NISTIR 5691

Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 1: Requirements and Design

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 1: Requirements and Design

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

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 1

Abstract

Current practice for examination of a high integrity software artifact is often a manual process that is slow, tedious, and prone to human errors. This report describes a Computer Aided Software Engineering (CASE) tool, **unravel**, that can assist evaluation of high integrity software by using program slices to extract computations for examination. The tool can currently be used to evaluate software written in ANSI C and is designed such that other languages can be added.

Program slicing is a static analysis technique that extracts all statements relevant to the computation of a given variable. Program slicing is useful in program debugging, software maintenance and program understanding. Application of program slicing to evaluation of high integrity software reduces the effort in examining software by allowing a software reviewer to focus attention on one computation at a time. Once a software reviewer has identified a variable for further investigation, the reviewer directs **unravel** to compute a program slice on the variable. Instead of examining the entire program, only the statements in the slice need to be examined by the reviewer. By speeding up the process of locating relevant code for examination by the reviewer, a larger sample of the software can be inspected with greater confidence that some relevant section of source code has not been missed.

The source code for **unravel** is available and requires a UNIX or POSIX environment, an ANSI C compiler and the MIT X Window System, version 11 release 5 or later.

Volume 1 of this report describes the requirements, design and evaluation of **unravel**. Volume 2 is a user manual and tutorial for the **unravel** software.

Key Words

Code Analysis; High Integrity Software; Inspections; Program Slicing; Program Understanding; Reviews; Software Safety; Software Tools; Static Analysis

Trademarks

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

Executive Summary

Unravel is a prototype Computer Aided Software Engineering (CASE) tool that can be used to statically evaluate ANSI C source code using program slicing. 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. Under the terms of those contracts, the National Institute of Standards and Technology (NIST) supplied the prototype to both funding parties.

Program slicing is a static analysis technique that extracts all statements relevant to the computation of a given variable. Program slicing is useful in program debugging, software maintenance and program understanding. Application of program slicing to evaluation of high integrity software reduces the effort in examining software by allowing a software reviewer to focus attention on one computation at a time.

By combining program slices using logical set operations, **unravel** can identify code that is executed in more than one computation. This information is immediately useful for addressing issues of high integrity software, since a failure involving this code may lead to a malfunction of more than one logical software component. In the case of safety systems, which commonly use several computations for protection, common code among them can provide a single point of failure. In the case of security, what may have been perceived as a secure path may be penetrated by an otherwise unsuspected approach. The identification of common code enables the developer to consider redesign or to emphasize verification and validation activities in those regions to provide assurance of the program.

Unravel was evaluated in the context of reviewing safety system software for quality. The evaluation considered the size of slices produced, time to compute slices and usability by a novice user. 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?

3. Is **unravel** usable by a novice user?

Two examples of typical safety system code were used to test and refine **unravel**. Demonstration of **unravel** using these and other examples were given to software reviewers. The demonstrations resulted in improvements to the user interface and in the identification of features to be explained in more depth in the user manual or to be included in a later version of **unravel**.

Examination of software is often a manual process that is slow, tedious, and prone to human errors. With **unravel**, once a reviewer has identified a variable for further investigation, the reviewer directs **unravel** to compute a program slice on the variable. Instead of examining the entire program, the reviewer only needs to examine the statements in the slice. By speeding up the process of locating relevant code for examination by the reviewer, a larger sample of the software can be inspected with greater confidence that some relevant section of source code has not been missed.

Without any tool, a reviewer evaluates the software for common code by manually searching for code shared between two computations until it is determined that there is no common code, or that the common code present will not compromise the mission of the safety critical software. With **unravel**, once two computations that could be vulnerable to common mode failure have been identified, program slices can be computed to find statements relevant to each computation. Source program statements that have potential to cause common mode failure would be present in the intersection of the program slices.

Unravel consists of three main components, called the analyzer, linker and slicer. The analyzer and linker components can process up to 100,000 lines of source code in less than 10 minutes. The linear behavior of the analyzer and linker leads to stable run time performance. The slicer component does not use a linear algorithm, but rather uses a quadratic algorithm that can have significant run time variability. It should be noted that there is potential for significant algorithm improvement. For example, after one small change in the slicer code the longest time on a SPARCstation 2 to compute a slice on code from a 4,000 line actual safety system dropped, from 10 hours to 3 hours. Other areas that can be improved include loop analysis and procedure calls.

Symbol Glossary

Symbol	Example	Meaning of Example
{ }	$\{x\}$	the *set* with x as a member
$\in$	$x \in A$	x *is a member* of set A
$\forall$	$\forall x, \ni x > 0$	*for all* x such that x is positive
$\exists$	$\exists x, \ni x > 0$	*there exists* an x that is positive
$\cap$	$A \cap B$	*intersection* of A and B; members in both
$\cup$	$A \cup B$	*union* of A and B; members in either
$\varnothing$	$\varnothing$	the *empty set*; the set with no members
$\ni$	$\forall x \ni x > 0$	for all x *such that* x is positive
$\notin$	$x \notin A$	x is *not a member* of set A

Table Of Contents

List of Tables

List of Figures

1 Introduction

This report describes the design and development of **unravel**, developed at 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. **Unravel** is a Computer Aided Software Engineering (CASE) tool that can be used to statically evaluate ANSI C[1] source code using program slicing. **Unravel** may also be used to examine software code for computer security functions. The tool can currently be used to evaluate software written in ANSI C and is designed such that other languages can be added.

Program slicing is a static analysis technique[3] that extracts all statements relevant to the computation of a given variable. Program slicing is useful in program debugging[4], software maintenance[5] and program understanding[6]. Application of program slicing to evaluation of high integrity software reduces the effort in examining software by allowing a software reviewer to focus attention on one computation at a time. Once a software reviewer has identified a variable for further investigation, the reviewer directs **unravel** to compute a program slice on the variable. Instead of examining the entire program, only the statements in the slice need to be examined by the reviewer. By speeding up the process of locating relevant code for examination by the reviewer, a larger sample of the software can be inspected with greater confidence that some relevant section of source code has not been missed.

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 the following standards: POSIX[9] operating system interface, ANSI C, and the X Window System[10].

The source code for **unravel** is available and requires a UNIX or POSIX environment, an ANSI C compiler and the MIT X Window System, version 11 release 5 or later.

Section 2 discusses slicing ANSI C. Section 3 contains the requirements for building **unravel**. Section 4 consists of the design of **unravel**. Section 5 provides the system evaluation and performance report. Appendices A and B contain the Language Independent Format and the Yacc ANSI C grammar respectively. Volume 2 is the **unravel** user manual.

2 Slicing 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 have been shown to aid program understanding, program debugging, program maintenance, and automatic integration of program variants[7].

This section describes the algorithms for building a program slicing tool[2] for ANSI C. Section 2.1 presents definitions and terminology; section 2.2 gives a general description of program slicing, its limitations, and assumptions about the expected users of **unravel**.

2.1 Definitions

This section contains definitions relevant to program slicing.

Active Set. The active set at statement *n* for slicing criterion <*L*, *V*> is a set of program variables, *active(n)*, such that the value of any member of active(*n*) just before execution of statement *n* could influence the value of *V* just before execution of statement *L*. Informally, the active set is the set of variables that determine the value of the criterion variable at the criterion location.

Code Generator. See Compiler.

Code Improver. See Compiler.

Compiler. Production of an object program from a source program is often modeled with concepts from linguistics. A source program is viewed as consisting of words from a vocabulary. The words are assembled into valid sentences of the language that form program statements. Each sentence is used to generate object code corresponding to each program statement. Modern compilers are usually designed in four components:

1. **Scanner** reads the program source code and collects strings of characters into the fundamental vocabulary units of the programming language called tokens. This vocabulary contains language key words, operators, text strings and identifiers.

2. **Parser** takes the tokens produced by the scanner and a grammar describing the language and checks that the sequence of tokens represents valid sentences in the language. A sequence of tokens that is not a valid sentence of the language grammar is a *syntax error*.

3. **Code Generator** is called by the **parser** for each valid sentence the **parser** recognizes to produce corresponding object code.

4. **Code Improver** Uses data-flow analysis to revise the produced object code so that the run-time execution is faster or requires less memory. This step is usually called *code optimization.*

Data-Flow Analysis. Using information about program structure, variable initialization, assignment of values to variables, and use of program variable values to answer questions about the behavior of program variables. Data-flow analysis is often used in compiler optimization of generated object code.

Defs(n). The set of variables defined (assigned to) at statement *n*. In data-flow analysis, modification of a variable is called a definition of the variable.

Dependence Graph. A program representation, defined by Ferrante[8], with many applications to program manipulation including program slicing. The PDG (*program dependence graph*) has the same nodes as a *flow-graph* but the edges represent control and data dependence within a program.

Flow Graph. A representation of the control structure of a program as a directed graph. The nodes of the graph correspond to statements or contiguous tokens of a source program. The edges correspond to program control flow.

Idefs(n). The set of variables specifying indirect assignment by a pointer at statement *n*. Each idef entry is a pair indicating a variable and a level of indirection. Level 0 represents a direct assignment to the variable. Level 1 represents an assignment to a variable whose address is contained in the idef entry variable.

Irefs(n). The set of variables specifying indirect reference by a pointer at statement *n*. Each iref entry is a pair indicating a variable and a level of indirection. Level 0 represents a direct reference to the variable. Level 1 represents a reference to a variable whose address is contained in the iref entry variable.

Parser. See Compiler.

Pointer. A variable that contains the address of a variable or other program object such as a procedure.

Pred(n). The set of statements that can be executed immediately before statement *n*. The predecessors of *n*.

Program Slice. 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 the following are true.

1. S is derived from P by deleting zero or more tokens from P.

2. S is syntactically correct.

3. The value of V just before control reaches location L is the same for EXECUTE(P) as EXECUTE(S).

Program Dice. The result of application of logical set operation to two or more program slices is a program dice. The intersection of two slices yields the statements in common to both computations. A software fault in the common code can be a single point of failure for both computations. The intersection of one slice with a logical complement of a second slice yields the set of statements in the first slice that has no influence on the second computation. This is useful when trying to isolate a fault in the second computation.

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

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

Scanner. See Compiler.

Slicing criterion. A slicing criterion, for a program is a tuple <*L,V*> where *L* is a statement in the program and *V* is a variable. A program slice is computed *on V, at statement L.* Where the meaning is clear from context, *V* is extended to a subset of program variables.

Succ(n). The set of statements that can be executed immediately after statement *n*.

Token. The output of the scanner and input to the parser; the fundamental lexical units of a language.

2.2 General Description

Unravel is divided into three main components: a source code analysis component, a link component, and an interactive slicing component. The analysis component collects from source files (with a **.c** extension) and included header files (usually with a **.h** extension) the information necessary for the computation of program slices. The information is translated to a representation independent of source language called language independent format (LIF). The analyzer is designed like a compiler with a scanner to break the source code into tokens that are recognized by a parser, but instead of generating object code, it produces LIF code. The analyzer also produces a tally of objects (**.T** file) such as procedures and variables, and a file to list global

objects (**.H** file) declared in each included header file. The link component operates in two parts. The first part, **map**, identifies for each program in the current directory its constituent files and then saves this information in a file named **SYSTEM**. The second part of the link component, **slink**, uses the **SYSTEM** file to merge data-flow information from the **.LIF**, **.T** and **.H** files created from separate source files into a single **.LINK** file and a single **.K** file. Under user control, the interactive component extracts and displays program slices and keeps a record of user activities in a **.LOG** file. The overall structure of **unravel** is presented in Figure 2-1.

