicing so that many apparently irrelevant statements are included in a static slice. The extra information provided by knowledge of the inputs permits static slices to be pared down to only those statements were executed the individual data objects that were that were used. For example, a static slice that references an array component would then require that all references to any component be included in the slice. The dynamic slice permits the individual array component to be treated as an independent data object; thus only references to the particular component need to be included in the slice. Which components are accessed is a function of the particular input. A static slice is independent of any input.

A decomposition slice captures all the computation on a given variable without regard to any specific program location by taking the union of all static slices on a given variable at all output statements[2]. Decomposition slices are useful for software maintainers, as appropriate combinations of decomposition slices can indicate to the maintainer which statements and variables can be modified without impacting the components that are to remain unchanged.

Slices exhibit a number of useful properties that can be exploited by SQA auditors:

1. Program slices fit in with the way programmers understand programs since, after trying to understand an unfamiliar program, programmers recognize slices from the program better than other chunks of code from a program[11].

2. The union of two slices is a slice with a slicing criterion that is the union of the respective slicing
criterions. This property is exploited in the definition of a decomposition slice.

3. A program can be viewed as the union of decomposition slices of each output variable.

4. Any statement not in the union of all decomposition slices must be useless code in the sense that it does not contribute to the calculation of any program output. Since no-ops inserted for timing would not do not contribute to the value of a variable, they would not be included in a slice. There is no theory of slicing for asynchronous events, i.e., interrupts.

5. The intersection of two slices is a slice with a slicing criterion that is related to the intersection of the respective slicing criterions.

6. Using intersections, a slice can be further partitioned into two parts, a backbone slice, the union of all intersections with other slices, and the set of statements not included in the backbone slice (called the residual set).

One typically tries to understand a program by a combination of reading program text, reading program documentation, ad hoc testing with different inputs, and dynamic tracing. Program documentation gives a high level picture of what is going on, but is usually not directly useful in code understanding. Ad hoc testing can give a clear understanding of what is happening, but it depends on careful input selection for useful results. Dynamic tracing shows exactly how a given result is produced, but produces a deluge of information which must then be understood. Reading program text also shows exactly how a given result is produced, but again one can be overwhelmed with details that have nothing to do with the task at hand.

Program slicing addresses this information overload problem in three ways:

1. A static program slice extracts all the statements that might be relevant to a computation.

2. A dynamic program slice extracts all the statements that are relevant to a computation for a given set of inputs.

3. A sliced-based model of program structure can be constructed.

From unions and intersections of slices, a slice-based model of program structure can be built. Figure 1 represents a view of an unfamiliar program. One sees some inputs, some outputs, and a shapeless mass of code with unknown connections among inputs and outputs. Figure 2 represents the knowledge obtained after computing a slice on Output3. To answer questions about the computation of Output3, examine the slice on Output3 (the shaded region labeled σ, the slice). The statements not in the slice (unshaded region ω) can be ignored. Figure 3 shows how the slice-based model can further refine knowledge about the program. The backbone slice relative to Output1 and Output2, β, is the intersection of the respective slices.

5 Using Slicing as an Aid to Validating Safety

We now show how to meld these various ideas into a coherent whole. First, the auditor uses system fault tree analysis to locate critical components. The software that is invoked when a hazardous condition occurs is identified in the system. The auditor then locates the software variables that are the indicators of unsafe conditions. Program slices are extracted on these "interesting" variables; i.e., slices are extracted using the variable(s) of interest at the points where the hazards should be detected. Software fault tree analysis is used on these slices. These slices can be used to validate that there are no interactions between critical components or with non-critical components. In other words, the backbone slices of safety critical components with any other component must be empty.

5.1 Diversity Assurance

The applications of slicing to software fault tree analysis are straightforward. In the simplest use, a static program slice will yield all inputs that any variable uses for its computation, so it can be used to validate diversity by showing that paths from particular inputs to particular outputs do not occur.

Program slices computed from the outputs of individual hazards can be examined to determine the logical independence of the events. For instance, if A and B are two critical conditions, the program slices on these two conditions can be compared to give partial information on whether or not both conditions can occur simultaneously. If the slices have no intersection then there is no way that the software can guarantee that both will not occur simultaneously (there may be other ways the verify that both will not occur). If the slices on A and B have a common intersection, inspection of the intersecting parts may prove that both conditions cannot occur together (although the functional diversity of such computations is suspect).
These program projections can also be highlighted for more vigorous analysis, inspection and testing. A static program slice can be further refined by examining the trajectory of specific inputs through the program: we can use a dynamic slice to observe individual instances of the computation. This simplifies the tedious task of the auditor and permits undivided attention to be focused on the analytic portions of the audit.

