USING A PROGRAM SLICING CASE TOOL FOR EVALUATING HIGH INTEGRITY SOFTWARE SYSTEMS

Leo Beltracchi*
lxb@nrc.gov
(301) 415-6558

James R. Lyle
jlyle@nist.gov
(301) 975-3270

Dolores R. Wallace
dwallace@nist.gov
(301) 975-3340

Technology Administration
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

ABSTRACT

This paper describes a Computer Aided Software Engineering (CASE) tool, **unravel**, that can assist in the evaluation of high integrity software by using program slices to extract a *software channel* of code for examination and test. The tool, available through the National Institute of Standards and Technology, can currently be used to evaluate software written in ANSI C and is designed such that other languages can be added. The opinions and viewpoints presented herein are those of the authors and do not necessarily represent the criteria, requirements and guidelines of the U.S. Nuclear Regulatory Commission.

I. INTRODUCTION

High integrity software systems are often used in environments where a lack of response can cause an accident or result in severe financial loss due to an operational failure. A typical digital computer-based reactor protection system uses sensors to measure process variables that are used to evaluate safety functions. A protective action is initiated when the process variable or safety function exceeds a set point. A fault in the code can result in the violation of a safety function. Detecting a fault in the code is difficult and costly. This paper describes a Computer Aided Software Engineering (CASE) tool, **unravel**, that can assist in the evaluation of high integrity software by using program slices to extract a *software channel* of code (analogous to a hardware channel) for examination and test. The informal notion of a *software channel,* is formally captured by the concept of a program slice. The tool, available through the National Institute of Standards and Technology[10], can currently be used to evaluate software written in ANSI C and is designed such that capability for slicing other languages can be added.

Program slicing is a static analysis technique that extracts all statements relevant to the computation of a given variable. This is accomplished by using *data-flow analysis* [6] to analyze the program source code without the need to actually execute the program. For example, all code relevant to a reactor flux trip signal may be defined by slicing on the reactor trip variable in the code. Application of program slicing to the evaluation of high integrity software reduces the effort of examining software by allowing a reviewer to focus attention on one computation at a time. Program slicing can be used to identify a *channel* of code associated with each trip variable. The process for identifying the *channel* of code is independent from the requirements and specifications for the code. Since the slice (channel) identifies all code and plant variables (sensor data) associated with a trip variable, test cases may be structured for the specific trip variable. Analysis of the code within the slice helps to define test requirements needed to evaluate the code. The use of random input sensor data to test the *channel* is also discussed.

By combining program slices using logical set operations (e.g., *union* or *intersection*), **unravel** can identify code that is common to each slice. Analysis and evaluation of the code common to each slice by an reviewer is important because it provides a measure of the independence and isolation between the slices (channels). This information is useful since a failure involving this code may lead to a malfunction of more than one protective action. Manual examination of a program for common code in which a reviewer searches for code shared between two trip variables, or until it is determined that there is no common code, or that common code present will not compromise the safety function is often a slow, tedious, error-prone process. With **unravel**, once two different trip variables have been identified, program

*Mr. Beltracchi is with the U.S. Nuclear Regulatory Commission slices can be identified to find statements relevant to each channel. A fault in any source program statements common to two slices has the potential to cause common mode failure. Review and the careful test of the common code between the two slices is critical to the assurance of high integrity software.

II. A PROGRAM SLICING TOOL

Program slicing, an application of data-flow analysis[6], can be used to transform a large program into a smaller one containing only those statements relevant to the computation of a given variable. Program slices have been shown to aid testing[4], debugging[9], program maintenance[2], program understanding[12], and automatic integration of program variants[5].

A **Program Slice** is defined as follows:
Given a syntactically correct source program P, in some programming language, and a *slicing criterion* $C =< L, V >$. Where L is a location in the program and V is a variable in the program. S is a slice of program P for criterion C if

(1) S is derived from P by deleting statements from P,

(2) S is syntactically correct, and

(3) for all executions of P and S, in any given execution of P and of S with the same inputs, the value of V in the execution of slice S just before control reaches location L is the same as the value of V in program P just before control reaches location L.