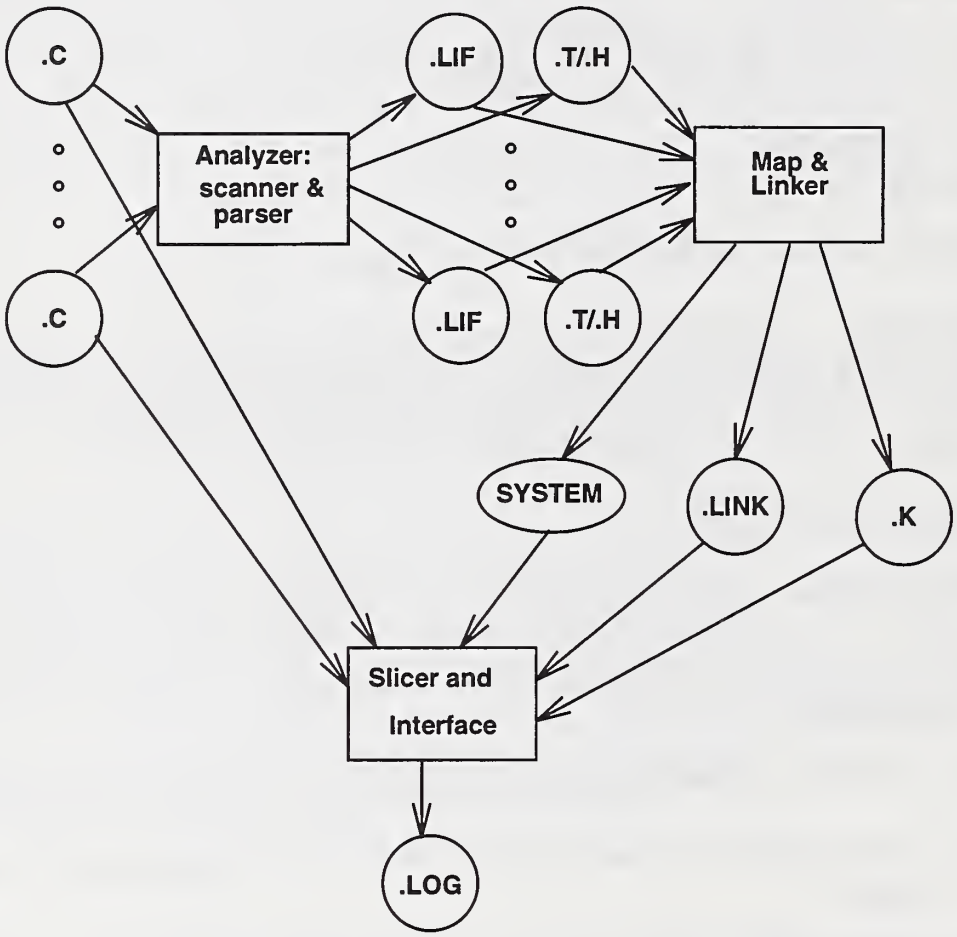

Figure 2-1: Unravel Structure Overview

2.2.1 Program Slicing Algorithm

A program slicing algorithm 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 that assigns a value to any variable in the slicing criterion and generate a new slicing criterion for the predecessor by deleting the assigned variables from the

original slicing criterion and adding any variables referenced by the predecessor. The slicing algorithm considers the following language features:

1 Expression statements
2 Compound control statements
3 Structure variables
4 Indirect assignment by pointer
5 Indirect reference by pointer
6 Dynamic structures
7 References to structure members by pointer
8 Assignment to structure members by pointer
9 Procedure calls

2.2.1.1 Expression Statements

An expression statement is the primary method in ANSI C of expressing an assignment of a computed value to a variable. For expression statement *n*, a predecessor of statement *m*, the *defs(n)* set and the slicing criterion determine if an expression statement is included in a slice. Statement *n* is included in a slice for criterion <*m, v*> if statement *n* assigns a value to variable *v*.

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

Table 2-1 presents the data-flow sets used in computing program slices of the program of Figure 2-2. For example, suppose we want to know how the value of the variable **sweet** printed at line 25 was 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 so, the criterion would be $S_{<18,sweet>}$. Applying this criterion generates a sequence of criteria:

$$S_{<18,sweet>} = S_{<17,sweet>} = \ldots = S_{<9,\ sweet>} = \{8\} \cup S_{<8,\ red>} \cup S_{<8,green>}$$

Nodes 9 through 18 do not assign a value to **sweet** and are not included in the slice. Node 8 assigns a value to **sweet** based on **red** and **green** and therefore node 8 (line 13) is included in the slice along with slices on **red** and **green** at node 8. The slice on **red** consists of nodes 7 and 3; the slice on **green** consists of node 5. The slice is now complete except for some syntactic dependencies (nodes 1, 2 and 20) that are captured by the *requires set*, explained in section 2.2.1.2. The nodes included in the slice are marked with an asterisk in Table 2-1.

Line	Statement	Node	Succ	Req	Defs	Refs
1	main()	1	2	17	--	--
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	i	--
19	}	14	15	17	i	--
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 25	printf("%d %d %d %d\n", sweet, sour, salty, bitter);	18	19	2	--	sweet, sour salty, bitter
26	exit(0);	19	--	2	--	--
27	}	20*	--	--	--	--

Table 2-1: Slicing Example Data-Flow Set

```
1  main()
2  {
3      int red, green, blue, yellow;
4      int sweet,sour,salty,bitter;
5      int i;
6
7      red = 1;
8      blue = 5;
9      green = 8;
10     yellow = 2;
11
12     red = 2*red;
13     sweet = red*green;
14     sour = 0;
15     i = 0;
16     while ( i < red) {
17         sour = sour + green;
18         i = i + 1;
19     }
20     salty = blue + yellow;
21     yellow = sour + 1;
22     bitter = yellow + green;
23
24     printf ("%d %d %d %d\n",
25         sweet,sour,salty,bitter);
26     exit(0);
27 }
```

Figure 2-2: Slicing Examples Program

2.2.1.2 Compound Control Statements

A compound control statement is a statement that has a condition directly controlling the execution of another statement (possibly also a compound statement). Control statements such as **if, switch, while, for,** and **do ... while** should be included in a program slice whenever any statement governed by the control statement is included in a slice. When control statement *k* is added to a program slice, the slice on the criterion *<k, refs(k)>* is added to the original slice computation.

A requires set is maintained for each statement to specify an enclosing control statement or to specify other statements and tokens that should always be included with the statement in a slice.

The rules for representing C statements as flow-graph nodes and for specifying requires sets are as follows:

1. A statement that is composed of noncontiguous tokens is divided into two or more data-flow nodes such that each group of contiguous tokens is one or more nodes. Examples are the matching braces of a compound statement and the **do ... while** statement.

2. An additional data-flow node is used to represent each C prefix (**++x**), postfix (**x++**) or comma (**x+y, z**) operator in an expression. A conditional operator uses three additional data-flow nodes.

3. Any compound statement that is represented with more than one data-flow node has one node designated for inclusion in requires sets. Any node controlled by the compound statement references the designated node in its requires set. The other nodes of the compound statement are referenced in the requires set of the designated node. This strategy reduces the size of the requires sets.

The rule for slicing expression statements with $v \in defs(n)$ given in section 2.2.1.1 is actually a special case of an empty requires set from the following rule:

$$S_{<m,v>} = \{n\} \cup \left(\bigcup_{x \in refs(n)} S_{<n,x>} \right) \cup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in req(n)} S_{<k,y>} \right)$$

The above rule states that when $v \in defs(n)$ add the following to the slice:

- statement n,
- the slice on each member of $refs(n)$ at statement n, and
- for each statement k, a member of $requires(n)$, slice on each variable referenced in statement k.

2.2.1.3 Structure Variables

A slice on a structure variable is a slice on each member of the structure.

2.2.1.4 Indirect Assignment by Pointer

Pointers interact with slices both by indirect assignments and by indirect references. To determine if an expression statement with an indirect assignment should be included in a slice, every possible location to which a pointer could be pointing must be known. This is often complicated by using more than one level of indirection. Figure 2-3 shows the pointer state for the variable *A* to three levels of indirection for the program fragment in Figure 2-4. $P_1(n,v)$ is the set of variables to which v might point (dereference to). $P_k(n,v)$ is the set of variables to which $*\cdots*v$ (where there are k *'s) might point. Note that $P_0(n,v) = v$ and for $i > 0$, $P_i(n,v) = \{x \mid x \in P_1(n,y) \; \forall \; y \in P_{i-1}(n,v)\}$.

Consider the following statement with an assignment through k levels of indirection.

$$\underbrace{\text{n: } * \ldots *}_{\text{k *'s}} \text{ A = } \ldots$$

If a statement is included in a slice due to an indirect assignment, then all intermediate indirect references must be sliced on and unioned with the slice. The function $R_{i,k}(n,v,x)$ returns the set of intermediate pointers that might be used at level i of indirection for an assignment at statement n of k levels of indirection through pointer variable x to variable v. The R function prunes away indirect pointers that are not relevant to the slicing criterion. At a given level of indirection, say i, $P_i(n,v)$ is the set of variables that v might point to. The only members of $P_i(n,v)$ that are relevant to the slice are the pointers that might dereference to v after k-i levels of indirection. Figure 2-5 shows the pruned sets of indirect pointers for the criteria $S_{<n,W>}$ and $S_{<n,Z>}$.

$$R_{i,k}(n,v,x) = \{r \mid r \in P_i(n,x) \& v \in P_{k-i}(n,r)\}$$

For example, there are two sets of intermediate pointers, $R_{1,3}(n,W,A)$ and $R_{2,3}(n,W,A)$ required for pruning Figure 2-5 for W. $R_{1,3}(n,W,A)=(\{B,D\})$ is the subset of pointer variables from *A that can dereference to W by two levels of indirection. $R_{1,3}(n,W,A)$ includes B (a member of $P_1(n,A)$) since W is a member of $P_2(n,B)$. D is also included in $R_{1,3}(n,W,A)$ since $W \in P_2(n,D)$, but C is not a member since $W \notin P_2(n,C)$. $R_{2,3}(n,W,A)=(\{E,I,J\})$ is the set of variables that dereference to W and can be reached by dereferencing A twice.

