# Slide Index
Software Quality Week (May 28, 1997):

20 slides
April 18, 1997

# UNIT 1

Unravel

## Unravel

# USING THE UNRAVEL PROGRAM SLICING TOOL TO EVALUATE HIGH INTEGRITY SOFTWARE

James R. Lyle JLYLE@NIST.GOV
Dolores R. Wallace DWALLACE@NIST.GOV

SOFTWARE QUALITY WEEK
San Francisco
May 27-30, 1997

Version

## Talk Outline

- Evaluating High Integrity Software
- Program Slicing
- Results Using Unravel
- Current Work
- Future Work
- Conclusions

## Evaluating High Integrity Software

- Failure of High Integrity Software ⇒ serious accident or severe financial loss ⇒ software needs careful evaluation & review.
- Simpler to compare spec of computation to code if code of other computations are removed.
- Different algorithms used to compute consistency check ⇒ possible point of common failure.
- Need to find software faults (bugs) quickly.
- All of the above are variations on the theme:

**Need to understand an existing program.**

## Program Slicing

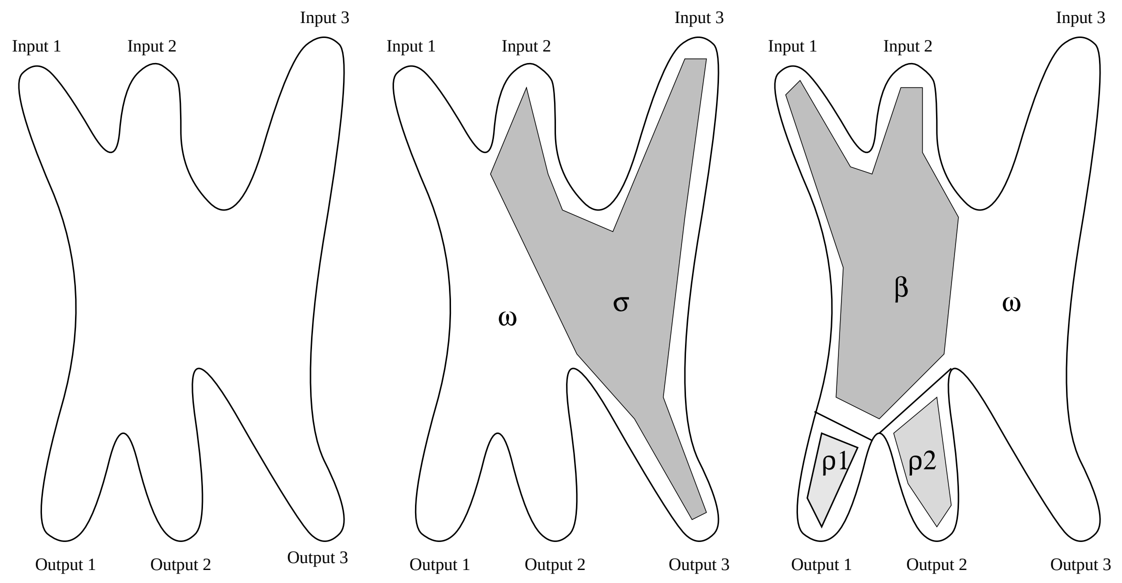
- **Program Slice:** All program statements relevant to a given computation.
- **Slicing Criterion:** Specifies the slice (computation) for a variable, **v**, at a statement, **n**.
- Program slices for a given slicing criterion are obtained from a given program, **P**, by deleting zero or more statements from **P**, but the slice still computes the same value for **v** at statement **n**.
- Data–flow analysis is used to identify statements that may be deleted without affecting the computation.

## Combining Slices

Program slices can be combined to find or exclude common code.

- The intersection of two slices (**backbone slice**) is the set of statements common to the two computations (A major concern in high integrity software).
- A slice minus a backbone slice gives the statements unique to a computation (**program dice**).