The function of the slicing criterion is to specify the program variable that is of interest along with a location in the program where the value of the variable is desired.

The program slicing tool, **unravel**, constructs program slices from the control structure of the program and the pattern of assignment and reference to variables by backward chaining from the slicing criterion to the beginning of the program. Figure 1 shows the **unravel** output for a short program of three inputs and three outputs with a slice on **output1** highlighted.

The following definitions are helpful in understanding how program slices are constructed.

Defs(n): The set of variables defined (assigned to) at statement n.

Refs(n): The set of variables referenced at statement n.

Req(n): A set of statements that is included in a slice along with statement n. The set is used to specify control statements (e.g., **if** or **while**) enclosing statement n or other characters that are syntactically part of statement n but are not contiguous with the main group of characters comprising the statement.

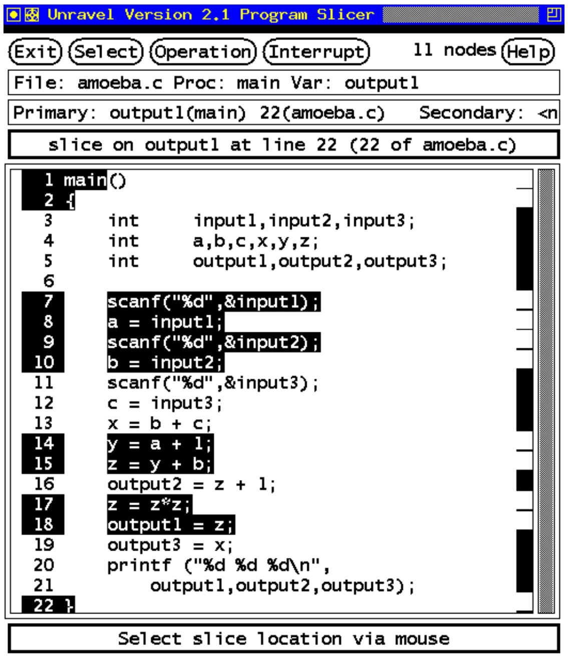

Figure 1: Slice on Output 1

An algorithm for constructing program slices must locate all statements relevant to a given slicing criterion. The essence of a slicing algorithm is the following: starting with the statement specified in the slicing criterion, include each predecessor statement that assigns a value to any variable in the slicing criterion, generate a new slicing criterion for the predecessor by deleting the assigned variables from the original slicing criterion, and add any variables referenced by the predecessor. The **unravel** slicing algorithm considers the following issues:

1 Assignment statements (expression statements in C)
2 Compound control statements
3 Declared structures
4 Pointers
5 Dynamic structures
6 References to structure members by pointer
7 Assignment to structure members by pointer
8 Procedure calls

Only assignment and compound control statements are discussed in this paper.

For expression statement n, a predecessor of statement m, the defs(n) set and the slicing criterion determines if an expression statement is included in a slice.

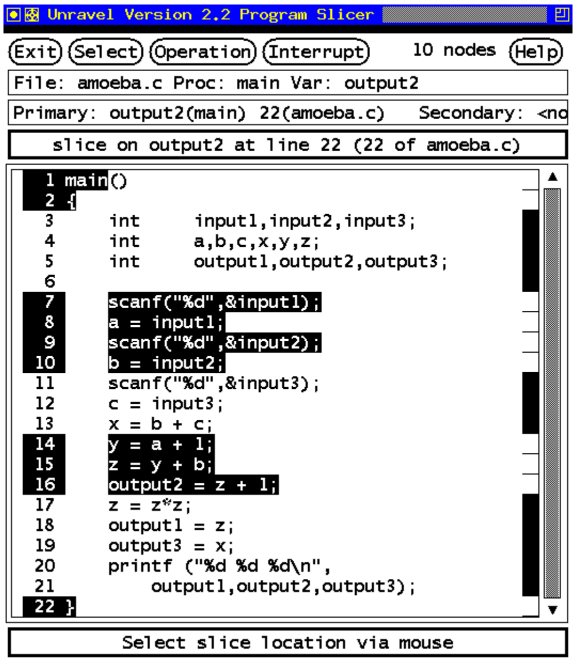

Figure 2: Slice on Output 2

$$S_{<m,v>} = \begin{cases} S_{<n,v>} & \text{if } v \notin \text{defs(n)} \\ \{n\} \cup S_{<n,x>} \forall x \in \text{refs(n)} & \text{otherwise} \end{cases}$$