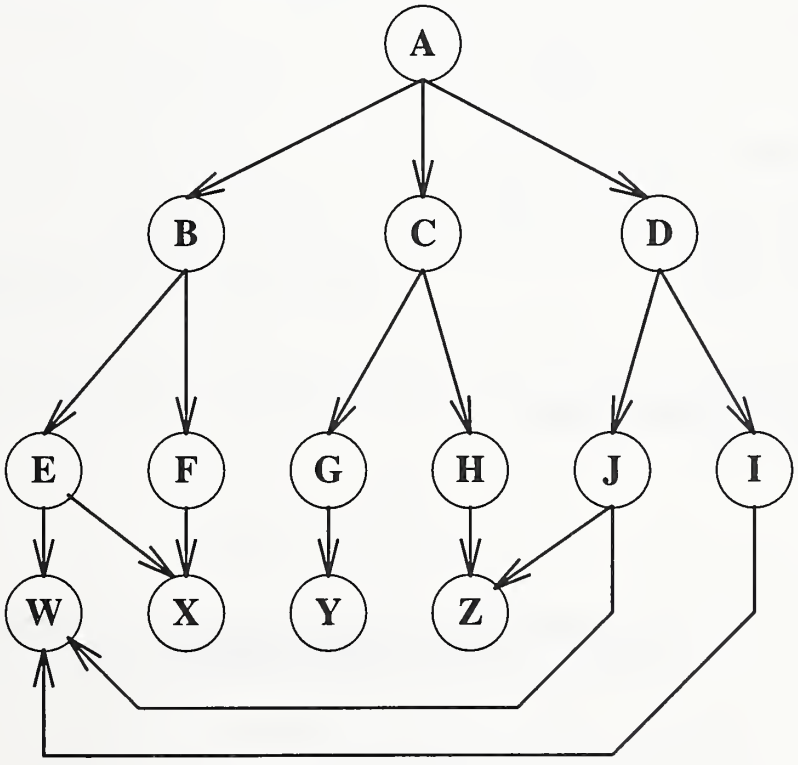

Figure 2-3: Pointer State For ***A

For slicing criterion $S_{<m,x>}$ where $n \in pred(m)$, and $(a,k) \in idefs(n) \& x \in P_k(n,a)$ then statement n should be included in the slice unioned with slices on variables referenced, the pruned pointer state and the original criterion variable if the pointer could point to more than one variable.

$$S_{<m,x>} = \begin{cases} \{n\} & \\ \cup\, S_{<n,v>} & \forall\ v \in refs(n) \\ \cup\, S_{<n,x>} & if\ |\ P_k(n,A)\ |\ >\ 1,\ \forall\ (a,k) \in idefs(n) \\ \cup\, S_{<n,y>} & \forall\ y \in R_{i,k}(n,a,x)\ \forall\ i,\ 0 < i < k,\ \forall\ (a,k) \in idefs(n) \end{cases}$$

```
/* integers: W X Y Z
   pointers to integers: E F G H I J
   pointers to pointer to integer: B C D
   pointer to pointer to pointer to integer: A

   cond() is some condition that is true or false

   K ? M : N; ANSI C conditional expression
   evaluate M if K is true, otherwise evaluate N
*/
...
  int ***A,**B,**C,**D,*E,*F,*G,*H,*I,*J;
  int W,X,Y,Z;
...
  A = cond() ? (cond() ? &B : &C) : &D;
...
  B = cond() ? &E : &F;
  C = cond() ? &G : &H;
  D = cond() ? &I : &J;
...
  E = cond() ? &W : &X;
  F = &X; G = &Y; H = &Z; I = &W;
  J = cond() ? &W : &Z;
...

  n: ***A = ...
```

Figure 2-4: Pointer Code Fragment

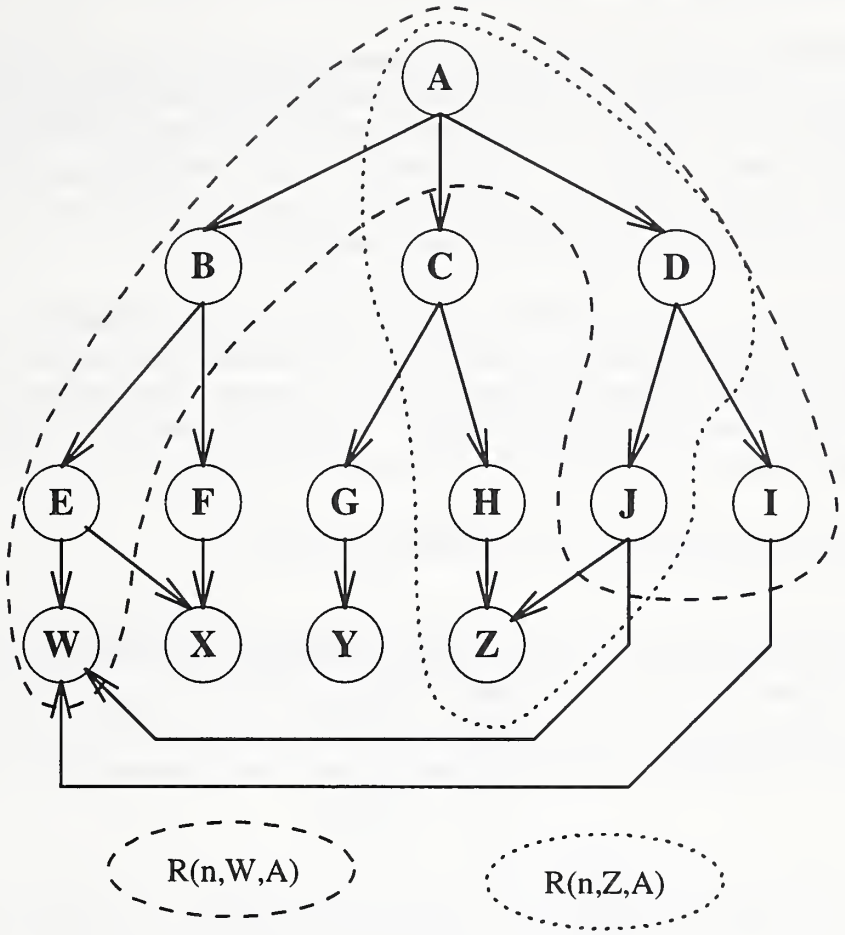

Figure 2-5: Pruned Pointer State For ***A

2.2.1.5 Indirect Reference by Pointer

If a statement with an indirect reference is included in a slice, each variable that might actually be referenced must be sliced on too. Consider a *k* level indirect reference such as:

$$\text{n:A= . . . } \overbrace{* \text{ . . . } *}^{\text{k *'s}} \text{ B . . .}$$

For slicing criterion: $S_{<m,A>}$ *where* $n \in pred(m)$, and statement *n* is included in the slice then the following should also be unioned to the slice:

$$S_{<n,v>} \; \forall \; v \in P_i(n,b) \forall i, \; 1 \leq i \leq k, \; \forall \; (b,k) \in irefs(n)$$

2.2.1.6 Dynamic Structures

Dynamic structures are created by obtaining a contiguous block of storage allocated to a program by the operating system. Since **unravel** is a static (not run time) analysis tool, an exact analysis of dynamic storage is not possible. The **unravel** solution is to assume that any assignment to a dynamic object is also a reference to the dynamic object. This reflects the possibility that although one object instance may be changed by some statement, other instances of the object are unchanged and may still be relevant to some computation.

Each storage allocation statement creates a pseudo-variable for the allocated block of memory that might contain a scaler, an array or a structure. It is assumed that storage is allocated more than once and the pseudo-variable represents several instances. Each pseudo-variable has an implicit type attribute that is determined by usage. If the pseudo-variable is used as a structure, additional pseudo-variables are created for each structure field used.

2.2.1.7 References to Structure Members by Pointer

A reference such as *n: y = · · · v->a · · ·* to a field of a structure when statement *n* is being included in a slice generates slices on the variable *v* and on the accessed field of each structure (each variable returned by *P(n,v)* is assumed to be a structure). A mapping from a *(variable, field)* pair to a structure member variable, *s=F(v,f)*, must be maintained. This gives the following modification to the slicing algorithm for references to structure members by pointer (v->a):

$$S_{<m,X>} = \begin{cases} \{n\} & \\ \vdots & \\ \bigcup S_{<n,v>} & v, \textit{ the pointer variable} \\ \bigcup S_{<n,x.a>} & \forall\ x \in P(n,v), \textit{ the member } a \end{cases}$$

A *pointer chain expression* is defined as a pointer variable followed by one or more structure field names separated by the token "->".

n: y = · · · v->f_1-> · · · ->f_i-> · · · ->f_k · · ·

To generalize to an arbitrary reference through a pointer chain, the following three functions are useful:

1. **crefs(n)** is the set of pointer chain expressions in statement *n*.

2. **clength(n,x)** is the length of (number of fields referenced in) pointer chain expression x.

3. **field(n,x,i)** is the i^{th} member of pointer chain expression *x* in statement *n*.

When a statement containing a pointer chain expression is included in a slice, in addition to generating slicing criteria for all variables the entire chain might reference, criteria must be

generated for each intermediate link in the pointer chain. For example, consider the following **typedef**s and **struct**s:

```
typedef struct alpha alpha_rec, *alpha_ptr;
typedef struct beta beta_rec, *beta_ptr;
struct alpha {
    int      value;
    beta_ptr b;
    };
struct beta {
    int      c;
    };
    alpha_ptr a;
    alpha_rec r,s,t,u;
    beta_rec  w,x,y,z;
    . . .
    n: . . .  a->b->c . . .
```

Suppose that at statement n the pointer state is such that:

$$\begin{aligned} P(n,a) &= \{s,t\} \\ P(n,r.b) &= \{w,x,z\} \\ P(n,s.b) &= \{w,y\} \\ P(n,t.b) &= \{y,z\} \\ P(n,u.b) &= \{w,x,y\} \end{aligned}$$

If statement n is included in a slice, then slicing criteria based on `a->b->c` must be generated for any variable that `a->b->c` might designate to account for variables that the entire chain might reference. Since a points to s and $s.b$ points to w then one of the criteria is $S_{<n,w.c>}$. The other criteria are for the variables $y.c$ and $z.c$. This example chain has one intermediate link, `a->b` that must also be accounted for by generating criteria on $s.b$ and $s.t$.

Generating criteria for an arbitrary chain generalizes as follows:

$$S_{<m,X>} = \begin{cases} \{n\} \\ \vdots \\ \bigcup S_{<n,t>} & \forall\ t \in\ w_i,\ w_i = F(z,f_i)\ \forall\ z \in P(n,r),\ \forall\ r \in\ w_{i-1} \\ & f_i = field(n,c,i)\ \forall\ i \ni 1 \le i \le k, \\ & where\ w_0 = field(n,c,0),\ k = clength(n,c),\ \forall\ c \in\ crefs(n) \end{cases}$$

2.2.1.8 Assignment to Structure Members by Pointer

A statement that assigns a value to a structure member by a pointer should be included in a slice if there is at least one variable in common between the active set and the set of variables to which the pointer chain points.

2.2.1.9 Procedure Calls

When procedure calls are included in a slice, the called procedure needs to be sliced. Procedure calls are included in a slice for the following reasons:

1. The procedure call assigns a value to a variable in the active set at the call site.

2. The call site is part of a statement included in a slice and a value returned by the called procedure is used in the statement at the call site.

3. The procedure is a member of the call tree of the procedure containing the slicing criterion.

To slice procedures lower in the call tree, the following steps are required:

1. Introduce statements before the procedure entry to assign each actual parameter to the corresponding formal parameter.

2. Slice the procedure at the last node using the variables in the active set of the procedure call.

2.2.2 Algorithm Limitations

There are a number of limitations to the current slicing algorithm:

- If the program to be analyzed is not ANSI C, the **parser** is not able to produce the various data structures needed for program slicing. This is likely to happen if a C compiler is used that has implemented extensions to the ANSI C specification. This limitation is the most likely to cause **unravel** to fail to analyze a program. The only recourse is to modify the program being analyzed to remove the non-standard code without changing the program semantics.

- Asynchronous UNIX system calls such as **signal** and **fork** are beyond the state of the art of program slicing and are ignored.

- Variables declared as an ANSI C **union** are treated as separate variables.

- Pointers to functions are ignored.

- The statements **goto**, **break** and **continue** are ignored.

- Arrays accessed by pointer instead of indexing are not recognized as arrays and therefore an assignment to a single array element is treated as an assignment to the entire array.

- Aliasing, multiple variables referencing the same memory location, is treated as distinct variables rather than a single variable.

- Functions must have a fixed number of formal parameters. The **varargs** and **stdargs** variable length parameter lists are ignored.

2.2.3 Assumptions About Users

The users of **unravel** are assumed to be knowledgeable about computers and ANSI C, but they may not be familiar with UNIX, POSIX or program slicing.

