NISTIR 5691

Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 2: User Manual

James R. Lyle
Dolores R. Wallace
James R. Graham
Keith B. Gallagher
Joseph P. Poole
David W. Binkley

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

NIST

NISTIR 5691

Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 2: User Manual

James R. Lyle
Dolores R. Wallace
James R. Graham
Keith B. Gallagher
Joseph P. Poole
David W. Binkley

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

August 1995

Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 2

Abstract

This is the second volume of a two volume report on **unravel**, a Computer Aided Software Engineering (CASE) tool for software written in ANSI C, that can assist evaluation of high integrity software by using program slices to extract computations for examination. In this volume, we provide a user manual for **unravel**. This manual is intended to provide the user with enough information to use **unravel** without any other reference. To this end, a brief simplified description of program slicing is provided in addition to a tutorial example and a detailed description of **unravel** operation. This user manual also discusses limitations of **unravel** and how to deal with code containing extensions to ANSI C that would inhibit the correct operation of **unravel**.

Trademarks

Contents

List of Tables

List of Figures

1 Introduction

This volume describes the operation of version 2.1 of the program slicing tool, **unravel**, developed by the National Institute of Standards and Technology (NIST). Development of **unravel** was funded by both the United States Nuclear Regulatory Commission (NRC) and the National Communications System (NCS) under contracts RES-92-005, FIN #L24803, and DNRO46115, respectively. The tool can be used to compute program slices of programs written in ANSI C.

Program slicing 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 aid program debugging, program maintenance, program understanding, and automatic integration of program variants.

The users of **unravel** are assumed to have a working knowledge of computers and ANSI C, but they may not be familiar with UNIX, POSIX[†] or program slicing.

Unravel is intended to support the understanding and evaluation of software by allowing the user to investigate a program through program slices.

To achieve the goal of making **unravel** a portable and easy to use slicing tool, the following general requirements were met:

- The user must be able to execute **unravel** with minimal knowledge of the platform on which it resides.
- The user must be able to interactively specify criteria for computing program slices.
- The user must be able to view program slices on-screen.
- The user must be able to perform logical set operations (e.g., intersections) on program slices.
- The user must be able to use **unravel** without needing to understand the intrinsics of the program; hence a user manual and user interface must contain all operational information.
- The implementation must comply with standards for operating system interface POSIX, ANSI C, and the X Window System.

[†]POSIX, Portable Operating Systems, FIPS 151-2 (ISO/IEC 9945-1).

Section 2 explains program slicing and gives an example of using **unravel** to examine a small program. Details of **unravel** operation are given in section 3. Section 4 discusses using **unravel** on non-ANSI C programs. The limitations of **unravel** and assumptions about source code to be analyzed are discussed in section 5.

2 Unravel Description

Unravel is a tool for program understanding that uses program slicing to identify statements relevant to some computation. This section describes program slicing, the software architecture of **unravel**, a scenario for using **unravel**, and presents a detailed example of using **unravel**. The description of program slicing is a simplified discussion of the theory followed by an example illustrating the steps in computing a program slice.

Some terminology about programs and program components needs to be clearly defined so that the reader understands how **unravel** views an ANSI C program. It is important to understand that **unravel** sees a program as a collection of procedures executed as a unit. A program has one main procedure (called **main**) and some number of other procedures that are arbitrarily grouped in one or more source files.

Source Program Statement: A statement written in a programming language. Statements may be *declarations*, that define data types and variables, or *executable*, that specify some action.

Procedure: A named set of source program statements, possibly with parameters, that performs some action. Functions and subroutines are considered procedures.

Source Program File: A file containing source program statements for zero or more procedures and declarations. A source program file is also called a *module*.

C Preprocessor: A standard part of any ANSI C compiler used to process *preprocessor directives* identified by a pound sign (#) as the first non-blank character on a line. Preprocessor directives are used to define constants, insert include files, select statements for conditional compilation and define source macros.

Include File: A source file, specified by a **`#include`** directive, that the C preprocessor inserts in the compilation of another source file. By convention, an include file name should end in `.h` and should contain only declarations and C preprocessor directives; it should not contain procedures or parts of procedures.

Program: A main procedure and the set of procedures called transitively by the main procedure.

Source Program: The set of source program files and include files that define all the procedures of a program. When a source program is compiled and linked, an executable program is produced.

2.1 Program Slicing

Program slicing is a family of program decomposition techniques based on extracting statements relevant to a computation in a program. A slice is a smaller program that reproduces a subset of the original program's behavior. This is advantageous since the slice can, by excluding irrelevant statements, collect an algorithm for a given calculation that may be scattered throughout a program. It is easier for a programmer interested in a subset of the program's behavior to understand the corresponding slice than to deal with the entire program. The utility and power of program slicing comes from the potential automation of tedious and error prone tasks. Research on program slicing is being conducted on program debugging, program testing, program integration, parallel program execution and software maintenance. Several variations on this theme have been developed, including *program dicing*, *dynamic slicing* and *decomposition slicing*.

An informal definition of a *program slice* (taken at statement *n* on variable *v*) is all statements that might affect the value that *v* has just before control reaches statement *n*. A *slicing criterion* defines the starting place for a program slice. It is composed of the statement *n* and the variable *v*. Statements are included in a program slice with either a direct influence or an indirect influence on the criterion variable. A statement assigning a value to a variable that could influence the value of the criterion variable at the criterion statement is a direct influence. A compound statement (e.g., **for** or **while**) that controls execution of another statement included in the slice is an indirect influence.

This is not intended to be a complete discussion of program slicing but, an overview to give the user insight into the behavior of **unravel**. Some language features such as pointers and procedure calls introduce complications that would require lengthy discussion.

2.1.1 Creating a Flow-Graph

To compute program slices, a program is first represented as a flow-graph of nodes annotated with lists of variables assigned a value or used at each node and directed edges indicating control-flow. A flow-graph node roughly corresponds to an executable statement, however some statements such as a **for** statement are divided into several nodes to represent the various parts of the **for** statement that do control variable initialization, loop termination testing and control variable modification. The edges of the flow-graph connect each node, *n*, to the nodes (statements) that could be executed after *n*. An example program is presented in Figure 2-1 with the corresponding flow-graph in Figure 2-2 and the node number to line number mapping in Figure 2-3. To understand the basics of transforming a program into a flow-graph here are the rules to follow for programs composed only of assignment statements, **if** statements and **while** statements. The flow-graph is composed of nodes that represent the statements and edges that represent execution flow between pairs of statements.

- Each statement is represented by one or more flow-graph nodes. If the statement generates more than one node then one node will be designated the *entry node* and one node designated the *exit* node. The entry and exit nodes are used to connect directed edges to the entry and exit nodes of other statements.

- An assignment statement is represented by a single node. The node functions as both entry and exit. An edge connects the node to the entry node of the next statement. An edge connects the exit node of the previous statement to the assignment statement node.

- In addition to the nodes representing the body, an **if** statement without an **else** part is represented by two nodes; an **if** statement with an **else** needs three nodes. The entry node represents the **if** and *conditional expression* and is connected to the entry node of the statement in the *then part* of the **if**. The exit node of the then part is connected to a separate exit node created for the **if** statement. If there is an else part, the **if** statement entry node connects to a third node that represents the **else** keyword. The else node connects to the entry node of the statement in the else part. The exit node of the else part connects to the **if** statement exit node. The exit node of the **if** statement connects to the entry node of the next statement. The exit node of the previous statement connects to the **if** statement entry node.

- A **while** statement is represented by two nodes, an entry node representing the **while** *(condition)* and an exit node. The entry node is connected to the entry node of the loop body and to the exit node of the **while**. The exit node of the loop body is connected to the entry node of the **while**. The exit node of the previous statement connects to the **while** statement entry node.

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

    red = 1;
    blue = 5;
    green = 8;
    yellow = 2;

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

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

Figure 2-1: Slicing Example 1 flavors.c

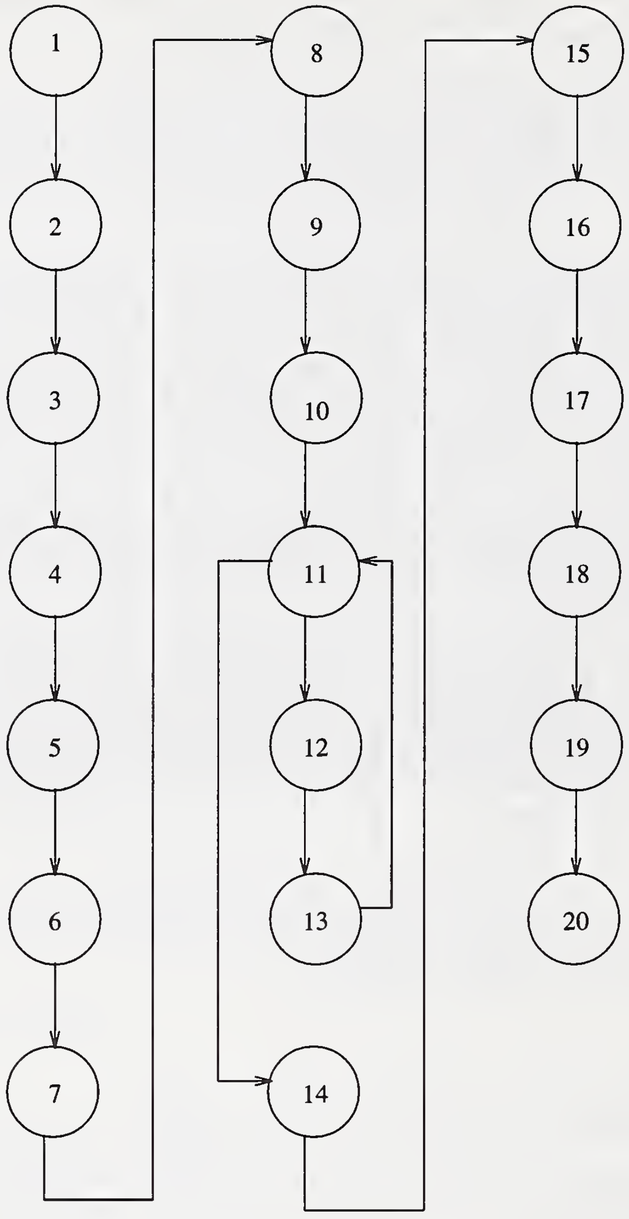


Figure 2-2: Unravel Slicing Example 1 Flow Graph

2.1.2 Computing Program Slices

A program slice is computed for a given slicing criterion from an annotated flow-graph. These annotations include variables referenced and defined at each flow-graph node and the active variable set. The active set is the set of variables that the criterion variable depends on just before program execution reaches the associated node.

A program can be represented by a flow-graph annotated by variables referenced and defined at each flow-graph node. A program slice can be computed on a program for a given slicing criterion with the help of one more annotation to the flow-graph, called the *active variable set* or *active set*. The active set is the set of variables that the criterion variable depends on just before program execution reaches the associated node.

