Using the Unravel Program Slicing Tool to Evaluate High Integrity Software

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
Information Technology Laboratory
Gaithersburg, MD 20899

ABSTRACT

This paper describes a program slicing tool, **unravel**, that can assist in the evaluation of high integrity software by using program slices to extract a single computation from a program 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.

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. Detecting a fault in the code is difficult and costly. This paper describes a program slicing tool, **unravel**, that can assist in the evaluation of high integrity software by using program slices to extract a single computation for examination and test. The tool, available through the National Institute of Standards and Technology[6], can currently be used to evaluate software written in ANSI C and is designed such that capability for slicing other languages can be added (we are currently considering FORTRAN, C++ and Java).

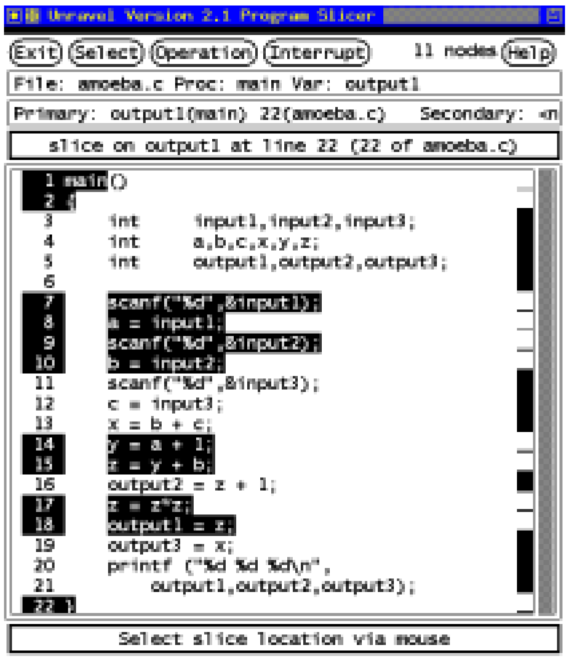
Program slicing is a static analysis technique that extracts all statements relevant to the computation of a given variable. This is accomplished by using *data-flow analysis* [4] to analyze the program source code without the need to actually execute the program. Application of program slicing to the 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 the code associated with some key variable. The process for identifying the code is independent of the requirements and specifications for the code. Since the slice identifies all code and variables associated with a variable, test cases may be structured for the specific variable. Analysis of the code within the slice helps to define test requirements needed to evaluate the code.

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 a reviewer is important because it provides a measure of the independence and degree of isolation between different computations. This information is useful since a failure involving this code may lead to a malfunction of more than one logical program component. Manual examination of a program is often a slow, tedious, error-prone process. With **unravel**, once two different computations have been identified, program slices can be identified to find statements relevant to each computation. 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 within 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[4], 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[2], debugging[5], program maintenance[1], program understanding[7], and automatic integration of program variants[3].

A **Program Slice** is defined as follows:



```
File: amoeba.c Proc: main Var: output1
Primary: output1(main) 22(amoeba.c)   Secondary: <n
slice on output1 at line 22 (22 of amoeba.c)
```

```
 1 main()
 2 {
 3     int     input1,input2,input3;
 4     int     a,b,c,x,y,z;
 5     int     output1,output2,output3;
 6
 7     scanf("%d",&input1);
 8     a = input1;
 9     scanf("%d",&input2);
10     b = input2;
11     scanf("%d",&input3);
12     c = input3;
13     x = b + c;
14     y = a + 1;
15     z = y + b;
16     output2 = z + 1;
17     z = z*z;
18     output1 = z;
19     output3 = x;
20     printf ("%d %d %d\n",
21         output1,output2,output3);
22 }
```

```
Select slice location via mouse
```

Figure 1: Slice on Output 1

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.