2.2.4 General Constraints and Assumptions

- **Unravel** assumes a UNIX environment with an ANSI C compiler, the X Window System, Version 11, Release 5 and the MIT Athena widget library.

- Programs to be analyzed by **unravel** are assumed to be syntactically correct ANSI C with no language extensions.

- Programs are assumed to use the standard ANSI C library[1] and no other libraries.

- Programs are assumed to be a single process, i.e., no multitasking, no asynchronous code, no *forks* and no *signals*.

- An ANSI C preprocessor is assumed to exist to process all # commands (e.g., **#define** and **#include**) and to remove comments. The ANSI C preprocessor should insert **#line** directives to indicate location and file of included statements from **#include** directives.

- Files included (`#include header.h`) by the ANSI C preprocessor contain only ANSI C preprocessor directives or ANSI C declarations. Files included by the ANSI C preprocessor do not contain procedure bodies or executable statements.

3 Unravel Requirements

The functional requirements are divided into *scanner, parser, LIF, slicing algorithm, system map, linker, user interface, and help system* requirements. The function of the *slicer* is to extract program slices for the *user interface* to display. The *slicer* depends on information about the program collected by the *scanner* and *parser* that is bound together by the *linker.*

Requirements that would enhance **unravel** (i.e., *nice to have*) but would not make **unravel** unacceptable if absent are labeled as *desirable.* Requirements that may not be worthwhile to add to **unravel** or that might be added to **unravel** if time permits are labeled *optional.* All unlabeled requirements are mandatory.

3.1 Scanner Requirements

Input is ANSI C source program files. Output is a token stream to the **parser**.

R1.1 Recognize all ANSI C keywords: **auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while**.

R1.2 Recognize all ANSI C operators: `>>= <<= += -= *= /= %= &= ^= |= >> << ++ -- -> && || <= =+ == !=+ ... , = [ ] . & ! ~ - + * / % < > ^ | ? } ; { ( ) :.`

R1.3 Recognize ANSI C numeric constants with optional suffix: decimal, octal, hexadecimal and float.

R1.4 Recognize ANSI C character constants with optional prefix.

R1.5 Recognize ANSI C octal escapes, hex escapes and character escapes: `\a \b \f \n \r \t \v \' \" \? \\`.

R1.6 Recognize ANSI C string constants with optional prefix.

R1.7 Merge multiple string constants into a single constant.

R1.8 Recognize ANSI C identifiers.

R1.9 Recognize ANSI C programmer defined *type names.*

R1.10 Remove white space (blank, tab and new line characters).

R1.11 Each character not recognized as a keyword, identifier, type name, constant, operator or string is passed to the **parser** as a token.

R1.12 (Desirable) Be able to optionally trace each token recognized.

3.2 Parser Requirements

Input is a string of tokens from the scanner. Output is a language independent representation of the program.

R2.1 Recognize ANSI C grammar.

R2.2 If the **parser** detects a syntax error, report the file and line within the file containing the error.

R2.3 Keep mapping of source code line and column to statement.

R2.4 Assign a statement number for each statement.

R2.5 Generate successor relation to add a statement to a statement list.

R2.6 Generate branching flow for the following statements: **if, else, while, do, for, switch, case, return**.

R2.7 Generate requires sets of syntactic components for the following statements: **if, else, while, do, for, switch, case, break, continue, return**.

R2.8 Generate a statement node for each statement label.

R2.9 Generate flow to target label of **goto**.

R2.10 **Switch** should branch to default case.

R2.11 **Continue** should branch to bottom of loop.

R2.12 **Break** should exit loop or case.

R2.13 Assign each identifier a unique id number.

R2.14 There is a separate name space for each of: 1) statement labels, 2) tags (struct tags, union tags, and enum tags), 3) members of each structure or union, and 4) other identifiers.

R2.15 Recognize type definitions.

R2.16 Keep the type of each identifier.

R2.17 Recognize the four kinds of scope: function (statement labels), block (within braces), file and function prototype.

R2.18 Create stacked local symbol table on block or procedure entry.

R2.19 Remove top symbol table on block or procedure exit.

R2.20 Recognize external declarations.

R2.21 Recognize references to external objects.

R2.22 Generate *refs* sets for each variable referenced.

R2.23 Generate *defs* sets for each variable assigned.

R2.24 Generate *defs* and *refs* for each compound assignment.

R2.25 Prefix operators generate a node for each operator that is before the current statement.

R2.26 Postfix operators generate a node for each operator that is after the current statement.

R2.27 Structure assignment generates *refs* and *defs* for the members.

R2.28 Assignment to a structure member is a define of the member and a reference and define to the structure.

R2.29 Assignment to an array element is a reference and define to the array.

R2.30 Resolve references of the form *ref_chain . identifier* or *ref_chain -> identifier*. For a structure or pointer to structure reference find the type definition for the ref_chain structure type to resolve the identifier.

R2.31 Identify operations that may create an alias to an area of memory.

R2.32 (Optional) Recognize that union members are aliases to the same area of memory.

R2.33 (Desirable) Identify C preprocessor token substitutions for identification of defines used.

R2.34 (Desirable) ANSI C syntax errors should be tolerated and noted but should not prevent the analyzer from continuing.

R2.35 (Optional) Try to determine if code to be analyzed contains common extensions to ANSI C that can be ignored for slicing (e.g., *far* pointers).

3.3 Language Independent Format Requirements

The Language Independent Format (LIF) is the output of the **analyzer** and, with some modifications from the **linker**, input to the **slicer**. The LIF file should capture all information necessary to compute a program slice for the ANSI C source program input to the analyzer.

R3.1 LIF should identify procedure start.

R3.2 LIF should identify a procedure end.

R3.3 LIF should identify formal procedure parameters.

R3.4 LIF should identify procedure calls.

R3.5 LIF should identify actual procedure parameters.

R3.6 LIF should identify a flow graph node for each arithmetic expression.

R3.7 LIF should identify flow graph nodes necessary to represent each statement.

R3.8 LIF should identify the flow graph nodes that are direct successors of each flow graph node.

R3.9 LIF should identify each program variable.

R3.10 LIF should identify variables that are **extern, pointer, static** or **array**.

R3.11 LIF should identify each program variable referenced (used) at each statement.

R3.12 LIF should identify each program variable assigned at each statement.

R3.13 LIF should identify level of indirection for references and assignments to pointer variables.

R3.14 LIF should identify **return** statements.

R3.15 LIF should identify **goto** statements.

R3.16 LIF should maintain a mapping between each flow graph node and source statements.

3.4 Slicing Algorithm Requirements

The following requirements refer to the slicing algorithm described in section 2.2.1. The background for requirements R4.1 - R4.4 is sections 2.2.1.1 and 2.2.1.2 (Expression Statements and Compound Control Statements), R4.5 - R4.10 refer to sections 2.2.1.3 - 2.2.1.8 (pointer expressions) and R4.11 - R4.14 refer to section 2.2.1.9 (procedure calls). The output of the slicing algorithm is the set of statements relevant to the computation of the variables in a given slicing criterion. The input to the slicing algorithm is:

- LIF representation of an ANSI C program.
- slicing criterion

R4.1 Given expression statement n and statement m, a successor to n, and slicing criterion $S_{<m,v>}$, for variable v the slicing algorithm includes expression statement n if n assigns a value to v.

R4.2 If an *expression statement* is included in the slice by the slicing algorithm, the following slicing criteria are generated: $S_{<n,x>}\ \forall\ x \in refs(n)$.

R4.3 Given statement n and statement m, a successor to n, and slicing criterion $S_{<m,v>}$, for variable v if n does not assign a value to v, (the slicing algorithm does not include expression statement n in the slice) then the following slicing criterion is generated: $S_{<n,v>}$

R4.4 If a statement n is included in a slice then all statements in the set requires(n) are included in the slice with the following generated criteria: $S_{<y,x>}\ \forall\ x \in refs(y)\ \forall\ y \in requires(n)$.

R4.5 The *pointer state function* $P_k(n,v)$ is a function that returns the set of addresses to which * · · · *v (where there are k *'s) could point.

R4.6 Given expression statement n with a k level indirect assignment through variable a, and statement m, a successor to n, and slicing criterion $S_{<m,v>}$, for variable v, the slicing algorithm includes expression statement n if $v \in P_k(n,a)$.

R4.7 If statement n with a k level indirect assignment through variable a, is included in a slice for slicing criterion $S_{<m,v>}$, then the following criteria are generated (to capture relevant assignments of addresses at each level of indirection):

$S_{<n,y>}\ \forall\ y \in R_{i,k}(n,v,a)\ \forall\ i,\ 0 < i < k,\ \forall(a,k) \in idefs(n)$

Where: $R_{i,k}(n,v,a) = \{\ r \mid r \in P_i(n,a) \& v \in P_{k-i}(n,r)\}$

R4.8 If statement n is included in a slice the following criteria are generated:

$$S_{<n,v>} \ \forall v \in P_i(n,b) \forall i,\ 1 \le i \le k,\ \forall (b,k) \in irefs(n)$$

R4.9 If statement n is included in a slice and the set crefs(n) is not empty then the following criteria are generated for each pointer chain, $v \rightarrow f_1 \rightarrow \cdots \rightarrow f_k$ in crefs(n):

$$S_{<m,X>} = \begin{cases} \{n\} \\ \vdots \\ \cup S_{<n,r>} & \forall\ t \in\ w_i,\ w_i = F(z,f_i)\ \forall\ z \in\ P(n,r),\ \forall\ r \in\ w_{i-1} \\ & f_i = field(n,c,i)\ \forall\ i \ni 1 \le i \le k, \\ & where\ w_0 = field(n,c,0),\ k = clength(n,c),\ \forall\ c \in\ crefs(n) \end{cases}$$

R4.10 For a slicing criterion $S_{<m,x.fk>}$ and statement n where $m = succ(n)$, with cdefs(n) not empty, n: $v \rightarrow f_1 \rightarrow \cdots \rightarrow f_k = \cdots$ if $x.f_k \in w_{k-1}$ where:

$$w_i = \{F(z,f_i)\ \forall\ z \in\ P(n,r)\ and\ r \in\ w_{i-1}\}$$
$$where\ w_0 = \{v\}$$

then include statement n in the slice and generate the following criteria:

$$S_{<n,t>} \quad \forall\ t \in\ w_i,\ for\ 0 \le i < k$$
$$w_i = \{F(z,f_i)\ \forall\ z \in\ P(n,r)\ and\ r \in\ w_{i-1}$$
$$where\ w_0 = \{v\}$$

R4.11 Given the set of procedure call sites, where a call site is represented as an ordered triple *(statement, calling procedure, called procedure)*. Let p_0 be the procedure containing the user supplied slicing criterion. Let C be the set of procedure call sites.

Let $Q = \{\forall (n_0,x_0,y_0) \in C \ni y_0 = p_0\ or\ \exists (n_1,y_0,y_1) \in Q\ \}$

The set Q represents the tree of procedure calls that invoked P_0, the procedure containing the slicing criterion. The statements containing each call site in Q are included in the slice and the following criteria are generated:

$S_{<n,t>}$ $\forall\ (n,p,q) \in\ Q$

$\forall\ t$ *actual parameter to* q *corresponding to a formal parameter of* q *or a non-local variable, in the active set of node* 0 *of* q

R4.12 If a statement, n, is included in a slice and statement n includes a function call, f, that returns a value, then the following slicing criteria are generated:

$S_{<r,v>}$ $\forall x \in refs(r)\ \forall\ r$ ***return*** *statement in* f

$S_{<n,a>}$ $\forall a \in formal\_to\_actual(v)\ \forall v \in f.begin$

$S_{<n,g>}$ $\forall g$ *(global variable)* $\in f.begin$

where f.begin is the set of generated criteria at statement 0 in procedure f

R4.13 If a statement, n, includes a function call, $f(a_1,a_2,\cdots)$, with slicing criteria $S_{<m,x>}$, generate the criterion: $S_{<f.last,x>}$ where *f.last* is the last statement of the procedure, and include statement n in the slice if the generated criterion includes any statements in the slice.