The computation begins with all the active sets except for the active set for the slicing criterion statement initialized to the empty set. The active set for the criterion statement is initialized to the criterion variable. The slice is computed by propagating active sets across flow-graph nodes until the active sets stabilize (i.e., stop changing). Computation of the active set for an arbitrary node, *n*, is controlled by comparing variables defined at node *n* with the active sets of immediate successor nodes by the following rules:

1. If none of the immediate successor active sets contain a variable defined at node *n*, then node *n* is not in the slice unless it is part of a compound statement controlling execution of other statements included in the slice. The active set of node *n* is the union of the immediate successor active sets.

2. If node *n* assigns a value to a variable that is a member of the active set of some immediate successor of *n*, then node *n* is included in the slice. The active set for node *n* is the set of variables referenced at node *n* unioned with the union of immediate successor active sets without the variable assigned.

When a statement is included in a slice that is part of a compound statement such as **if** or **while**, the framework of the compound statement is included in the slice and any variables used in a controlling condition are added to the active set of the compound statement.

To summarize the algorithm for computing a slice:

1. Create the flow-graph.

2. Select slicing criterion.

3. Set active set for criterion location to the criterion variable.

4. Propagate active sets over the entire flow-graph until no changes occur.

5. A statement is included in the slice if it changes the active set or is a compound statement containing a statement in the slice.

2.1.3 Slicing Example

Figure 2-3 presents the data-flow sets annotating the flow-graph from Figure 2-2 used in computing program slices on the program of Figure 2-1. The columns labeled *Node* and *Succ* (successor) represent the flow-graph of the program. The columns labeled *Refs* and *Defs* contain the variables referenced and assigned-to (defined) at each node. The column labeled *Req* is the *required set* for the node, used to list other nodes that are required by any slice containing the given node. The required set is used to capture both syntactic relations such as the inclusion of the opening and closing program braces (nodes 2 and 20), and control relations of **if**, **while** and other compound statements (e.g., node 12 at line 17 requires node 11, the **while** on line 16).

For example, suppose we want to know how the value of the variable **sweet** printed at line 25 of Figure 2-3 is computed. The specification of a slicing criterion requires a variable and a node in the flow-graph. Node 18 corresponds to the **printf** statement at line 25, therefore, the criterion would be $S_{<18,sweet>}$. The active set for node 18 is initially {sweet}. Since nodes 9 through 18 do not assign a value to **sweet** no changes take place to the active set as it is propagated backward, and these nodes are not included in the slice. Node 8 assigns a value to **sweet** based on **red** and **green** and so node 8 (line 13) is included in the slice and **sweet** is replaced in the active set by **red** and **green** at node 8. Node 7 is included in the slice because the active variable **red** is assigned a value, however the active set does not really change with **red** being replaced by **red**. At node 3, **red** is assigned a constant value and the node is added to the slice and **red** is dropped from the active set. Node 5 is included in the slice and **green** is dropped from the active set since **green** is assigned a constant value at node 5. The active sets are summarized in Figure 2-4. The slice is now complete except for some syntactic dependencies (nodes 1, 2 and 20) that are captured by the *requires set.*

Line	Statement	Node	Succ	Req	Defs	Refs
1	main()	1	2	--	--	--
2	{	2	3	1,20	--	--
7	red = 1;	3	4	2	red	--
8	blue = 5;	4	5	2	blue	--
9	green = 8;	5	6	2	green	--
10	yellow = 2;	6	7	2	yellow	--
12	red = 2*red;	7	8	2	red	red
13	sweet = red*green	8	9	2	sweet	red,green
14	sour = 0;	9	10	2	sour	--
15	i = 0;	10	11	2	i	--
16	while (i < red) {	11	12,14	2,14	--	i,red
17	sour = sour + green;	12	13	11	sour	sour,green
18	i = i + 1;	13	11	11	i	i
19	}	14	15	--	--	--
20	salty = blue + yellow;	15	16	2	salty	blue,yellow
21	yellow = sour + 1;	16	17	2	yellow	sour
22	bitter = yellow + green;	17	18	2	bitter	yellow,green
24	printf ("%d %d %d %d\n",	18	19	2	--	sweet,sour
25	sweet,sour,salty,bitter);					salty,bitter
26	exit(0);	19	--	--	--	--
27	}	20			--	--

Figure 2-3: Slicing Example 1 Data-Flow Sets

Line	Statement	Active	Node
1	main()		1
2	{		2
7	red = 1;		3
8	blue = 5;	**red**	4
9	green = 8;	**red**	5
10	yellow = 2;	**red,green**	6
12	red = 2*red;	**red,green**	7
13	sweet = red*green;	**red,green**	8
14	sour = 0;	**sweet**	9
15	i = 0;	**sweet**	10
16	while (i < red) {	**sweet**	11
17	sour = sour + green;	**sweet**	12
18	i = i + 1;	**sweet**	13
19	}	**sweet**	14
20	salty = blue + yellow;	**sweet**	15
21	yellow = sour + 1;	**sweet**	16
22	bitter = yellow + green;	**sweet**	17
24 25	printf ("%d %d %d %d\n", sweet,sour,salty,bitter);	**sweet**	18
26	exit(0);		19
27	}		20

Figure 2-4: Slicing Example 1 Active Sets For $S_{<18,sweet>}$

2.2 Unravel Architecture

This section describes the software architecture of **unravel**. **Unravel** operates on the source files from a single directory. Figure 2-5 presents an overview of **unravel**. The circles represent files and the boxes represent processing steps. Source files are transformed to corresponding *language independent format* (LIF) files by the analyzer. The LIF files for a given program are bound together by the *linker* into a single *link* file. The link file is the primary input to the slicer. To use **unravel**, the source program files must first be analyzed and linked. A window based user interface gives the user of **unravel** access to these components and helps manage the results.

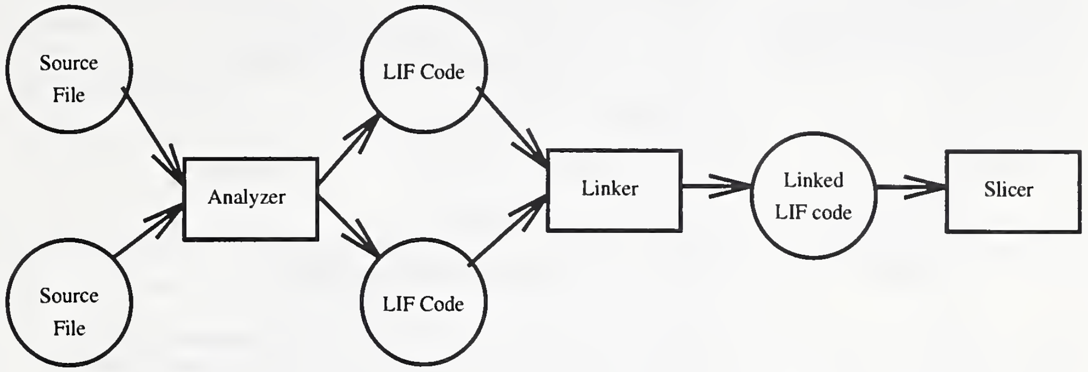


Figure 2-5: Unravel Structure Overview

2.3 Operational Scenario

This section is a tutorial talking a user through an example of using **unravel**. The two most important steps for using **unravel** successfully take place before **unravel** is executed. The user should have a general understanding of the architecture of the source code in order to select slicing criteria that can provide relevant information. Figure 2-6 presents an operational checklist that serves as an overview and guide to **unravel** operation. The figure is repeated in Appendix A for quick reference. The steps to using **unravel** follow.

1. Receive orientation to source code. Determine all the source files and **`#include`** files that make up the program. The **unravel** user should have a general idea about global variables, procedures and program structure in the source code.

2. Select slicing criteria. The **unravel** user should develop a list of questions about the source code that can be answered by program slicing. For each planned slice, the user should note the file name and line number where the slice should be computed along with the file name, where the variable of a slicing criterion is declared. For local variables, the procedure where the local variable is declared is needed for specifying the slicing criterion.

3. Make the directory containing the program to analyze the current directory.

4. Execute the command: **unravel**. The **unravel** command displays a control panel called the **Main Control Panel**.

1 Orientation	Examine program structure
2 Select Slices	Plan slices to compute
3 cd to source directory	Make source code directory current
4 **Unravel**	Execute **unravel**
5 Analyzer	Click on **Run Analyzer**
Select Files	Select files to analyze
Analyze	Click on **Analyze Selected Files**
Exit	Click on **Exit** to return to **Main Control Panel**
6 Review Results	Examine **Last Analysis** in **Review History** menu
Navigate	Use scroll bar to navigate
Done	Click on **Done** to exit
7 Slicer	Click on **Run Slicer**
Select Variable	Use **Select** menu to pick variable
Select Location	Click on criterion location to do slice
Navigate	Use scroll bar to examine source code
Operations	Use **Operations** menu to combine slices
Exit	Click on **Exit** to return to **Main Control Panel**
8 Exit	Click on **Exit** to stop **unravel**

Figure 2-6: Unravel Operation Checklist

5. Click on the **Run Analyzer** button from the **Main Control Panel**. This displays the **Analyzer Control Panel**. Select source files to analyze, then click on the **Analyze Selected Files** button to analyze each source file in turn. A message is displayed if any source files are not ANSI C. Click on the **Exit Analyzer** button to return to the **Main Control Panel**.

6. Review analysis results. The **Last Analysis** entry of the **Review History** menu of the **Main Control Panel** pops-up a summary of each analysis. If any source files are not ANSI C, an error message identifies the problem location. The source file needs to be brought into conformance with ANSI C by changing the source file and running the analyzer again.

7. Click on the **Run Slicer** button. If there is more than one program the **Selection Control Panel** is displayed; click on the desired program to start the slicer on the selected program. If there is only one main program, the **Selection Control Panel** is skipped and the slicer starts on that program. The user selects slicing criteria and displays the slices. When the user is finished computing slices, click on the **Exit** button to return to the **Main Control Panel**.

2.4 Tutorial Example

This section describes using **unravel** on a short tutorial example. We will go through the steps to run the **analyzer** and the **slicer**.

2.4.1 Orientation

The example source code in Figures 2-7 through 2-11 and in Figure 2-1 is provided with the **unravel** distribution. The source code is located in the **example** subdirectory of the **unravel** distribution. There are two example programs, **flavors** and **refinery**. The program **flavors** is completely contained in the file **flavors.c**. The other program, **refinery**, is spread over five files: **refinery.c**, **refinery.h**, **input.c**, **cool.c** and **pressure.c**.

```
typedef struct {
    int value;
    } sensor_rec,*sensor_ptr;

int     pressure,flow,volt,level;
int     pump_ok,flow_ok;
int     presure_ok,level_ok;
```

Figure 2-7: Slicing Example 2 refinery.h

```
# include "refinery.h"
get_sensor_v (sensor_ptr s) { s->value = read_sensor(); }
get_sensor_f (sensor_ptr s) { s->value = read_sensor(); }
get_sensor_l (sensor_ptr s) { s->value = read_sensor(); }
get_sensor_p (sensor_ptr s) { s->value = read_sensor(); }
```

Figure 2-8: Slicing Example 2 input.c

```
# include "refinery.h"
int coolant_sys(p,f)
    sensor_ptr  p,f;
{

    volt = p->value;
    flow = f->value;
    pump_ok = pump_undervolt(volt);
    flow_ok = coolant_flow(flow);
}
```

Figure 2-9: Slicing Example 2 cool.c

```
# include "refinery.h"
int pressure_sys(p,w)
    sensor_ptr  p,w;
{

    pressure = p->value;
    level = w->value;
    pressure_ok = check_pressure(pressure);
    level_ok = water_level(level);
}
```

Figure 2-10: Slicing Example 2 pressure.c

```
# include "refinery.h"
main()
{
    int p_alarm,c_alarm,alarm;
    sensor_rec  pump_sensor,flow_sensor;
    sensor_rec  pressure_sensor,level_sensor;

    while (1){

        get_sensor_v(&pump_sensor);
        get_sensor_f(&flow_sensor);
        get_sensor_p(&pressure_sensor);
        get_sensor_l(&level_sensor);

        pressure_sys(&pressure_sensor,&level_sensor);
        coolant_sys(&pump_sensor,&flow_sensor);
        p_alarm = !(pressure_ok && level_ok);
        c_alarm = !(pump_ok && flow_ok);
        alarm = c_alarm || p_alarm;
        if (alarm) system_shutdown();
    }
}
```

Figure 2-11: Slicing Example 2 refinery.c

2.4.2 Slice Criteria Selection

For the program **flavors** we would like to examine the computation of **sweet** and **sour** and evaluate any code common to these two computations. For the program **refinery** we would like to examine the computation of **pump_ok**, **level_ok**, and **a_dump**.