For example, to use the above rule for slicing on assignment statements to determine the value of y at line 15 of the program in Figure 2 the criterion would be $< 15, y >$. The rule for assignment statements yields one of two results based on whether **y** is assigned to at line 14 (the predecessor of 15). Since **y** is assigned a value at line 14 the second part of the rule would be used so that line 14 is included in the slice and new slicing criteria are generated for any variables that **y** depends on at line 14. In this case, the criterion $< 14, a >$ would be generated and the slice on that criterion would be a subset of the slice $< 15, y >$. To construct the slice on $< 14, a >$ the first part of the rule is used, since **a** is not assigned a value at line 13. The generated criterion is $< 13, a >$ which again generates a criterion without adding a statement to the slice. This would continue until line 8 was added into the slice by the criterion $< 9, a >$. Constructing the slice on **output2** at line 21 presented in Figure 2 generates the criterion $< 15, y >$ as an intermediate step in the construction of the slice on $< 21, output2 >$.

A compound control statement is a statement that has a condition directly controlling the execution of another statement (possibly also a compound statement). Control statements such as **if, switch, while, for** and **do** ... **while** should be included in a program slice whenever any statement governed by the control statement is included in a slice. When control statement n is added to a program slice, the slice on the criterion $< n, \text{refs(n)} >$ is added to the original slice. For each statement, n, associate a set, *req(n)*, of statements that are required to be included in any slice containing statement n. The slicing rule for $v \in \text{defs(n)}$ becomes:

$$\begin{aligned} S_{<m,v>} \quad = \quad & \{n\} \bigcup \left(\bigcup_{x \in \text{refs(n)}} S_{<n,x>} \right) \bigcup \\ & \left(\bigcup_{\ y \in \text{refs(k)}} \ \bigcup_{k \in \text{req(n)}} S_{<k,y>} \right) \end{aligned}$$

The result of the revised rule is to include the set of required statements for statement n, *req(n)*, whenever statement n is included in a slice. In the program in Figure 1 statements on lines 7-21 require lines 1, 2 and 21.

From unions and intersections of slices, a slice-based model of program structure can be built that has applications to program understanding tasks, such as software reviewing. Figure 3 illustrates how slices can be used to quickly examine a program's structure. The initial view of an unfamiliar program, left part of Figure 3, usually contains some inputs, some outputs, and a shapeless mass of code with unknown connections among inputs and outputs. After constructing a slice on some variable, the program is partitioned into two parts, statements relevant to the computation of the slice variable and statements not relevant to the computation of the slice variable. The middle part of Figure 3 represents what is known about the program after constructing a slice on **Output3**. To answer questions about the computation of **Output3**, the slice on **Output3** (shaded region labeled σ), should be examined and the statements not in the slice (unshaded region ω) can be ignored.

Program slices can be combined with logical set operations to explore dependencies between two computations. The right part of Figure 3 shows how program slices can further refine knowledge about the program. The intersection of slices on **Output1** and **Output2**, labeled β, contains statements relevant to both variables. A single bug could cause both outputs to be incorrect. In a debugging task, if a programmer suspects that a single bug is causing both outputs to be incorrect, then the intersection of slices should be examined. However, if the programmer has some confidence that one output is correctly computed, then a bug is more likely to be found among the statements not in the intersection of slices. For example, if **Output1** fails some set of test data but **Output2** appears to be correct, then the programmer should look for the error among the statements unique to **Output1** (shaded region labeled $\rho 1$ in Figure 3).