R4.14 If a statement, n, includes a function call, $f(a_1,a_2,\cdots)$, with slicing criterion $S_{<m,x>}$, where $\&x$ is one of the a_i, generate the criterion: $S_{<f.last,x>}$ where *f.last* is the last statement of the procedure. If the generated criterion includes any statements in the slice, then include statement n in the slice and generate the criterion $S_{<n,ai>}$ where f.last is the end of the procedure.

3.5 System Map Requirements

The link component operates in two parts. The first part, **map**, identifies for each program, in the current directory and its constituent files and then saves the information in a file named **SYSTEM**. The goal of **map** is to identify the files containing the procedure definitions of all procedures required to link each program in a given file system directory. A program is defined to be a **main** procedure and the set of procedures that must be defined for the program to *link*. This set of procedures is called the *required procedure set*, RPS, and the set of files containing the definitions of the RPS is called the *required file set*, RFS.

The input to **map** is the set of LIF files and T files produced by the **parser**. The output of **map**, a file named **SYSTEM**, defines the RFS of each program for **slink** to link the **.LIF**, **.H** and **.T** files together. The **SYSTEM** file contains a list of programs, identified by the file containing the **main** procedure for the program. For each program there is a list of files (the RFS) that contain the definitions of procedures called by the program either directly or by a sequence of intermediate calls. Any procedures that are not defined within the directory are classified as library functions.

If a procedure used by the program (i.e., in the RPS) is externally defined in more than one file, **map** fails for the given program since **map** cannot determine the file that should be used. The procedure is identified as ambiguous in the **SYSTEM** file entry for any programs that try to use that procedure. If **map** fails, the list of required files contains a partial list of the RFS and a partial list of library procedures corresponding to the stage of computation where **map** discovered a called procedure defined in more than one file. The **SYSTEM** file entry for this **main** must be built manually.

Map is limited to assuming that a **main** procedure is used for a single program.

R5.1 By default, **map** operates on the current directory.

R5.2 A directory may optionally be given on the command line.

R5.3 **Map** should make a *program entry* in the SYSTEM file for each **main** procedure found in a LIF file.

R5.4 A *program entry* consists of three sections: required source files, library functions and ambiguous functions.

R5.5 The source file containing the **main** procedure should be listed in the required source files.

R5.6 The required file set (RFS) for program M is the set of source files that contain the definitions of procedures (but not library procedures) called somewhere in the program. In addition to the source file containing the **main** procedure, a source file, F, that meets the following conditions should be in the required source files for program M.

1. P is a member of RPS of M and defined **extern** only in F.

2. F does not contain a **main** procedure.

R5.7 Procedure P is included in the library procedure section of program M if there is at least one call to P that meets the following conditions:

1. P is a member of RPS of M.

2. P is not defined **extern** in any RSF file of M.

3. There is no static definition of P visible at the call site.

R5.8 Procedure P is included in the ambiguous procedure section of program M if the following conditions are met:

1. P is a member of the RPS of M.

2. P is defined **extern** in at least two files, G and H.

3. Neither G nor H contain a **main** procedure.

3.6 Linker Requirements

After **map** has produced a **SYSTEM** file, the second part of the linker, **slink**, uses the **SYSTEM** file to merge data-flow information from the **.LIF**, **.T**, and **.H** files created from separate source files into a single **.LINK** file and a single **.K** file. Items such as object addresses, chains of pointers to structure fields, global variables or procedures must be resolved by the linker. The following files are input to the linker:

SYSTEM The **SYSTEM** file identifies all source files required for each **main** program in the current directory. This file is an output of the **map** component.

.LIF There is one **.LIF** file for each source file. All the **.LIF** files are combined into one **.LINK** file.

.T There is one **.T** file for each source file. All the **.T** files are combined with the **.H** files into one **.K** file.

.H There is one **.H** file for each source file. All the **.H** files are combined with the **.T** files into one **.K** file.

R6.1 Each of the following records (see sec. 4.2) in a **.LIF** file should be passed unchanged to the **.LINK** file:

PROC_END	FORMAL_ID	ACTUAL_SEP
CALL_END	RETURN	GOTO
SUCC	REQUIRES	SOURCE_MAP
LOCAL_ID		

R6.2 Each named procedure is given a unique procedure identifier, *pid*. If two **.T** files each refer to a procedure with the same name, the procedures are assigned the same pid if neither one is **static**. If at least one is **static**, then each procedure is assigned a different pid.

R6.3 For each unique pid the **.K** file should contain one line with the following information: pid, entry statement, exit statement, number of local variables, flag indicating **static** or **extern** and the procedure name.

R6.4 A procedure that is not defined in any of the source files should have an entry statement of -1.

R6.5 Each PROC_START record input from a **.LIF** file yields a PROC_START record in the **.LINK** file with the pid field updated to contain the unique pid assigned to the procedure.

R6.6 Each CALL_START record input from a **.LIF** file yields a CALL_START record in the output **.LINK** file with the unique pid corresponding to the pid in the **.LIF** file.

R6.7 REF or DEF records that refer to an identifier that has a LOCAL_ID record with an **X** flag are replaced in the **.LINK** file with a GREF or GDEF record referring to the corresponding global id. The node field stays the same.

R6.8 Other REF or DEF records are passed to the **.LINK** file unchanged.

R6.9 Each GLOBAL_ID record with a unique name not found in any other LIF file is assigned a unique id number and a GLOBAL_ID record using the new id number is output to the **.LINK** file. All other fields in the GLOBAL_ID record are unchanged.

R6.10 For a GLOBAL_ID record that does not have a unique name, but is flagged **static**, a new unique id is assigned and a GLOBAL_ID record is output for each **static** id.

R6.11 For a GLOBAL_ID record name that appears in multiple records, the instances that are not flagged **static** are considered to refer to the same object and a single GLOBAL_ID record with a unique id is generated to the **.LINK** file.

R6.12 GREFS, GDEFS, GCHAIN, ADDRESS and STRUCT records are passed from the **.LIF** files to the **.LINK** files with the id field updated to the assigned unique id.

R6.13 Duplicate chains should be eliminated and a unique new chain id assigned. A single CHAIN or GCHAIN should be generated to the **.LINK** file.

R6.14 CREF, CDEF and FIELD records should be updated with the new chain id in the **.LINK** file.

R6.15 Duplicate ADDRESS records should be eliminated and reassigned unique address ids for the **.LINK** file.

R6.16 AREF records should be updated with the new address ids in the **.LINK** file.

R6.17 FILE records should be inserted in the **.LINK** file just before records from the corresponding **.LIF** file are inserted into the **.LINK** file.

R6.18 The **.H** files should be merged together into the **.K** file as *header file groups* consisting of the header file name and a list of variable ids and variables for each variable declared in a header file.

R6.19 A header file group name should appear only once in the **.K** file.

R6.20 The **.K** file should contain the following information for each source file: file id number (starting from 0), number of procedures, number of statements, number of lines, number of characters and file name.

R6.21 The **.K** file should contain a count of the number of each of the following items: global variables, procedures, object addresses, pointer to structure field chains, header file groups and source files.

3.7 User Interface And Help System Requirements

The function of the *user interface* and *help system* is to provide controlled access to **unravel** components by the **unravel** user. The goals of the user interface are to insulate the user from detailed knowledge of the underlying software and hardware, assist the user in saving the results obtained, give the user feedback on progress for lengthy tasks and provide access to additional information on using **unravel**. The user interface uses a mouse driven window system that provides a set of control panels (windows with buttons and other objects) to allow the user to invoke **unravel** components and display the results.

R7.1 The user interface shall display the following information about the current directory: directory name, number of source files, number of ANSI C source files analyzed, number of main programs analyzed, number of main programs linked and number of procedures that appear in more than one file.

R7.2 The user shall be able to change directories.

R7.3 The user interface shall allow the user to select from all source files in the current directory a subset for operations (analyze or clear analysis results).

R7.4 The user shall be able to specify command line options for the *C preprocessor* and the *analyzer*.

R7.5 The user shall be able to invoke the analyzer on the selected set of files (see above).

R7.6 The user shall be able to remove any analysis files created by the analyzer on the selected set of files (see above).

R7.7 The user interface shall display the name of the file currently being analyzed when a set of source files is analyzed.

R7.8 The user interface shall display a summary of analysis results indicating any non-ANSI source files.

R7.9 The user interface shall allow the user to display all messages produced by the analyzer for each analyzed source file.

R7.10 The user interface should update displayed information about source files after analysis of a set of source files is completed.

R7.11 After the analyzer is run on a set of source files, the **map** program shall be run, (**map** identifies main programs).

R7.12 The user shall be able to select a main program for slicing.

R7.13 The user interface shall automatically invoke the linker when a main program is selected.

R7.14 The user shall be able to select a slicing criterion (program variable and location) for slicing from all program variables and statements.

R7.15 The user interface shall be able to display all the source files linked with a selected main program.

R7.16 If the entire program cannot be displayed at one time, the user interface shall display a contiguous block of statements such that any given statement is displayed in at least one block of statements (i.e., every statement can be displayed somehow in a scrollable window).

R7.17 The user interface shall be able to display statements identified for user attention in a manner easily identified by the user (e.g., reverse video or contrasting color).

R7.18 The following statements are identified for user attention:

- Statements in a slice.
- Statements in an operation on two slices.
- Statements marked to indicate the location of a procedure.
- Statements marked to indicate the location of a procedure's call tree.

R7.19 The user interface shall display an indication of progress during the slice computation.

R7.20 Each computed slice shall be saved to a file.

R7.21 The user interface shall allow the user to select and display a saved slice.

R7.22 The user interface shall allow the user to select two previously computed slices for display of the intersection, union and program dice of the two slices.

R7.23 The user interface shall allow the user to halt the computation of a slice and display the partial results.

R7.24 The user interface shall display a visual summary of the set of statements identified for user attention that indicates the approximate set size and statement location relative to the entire program (e.g., an object like a scroll bar with tick marks at the location corresponding to each identified statement).

R7.25 The user interface shall provide for the display of information describing each control panel, the function of each interface object on the control panel and guidance in using the control panel (i.e., a *help button*).

R7.26 The user interface shall always display a brief description of the interface object currently under the mouse pointer.

4 Unravel Design

This section describes the design of **unravel**. The description of procedures and data structures is a high-level abstraction of the actual implementation presented in an informal pseudo-code.

Unravel is divided into three main components: a source code analysis component to collect information necessary for the computation of program slices into a source language independent format; a link component to merge flow information from separate source files; and, an interactive slicing component that the user can use to extract program components and statements to answer questions about the software being examined.