After due consideration, the slices in Table 2-1 were selected.

Slicing Criteria					
Variable			Location		
File	Procedure	Name	File	Procedure	Line No.
refinery.h	*global*	pump_ok	cool.c	main	10
refinery.h	*global*	level_ok	cool.c	main	10
refinery.c	main	a_dump	refinery.c	main	22
flavors.c	main	sweet	flavors.c	main	25
flavors.c	main	sour	flavors.c	main	25

Table 2-1: Planned Slices

2.4.3 Setting Source Directory

The user should locate the source directory for this example, then use the UNIX **cd** command to set the current directory.

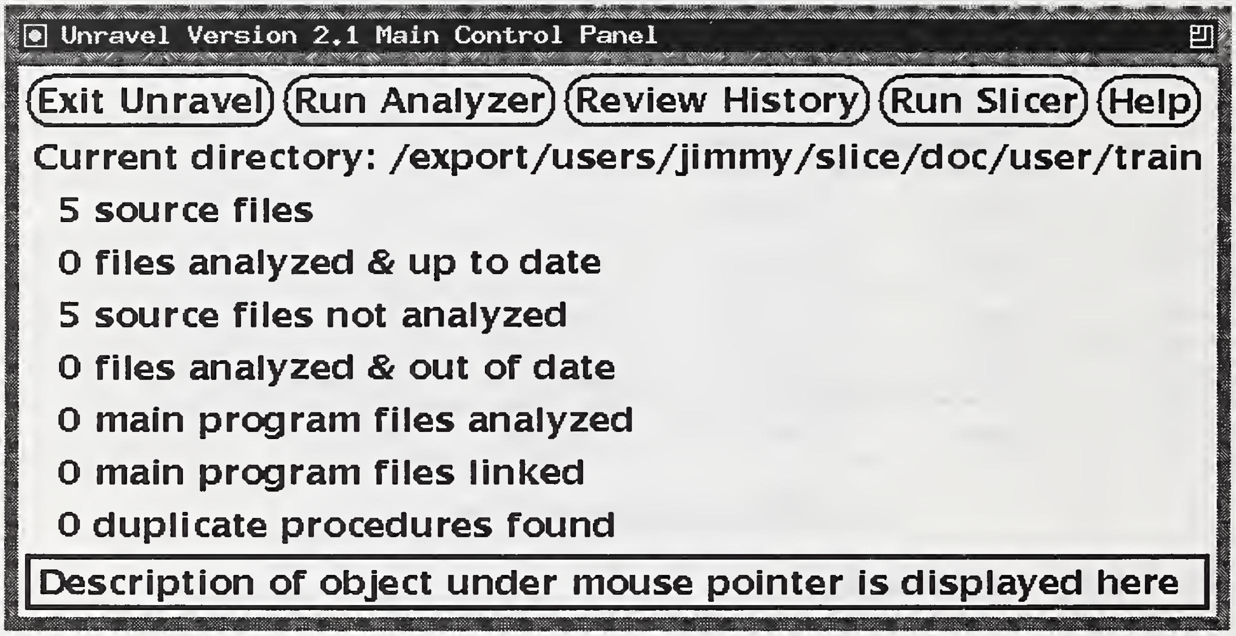
2.4.4 Executing Unravel

Figure 2-13 displays the initial **Main Control Panel** on the **example** directory. Note the following:

- Five source files are identified.
- None of the files have been analyzed or linked. Even though there are two **main** programs in the directory, the programs have not been analyzed yet by **unravel**.
- At this point the user should run the analyzer. After the analysis is finished, then the slicer can be run.
- To run the analyzer, the user moves the mouse cursor to the button labeled **Run Analyzer** and clicks the left mouse button. The **Analyzer Control Panel** should appear in a few seconds.

2.4.5 Executing the Analyzer

The **analyzer** examines the current directory for source program files and automatically places the file on either a list of files to be analyzed or a list to be ignored. A source file that has been analyzed since the last change to the source file is placed on the **Files Not Selected** list; a source file that has not been analyzed or has been changed since its last analysis is placed on the **Selected Files** list.



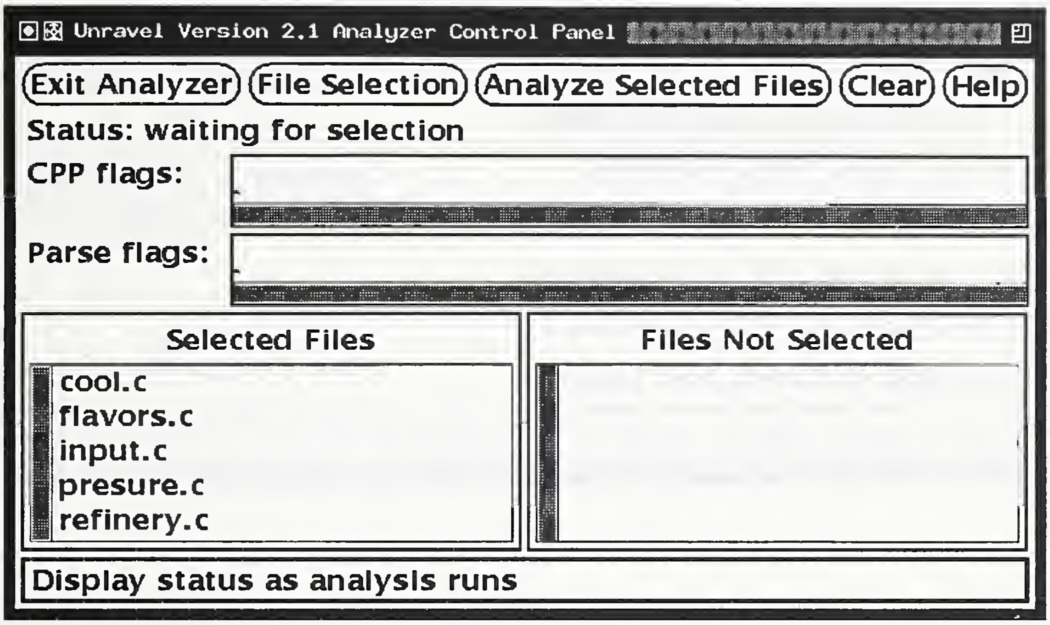
```
Current directory: /export/users/jimmy/slice/doc/user/train
  5 source files
  0 files analyzed & up to date
  5 source files not analyzed
  0 files analyzed & out of date
  0 main program files analyzed
  0 main program files linked
  0 duplicate procedures found
Description of object under mouse pointer is displayed here
```

Figure 2-13: Initial Main Control Panel

Figure 2-14 displays the initial **Analyzer Control Panel**. Note the following:

- The five files are already selected since they have not been analyzed. All files need to be analyzed so the user should click on the **Analyze Selected Files** button.
- When analysis starts, the **Analyze Selected Files** button changes its label to **Stop Analysis** as in Figure 2-15. If the user clicks the left mouse button on **Stop Analysis** the analysis stops after the current program analysis is finished.
- While the analysis is being performed, the status line indicates which program is being analyzed and how many programs are selected for analysis.
- The status line indicates if any source programs were not ANSI C and could therefor not be analyzed. This is usually caused by a missing include file or a non-ANSI language.

- After the analysis is complete, the user clicks on **Exit Analysis** to return to the **main control panel**. Figure 2-16 displays the **Main Control Panel** after the analysis is complete.



Status: waiting for selection

CPP flags:

Parse flags:

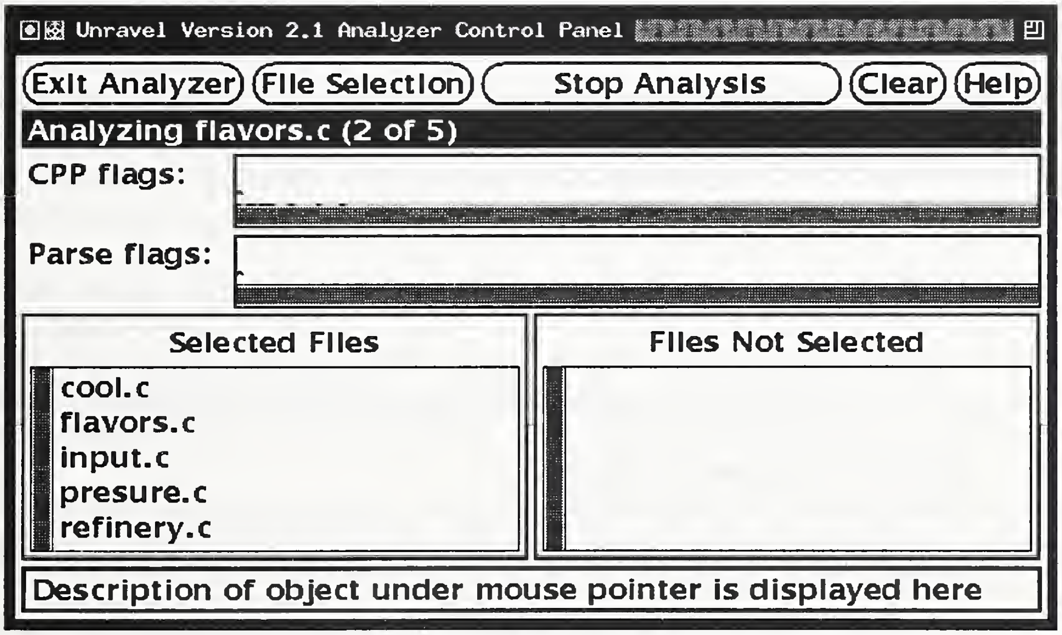
Selected Files

cool.c
flavors.c
input.c
presure.c
refinery.c

Files Not Selected

Display status as analysis runs

Figure 2-14: Initial Analyzer Control Panel



Analyzing flavors.c (2 of 5)

CPP flags:

Parse flags:

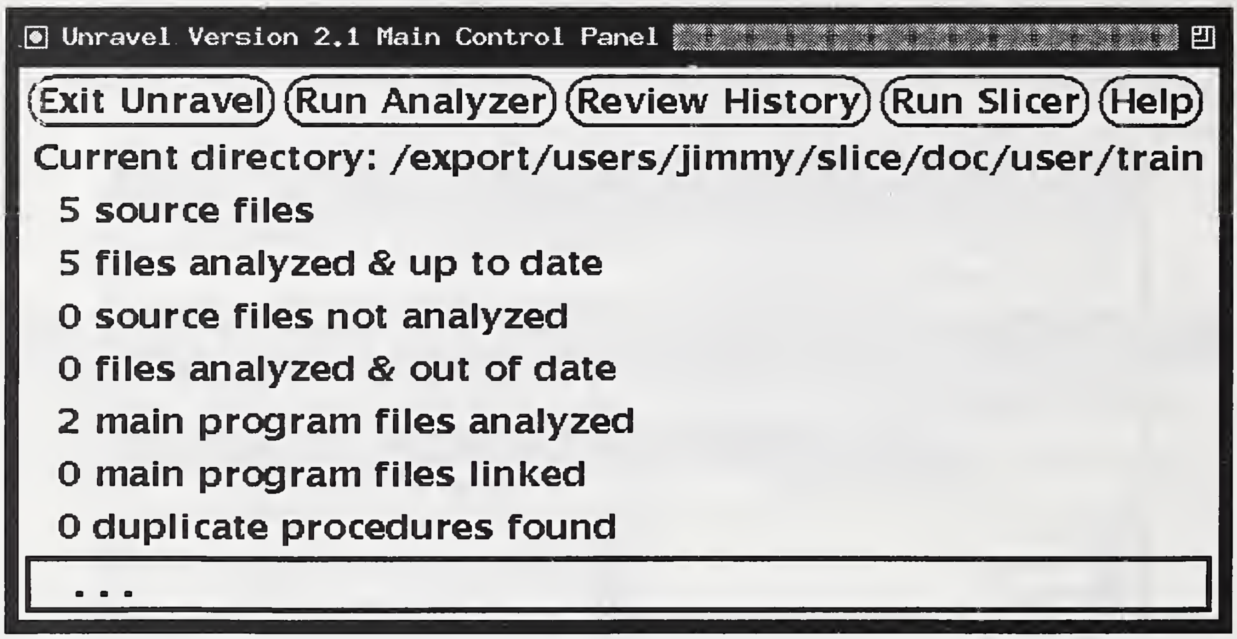
Selected Files

cool.c
flavors.c
input.c
presure.c
refinery.c

Files Not Selected

Description of object under mouse pointer is displayed here

Figure 2-15: Analyzer Control Panel During Analysis



```
Current directory: /export/users/jimmy/slice/doc/user/train
  5 source files
  5 files analyzed & up to date
  0 source files not analyzed
  0 files analyzed & out of date
  2 main program files analyzed
  0 main program files linked
  0 duplicate procedures found
  ...
```

Figure 2-16: Main Control Panel After Analysis

2.4.6 Analysis Review

After the analysis is done the user should return to the **Main Control Panel** and review the analysis results. To display the analysis results select the **Last Analysis** entry on the **Review History** menu.

A summary of each source file analyzed is presented. If any syntax errors were found (i.e., the source is not ANSI C), the location of the error in the source file is indicated. The error should be repaired and the analyzer run again until there are no errors. Section 5.1 discusses what to do with some common extensions to ANSI C.

2.4.7 Using the Slicer

This section describes invoking and using the slicer. This is a two step process; first, the program to be sliced is selected, then the slicing criteria is specified and the slices examined.

2.4.7.1 Selecting a Program to Slice

After the analysis is complete and any syntax errors have been repaired, the user runs the slicer by clicking on the **Run Slicer** button of the **Main Control Panel**. If there is more than one program to choose from, the **Selection Control Panel** is displayed as in Figure 2-17. The user should either click on a program file name or click on the **Exit** button. If the user clicks on the **Exit** button, the **Selection Control Panel** pops-down and control returns to the **main panel**. To select the program **refinery.c** the user clicks on that program name.

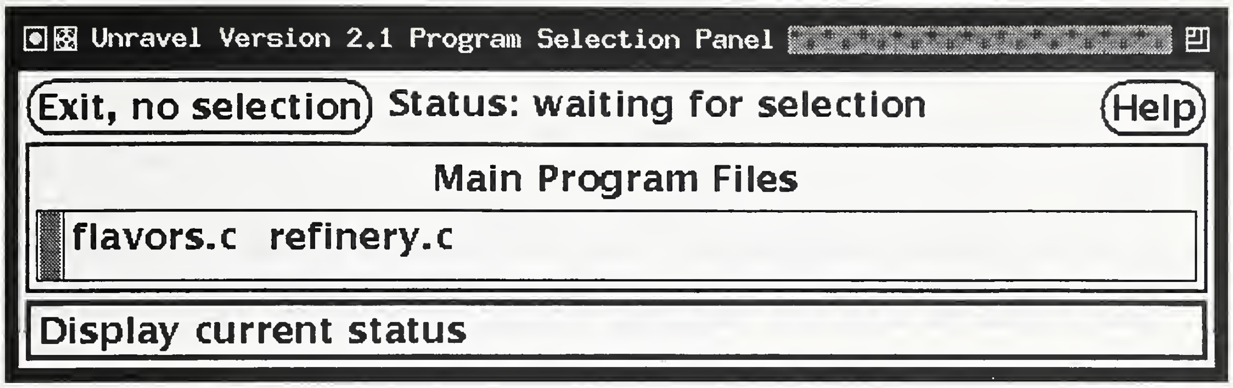


Figure 2-17: Selection Control Panel

2.4.7.2 Executing the Slicer

After a program is selected from the **Selection Control Panel**, the **Slice Control Panel** pops-up. The user can do the following with the **Slice Control Panel**:

1. Select a variable for a slicing criterion.
2. Select a statement location for a slicing criterion.
3. Navigate through the source file to examine the results of the slice.
4. Select an operation for combining two slices.

To compute the first slice from our list of planned slices in Table 2.1, the user needs to select the global variable **pump_ok**. To select **pump_ok** the user does the following:

1. Move the mouse pointer over the **Select** button.
2. Push and hold the left mouse button. A menu with several choices should appear under the **Select** button.
3. Drag the mouse down to the choice labeled **Global Variable**. The text should highlight in reverse video.
4. Release the left mouse button. A pop-up list with two file names should appear. This is a list of files where global variables are declared.
5. Click the left mouse button on the entry **refinery.h**. A second pop-up list of the global variables declared in the selected file should appear.

6. Click the left mouse button on **pump_ok**. Note that the second line on the display contains the currently selected criterion variable.

To specify the location for the slicing criterion and initiate computing the slice do the following:

1. Find the location for the slice, line 10 in file **cool.c**, by moving the scroll bar until line 10 of the file **cool.c** comes into view. To move the scroll bar, position the mouse cursor in the scroll bar, push and hold the middle mouse button, then drag the mouse up and down to move the source text. When line 10 comes into view release the mouse button.

2. Move the mouse pointer to line 10 and click the left mouse button. This starts computing the slice. The fourth line of the window should change color to indicate that a slice is in progress. When the slice is finished, the color returns to normal.

3. Statements in the slice are highlighted in reverse video. Along the right edge of the source text display window there is a vertical stripe called a *tick bar* that gives a visual summary over the entire program of statements in the slice. Figure 2-18 illustrates the results on the screen.

Figure 2-19 presents the results of slicing on **level_ok** at the same location. Note from the figure that this slice is labeled *secondary slice* while the other slice is labeled *primary slice*. The terms *primary* and *secondary* are used to label two slices for computing set operations on the slices. To compute a secondary slice, the user clicks the criterion location with the middle mouse button rather than the left mouse button.

Now that we have two slices we can intersect slices to determine if there is shared code. To compute the intersection displayed in Figure 2-20, the user moves the mouse cursor to the **Operation** button, pushes and holds the left mouse button, drags the mouse to the **Intersection** entry and releases.

The slice on **a_dump** can be computed in similar fashion:

1. Select criterion variable (**a_dump**) by dragging the left mouse button down the *Select* button menu to the **Local Variable** entry and then release the left mouse button.

2. To display the list of variables defined in **main**, select the **main** procedure in the pop-up list of procedure names by clicking the left mouse button while the mouse cursor is over **main**.

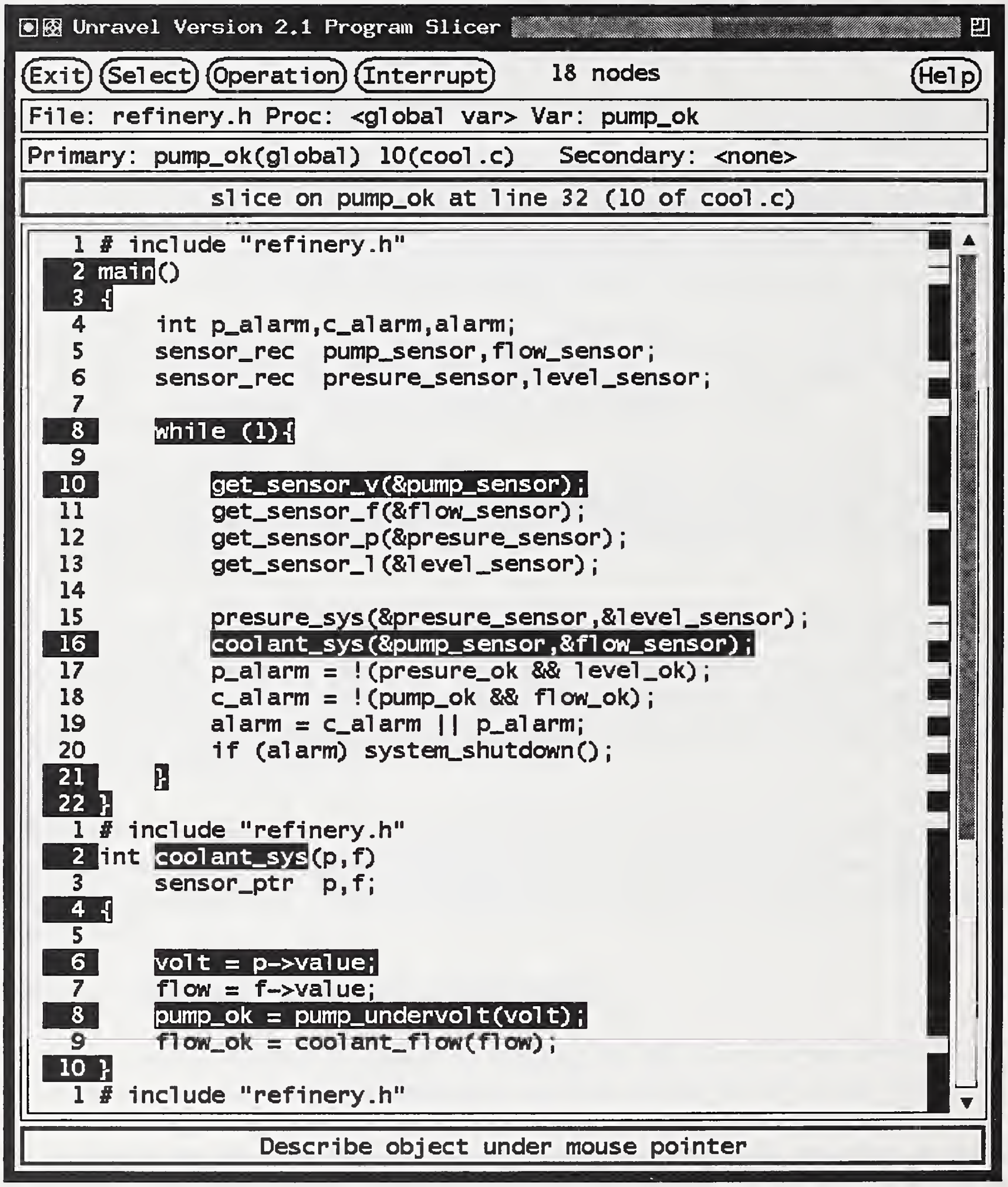
3. When the pop-up list of local variables declared in **main** appears select the variable **a_dump** by clicking the left mouse button over **a_dump**.