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:

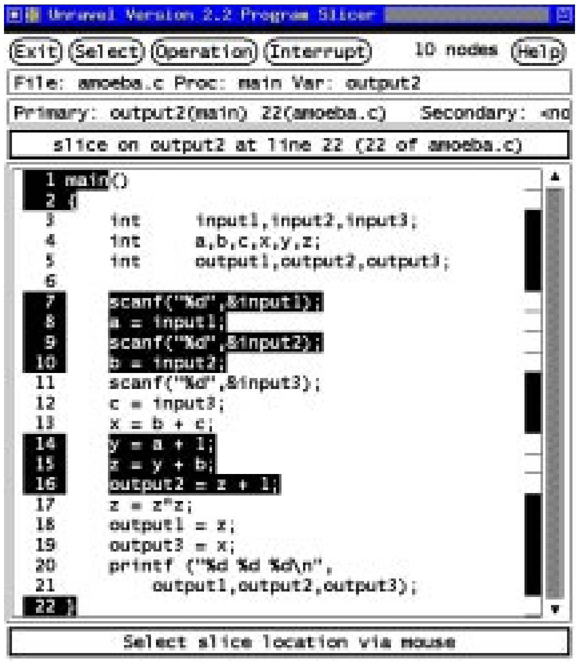
1 Assignment statements (expression statements in C)
2 Compound control statements
3 Declared structures
4 Pointers
5 Dynamic structures
6 References to structure members by pointer
7 Assignment to structure members by pointer
8 Procedure calls

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.

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

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 a value or not 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



```
File: amoeba.c Proc: main Var: output2
Primary: output2(main) 22(amoeba.c)   Secondary: <no
slice on output2 at line 22 (22 of amoeba.c)
 1 main()
 2 {
 3       int      input1,input2,input3;
 4       int      a,b,c,x,y,z;
 5       int      output1,output2,output3;
 6
 7       scanf("%d",&input1);
 8       a = input1;
 9       scanf("%d",&input2);
10       b = input2;
11       scanf("%d",&input3);
12       c = input3;
13       x = b + c;
14       y = a + 1;
15       z = y + b;
16       output2 = z + 1;
17       z = z*z;
18       output1 = z;
19       output3 = x;
20       printf ("%d %d %d\n",
21           output1,output2,output3);
22 }
Select slice location via mouse
```

Figure 2: Slice on Output 2

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 a compound statement). Simple compound statements (a list of statements enclosed in braces) and procedure definitions (a procedure header and braces to enclose the procedure statements) are treated like compound control statements with no condition. 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:

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

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 22 to capture the enclosing procedure definition and enclosing braces.

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 $\rho1$ in Figure 3).

III. USING UNRAVEL

Unravel is useful in assisting a designer or reviewer in planning assurance activities as well as the analysis of software.

To verify code with respect to the specifications for the software, compute a slice and then compare the slice to the specification. 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**. The program requirements for **output3** can be quickly examined and compared to the slice. If the requirements indicate that **input1** should be relevant to **output3** then an error is clearly present. In a similar fashion, if the requirements indicate that either **input2** or **input3**