# Programmer's View After a Slice



Initial View | After Slice on Output 3 | Relations Between Two Slices

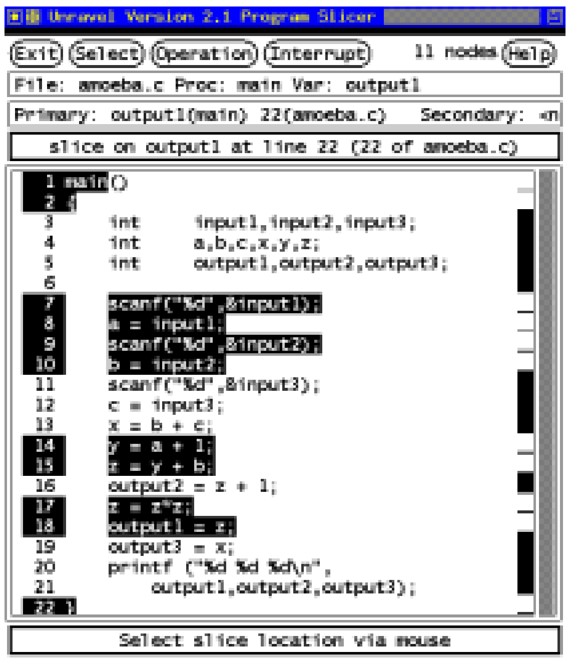
## Computing a Program Slice

To locate the statements that influence variable $v$ just before execution reaches statement $m$ we would compute a program slice for the criterion $< m, v >$.

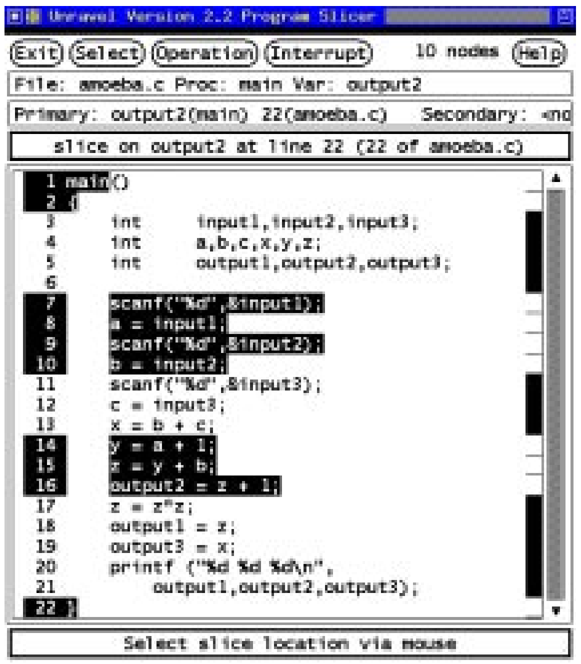
1. Start with program flow-graph
2. Annotate with variables referenced and assigned to
3. the **defs(n)** set and the slicing criterion determine inclusion
4. the **refs(n)** set gives new criterion

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

# Slices on Output 1 and Output 2

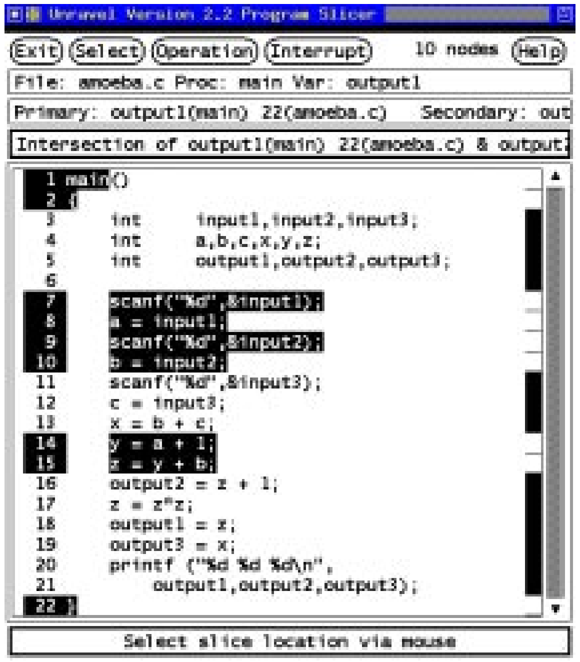


```
File:  amoeba.c Proc:  main Var:  output1
Primary:  output1(main) 22(amoeba.c)    Secondary:  <n
slice on output1 at line 22 (22 of amoeba.c)
  1 main()
  2 {
  3     int     input1,input2,input3;
  4     int     a,b,c,x,y,z;
  5     int     output1,output2,output3;
  6
  7     scanf("%d",&input1);
  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
```



```
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
```