4. Move the mouse pointer over line 22 and click the left mouse button to compute a slice labeled *primary*, or click the middle mouse button to compute a slice labeled *secondary*.

When the user is finished computing slices, the user clicks on **Exit** in the **Slice Control Panel**.

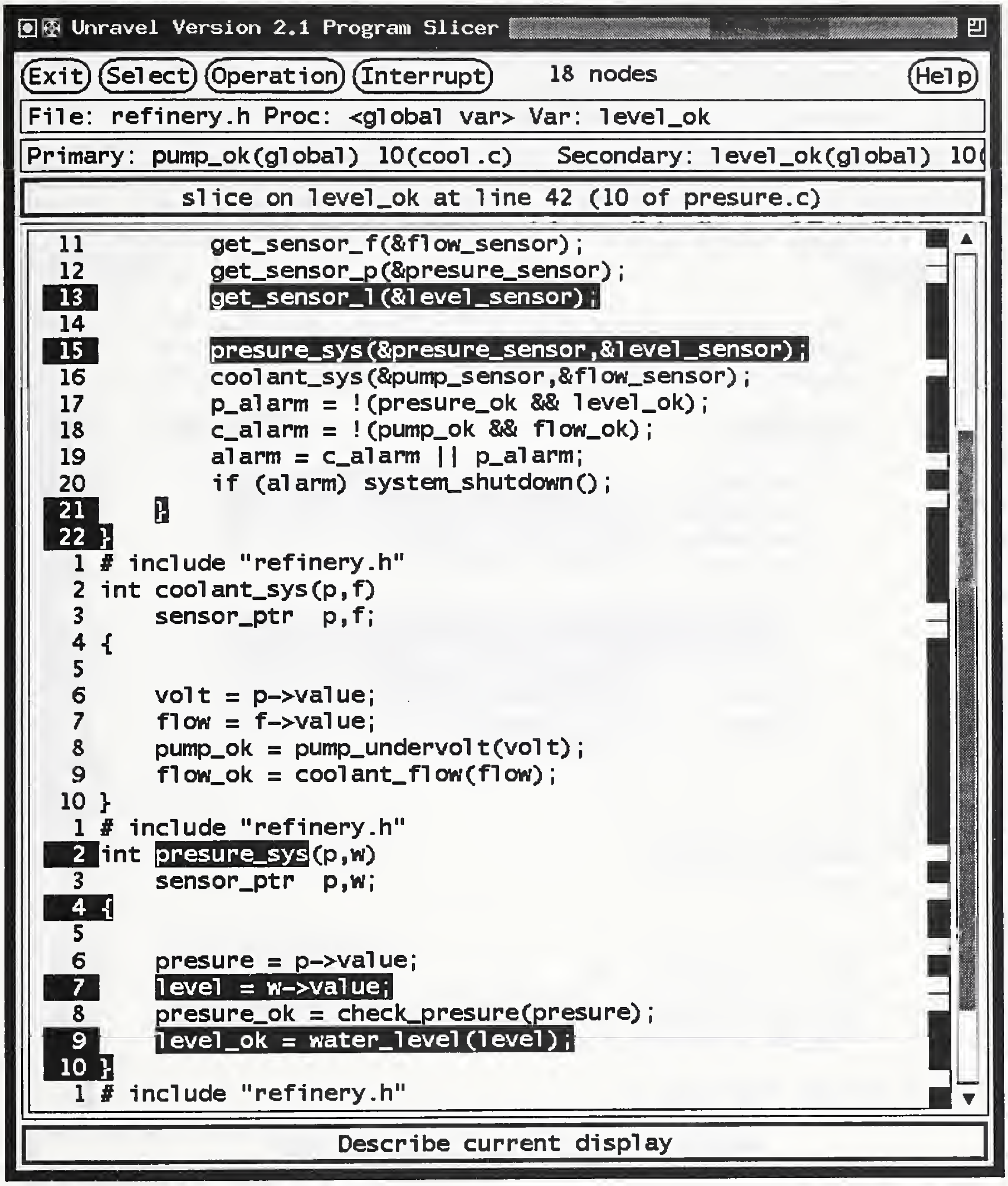
2.4.8 Exiting Unravel

When the user is finished with **unravel**, the user clicks on **Exit** in the **Main Control Panel**.



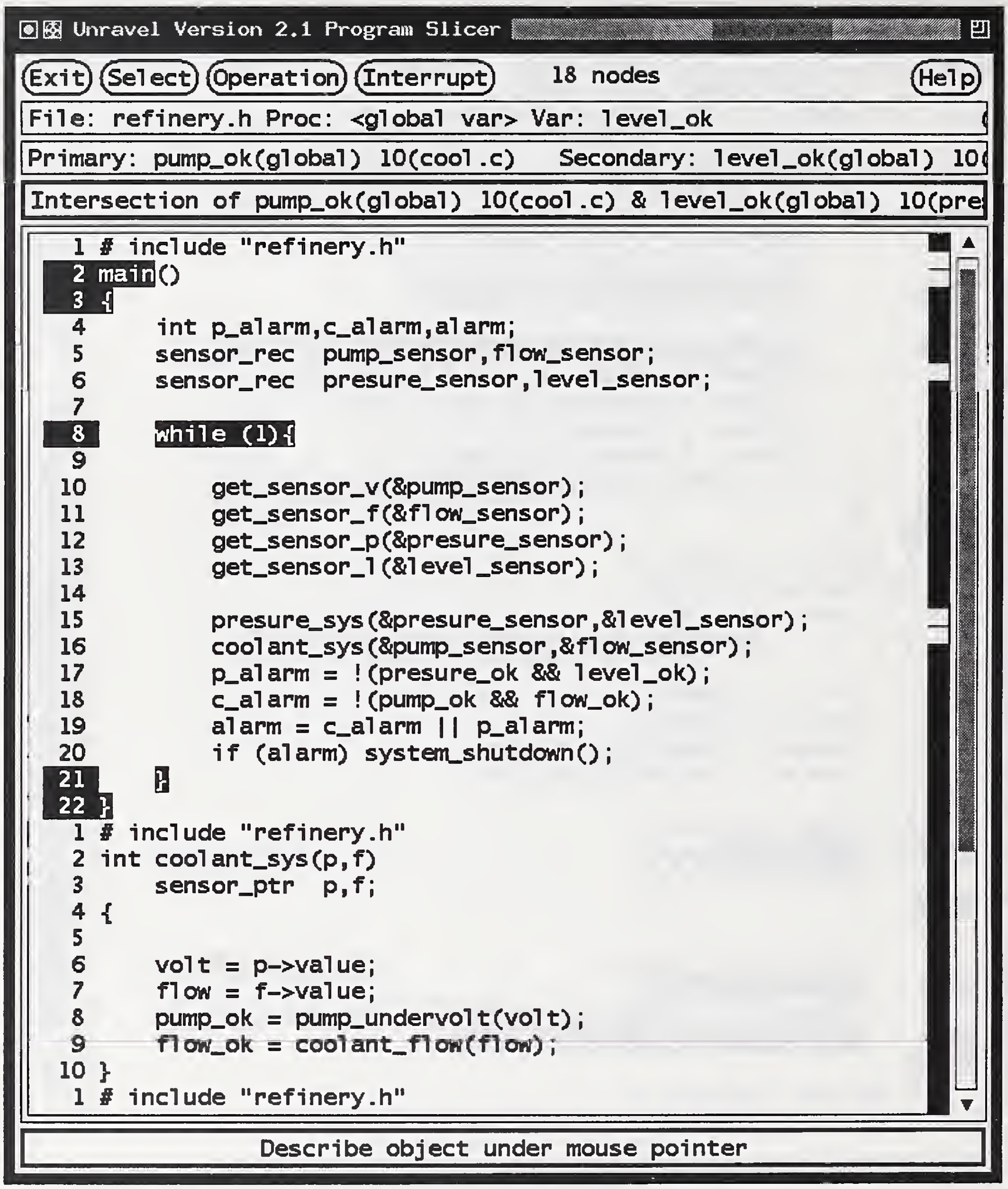
```
18 nodes
File: refinery.h Proc: <global var> Var: pump_ok
Primary: pump_ok(global) 10(cool.c)    Secondary: <none>
slice on pump_ok at line 32 (10 of cool.c)

 1 # include "refinery.h"
 2 main()
 3 {
 4     int p_alarm,c_alarm,alarm;
 5     sensor_rec  pump_sensor,flow_sensor;
 6     sensor_rec  presure_sensor,level_sensor;
 7
 8     while (1){
 9
10         get_sensor_v(&pump_sensor);
11         get_sensor_f(&flow_sensor);
12         get_sensor_p(&presure_sensor);
13         get_sensor_l(&level_sensor);
14
15         presure_sys(&presure_sensor,&level_sensor);
16         coolant_sys(&pump_sensor,&flow_sensor);
17         p_alarm = !(presure_ok && level_ok);
18         c_alarm = !(pump_ok && flow_ok);
19         alarm = c_alarm || p_alarm;
20         if (alarm) system_shutdown();
21     }
22 }
 1 # include "refinery.h"
 2 int coolant_sys(p,f)
 3     sensor_ptr  p,f;
 4 {
 5
 6     volt = p->value;
 7     flow = f->value;
 8     pump_ok = pump_undervolt(volt);
 9     flow_ok = coolant_flow(flow);
10 }
 1 # include "refinery.h"

Describe object under mouse pointer
```