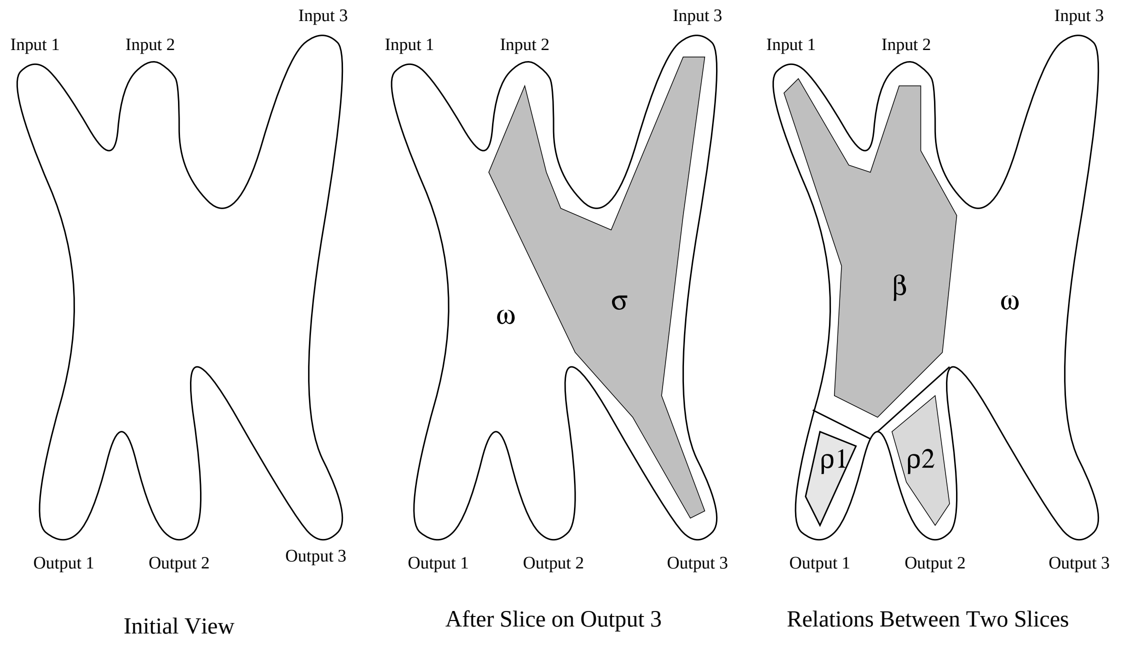
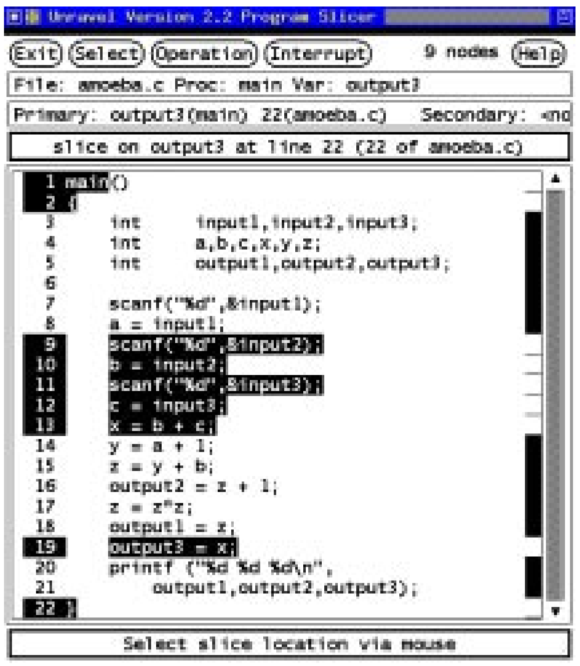


Figure 3: Slice Based Model of Program Structure

have no relation to **output3** then a closer examination of the slice and requirements is called for. There may be an error or it may be that there is a subtle relationship among the inputs that was not recognized by the requirements writer.

Another type of functional analysis is the evaluation of the independence and isolation between two computations. This feature of **unravel** allows a reviewer to evaluate if common software exists (and potential for common failure) between two computations. The ideal situation is to have no common software shared by critical computations. However, if common software is shared by critical computations, then this software must be thoroughly analyzed and tested to ensure no faults exist within it.

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.

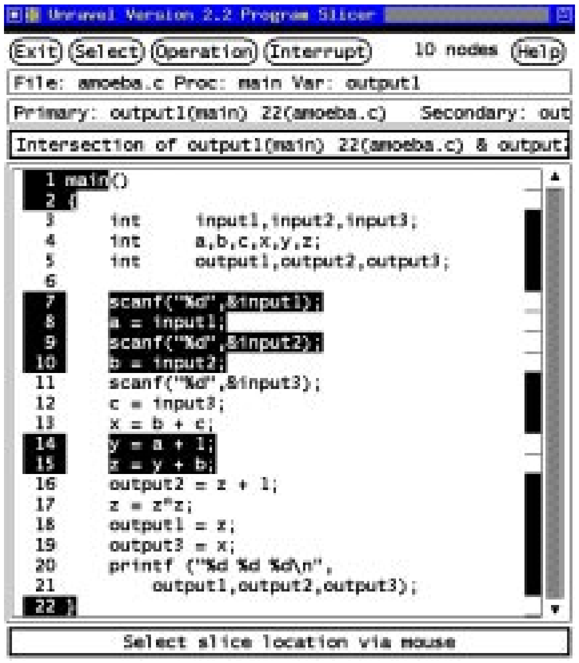


```
File: amoeba.c Proc: main Var: output3
Primary: output3(main) 22(amoeba.c)   Secondary: <no
        slice on output3 at line 22 (22 of amoeba.c)
  1 main()
  2 {
  3       int     input1,input2,input3;
  4       int     a,b,c,x,y,z;
  5       int     output1,output2,output3;
  6
  7       scanf("%d",&input1);
  8       a = input1;
  9       scanf("%d",&input2);
 10       b = input2;
 11       scanf("%d",&input3);
 12       c = input3;
 13       x = b + c;
 14       y = a + 1;
 15       z = y + b;
 16       output2 = z + 1;
 17       z = z*z;
 18       output1 = z;
 19       output3 = x;
 20       printf ("%d %d %d\n",
 21           output1,output2,output3);
 22 }
        Select slice location via mouse
```