# Combined Slices on Output 1 and Output 2



File: amoeba.c Proc: main Var: output1

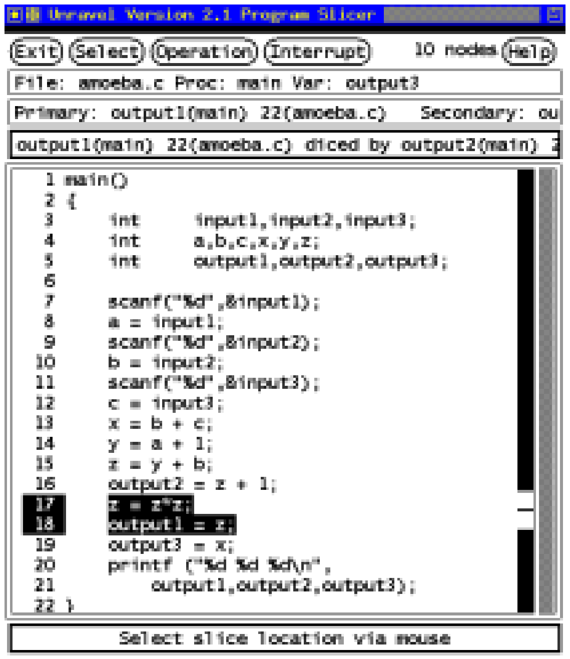
Primary: output1(main) 22(amoeba.c)    Secondary: out

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



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

## Using Unravel on Sample Code

**Unravel**[a] was tried on sample high integrity code, generated by a software reviewer.

- Allowed auditor to extract a computation for manual examination
- Found questionable sharing of code

[a]Development of **unravel** was supported under interagency agreement DNRO46115 between the Department of Defense and NIST. Currently **unravel** is supported by NIST and the NRC. Development of the **unravel** tool is being supported under interagency agreement RES-92-005 between the U.S. Nuclear Regulatory Commission and NIST. The opinions, findings, conclusions and recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of the NRC.

## Unravel Performance

(1) **Unravel** significantly enhances ability to analyze code.

(2) **Unravel** is easy to operate.

(3) **Unravel** can disclose subtle relationships in 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 (some as small as 10 percent of the original).