Figure 2-18: Slice on *pump_ok*



```
Unravel Version 2.1 Program Slicer
18 nodes
File: refinery.h Proc: <global var> Var: level_ok
Primary: pump_ok(global) 10(cool.c)    Secondary: level_ok(global) 10(
slice on level_ok at line 42 (10 of presure.c)
 11          get_sensor_f(&flow_sensor);
 12          get_sensor_p(&presure_sensor);
 13          get_sensor_l(&level_sensor);
 14
 15          presure_sys(&presure_sensor,&level_sensor);
 16          coolant_sys(&pump_sensor,&flow_sensor);
 17          p_alarm = !(presure_ok && level_ok);
 18          c_alarm = !(pump_ok && flow_ok);
 19          alarm = c_alarm || p_alarm;
 20          if (alarm) system_shutdown();
 21      }
 22 }
  1 # include "refinery.h"
  2 int coolant_sys(p,f)
  3     sensor_ptr  p,f;
  4 {
  5
  6     volt = p->value;
  7     flow = f->value;
  8     pump_ok = pump_undervolt(volt);
  9     flow_ok = coolant_flow(flow);
 10 }
  1 # include "refinery.h"
  2 int presure_sys(p,w)
  3     sensor_ptr  p,w;
  4 {
  5
  6     presure = p->value;
  7     level = w->value;
  8     presure_ok = check_presure(presure);
  9     level_ok = water_level(level);
 10 }
  1 # include "refinery.h"
Describe current display
```

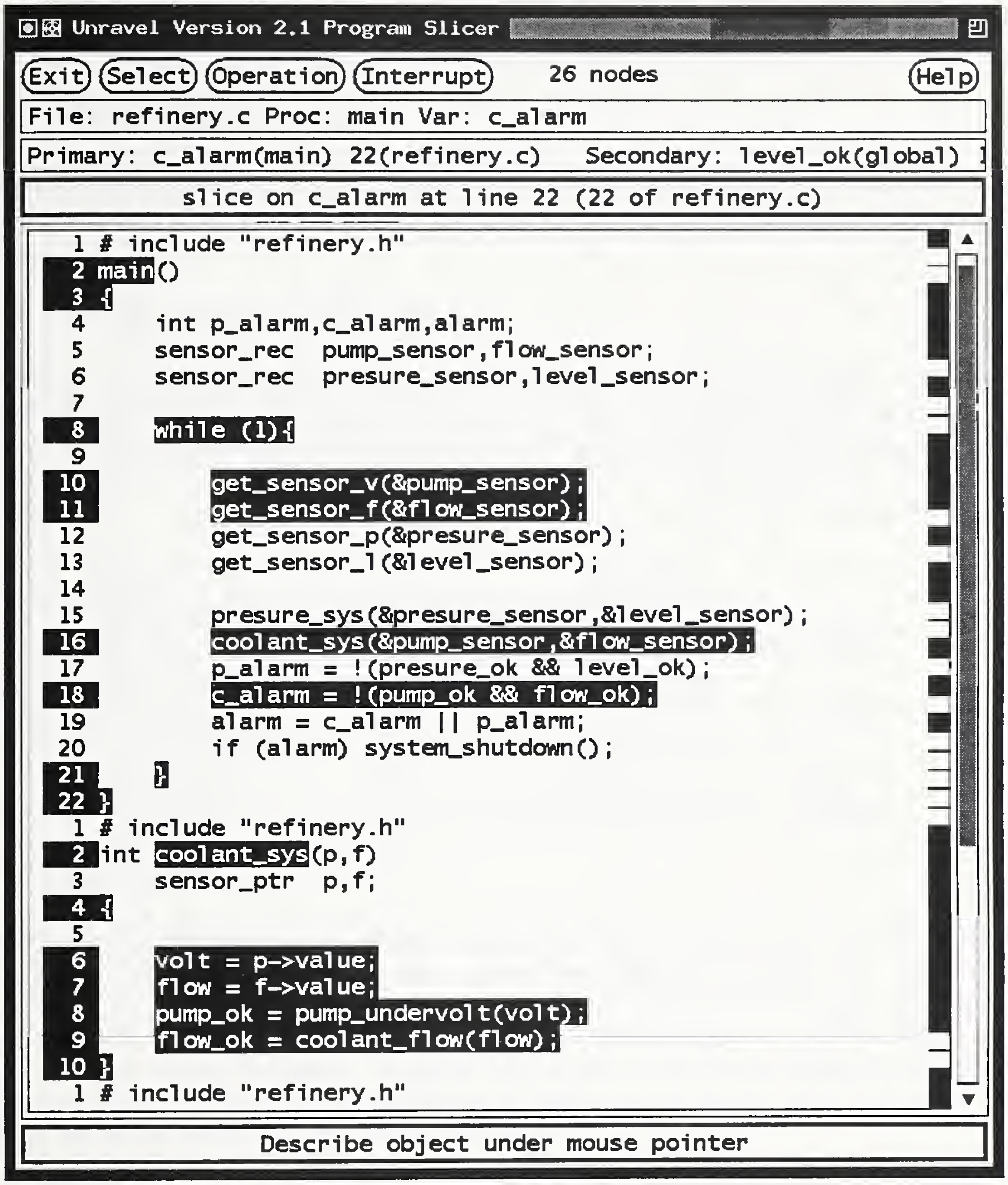
Figure 2-19: Slice on *level_ok*


Unravel Version 2.1 Program Slicer
Exit
Select
Operation
Interrupt
18 nodes
Help
Describe object under mouse pointer

```
File: refinery.h Proc: <global var> Var: level_ok
Primary: pump_ok(global) 10(cool.c)   Secondary: level_ok(global) 10
Intersection of pump_ok(global) 10(cool.c) & level_ok(global) 10(pre
```

```
# include "refinery.h"
main()
{
    int p_alarm,c_alarm,alarm;
    sensor_rec  pump_sensor,flow_sensor;
    sensor_rec  presure_sensor,level_sensor;

    while (1){

        get_sensor_v(&pump_sensor);
        get_sensor_f(&flow_sensor);
        get_sensor_p(&presure_sensor);
        get_sensor_l(&level_sensor);

        presure_sys(&presure_sensor,&level_sensor);
        coolant_sys(&pump_sensor,&flow_sensor);
        p_alarm = !(presure_ok && level_ok);
        c_alarm = !(pump_ok && flow_ok);
        alarm = c_alarm || p_alarm;
        if (alarm) system_shutdown();
    }
}
# include "refinery.h"
int coolant_sys(p,f)
    sensor_ptr  p,f;
{

    volt = p->value;
    flow = f->value;
    pump_ok = pump_undervolt(volt);
    flow_ok = coolant_flow(flow);
}
# include "refinery.h"
```

Figure 2-20: Intersection of Slices on *pump_ok* and *level_ok*



```
26 nodes
File: refinery.c Proc: main Var: c_alarm
Primary: c_alarm(main) 22(refinery.c)    Secondary: level_ok(global) ]
                slice on c_alarm at line 22 (22 of refinery.c)
  1 # include "refinery.h"
  2 main()
  3 {
  4     int p_alarm,c_alarm,alarm;
  5     sensor_rec  pump_sensor,flow_sensor;
  6     sensor_rec  presure_sensor,level_sensor;
  7
  8     while (1){
  9
 10         get_sensor_v(&pump_sensor);
 11         get_sensor_f(&flow_sensor);
 12         get_sensor_p(&presure_sensor);
 13         get_sensor_l(&level_sensor);
 14
 15         presure_sys(&presure_sensor,&level_sensor);
 16         coolant_sys(&pump_sensor,&flow_sensor);
 17         p_alarm = !(presure_ok && level_ok);
 18         c_alarm = !(pump_ok && flow_ok);
 19         alarm = c_alarm || p_alarm;
 20         if (alarm) system_shutdown();
 21     }
 22 }
  1 # include "refinery.h"
  2 int coolant_sys(p,f)
  3     sensor_ptr  p,f;
  4 {
  5
  6     volt = p->value;
  7     flow = f->value;
  8     pump_ok = pump_undervolt(volt);
  9     flow_ok = coolant_flow(flow);
 10 }
  1 # include "refinery.h"
            Describe object under mouse pointer
```

Figure 2-21: Slice on *a_dump*

3 Operating Unravel

Unravel is implemented in ANSI C under UNIX with a user interface built with the Athena Widgets of the MIT X Window System, running the X window system under the UNIX operating system.

3.1 Using The Interface Control Panels

The interface control panels are each composed of several interface objects from the MIT Athena Widget Set that the user can manipulate using the mouse to communicate with **unravel**. This section discusses how the user can interact with each Athena Widget used by **unravel**.

Unravel uses the following interface objects, each of which is described in subsequent sections.

Object	Function
Display Label	Presents some information
Button	Invokes some action (function) when pushed
Pull Down Menu	Displays a fixed list of buttons
Pop-up List	Displays a variable list of items for selection
Text Input Window	Allows the user to enter text
Text Window	Used to display more than one line of text
Scroll Bar	Used to specify visible subset of text window

3.1.1 Using a Mouse

Unravel uses a mouse with three buttons, referred to as *left*, *middle* and *right*. To interact with an object the user first positions the mouse cursor, an image on the screen that tracks mouse motion, over the object. Then the user does one of the following: types on the keyboard, clicks a mouse button or drags the mouse. To *click* a mouse button means to push the button on the mouse and then release the button. A *double click* is two mouse button clicks in a short (about one second) time. To *drag* the mouse means to push a mouse button, move the mouse while holding the button down and then release the button.

3.1.2 Display Label

Display labels are used to display information to the user. As the user requests **unravel** to do tasks the displayed information changes to inform the user of actions taken. The user has no direct input to a display label.

3.1.3 Button

A button on an **unravel** control panel has an oval border. To push a button, position the mouse cursor over the button and click the left mouse button. When the mouse button is pushed, the button is highlighted in reverse video. If the mouse cursor is dragged outside the button, highlighting is turned off and nothing happens if the button is released, otherwise, the button invokes its function when the button is released.

3.1.4 Pull Down Menu

A pull down menu button has an oval border. **Unravel** uses menus to group related actions for invocation by the user dragging the mouse. The user positions the mouse cursor over the *menu button*, then presses and holds the left mouse button. The menu appears in a window under the *menu button*. The user may drag the mouse cursor over the menu entries and release the button on the entry that should be invoked. The current entry is indicated by reverse video. If the button is released outside the menu nothing happens.

3.1.5 Pop-up List

Unravel uses *pop-up lists* for the user to select files, procedures or variables, usually for specifying a variable in a slicing criterion. To select an item from a pop-up list, position the mouse cursor over the item and click the left mouse button.

3.1.6 Text Input Window

A *text input window* is used to get text from the user. The mouse cursor must be positioned somewhere within the *text input window*. The *text input window* has a text cursor shaped like the caret character (^). Any characters typed are inserted at the text cursor. The text cursor can be moved by positioning the mouse cursor over the desired location and clicking the left mouse button. The character to the left of the text cursor can be removed by the **delete** key.

3.1.7 Scroll Bar

A *scroll bar* is usually used to control the display of information that needs more space than allocated on the screen. **Unravel** uses scroll bars on text windows, text input windows and pop-up lists. Scroll bars may be either horizontal or vertical. All three mouse buttons can be used to manipulate the scroll bar when the mouse cursor is positioned within the scroll bar.

Left Click: Move the text forward by some amount (depends on the actual object but is usually about 45% of the window length). The text in the controlled window does not change until the button is released. A drag has no effect.

Middle Drag: The length of the scroll bar is scaled to the length of the text being displayed. When the middle mouse button is pushed the text displayed in the window is adjusted so that the location of the start of displayed text corresponds to the location of the mouse cursor in the scroll bar. If the mouse is dragged, the displayed text is continuously updated to reflect the mouse cursor position. The drag does not have to remain within the bounds of the scroll bar.