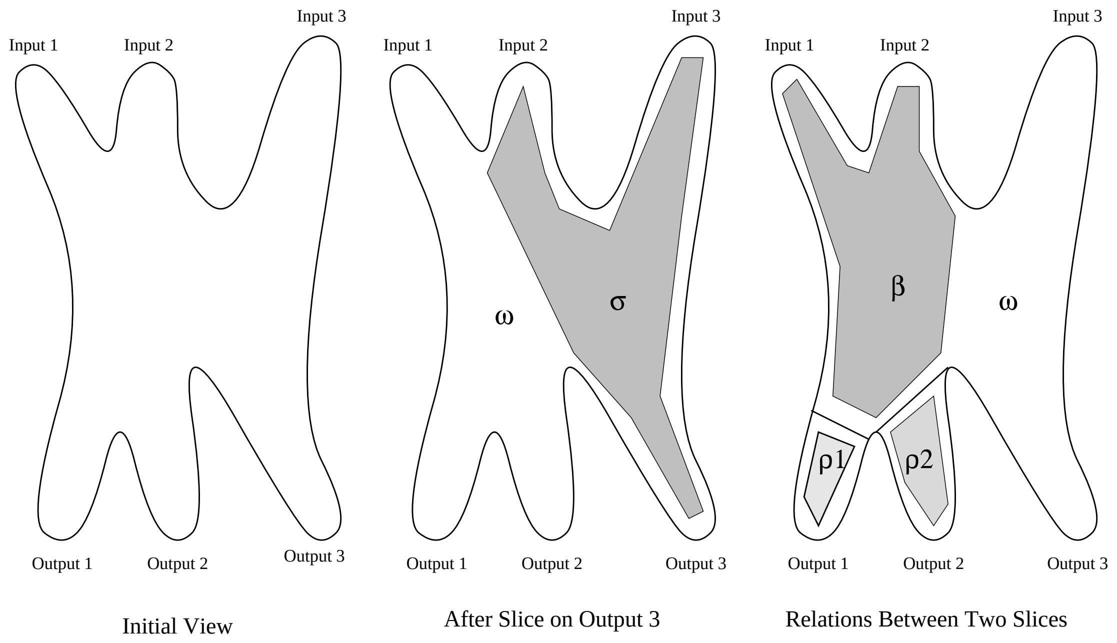

Figure 3: Slice Based Model of Program Structure

III. FUNCTIONAL ANALYSIS OF SOFTWARE

Unravel is useful in assisting a designer or reviewer in the functional analysis of software as well as planning assurance activities. This section discusses both of these topics.

Establishing a *channel* for each trip variable within nuclear safety system software is the first step in a functional analysis. The next step is to analyze the contents of each channel. The goal of this step is to verify the code with respect to the specifications for the software. For example, after constructing a program slice on **output3** it is trivial to verify from the **unravel** output in Figure 4 that **output3** depends on **input2** and **input3** and does not depend on **input1**. An analysis of each channel identifies functions and tasks useful in verifying adherence to requirements. The analysis of each channel is also useful in identifying internal variables calculated, the sensed plant variables used by the channel, and memory stored variables used by the channel as programmed. The analysis of each channel is a much greater cognitive challenge than the task of tracing requirements from a specification.

Another type of functional analysis is the evaluation of the independence and isolation between any two channels of software. This feature of **unravel** allows a reviewer to evaluate if common software exists between two diverse safety functions. The ideal situation is to have no common software shared by diverse safety func-