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

## Current Work

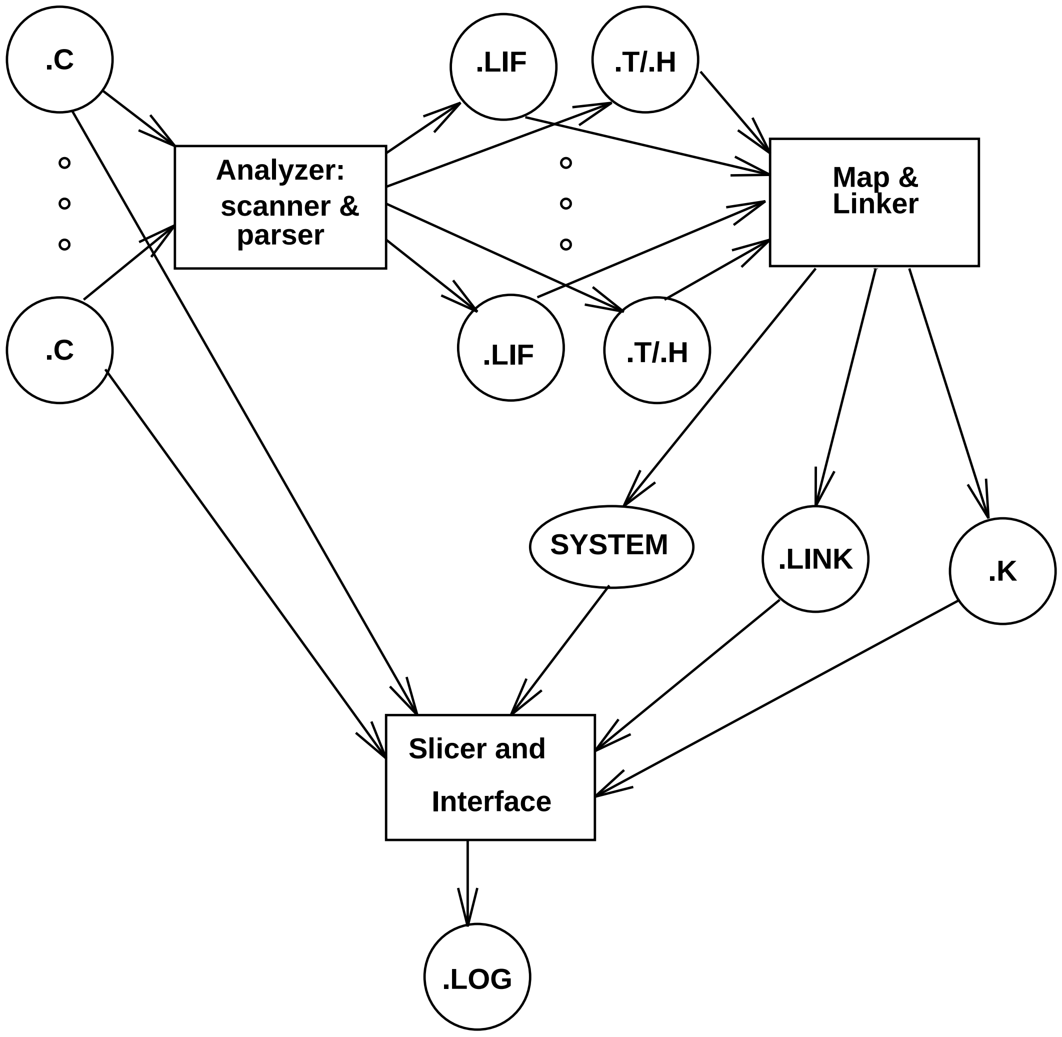
- Make **unravel** run faster. Current version uses simple algorithms that can be easily improved.
- Cosmetic user interface changes. E.g., Selecting variables: now global variables presented by file and locals presented by procedure. Add list of all variables.
- Parsing C dialects complicated by new keywords or syntax.
- No semantic knowledge for libraries.
- Add display of call tree with slice information.

## Call Tree Example 1

## Call Tree Example 2

## Call Tree Example 3

# Basic Unravel Design



## Future Work

- **Unravel** is based on a FORTRAN slicing tool. We have been requested to update the FORTRAN component and integrate it into **unravel**.
- Add Java and C++.

## C++ Parsing

C++ is hard for many reasons, some that stand out to us:

- **Exceptions** are hard to slice and used more often in C++ than **signals** in ANSI C.
- Templates must be expanded. In C we used the cpp to expand macros.
- Pointer usage is much richer in C++ than C, e.g., pointers to functions more common.

## Conclusions

In an initial evaluation an independent software reviewer found:

- **Unravel** easy to use with little training.
- Simplified the task of extracting a computation for evaluation.
- Slices were much smaller than the original program.
- Slices were found quickly.

Down load by anonymous ftp from:
HISSA.NCSL.NIST.GOV

Web page: HTTP:HISSA.NCSL.NIST.GOV/UNRAVEL