Right Click: Move the text backward by some amount (depends on the actual object but is usually about 45% of the window length). The text in the controlled window does not change until the button is released. A drag has no effect.

3.1.8 Text Window

A *text window* is used by **unravel** to display history logs and help screens. A text window has scroll bars if the window is too small to display the text. The user can navigate through the file by either the scroll bar or by searching for a string. To search for a string type **CTRL-S** to pop-up a search control window. Enter the target string in the text input box just after the `Search for:` label. Either the **return** key or the **Search** button starts the search. The next instance of the search target can be found by either the **return** key or the **Search** button. There is a **Cancel** button to pop-down the search control window when done.

3.2 User Interface Control Panels

The user interface displays four control panels and two pop-up information windows. The four control panels are the following:

Main Control Panel: Allows the user to invoke analyzer and slicer and provides relevant information about the current directory, directory name, number of source files, number analyzed, number of main programs found and other information. The main control panel is invoked by running the program **unravel**. The command takes an optional command line argument of a directory name for the location of the source files to analyze. If the source files are in the current directory, the command line argument can be omitted. All the other control panels are invoked from the main control panel.

Analyzer Control Panel: Allows the user to select files, run the analyzer and automatically scan for **main** programs.

Selection Control Panel: Allows the user to select a **main** program and runs the **linker** on the selected **main** program followed by running the slicer on the linked program.

Slice Control Panel: Gives the user access to the program slicer, accepts a slicing criterion interactively and displays the source program text in a scrollable window with slice statements highlighted.

All control panels have the following features:

- Control buttons on top row of the panel
- Leftmost button pops-down (exits) the panel
- Rightmost button pops-up the *help* display for the panel
- **Help** button sticks to right window edge on resize
- Other buttons keep same distance from left edge on resize
- Panel name in the window title bar
- Last line of panel displays a brief description of the object under the mouse pointer
- **Help** button short cuts (accelerators): pressing anywhere on the panel outside a text window *h, H* or *?* invokes help
- **Exit** button short cuts (accelerators): pressing anywhere on the panel outside a text window *q* or *Q* exits the panel
- All top level control panel windows are created with an X Windows application class name of **Unravel** so that X resources can be set for all panels at once (e.g., to set the foreground color to red for all the control panels give the following resource specification to **xrdb**: **`*Unravel*Foreground: red`**.)

Two application resources, **runningFG** and **runningBG**, are defined for **analyzer** and **slicer**. These resources are a foreground and a background color that are used to indicate a lengthy operation is in progress.

The two information pop-ups display a history of user activities and help text for each panel. Both pop-ups are displayed by the same program. An information pop-up window consists of a **done** button to dismiss (pop-down) the window and a scrollable text window.

3.2.1 Main Control Panel

The function of the **Main Control Panel** is to respond to user interaction with the panel. All files are assumed to be in the current directory.

Unravel performs the following initializations:

1. Change to the directory specified on the command line, if one is provided.

2. Initialize slice history to **No slices computed this session**

3. Initialize analysis history to **No analysis done this session**

4. Initialize history of current session to **UNRAVEL** *directory name* **current date and time**

The **Main Control Panel** displays the following information:

- Current directory name.

- Number of source files. This is a count of files with a **.c** extension.

- Number of files analyzed and up to date. The number of C files that have analysis files such that the C file is older than the analysis files (i.e., the C file has not been changed since the analysis files were created).

- Number of source files not analyzed. This is a count of files with the **.c** extension that do not have analysis files.

- Number of files analyzed and out of date. The number of C files where the C file is younger than the analysis files.

- Number of main program files analyzed. The number of main program files identified.

- Number of main program files linked. The number of main program files identified that also have been linked.

- Number of duplicate procedures found. The number of procedures identified as ambiguous (appearing in more than one file). This indicates that **unravel** cannot determine which file contains the source code for some procedures. See section 5.2 for an explanation of how to remove ambiguous procedures.

- Last line of panel displays a brief description of the object under the mouse pointer.

The **Main Control Panel** buttons invoke the following actions:

Exit: Performs the following:

1. Append the file HISTORY to HISTORY.LOG
2. Delete HISTORY file
3. Exit

Run Analyzer: The **Run Analyzer** button pops-up the **Analyzer Control Panel**.

Review History: Pops-up a four item menu (**Last Analysis**, **Last Slice**, **This Session** or **All History**), and displays the indicated history in a pop-up window.

Run Slicer: Pop-up a list of main programs that can be selected for slicing.

Help: Display general information about **unravel** and the **Main Control Panel** in a pop-up window.

3.2.2 Analyzer Control Panel

The **Analyzer Control Panel** presents the user with:

- Buttons to control file selection, run the analyzer, clear analysis files and pop-up a help window.
- A status line to give the user feedback on progress of the analysis of a set of files.
- Two text windows for specifying command line options to the C preprocessor and to the unravel analyzer.
- A list of *selected* source files from the current directory.
- A list of *not selected* source files in the current directory.
- Last line of panel displays a brief description of the object under the mouse pointer

The **Analyzer Control Panel** buttons invoke the following actions:

Exit Analyzer This button Appends the file **HISTORY-A** to **HISTORY** and then exits.

File Selection The **File Selection** button pops-up a menu with the following four choices and actions:

All Files: All source file names are placed in the *selected* list window. The *not selected* list window will be empty.

No Files: All source file names are placed in the *not selected* list window. The *selected* list window will be empty.

Analyzed Files: All source file names of files that have older **.LIF**, **.T** and **.H** files are placed in the *selected* list window. The remaining source file names are placed in the *not selected* list window.

Files Not Analyzed: All source file names of files that have older **.LIF**, **.T** and **.H** files are placed in the *not selected* list window. The remaining source file names are placed in the *selected* list window.

Analyze Selected Files/Stop Analysis This button runs the analyzer on each selected file, adding the contents of the C preprocessor options window to the C preprocessor command line and adding the contents of the parser options window to the parser command line. When the **Analyzer Control Panel** button is pushed the button label is changed from **Analyze Selected Files** to **Stop Analysis**. If the **Stop Analysis** button is pressed, **unravel** will not analyze any more of the selected files after the file currently being analyzed is finished. As each file is analyzed, the file name currently being analyzed is displayed on the status line along with a progress indication. The progress indication is defined by the following: number each file in sequence starting from 1 in the order that the files will be analyzed. Display the file's sequence number and the total number of files. The status line is set to the foreground and background colors specified in the application resources **runningFG** and **runningBG**.

After all selected files have been analyzed, the **map** program is run automatically.

Clear Deletes the analysis files (**.LIF**, **.H** and **.T**) for each selected file and deletes the **SYSTEM** file.

Help: The **Help** button displays a pop-up window of information about the analyzer control panel.

The C preprocessor options text window allows the user to specify any C preprocessor options that are normally used to compile the source code to be analyzed. The two most common options are *preprocessor symbol definition (-D)* and *include directory path (-I)*. When and how to use the *-D* option depends on how the source code is usually compiled. The *-I* option is used to specify the location (directory path) of include files not located in the current directory.

The *parser options* text window allows the user to specify options to the parser. The parser has two options related to dynamic memory allocation that might be needed. There is a list of function names that are assumed to allocate dynamic storage. When **unravel** encounters a call to one of these functions a dynamic variable is created in the form `@`*`file_name`*`#`*`line number`*`[`*`sequence number`*`]` to represent any memory allocated by the statement. If the function call result is cast to a *pointer to a base type*, the generated variable is declared to be of the base type.

1. The **-a** option is specified to remove the default list of memory allocation procedures. The default list is: **malloc()**, **realloc()** and **calloc()**.

2. The option **-f** *name* adds procedure *name* to the list of memory allocation functions.

There are several other options to the parser, intentionally undocumented, that are not useful to the user of **unravel** but are important for debugging and performance analysis.

3.2.3 Selection Control Panel

The Selection Control Panel presents the user with:

- **Exit** and **Help** buttons
- A status line
- A list of main program source files from the current directory
- Last line of panel displays a brief description of the object under the mouse pointer

The **Exit** button pops-down the panel with no further action.

If a file from the list is selected, the file is linked, the **Selection Control Panel** is popped-down and the **slicer** is called.

The *status line* initially indicates that **select** is waiting for the user to make a selection. After a file is selected, the status line indicates that a file is being linked.

If there is exactly one main program file, the file is linked and the slicer is called without bringing up the **Selection Control Panel**.

The **Help** button pops-up a file of information about the **Selection Control Panel**.
The slice history file is updated with a message indicating the file to be linked before the linker is called. Any linker output is appended to the slice history file.

3.2.4 Slice Control Panel

The slicer accepts slicing criteria from the user, computes a program slice for each criterion given, saves each slice for later recall and displays the program in a scrollable window. The slicer presents the user with:

- Buttons to exit, to pop-up help and to interrupt a lengthy slice calculation.
- A display indicating slice size and slice calculation progress.
- A display of the currently selected slicing criterion variable.
- Menu of selection options for selecting slicing criterion variables, or previously computed slices.

- Menu of operations that can be performed on two selected slices.
- Display describing the contents of the scrollable window.
- Display of program source text in a scrollable window.
- Last line of panel displays a brief description of the object under the mouse pointer.
- Clicking a mouse button in the text window specifies the statement for the slicing criterion and initiates the slice computation.

Primary slice and *secondary slice* has no significance other than being convenient names for two slices when an operation such as intersection is performed on two slices.

The buttons do the following:

Exit: Append slice history to session history and then exit.

Interrupt: The **Interrupt** button does the following:

1. Stop computation of the slice and display partial results.
2. Mark the slice as partial and save.

Help: Pop-up the panel help file.

There are six information display windows on the control panel.

1. **Slice Progress Window** displays the current size of the slice being computed (or last computed) in units of flow graph nodes. This window is located between the **Interrupt** and **Help** buttons on the top line of the panel.
2. **Criterion Variable Window** displays the currently selected criterion variable, the file where the variable is declared and the declaration scope. If the variable is *global* then the scope is the word *global*, otherwise the name of the local procedure containing the variable declaration. If an element is not defined, the word **none** is displayed. This window is the second line of the panel.
3. **Primary-Secondary Window** displays the criteria for the current primary and secondary slices. If there is no such slice, the word **none** is displayed. This window is the third line of the panel.
4. **Text Description Window** describes the contents of the **Text Window** using one Message in Table 3-1. This window is the fourth line of the panel.

Contents	Message
None	Source File: *file name*
Slice	Slice on *criterion*
Intersection	Intersection of *primary criterion & secondary criterion*
Union	Union of *primary criterion & secondary criterion*
Dice	*Primary criterion* diced by *secondary criterion*
Dice S-P	*Secondary criterion* diced by *primary criterion*
Marked	Location of *procedure name* in *file name*
Call Tree	Call tree of *procedure name*

Table 3-1: Operation Description

5. **Text Window** displays the program text with a scroll bar for navigation. Statements can be designated for highlighting by the text window. Highlighting is used to indicate statements that are members of a slice or the results of an operation on two slices. The right margin of the text window contains a **tick bar** that is used to visually indicate the location of highlighted statements throughout the entire program. The vertical length of the tick bar is scaled to the length of the program in source file lines. A tick (horizontal line) in the tick bar indicates that at that relative position in the display there are one or more highlighted lines. The tick bar is adjacent to the scroll bar to facilitate scrolling to highlighted regions of the text. Clicking the left or middle mouse button on the tick bar scrolls the text window to the corresponding area of the program. Above the scroll bar is an arrow shaped button that scrolls the text window up one line each time the left mouse button is clicked when the mouse cursor is over the arrow button. Below the scroll bar is a similar arrow shaped button to scroll text down.