4.1 Analyzer

The analyzer is similar to a compiler with a scanner, a parser and a code generator. The analyzer translates each ANSI C source code file into a language independent format (LIF) file. The UNIX compiler writing tools **lex** and **yacc** handle the scanning and parsing of the source code. The code generator is a collection of semantic action routines, code fragments suitable for insertion as a **case** in a **switch** statement. Each semantic action routine is attached to a **yacc** grammar production and is called to output LIF when a grammar production is recognized (reduced). Figure 4-1 presents the structure of the analyzer. A *main* procedure calls the parser (*yyparse()*), which returns zero if the parse is successful, and one if the parse is unsuccessful. The semantic actions for declarations record information about variables and types in the symbol table and the **.LIF** file. The semantic actions for expressions, statements and external objects using the symbol table and information passed up from grammar productions already recognized also generate entries in the **.LIF** file.

4.1.1 Scanner

The **scanner**, coded in **lex**, is called by the **parser** to read the source code and return tokens to the **parser**. The source code is assumed to have been already processed by an ANSI C preprocessor. The major difficulty for the scanner is to correctly recognize IDENTIFIER and TYPENAME tokens. The problem comes up when a name declared as a TYPENAME in an outer scope is redeclared in an inner scope. The name must be recognized as an IDENTIFIER token in the inner scope when it is redeclared. This is somewhat ambiguous since the context determines if the name is used as a TYPENAME or an IDENTIFIER.

```
read characters until pattern match
set yytext to matching characters
if matched pattern is IDENTIFIER then
    if yytext is found as a typename then
        if typename expected return TYPENAME token
        else return IDENTIFIER token
    else return IDENTIFIER token
else return token found
end if
```

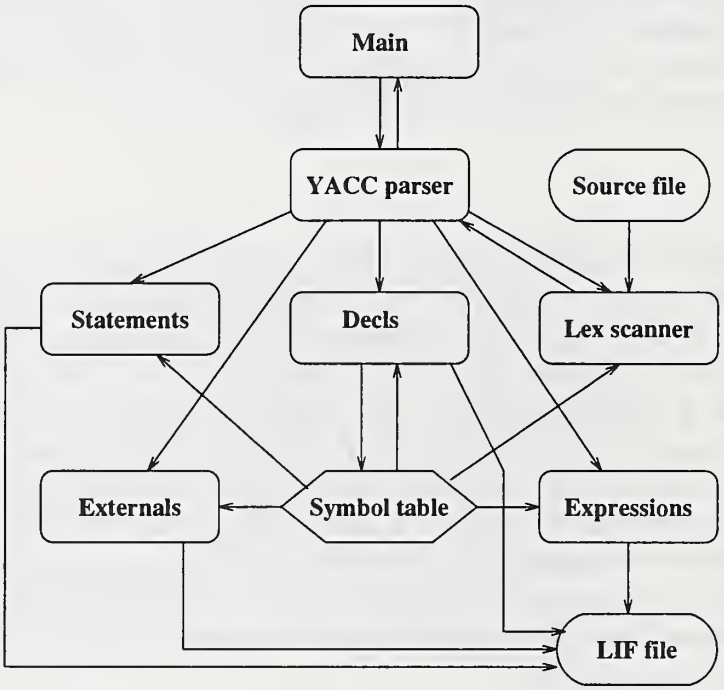

Figure 4-1: Unravel Analyzer Structure Design

4.1.2 Parser

The flow of *yyparse()* is to call *yylex()* (the **lex** scanner) for a token, then either shift the token onto a stack or reduce the stack by a matching grammar production. The **yacc** grammar (given in Appendix B) productions are arranged in four groups: declarations, statements, expressions and external objects.

```
do
    get token from scanner
    if no match then
        shift token to parse stack
    else
        pop parse stack
        reduce production
        switch (production)
            case expression:
                output to LIF: variables referenced
                output to LIF: variables defined
                output to LIF: procedure calls
            case statement:
                output to LIF: flow graph node
            case declaration:
                output to LIF: variables declared
                save declaration in symbol table
            case external objects:
                output to LIF: external variables declared
                output to LIF: procedures defined
                output to LIF: formal procedure parameters
        end switch
    end if
while not EOF
```

4.2 Language Independent Representation

The language independent representation captures the details of the program required for the slicing algorithm to compute program slices. The program representation is contained in several files as shown in Table 4-1.

File	Contents	Produced by . . .
file.c	source code to be examined (source file)	programmer
file.h	declarations for C preprocessor to include	programmer or ANSI C
file.LIF	translation of source code	analyzer
file.T	count of objects in file.LIF	analyzer
file.H	mapping of variable names to header (.h) files	analyzer
SYSTEM	summary of all programs in directory	map
file.LINK	merged LIF from all modules	linker
file.K	merged T and H files	linker

Table 4-1: Unravel System Files

The *language independent format* represents the program as an annotated flow graph of nodes and edges. Nodes are generated to represent semantic or syntactic units of the program that correspond to statements or parts of statements. Edges are of two types, *control flow* and *requires*. A *control flow edge* between two nodes indicates the flow of control from one node to the other. The *requires* edges from a node indicate other nodes that should be included in any slice containing the node the edges are from. The *requires* edges are a general mechanism for specifying control dependence between nodes, pieces of required source code, or other slicing dependencies. The annotations specify location of corresponding source code, variables referenced or assigned and special statements such as **goto** and **return**.

The following subsections describe the format of each file.

4.2.1 File.c

The source code as produced by the programmer.

4.2.2 File.h

The source file is assumed to use the **#include** preprocessor directive to include only header files containing *declarations, typedefs* and *defines*. No procedure bodies should be in a header file.

4.2.3 File.LIF

The general form of the **.LIF** file is a sequence of one line records. Each record is one of the LIF codes followed by a comma separated list of parameters in parenthesis. The LIF codes, found in Appendix A, can be grouped in codes for flow-graph, procedure headers, declarations, expressions, procedure calls and structure fields.

4.2.3.1 Flow-Graph

The following LIF codes are used to specify the flow-graph:

```
#define LIF_REQUIRES    17  /* 17(node,required_node)                    */
#define LIF_SOURCE_MAP 18  /* 18(node,from_ln,from_cl,to_ln,to_cl*/
#define LIF_RETURN      14  /* 14(node,1|0)                              */
#define LIF_GOTO        15  /* 15(node,G|B|C)                            */
#define LIF_SUCC        16  /* 16(from_node,to_node)                     */
```

For producing LIF code, ANSI C source statements are classified as declarative, expression, compound control and branch. Each flow-graph node produced is annotated by `LIF_SOURCE_MAP` to provide a mapping from flow-graph nodes to source code statements.

The declarative statements are declarations and procedure headers. Declarations generate no flow-graph nodes. Procedure headers generate an entry node corresponding to the procedure header and an exit node corresponding to the closing brace of the procedure.

An expression generates one flow node for the expression plus one flow node for each postfix, prefix or comma operator in the statement.

The compound control statements are: **if, switch, while, do ... while, for** and compound statement ({ . . . }). Any nodes directly within the scope of control of a compound control statement specify the control statement with a `LIF_REQUIRES` entry in the LIF file. When control statements are nested, only the next layer of control out from a node is specified with a `LIF_REQUIRES` entry.

The compound statement generates a flow-graph node for the beginning bracket and one for the ending bracket.

An **if** statement without an **else** generates at least two nodes: first a node for the **if**, left parenthesis token and the condition expression, second a node for the right parenthesis to serve as an exit point from the statement. The nodes for the controlled statement must exit through the right parenthesis node. The controlled statement generates a `LIF_REQUIRES` entry for the **if** node. The **if** node requires the parenthesis node.

An if statement with an **else** generates an additional node for the **else**. Nodes of the second controlled statement require the **else** node. The **else** node requires the **if** node. The flow-graph of an **if** statement is presented in Figure 4-2.

A **switch** statement generates two nodes, one for the **switch** token and expression and one for the right parenthesis token. The right parenthesis token is used as an exit point for each **case** in the controlled statement. The controlled statement generates a `LIF_REQUIRES` entry for the **switch** node.

A **while** statement generates two nodes, one for the **while**, left parenthesis and expression and one for the right parenthesis. The right parenthesis node is a successor to the **while** node and the last node of the controlled statement. The controlled statement generates a `LIF_REQUIRES` entry for the **while** node. The **while** node requires the right parenthesis node. The flow-graph of a **while** statement is presented in Figure 4-3.

The **do . . . while** generates three nodes: the **do**, the **while** and condition, and the right parenthesis. The successor of the **do** node is the first node of the controlled statement. The **while** node is the successor of the last node of the controlled statement. The **while** node has two successors: the **do** node and the right parenthesis. The **do** node is required by the controlled statement, and the **do** node requires the **while** node and the right parenthesis.

The **for** statement generates three nodes. The first node contains the **for**, left parenthesis, and the initialization. The second node encompasses the test expression, and the third node contains the increment and the right parenthesis. The test is a successor of the **for** and initialization. The statement is a successor of the test, and the increment is a successor of the statement. The **for** and the initialization expression require the test, the increment and the statement. The statement

and increment are both required by the test and the right parenthesis requires the **for**. The flow-graph of a **for** statement is represented by Figure 4-4.

Nodes corresponding to **return** statements are identified by `LIF_RETURN`. The second field of the `LIF_RETURN` indicates a **return** with expression by **1** and a **return** without expression by **0**. The statements **goto, break** and **continue** are identified by a corresponding `G B` or `C` code in a `LIF_GOTO` entry.