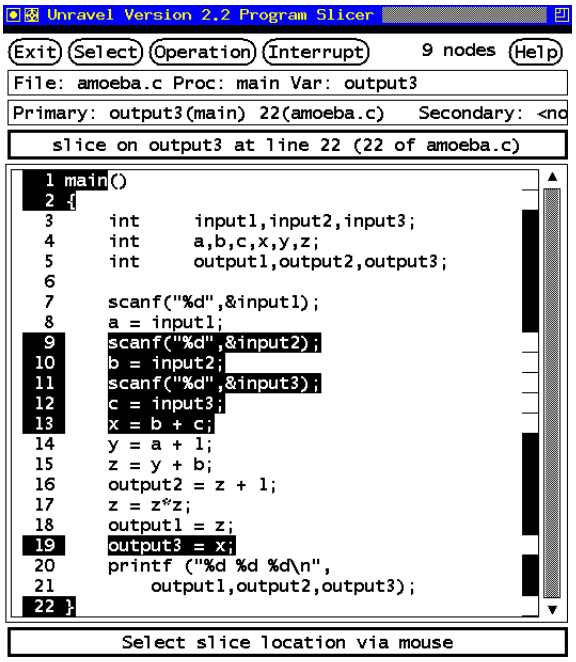

Figure 4: Slice on Output 3

tions. However, if common software is shared by diverse safety functions, then this software must be thoroughly analyzed and tested to ensure no faults exist within it. In addition to common code, a reviewer may also evaluate if common plant data or common memory stored data are used by the channels under study by performing a manual comparison of the data between the channels. Verification of the accuracy of shared plant data or shared memory stored data is critical to the integrity of the functions performed within each channel.

In addition to the independence and isolation analysis between channels performing safety functions, an analysis of channels performing safety functions and other functions in the code is also necessary to complete the isolation study. For example, the isolation between a safety function and a fault tolerance function is also important. An undetected fault in shared code between these channels will, when activated, result in the loss of each function. Similarly, a loss of each function will also occur should faulty common plant data or common memory data be used by each function.

Figure 5 shows the **unravel** output for the intersection of slices on **output1** (Figure 1) and **output2** (Figure 2). Figure 6 shows the **unravel** output for the code in the slice of **output1** that is not shared with the computation of **output2**. By automatically locating statements shared between two computations or code unique to a single computation, the task of evaluating the interaction between the functions implemented by the computations is simplified.

The products from the functional analysis of channels are useful in planning and performing assurance activities. One of the assurance activities is to test each channel for faults and conformance to requirements. Dahll [1] found that an effective form of test data in the detection of faults is the use of uniform random input data and a test oracle. The use of uniform random test data proved to be more effective in the detection of faults than the use of systematic data (which exercises functions within the specifications) or the use of plant simulation data (which attempts to simulate actual plant scenarios). The use of uniform random test data should result in near uniform executions of the various paths through a channel and thus enhance the probability of detecting faults. A fault within the channel that exhibits itself over a wide range of the input plant sensor data should be easily detected. However, a fault in the channel that exhibits itself as a spike over the range of input plant sensor data will be difficult to detect.

The use of time dependent sets of input plant data is also necessary to evaluate the functional performance of each channel and the states that may arise therein. Parnas [11] reports that there may be states that may arise only after a period of time in the processing of