6. **Current Object Window** describes the function of the object currently under the mouse pointer. This window is the last line of the panel.

The **Select** menu is used to specify a criterion variable, to aid navigation by marking the location of a procedure in the tick bar and to select a previously computed slice for display.

The **Select** menu has the following selections:

Local Variable: This entry is a two-step selection. First, a list of procedure names is popped-up for the user to select one item. The list consists of all procedures that are defined somewhere in the program. Procedures such as library routines that are used, but not defined are not included in the list. The first entry in the list is **No Selection**. If a procedure is selected, the procedure header, opening brace and closing brace are highlighted, a list of variables declared local to the selected procedure is popped-up and the list of procedure names is popped-down. If no procedure is selected, the procedure list is popped-down. The **Criterion Variable Window** is updated with the selected items.

Global Variable: This entry is a two-step selection. First, a list of file names is popped-up for the user to select one item. The list includes all source files (.c) in the program and all header files (.h) that are included in the program. The first entry in the list is **No Selection**. If a file is selected, a list of global variables declared in the selected file is popped-up and the file list is popped-down. If no file is selected, the file list is popped-down. The **Criterion Variable Window** is updated with the selected items.

Mark Proc: A list of procedure names is popped-up for the user to select one item. The first entry in the list is **No Selection**. If a procedure is selected, the procedure header, opening brace and closing brace are highlighted. The list is popped-down.

Show Call Tree: A list of procedure names is popped-up for the user to select one item. The first entry in the list is **No Selection**. If a procedure is selected, the procedure header, opening brace, closing brace and all the call sites for the selected procedure are highlighted. The highlighting continues for each procedure containing a highlighted call site until no more unhighlighted procedures are found. If a call site in controlled by a conditional statement (e.g., **if** or **while**), the conditional statement is highlighted. The list is popped-down.

Primary: Pops-up a list of previously computed slices. The first entry in the list is **No Selection**. If an entry is selected, make the slice the *primary* and display the slice. The list is popped-down.

Secondary: Pops-up a list of previously computed slices. The first entry in the list is **No Selection**. If an entry is selected, make the slice the *secondary* and display the slice. The list is popped-down.

The **Operation** menu has the following selections:

Dice: Highlights the statements of the *primary* slice that are not members of the *secondary* slice and updates the **text description window**.

Dice S-P: Highlights the statements of the *secondary* slice that are not members of the *primary* slice and updates the **text description window**.

Intersection: Highlights the statements in both the *primary* and *secondary* slice and updates the **Text Description Window**.

Union: Highlights the statements in either the *primary* or *secondary* slice and updates the **Text Description Window**.

Clear: Removes all highlighting and updates the **Text Description Window**.

Clear Slice History: Deletes the saved slices.

The text window has four actions triggered by the mouse.

1. Clicking a mouse button in the tick bar area scrolls the window to the corresponding area of the program text.

2. The leftmost mouse button computes a primary slice.

3. The middle mouse button computes a secondary slice.

4. The rightmost mouse button highlights the current line.

The source program line under the mouse pointer when the mouse button is clicked specifies the statement for the slicing criterion. If the specified slicing criterion has already been used to compute a slice (without interruption), then the slice is not computed, but is retrieved and displayed.

3.2.5 Help/History Pop-up

History and help information pop-ups are handled by a single program. The text may have either horizontal or vertical scroll bars as needed. It is possible to search for a string by typing **CTRL-S** and typing the target string in the search pop-up. There is a **Cancel** button to pop-down the search control window when done.

4 Solving Problems

This section discusses how to resolve two problems that can arise when using **unravel**. The first problem occurs when the software to be analyzed is not strict ANSI C. The second problem can occur when more than one program is being analyzed.

4.1 Source Code Not ANSI C

Unravel is designed for ANSI C and is strict about the language accepted. There exist C compilers that for a variety of reasons have implemented extensions to the C language. When these extensions involve additional data types, additional data attributes, additional keywords, or changes to the language syntax, **unravel** cannot be used without first modifying the source code. This section discusses how to accomplish these modifications.

Any unknown keywords or data types bring the **unravel analyzer** to a stop at that point with a message indicating the line number and file. If there are a few deviations from ANSI C, it is sometimes possible to make the program acceptable to **unravel** with a few small changes to the source program under analysis. If there are a large number of extensions or there are any significant syntactic additions then modifications to the source program may be too time-consuming to be practical.

4.1.1 Additional Data Types

A common extension is to include additional data types in the C language. Provided that the added data types do not also add significantly new semantics they can be handled by inserting a `#define` preprocessor command to equate the new type to an existing builtin type. For example, if a compiler adds a *boolean* type, the following preprocessor statement could be used to equate **boolean** with **int**:

```
#define boolean int
```

4.1.2 Additional Data Attributes

Another common extension is additional data attributes. For example, a compiler might add the keyword *extended* to indicate that floating point numbers should use 128 bits rather than the 64 bits used in double precision. This keyword can be eliminated by defining the keyword to be null:

```
#define extended
```

4.1.3 Preprocessor Extensions

Sometimes the C preprocessor is extended. The extensions must be replaced in such a way that the result of the ANSI C preprocessor used by **unravel** produces the same result as the vendor C preprocessor. An alternative to replacing the extensions is to use the extended preprocessor to preprocess the source program.

4.2 Ambiguous procedures

If the **unravel** user is interested in more than one source program the source files for all the source program files can be in the same directory, however, if there are two or more functions with the same name, **unravel** cannot determine which is the procedure that belongs with a given **main** program.

The example in Figure 4-1 presents five source files (*alpha.c*, *beta.c*, *theta.c*, *phi.c* and *web.c*) that correspond to two source programs. Source program **alpha** requires the files *alpha.c, theta.c* and *web.c*. The second source program, **beta**, requires the files *beta.c, phi.c* and *web.c*.

If we run the **unravel** analyzer on all the these files, the analyzer reports that the procedure **angle** is defined in more than one file (**theta.c** and **phi.c**). There are two possible ways to deal with the problem, either move each set of source files for a given source program to a separate directory or only analyze the source files for one source program at a time.

```
  alpha.c                  beta.c
int x;                   int y;
main(){                  main(){
     angle();                angle();
     mangle();               tangle();
}                        }

    theta.c                   phi.c
int x;                   int y;
angle(){ x = 1;}         angle(){ y = 2;}

       web.c
int x,y;
mangle(){ printf ("x = %d\n",x);}
tangle(){ printf ("y = %d\n",y);}
```

Figure 4-1: Ambiguous Example

4.2.1 Separate Directories

To put each source program in separate directories, use the UNIX **mkdir** command to create a directory for each program (i.e., one subdirectory for each **main** procedure), then move source files (with the **mv** command) to the subdirectories where they belong. If a source file is used by more than one source program, use the link command, **ln**, to place a link to the source file in each subdirectory.

```
mkdir alpha beta
mv alpha.c theta.c alpha
mv beta.c phi.c beta
ln web.c alpha
ln web.c beta
```

4.2.2 One at a Time

Alternatively, the user can analyze only files **alpha.c**, **theta.c** and **web.c**. Then the user runs the slicer and selects **alpha**. After slicing on **alpha** is finished, the user can go back to the **Analyzer Control Panel** and clear **alpha.c** and **theta.c**, then analyze **beta.c** and **phi.c** and run the slicer on **beta**.

5 Unravel Limitations and Assumptions

Unravel has limitations for several reasons. For example, an approximate solution to some aspect of slicing may be implemented to avoid a severe performance penalty. Other limitations arise from the static nature of program slicing. This section presents the most serious limitations.

5.1 Casts

Unravel ignores *cast* operations except where the cast is on the return value of the **malloc** function. This can cause a problem when a variable is declared as one data type and then cast to another data type. For example, consider the following code:

```
typedef struct {
     int      a;
     int      b; } ab_type;
int           x[2];
ab_type       *y;
. . .
y = (ab_type *) x;
y->b = 10;
```

Unravel loses the connection between the variable **x** and the object pointed to by **y**. **Unravel** expects that **y** points to an object of type **ab_type**, however when **unravel** finds that **y** points to **x**, **unravel** is unable to see that **y->b** is really **x[1]** because **unravel** expects to find **x.b**.

As a result, statements that influence the value of **x** are not in the slice. If there is one such cast then a second slice could be computed on **x** at the cast statement and the two slices unioned together.

5.2 Address Operator

Unravel expects the address operator (&) to be applied only to variables and not applied to expressions. This turns out to be a design error that causes expression such as `&a->b` to be ignored. This can cause statements to be omitted from a slice.

5.3 Pointers

When computing *active sets* for a statement that dereferences a pointer variable each object that the variable might point to should be added to the active set. At different locations in a program the set of objects that a pointer could point to may be different. For each pointer, **unravel** identifies all objects that the pointer might point to. This can cause statements to be included in a slice that could be excluded with a more precise pointer tracking algorithm.

Pointers to functions are ignored.

5.4 Unions

Unions are treated like structures by **unravel**. The union members appear to **unravel** as separate data objects rather than overlapping objects.

5.5 Goto and Branch Statements

Unravel ignores unconditional branch statements (i.e., **goto**, **break**, and **continue**). In most programs this does not change the content of computed slices in a significant way however, some statements that should be in some slices are omitted. For example, in the following code a slice on **x** should include all lines shown, however, if the **break** statement is ignored, lines 4 and 1 are omitted. Note that the statements missing are concerned with the calculation of another variable (**z**) and do not have a direct role in the calculation of **x**. This problem will be addressed in a later release of **unravel**.

```
1 z = a;
2 while (y) {
3    y--;
4    if (z) break;
5    x = w;
6 }
```

5.6 Libraries

Unravel has no knowledge of any libraries that might be used by vendor code. This includes the ANSI C library. Any library procedure call is assumed to not use or change any global variable. Any variable whose address is passed to a library procedure is assumed to be changed, and if a structure all members are assumed to be changed.

Appendix A: Unravel Quick Reference

1 Orientation	Examine program structure
2 Select Slices	Plan slices to compute
3 cd to source directory	Make source code directory current
4 **Unravel**	Execute **unravel**
5 Analyzer	Click on **Run Analyzer**
Select Files	Select files to analyze
Analyze	Click on **Analyze Selected Files**
Exit	Click on **Exit** to return to **Main Control Panel**
6 Review Results	Examine **Last Analysis** in **Review History** menu
Navigate	Use scroll bar to navigate
Done	Click on **Done** to exit
7 Slicer	Click on **Run Slicer**
Select Variable	Use **Select** menu to pick variable
Select Location	Click on criterion location to do slice
Navigate	Use scroll bar to examine source code
Operations	Use **Operations** menu to combine slices
Exit	Click on **Exit** to return to **Main Control Panel**
8 Exit	Click on **Exit** to stop **unravel**

Figure A-1: Unravel Operation Checklist