Figure 4: Slice on Output 3

IV. EMPIRICAL EVALUATION OF UNRAVEL

Unravel was initially evaluated[6] by a software reviewer in the context of reviewing safety system software for quality. This preliminary evaluation considered the



File: amoeba.c Proc: main Var: output1

Primary: output1(main) 22(amoeba.c)   Secondary: out

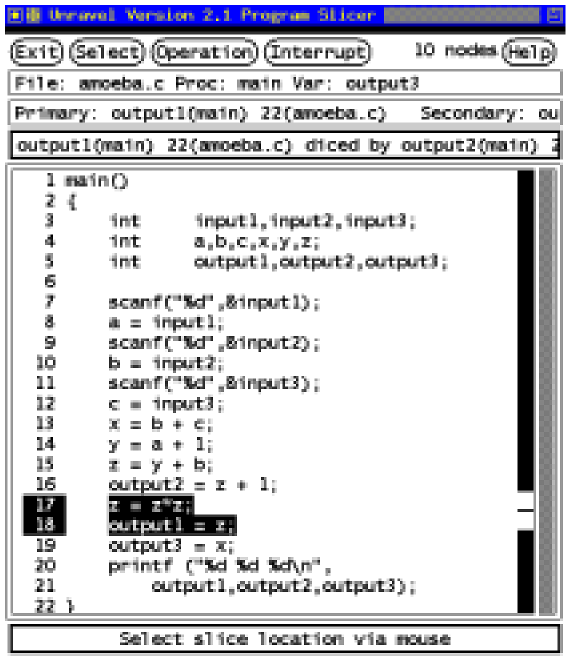
Intersection of output1(main) 22(amoeba.c) & output

```
main()
{
     int       input1,input2,input3;
     int       a,b,c,x,y,z;
     int       output1,output2,output3;

     scanf("%d",&input1);
     a = input1;
     scanf("%d",&input2);
     b = input2;
     scanf("%d",&input3);
     c = input3;
     x = b + c;
     y = a + 1;
     z = y + b;
     output2 = z + 1;
     z = z*z;
     output1 = z;
     output3 = x;
     printf ("%d %d %d\n",
          output1,output2,output3);
}
```

Select slice location via mouse

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



File: amoeba.c Proc: main Var: output3

Primary: output1(main) 22(amoeba.c)   Secondary: ou

output1(main) 22(amoeba.c) diced by output2(main) 2

```
main()
{
     int       input1,input2,input3;
     int       a,b,c,x,y,z;
     int       output1,output2,output3;

     scanf("%d",&input1);
     a = input1;
     scanf("%d",&input2);
     b = input2;
     scanf("%d",&input3);
     c = input3;
     x = b + c;
     y = a + 1;
     z = y + b;
     output2 = z + 1;
     z = z*z;
     output1 = z;
     output3 = x;
     printf ("%d %d %d\n",
          output1,output2,output3);
}
```

Select slice location via mouse

Figure 6: Statements Unique to Output 1

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 a review?

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

An example of typical high integrity system code was used to test and refine **unravel**. Demonstration of **unravel** using this and other examples were given to the software reviewer. The reviewer provided useful suggestions that resulted in improvements to the user interface and in the identification of features to be explained in more depth in a 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 avoid common code between separate critical computations 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 versions and show the absence of common code in the diverse version.

The 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 a review should significantly enhance the ability to extract a given computation for analysis.

(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 (90 percent for this example) were less than 25 percent of the size of the original program. 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 in less than one minute.

V. CURRENT WORK

We are currently developing several enhancements to **unravel**. These include the following:

Interface: Cosmetic changes to the user interface to make source navigation and slicing criterion specification easier.