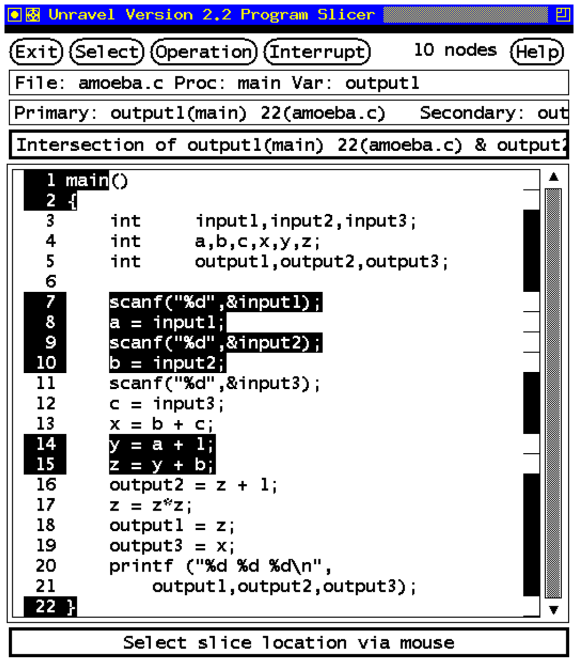

Figure 5: Intersection of Slices on Output 1 and Output 2

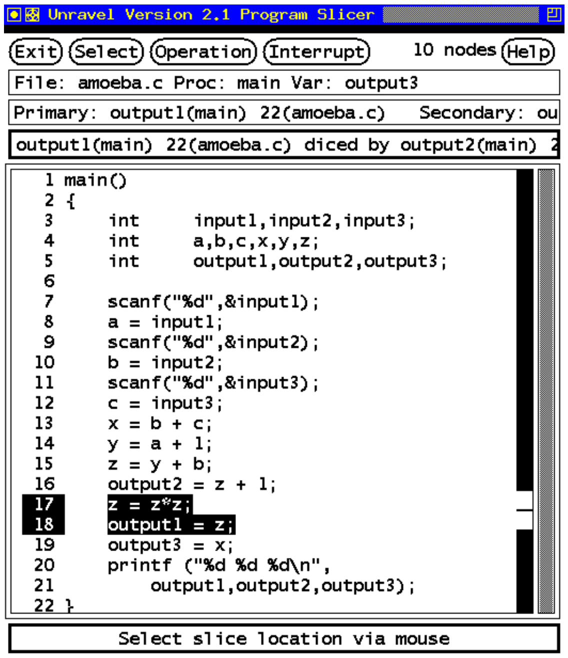

Figure 6: Statements Unique to Output 1

data. One example of such a state is a moving average that needs several time sequential data points to arrive at a data point. If six time sequential data points are needed to arrive at one moving average data point, then the test of this function should use a minimum of six data points to verify the function. Additional input data points are necessary in order to evaluate the various states wherein the moving average is changing rapidly with time. These input plant data states should be representative of the various operating scenarios, that are the design basis for the safety system. The limiting design basis events should define the most challenging time response scenarios from which to test the software channel. Furthermore, the initial state of the software channel must be reset prior to a new test case in order not to be influenced by memory stored data from the previous test case.

For a thorough assessment of a channel of software, it is also necessary to evaluate internal variables. A test of internal variables within a channel is similar to point to point testing in a channel of hardware. For a specific set of plant sensor data for a channel of software, one progresses forward from the input data to evaluate the computed value of each internal variable. The computed value of the internal variable is compared to a precalculated value to determine its validity. The printout of internal variables may be achieved through the use of a simple assertion when a value for a key partial result is satisfied. The successful test of each internal variable minimizes the potential of a fault within the channel.

IV. EMPIRICAL EVALUATION OF UNRAVEL

Unravel was initially evaluated[10] by one NRC reviewer in the context of reviewing safety system software for quality as is done by the NRC. This preliminary evaluation considered the size of slices produced, time to compute slices, and usability by a novice user. This should not be considered a complete evaluation, but rather a demonstration of the potential of the tool to be confirmed by further use.

The objectives of the evaluation were to determine the following:

(1) Are program slices smaller than the original program to an extent that is useful to a software reviewer evaluating a program?

(2) Can program slices be computed quickly enough to be useful in an review?

(3) Is **unravel** usable by a novice user?

An example of typical safety system code was used to test and refine **unravel**. Demonstration of **unravel** using this and other examples were given to the NRC reviewer. The demonstrations provided useful suggestions that resulted in improvements to the user interface and in the identification of features to be explained in more depth in the user manual or to be included in a later version of **unravel**.

The safety system example (1200 lines) came in three versions. One version was written to conform to safety system diversity requirements while the other two were deliberately seeded with common code. **Unravel** was able to verify and display the presence of the common code in the seeded version and show the absence of common code in the diverse version.

The NRC reviewer directed **unravel** to compute slices for both safety and nonsafety related process variables. The reviewer was able to identify several unanticipated connections between subsystems. The following observations by the reviewer are relevant to the evaluation of **unravel**:

(1) Use of **unravel** in an review should significantly enhance the ability to perform and analyze string checks.