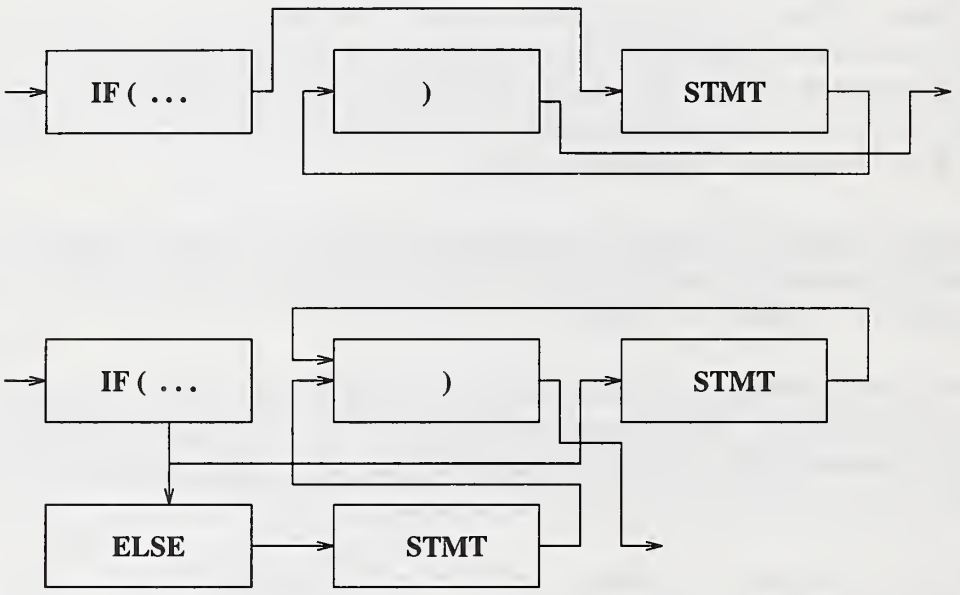

Figure 4-2: IF Statement Control Flow

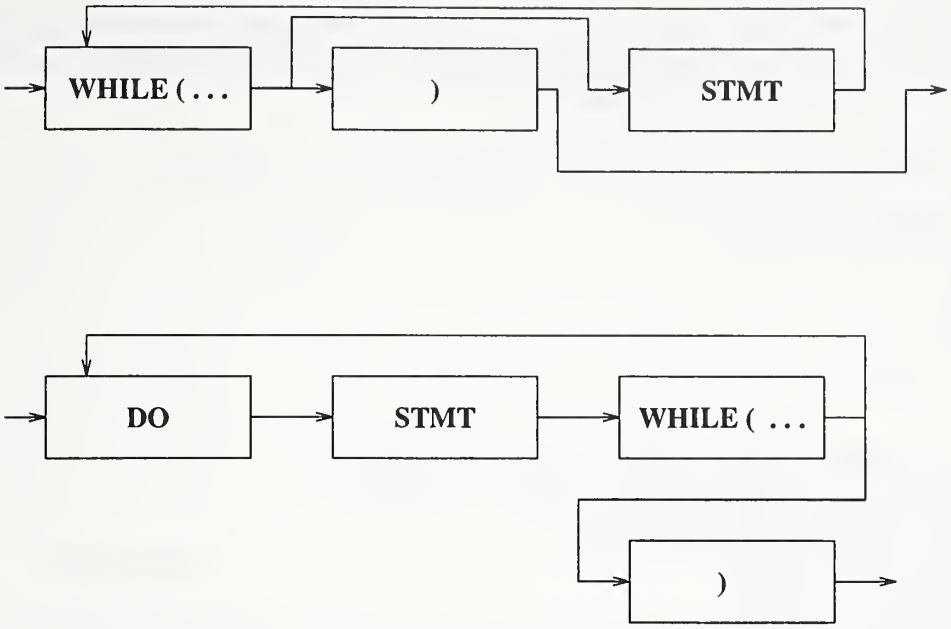

Figure 4-3: WHILE Statement Control Flow

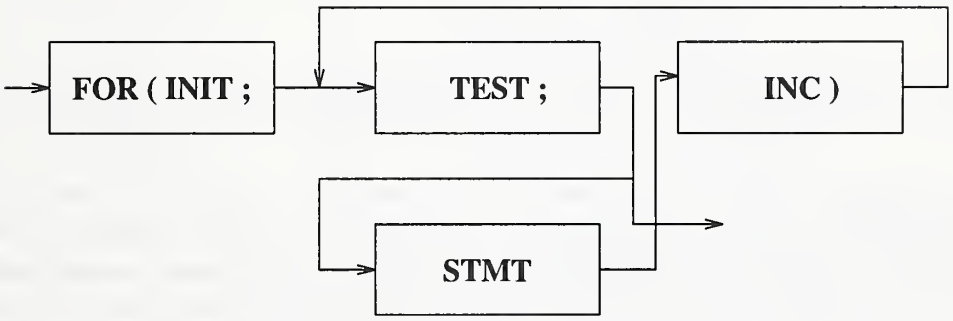

Figure 4-4: FOR Statement Control Flow

4.2.3.2 Procedure Headers

A procedure header, **function_name (f1,f2,...fn)**, uses the following LIF codes:

```
# define LIF_PROC_START   1   /*  1(node,pid,name)                */
# define LIF_PROC_END     2   /*  2(node[,S][,R])                 */
# define LIF_FORMAL_ID    3   /*  3(id,name[,A][,P])              */
```

The `LIF_PROC_END` indicates a **static** declared procedure with an **S** flag. Procedures that return an expression are indicated with an **R** flag.

For formal parameters, the variable attributes *pointer* and *array* are indicated by the codes: **P** and **A** in the `LIF_FORMAL_ID` record.

All local variable declarations, (LIF_LOCAL_ID), and flow graph node related LIF codes appear between the LIF_PROC_START and the LIF_PROC_END. The following is an example of a procedure header and generated LIF codes:

```
add_to_result(int a, int *b, int c[]){ ... body ...}

1(13,5,add_to_result)
3(1,a)
3(2,b,P)
3(3,c,A)
     ... body ...
2(30)
```

The procedure starts at node 13 in the flow graph.
The last node of the procedure is 30.
There are three formal parameters.
Formal **b** is a pointer.
Formal **c** is an array.

4.2.3.3 Declarations

Declarations generate the following:

```
# define LIF_LOCAL_ID     4   /*  4(id,name[,S][,P][,X][,A])     */
# define LIF_GLOBAL_ID    5   /*  5(id,name[,S][,P][,X][,A])     */
```

Declarations generate a positive, id-code for each variable. Each global variable (declared outside a procedure) is allocated a unique id-code. Each procedure has a separate set of id-codes for local variables and formal parameters, starting from 1. All local variable declarations, (LIF_LOCAL_ID), appear between the LIF_PROC_START and the LIF_PROC_END. Global variable declarations, (LIF_GLOBAL_ID) may appear anywhere outside of a LIF_PROC_START and LIF_PROC_END pair. The variable attributes, static, pointer, external, and array are indicated by the codes: **S, P, X** and **A**. An example of LIF codes generated for declarations follows:

```
    static int a,*b,c[10];
    extern x;
fun(int y)
{
    int u,v,w,x;
}

proc(int *z)
{
    int a,w;
}
```

```
1(1,1,fun)
3(1,y)
4(2,u)
4(3,v)
4(4,w)
4(5,x)
2(2)
1(3,2,proc)
3(1,z,P)
4(2,a)
4(3,w)
2(4)
5(1,a,S)
5(2,b,S,P)
5(3,c,S,A)
5(4,x,X)
```

4.2.3.4 Expressions

Expressions generate the following LIF codes for variables referenced and variables assigned at a node.

```
# define LIF_AREF        24  /* 24(node,addr)                    */
# define LIF_ADDRESS     25  /* 25(addr,pid,id)                  */
# define LIF_REF          7  /*  7(node,id[,level])              */
# define LIF_DEF          8  /*  8(node,id[,level])              */
# define LIF_GREF         9  /*  9(node,id[,level])              */
# define LIF_GDEF        10  /* 10(node,id[,level])              */
```

An expression generates one flow node for the expression plus one flow node for each postfix (**x++**), prefix (**++x**) or comma operator (**x+y,z**) in the statement. The nodes are ordered in the flow-graph as follows: prefix operators from left to right, the expression flow node, and the postfix operators from left to right.

`LIF_REF` code is generated for local variables whose values are used. `LIF_GREF` code is generated for global variables whose values are used. Variables that are assigned a new value generate `LIF_DEF` for assignment to local variables and `LIF_GDEF` for assignment to global variables.

The level indicates the level of indirection of the ref or def. A level of zero, no indirection, is omitted. A level of -1 indicated the *address of* operator (&). An `LIF_ADDRESS` is generated for each object of the *address of* operator, indicating the variable, the procedure where the variable is declared (zero for global declaration) and a unique address id. Address ids are assigned sequentially from 1. An example of expressions and corresponding LIF codes follows:

```
    int x,y; /* global ids 23 (x) and 24 (y) */
    ...
    int a,b,*c; /* local ids 18(a), 19(b) and 20(c) */
    ...
    c = &x;       /* node 46, x is address 14 */
    x = y - (b++) + *c; /* nodes 47 and 48  */

16(46,47)
9(46,23,-1)
8(46,20)
25(14,0,23)
24(46,14)
16(47,48)
7(47,19)
8(47,19)
9(48,24)
7(48,19)
7(48,20,1)
10(48,23)
```

4.2.3.5 Procedure Calls

Procedure calls are handled as part of an expression and use the following LIF codes:

```
# define LIF_CALL_START 11  /* 11(node,pid)                    */
# define LIF_ACTUAL_SEP 12  /* 12    *** void ***              */
# define LIF_CALL_END   13  /* 13    *** void ***              */
```

The actual parameters are listed as expressions in order separated by `LIF_ACTUAL_SEP` entries.

```
    int x,y,z;            /* local ids x(72) y(73) z(74) */
     ...
     x = fun(x+y,*z);     /* node 47, fun is pid 18 */

11(47,18)
7(47,72)
7(47,73)
12
7(47,74,1)
13
8(47,72)
```

4.2.3.6 Structure Fields

Structure fields generate the following LIF codes:

```
# define LIF_CHAIN      19  /* 19(node,chain,id)               */
# define LIF_GCHAIN     20  /* 20(node,chain,id)               */
# define LIF_FIELD      21  /* 21(node,chain,seq,fid,field)    */
```

```
# define LIF_CREF        22   /* 22(node,chain)                    */
# define LIF_CDEF        23   /* 23(node,chain)                    */
# define LIF_STRUCT      26   /* 26(pid,id,offset)                 */
```

At an expression node, each reference or assignment through a pointer to the fields of a structure generates a chain (LIF_CHAIN or LIF_GCHAIN). The chains of a node are given a chain number sequentially from 1. The variable at the head of the chain is specified in the *id* field of the LIF_CHAIN for local variables and in the *id* field of the LIF_GCHAIN for global variables.

LIF_CREF and LIF_CDEF indicate if the chain specifies a reference or a define. LIF_FIELD is used to specify each field of a chain by sequence number. The **fid** is the sequence number of the field within the data structure and **field** is the field name.

LIF_STRUCT indicates that the variable identified by the *pid* and *id* is a structure with *offset* members. For example, consider the following:

```
typedef struct a_struct a_rec,*a_ptr;
typedef struct b_struct b_rec,*b_ptr;
struct a_struct {a_ptr  a1; b_ptr a2; int a3};
struct b_struct {b_ptr  b1; int b3};

    a_rec   ar; /* Global ID 9(ar) 10(ar.a1) 11(ar.a2) 12 (ar.a3)*/
    a_ptr    a;  /* Global ID 13(a) fields 1(a1) 2(a2) & 3(a3) */
    . . .
    b_ptr    b;  /* Local ID 7(b) fields 1(b1) 2(b2) & 3(b3) */

    . . .

    a->a3 = b->b3 + a->a1->a2->b1->b3;       /* node 88 */

26(0,9,3)         structure ar has 3 fields
20(88,1,13)       a->a3
19(88,2,7)        b->b3
20(88,3,13)       a->a1->a2->b1->b3
23(88,1)
22(88,2)
22(88,3)
21(88,1,1,3,a3)       ->a3
21(88,2,1,3,b3)       ->b3
21(88,3,1,1,a1)       ->a1->
21(88,3,2,2,a2)       ->a2->
21(88,3,3,1,b1)       ->b1->
21(88,3,4,3,b3)       ->b3
```

4.2.4 File.T

The **.T** file is produced by the analyzer at the same time as the **.LIF** file. The purpose of the **.T** file is to provide for sizing of dynamic objects by the linker and slicer. The format of the **.T** file is as follows:

1. The first line has two fields, the number of procedures defined or referenced and the number of flow-graph nodes in the source file.

2. One line for each procedure either called or defined within the source file. Each line has 6 fields as shown in Table 4-2.

3. One line with three fields: number of global variables declared, number of pointer chains (i.e., expressions using ->), and number of address objects (i.e., & operator applied).

4. The last line has four fields, number of lines, number of words, number of characters and the file name (i.e., the output of the UNIX command **wc**).

Field	Contents
1	unique procedure id number
2	entry statement number if defined, else -1
3	exit statement number if defined, else 0
4	number of local variables
5	an **X** if **extern**, an **S** if **static**
6	procedure name

Table 4-2: File.T Fields For Defined Or Called Procedures

Example source and **.T** files:

```
    -----> z.c <-----
# include "a.h"
    int     zed,zulu;
main (n,p)
    int     n;
    char    *p[];
{
    int     kappa,lambda,mu;

    zircon (kappa+lambda-zed,zulu,z2->zeta,z1.zeta,&z1);
}
```

```
     -----> a.h <-----
int alpha,beta,omega;
typedef struct zeta_struct zeta_rec, *zeta_ptr;
struct zeta_struct {
    int     zeta;
    zeta_ptr za,zb,zc;
};
zeta_rec    z1,*z2;
zeta_ptr    z3;


    -----> z.T <-----
2 4
  1    1    4    5 X main
  2   -1    0    0 X zircon
12 1 1
      10      17     155 z.c
```

4.2.5 File.H

A program may have global variables either declared directly in the source files or declared in included header files. If a program has many include files then a large number of global variables could be declared. If the user of **unravel** needs to select a global variable then the set of global variables should be organized so that it is easy to locate global variables declared in source files and global variables declared in included files.