C Dialects: Some compiler writers extend ANSI C in ways that make it difficult to write general code analysis tools. Some examples from our experience with **unravel** are the following:

- New keywords, e.g., **near**, **far**, **_near**, used as data (declaration) attributes. Usually such keywords could be ignored for computing a program slice.
- New keywords that define new data types, e.g., **bit**, **complex**. These keywords cannot be ignored but, they can be replaced with another keyword that **unravel** recognizes as a type name.
- New syntax cannot in general be recognized. The **unravel** parser is being modified to allow extended syntax that we have encountered.
- One compiler writer decided to extend the C preprocessor to allow the keyword **sizeof** even though the ANSI standard forbids the extension, thus making code using this feature difficult to examine by a software analysis tool without manual replacement of preprocessor statements using **sizeof**. This impacts **unravel** because we want to allow **unravel** to be run on a system other than the system containing the development environment but, we wanted to avoid writing our own version of **cpp**. The practical resolution is to use the native **cpp** of the development environment before taking the source code to the system running **unravel** for analysis.

The solution to recognizing extended keywords is in two parts.

1. Add a run time switch that toggles recognition of common extended keywords.
2. Add a configuration file that contains a list of keywords paired with a (possibly null) substitution.

This takes care of the common extensions and allows the **unravel** user to handle unexpected keyword extensions that have no impact on data-flow semantics.

Libraries: Library or system calls are a mystery to **unravel** since the source code of the library routine is not available for analysis. The current approach is for **unravel** to guess by assuming that if a value is passed it is referenced and if an address is passed then the object is changed. Nothing is assumed about any global variable. We are designing a simple scripting language that would allow an **unravel** user to prepare summary of variable dependence relationships in the library call among input parameters, output parameters and global variables.

Call Tree: We are adding the ability to **unravel** for displaying the slice call tree with an indication of the parts of each called procedure that are included in the slice.

VI. 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.

We have studied the feasibility of extending **unravel** to C++. There are C++ features that would be difficult for **unravel** to analyze. Many of these are inappropriate for high integrity code. **Unravel** can draw attention to the use of these patterns and features.

It is important to note that C++ is a much larger language than C. The major addition to C provided by C++ is the notion of classes, which provide data hiding, guaranteed initialization, user defined conversions, dynamic binding (through virtual functions) and multiple inheritance. Other significant additions include strong type checking, function name overloading, operator overloading, function inlining, constant data objects and member functions, a reference type, heap operators `new` and `delete`, declarations occurring anywhere in a block, templates, and exception handling. These features and their interactions pose new challenges for **unravel**. At the same time, features such as C++'s strong typing will allow **unravel** to produce more accurate slices. Extending **unravel** to analyze C++ programs will require substantial effort.

The following features of C++ present potential difficulties in any analysis of the kind performed by **unravel**. "Being hard to analyze" indicates that these features are hard to understand, which in turn indicates that they should be avoided in high integrity programs. The last two, `unions` and `varargs`, exist in C.

exceptions Exceptions provide an error handling technique or alternatively another control flow structure. Exceptions are difficult to analyze because they represent non-local (often interprocedural) control transfers (gotos). Besides being difficult for **unravel** to analyze, exceptions are new to C++. Their implementation and semantics represent a moving target, not suitable for high integrity code.

pointers to members A pointer to a class member when combined with an object of the class yields a class member. Like pointers to functions, pointers to class members significantly complicate the data-flow analysis that **unravel** must perform.

templates A template provides a generic declaration (of a function or class) that includes a type parameter. By supplying a concrete type for this parameter a concrete declaration is formed. Analyzing templates would add significantly to the complexity of **unravel**.

unions A union, where two or more fields share the same memory, must have a tag field to discriminate through which field the shared memory should be accessed. Unions are taken from C; consequently, like C, C++ provides no checking of union access fields and the present values of the tags. This prevents **unravel** from correctly knowing which field may be accessed.

varargs Varargs allows a function to have an undetermined number of arguments having undetermined type. It is very difficult to track the data-flow through such calls to such functions.

VII. CONCLUSIONS

Unravel is a tool to assist a designer or reviewer to understand 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 high integrity software.

VIII. OBTAINING UNRAVEL

Additional information about **unravel** can be found on the **unravel World Wide Web** home page:
`http://hissa.ncsl.nist.gov/unravel`
Unravel can be obtained by **anonymous ftp** from:
`hissa.ncsl.nist.gov`

References

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

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

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

[4] 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.

[5] 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.

[6] 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 NISTIR 5691, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

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