(2) **Unravel** is easy to operate for a person with computer skills.

(3) **Unravel** can disclose subtle relationships between safety related and nonsafety related code that would require a C expert to discover.

(4) The majority of the slices were less than 25 percent of the size of the original program (90 percent for the safety system example). The user of **unravel** can expect to eliminate a significant portion of code from consideration when using program slicing to extract a given computation for examination.

(5) Requested slices were computed is less than one minute.

V. FUTURE WORK

Using program slicing in software development and the analysis of the final software product has potential to increase software quality in several ways that we plan to explore.

Once the functional structure of a channel has been established, then a fault tree analysis may be performed. A fault tree analysis is a safety analysis wherein the output trip variable of the channel is assumed to be in a faulty state, i.e., failure to trip when unsafe plant conditions exist. An analysis of the computations leading to the output is then performed to determine how conditions could exist that support the faulty output state. Should such a state be identified, then modifications to the code are necessary to avoid the loss of the safety function [8, 3].

Leveson and Shimeall [7] report on the results of research that evaluated the effectiveness of various fault detection techniques. This research evaluated five different fault detection techniques consisting of code reading by stepwise abstraction, static-data flow analysis, run time assertions, multiversion voting, and functional testing with follow on structural testing. They report that each of these test techniques were useful in the unique detection of faults not found by the other techniques. However, they also found other faults that were detected by multiple test techniques.

A run time assertion generates reports when faults produce an erroneous run-time state, e.g., an incorrect internal variable. Assertions examine the system state at specific points in the execution of the software. For safety critical software, this is an important result from Leveson and Shimeall because test and evaluation of internal variables has not received a great deal of attention.

VI. CONCLUSIONS

Assurance activities of safety critical software adopt many of the techniques used in the test of a channel of safety critical hardware. **Unravel** is a tool to assist a designer or reviewer in implementing this activity for software. While the initial evaluation of **unravel** considered only a single reviewer applying the tool on one program, the results suggest that further use may confirm that **unravel** is a powerful tool for assurance activities of safety critical software.

VII. ACKNOWLEDGMENTS

Funding was provided by the U.S. Nuclear Regulatory Commission and and the National Institute of Standards and Technology. Additional information about **unravel** can be found on the **unravel World Wide Web** home page:
`http://hissa.ncsl.nist.gov/~jimmy/unravel.html`
Unravel can be obtained by **anonymous ftp** from: `hissa.ncsl.nist.gov`

References

[1] G. Dahll. Software diversity - a way to enhance safety? In *Second European Conference On Software Quality Assurance*, Oslo, Norway, May 30-June 1, 1990.

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

[3] K. B. Gallagher and J. R. Lyle. Software safety and program slicing. In *Proceedings of the Eighth Annual Conference on Computer Assurance*, pages 71–76, Gaithersburg, Maryland, June 14-17, 1993. National Institute of Standards and Technology.

[4] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5:143–162, September 1995.

[5] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[6] K. Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[7] N. Leveson, S. Cha, J. C. Knight, and T. Shimeall. The use of self tests and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[8] N. Leveson, S. Cha, and T. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Computer*, 8(4):48–59, May 1991.

[9] J. R. Lyle and M. D. Weiser. *Experiments in Slicing-Based Debugging Aids.* In *Empirical Studies of Programmers,* Elliot Soloway and Sitharama Iyengar, eds. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[10] J.R. Lyle, D.R. Wallace, J.R. Graham, K.B. Gallagher, J.P. Poole, and D.W. Binkley. A case tool to evaluate functional diversity in high integrity software. Technical Report IR 5691, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

[11] D.L. Parnas, A.J. van Schouwen, and S.P. Kwan. Evaluation standards for safety critical software. Technical Report Technical Report 88-220, Department of Computing & Information Science, Queen's University, Kingston, Ontario, K7L 3N6, May 1988.

[12] M. Weiser. Programmers use slicing when debugging. *CACM*, 25(7):446–452, July 1982.