5.2 Example Using Unravel

The utility of a slicing tool comes from automating the task of finding statements that are relevant to a computation. Without any tool, the SQA auditor evaluating functional diversity would examine the program under consideration until outputs were identified that should be computed independently. The auditor would then try to verify independence by reading code.

**Unravel** is a static program slicer, currently under development at NIST as part of a research project. Unravel works with the variable names that the developer uses, and does not require explicit consultation with other documentation. The tool is divided into three main components: a source code analysis component to collect information necessary for the computation of program slices, a link component to link flow information from separate source files together and an interactive slicing component that the SQA auditor can use to extract program components and statements for answering questions about the software being audited.

The objective of the source code analysis component is to transform the source code into a flow graph representation. Each node of the flow graph is annotated with lists of variables defined or referenced and with the location of the source code statement that corresponds to the node. Since the flow graph representation is independent of the source programming language, the source analyzer is the only component that must be changed for a different programming language. This significantly reduces the cost of implementing Unravel for other programming languages.

We illustrate the SQA auditor's use of Unravel for program slicing. The following three functions are performed:

1. Determining the code that is executed when a system-critical activity occurs.
2. Evaluating functional diversity in a program.
3. Performing a string check, a trace of a program input through the code until an output is reached.

Once a variable is identified as interesting, the SQA auditor directs a slicer to compute a program slice on the variable. Instead of examining the entire program, only the statements in the slice need to be examined by the auditor.

As an example, consider the program of figure 4. The program has four inputs, red, green, blue and yellow, and four outputs, sweet, sour, salty, and bitter. The four inputs might be sensor readings and the four outputs might be control device activations.

To determine the code that is executed when critical control variable sweet is activated the auditor would compute a slice on sweet. Lines 14, 16, 19 and 20 would be identified as members of this slice. The Unravel display of this is in figure 5.

An SQA auditor might then wish to establish the independence of the calculation of sweet and sour. The auditor obtains a slice on sweet as before. Then the auditor would then direct the tool to slice on sour to obtain lines 14, 16, 19, 21, 22 and 23. (figure 6) As a last step, the auditor would intersect the two slices to identify statements common to the two computations. In this case, statements 14, 16 and 19 are used in both computations (figure 7). At this point the SQA auditor (not the tool) evaluates functional diversity of the code.

Now consider a string check where an SQA auditor follows a sensor signal through the code to an output, i.e., computes a dynamic slice. The goal of the string check is to connect inputs (sensor readings) with outputs (control activations). Starting with input red, at line 14, a dynamic slice collects line 19 (a recomputation and use of red) and line 20 (a use of red). At this point the variable sweet, is of interest. Lines 22 and 23 are collected since red is used; sour is now of interest. With no more uses of red, the dynamic slice is complete and we have determined that outputs sweet and sour depend on red.

By constructing a slice-based model of the program, the auditor can observe that output sweet depends on the inputs red and green, output sour also depends on red and green, salty depends on blue and yellow, and bitter depends on yellow and green.

6 Conclusion

We have presented a novel application of program slicing that aids software quality assurance auditors in the validation of safety critical code from a functional diversity perspective. While this approach does not solve all the problems that an auditor faces, it does ameliorate those which are amenable to computation.
7 Acknowledgements

We are grateful for the encouragement and suggestions of D. Wallace and the support of the Computer Systems Laboratory at NIST.

```c
int fetch()
{
    int value;
    scanf("%d", &value);
    return value;
}

main()
{
    int red, green, blue, yellow;
    int sweet, sour, salty, bitter;
    int i;

    red = fetch();
    blue = fetch();
    green = fetch();
    yellow = fetch();

    red = 2 * red;
    sweet = red * green;
    sour = 0;
    for (i = 0; i < red; i++)
    {
        sour += green;
    }
    salty = blue + yellow;
    green = green + 1;
    bitter = yellow + green;

    printf("%d %d %d %d\n", sweet, sour, salty, bitter);
    exit(0);
}
```

Figure 4

References


Figure 5
```c
int red, green, blue, yellow;
int sweet, sour, salty, bitter;
int i;

red = fetch();
blue = fetch();
green = fetch();
yellow = fetch();

red = 2 * red;
sweet = red * green;
sour = 0;
for (i = 0; i < red; i++)
    sour += green;
salty = blue + yellow;
green = green + 1;
bitter = yellow + green;

printf("%d %d %d %d\n", sweet, sour, salty, bitter);
exit(0);
```

Figure 6
int red, green, blue, yellow;

int sweet, sour, salty, bitter;

int i;

red = fetch();
blue = fetch();
green = fetch();
yellow = fetch();

red = 2*red;
sweet = red*green;
sour = 0;
for (i = 0; i < red; i++)
    sour += green;
salty = blue + yellow;
green = green + 1;
bitter = yellow + green;

printf ("%d %d %d %d\n", sweet, sour, salty, bitter);
exit(0);