The purpose of the **.H** file is to partition the set of global variables according to the location of the variable declaration. The **.H** file consists of two types of records, file name records and variable name records. The **.H** file is organized as a series of file name records followed by variable name records for each global variable declared in the file. The same file name may appear more than once in the **.H** file. A variable may be declared in more than one file.

The format of a file name record is `@␣file_name` with `@␣` in columns one and two and the file name starting in column 3 and extending to the end of the line. The file name is as produced by the C preprocessor for a change of file from a `#include` statement.

The variable record is a tab character in column 1 followed by the variable name extending to the end of the line. Example **.c**, **.h** and **.H** files follow:

```
  -----> c File (yy.c) <-----
int    a,tip;
# include "a.h"
int    rip,TRIP,sip;


  -----> h File (a.h) <-----
int    alpha,beta,omega,z1,z2,z3;
```

```
  -----> H File (yy.H) <-----
@ yy.c
        a
        tip
@ a.h
        alpha
        beta
        omega
        z1
        z2
        z3
@ yy.c
        rip
        TRIP
        sip
```

4.2.6 SYSTEM

The **SYSTEM** file is the output of the linker component **map** using the **.T** and **.LIF** files as input. The purpose of the **SYSTEM** file is to describe each main program in a directory in terms of required source files, library functions called and called procedures defined in more than one source file. The **SYSTEM** file consists of 6 kinds of records (one record to a line) described in Table 4-3.

#	Record name	Format
1	main file	**MAIN** *file_name file_count*
2	file separator	**FILE**
3	ambiguous procedure separator	**AMBIG**
4	library separator	**LIB**
5	file name	tab character *file_name*
6	procedure name	tab character *proc_name*

Table 4-3: SYSTEM File Records

The arrangement of records in the **SYSTEM** file to describe a single program divided among several files is as follows:

1 main file record
2 file separator
3 source (**.c**) files required by the given main file
4 ambiguous procedure separator

5 zero or more procedure name records
6 library separator
7 zero or more procedure name records

The above organization is repeated for additional programs in the current directory.

The following example has two main programs (one in **ex-1.c**, the other in **b.c**). The main in **ex-1.c** uses four library functions and the entire program is contained in one file. The main in **b.c** uses procedures defined in three other files. One procedure, **a4**, is ambiguously defined since it has a procedure body defined in more than one source file.

```
MAIN ex-1.c 1
FILES
        ex-1.c
AMBIG
LIB
        scanf
        printf
        exit
MAIN b.c 4
FILES
        b.c
        bb.c
        bbb.c
        abc.c
AMBIG
        a4
LIB
        exit
```

4.2.7 File.K

The linker merges the **.H** and **.T** files for all the files of a program into one **.K** file. The **.K** file is organized into 4 sections:

1. The first line of the file gives counts for number of global variables, number of procedures, number of address objects, number of pointer chains, number of header files and number of source files. Each value is preceded by an identifying string.

2. One line of procedure description for each procedure. The content is similar to the **.T** file. Each line contains, procedure id, entry statement, exit statement, number of local variables, **static** or **extern** flag as an **S** or an **X**, and procedure name.

3. One line of description for each file. Each line contains a file id, number of procedures, number of statements, number of lines, number of characters and file name minus extension.

4. The last section is the merge of the **.H** files. There should be a file record for each file where global variables are declared. Each file with declared global variables should have only one file record. A file record has two fields, a count of the number of globals declared in the file and the file name. Each file record is followed by a set of global records. A global record has two fields, the id number assigned to the global variable by the linker and the variable name.

4.2.8 File.LINK

The **.LINK** file contains the merged **.LIF** files from all the files that are required the main program in **file.c**.

One LIF code, `LIF_FILE`, not found in **.LIF** files is added to the **.LINK** file to indicate the source file associated with each procedure.

```
# define LIF_FILE         6   /*   6(file_id,file_name)          */
```

4.3 Slicer

This section presents by informal pseudo-code the procedures and data structures of the program slicing component. There is one procedure, **slice**, that serves to control invocation of the other procedures that are used to compute program slices. The **slice** procedure is invoked from a user interface component that obtains the slicing criterion from the user and displays the program slice to the user.

The data structures fall into two categories, representation of the program being sliced and dynamic support for slice computation.

4.3.1 Procedures

slice This procedure controls computation of a program slice and is called by the user interface after the slicing criterion has been obtained. The slice is returned to the caller as a set of flow-graph nodes.

```
slice (criterion c, set slice_set)
{
    clear criteria for each node
    clear slice_set
    set initial criterion
    call main_slice (c, slice_set)
}
```

main_slice Sweep over the program applying the slicing criteria until no more changes occur (and no more statements are added to the slice.

```
main_slice (criterion c, set slice_set)
{
    change = 1
    while (change) {
        change = 0
        for each node, n, {
            for each successor, m, of n{
                change += include_or_not (n,m,slice_set)
            }
        }
    }
}
```

include_or_not Test a statement for inclusion in a slice and return a count of changes. The change count is the sum of number of statements added to the slice and number of new criteria generated. If the statement changes a variable in the criteria for the statement's successor then the statement should be added to the slice and additional criteria should be generated.

```
int include_or_not (node_index n, node_index m, set slice_set)
{
    change = false
    if (defs(n) intersects criteria(m)){
        change += add_to_slice (n, slice_set)
    }
    if (idefs(n) intersects criteria(m)){
        change += add_to_slice (n, slice_set)
        change += add_idef_criteria(n)
    }
    if (cdefs(n) intersects criteria(m)){
        change += add_to_slice (n, slice_set)
        change += add_cdef_criteria(n)
    }
    return change
}
```

add_to_slice If statement n is not in the slice add n to the slice and adjust slicing criteria. Add any statements in the requires set and return the count of changes.

```
int add_to_slice (node_index n, set slice_set)
{
    count = 0
    if n already in slice then return 0
    add n to slice_set
    count = 1 + add_criteria(n)
    for each node, r, in requires(n){
        count += add_to_slice(r)
    }
    return count
}
```

add_criteria Add the criteria generated by inclusion of any statement in a slice.

```
int add_criteria (node_index n)
{
    add refs(n) to criteria(n)
    add criteria implied by irefs(n)
    add criteria implied by crefs(n)
    return change count
}
```

add_idef_criteria Add criteria for variables referenced in the specification of the variable indirectly defined (assigned).

```
int add_idef_criteria (node_index n)
{
    add criteria implied by idefs(n)
    return change count
}
```

add_cdef_criteria Add criteria for variables referenced in the specification of the variable indirectly defined (assigned).

```
int add_cdef_criteria (node_index n)
{
    add criteria implied by cdefs(n)
    return change count
}
```

4.3.2 Data Structures

The following data structures, built from the files produced by the *analyzer* and *linker*, are used to represent the program being sliced.

```
typedef struct source_file_struct source_file_rec,*source_file_ptr;
typedef struct flow_node_struct flow_node_rec,*flow_node_ptr;
typedef struct proc_struct proc_rec,*proc_ptr;
typedef struct ptr_state_struct ptr_state_rec,*ptr_state_ptr;

typedef struct {
    int             start_line;
    int             start_col;
    int             end_line;
    int             end_col;
} src_map;

struct source_file_struct {
    proc_ptr        procs[];
}
```

```
struct flow_node_struct {
    set             refs,defs,irefs,idefs,crefs,cdefs;
    set             successors,requires;
    src_map         source;
    set             criteria;
    chain           chains[];
    ptr_state_ptr   p;
}

struct proc_struct {
    flow_node_ptr   nodes[];
    set             globals_defed,globals_refed;
    set             formals_defed,formals_refed;
}

struct ptr_state_struct {
    set             state[];
}
```

4.4 Linker

The *linker* component operates in two parts. The first part, **map**, identifies for each program in the current directory its constituent files and then saves the information in a file named **SYSTEM** as discussed in sec 4.2.6. The second part of the link component, **slink**, uses the **SYSTEM** file to merge data-flow information from the **.LIF**, **.T** and **.H** files created from separate source files into a single **.LINK** (sec. 4.2.8) file and a single **.K** (sec. 4.2.7) file.

4.4.1 Map

The task of **map** is to identify all the source files that make up each main program. While this is not solvable in general, it is possible for most situations the user is likely to encounter. It is also possible to identify a situation where **map** cannot complete its task and the user must provide assistance.

The input to **map** is the set of all **.LIF**, **.T** and **.H** files in the current directory.

The output is a **SYSTEM** file that identifies each main program and its associated source (**.c**) files.

4.4.2 Slink

The input to **slink** is a file name (to identify the main program), the **SYSTEM** file and the **.LIF**, **.T** and **.H** files that the **SYSTEM** file indicates belong to the main program.

The output is the merged **.LIF** files in a **.LINK** file and the merged **.T** and **.H** files in a **.K** file.

4.5 User Interface and Help System

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

The **Main Control Panel** is used to invoke **Analyzer Control Panel** and **Slicer Control Panel** and provides relevant information about the current directory.

The **Analyzer Control Panel** allows the user to select files, run the analyzer and automatically run **map** to scan for **main** programs.

The **Selection Control Panel** allows the user to select a **main** program and runs the **linker** on the selected **main** program followed by invoking the **Slicer Control Panel** for the selected program.

The **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* give the following resource specification to **xrdb**: **`*Unravel*Foreground: red`**).

Two application resources, **runningFG** and **runningBG**, are defined. 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. An information pop-up window consists of a **done** button to dismiss (pop-down) the window and a scrollable text window.

The user interface keeps the following log files:

1. The file **HISTORY.LOG** is a log of user analysis and slicing activity in the current directory for past invocations of **unravel**. The **HISTORY.LOG** is updated with current activity when the **Main Control Panel** exits.

2. The file **HISTORY** is a log of user analysis and slicing activity in the current directory for the current invocation of **unravel**. The **HISTORY** file is updated with current analysis activity when the **Analyzer Control Panel** exits and is updated with slicing activity when the **Slicer Control Panel** exits.

3. The file **HISTORY-A** is a log of user analysis activity for the current invocation of the **Analyzer Control Panel**. Results of analysis of each file is recorded, including syntax errors found in the source code.

4. The file **HISTORY-S** is a log of user slicing activity for the current invocation of the **Slicer Control Panel**.

4.4.1 Main Control Panel

The function of the **Main Control Panel** is to respond to user interaction with the panel.

The **Main Control Panel** is invoked by running the program **unravel**. The input to **unravel** is a single directory name on the command line to specify a working directory. If the command line is empty, the current directory is used as the working directory. After a working directory is obtained, **unravel** makes the working directory the current directory.

Invoking the **Main Control Panel** does the following initializations:

1. Change to the directory specified on the command line.

2. Initialize **HISTORY-S** to **No slices computed this session**.

3. Initialize **HISTORY-A** to **No analysis done this session**.

4. Initialize **HISTORY** 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 source files that have **.LIF** and **.T** files such that the C file is older than the **.LIF** and **.T** files.
- Number of source files not analyzed. This is a count of files with the **.c** extension that do not have either an **.LIF**, **.T** or an **.H** file.
- Number of files analyzed and out of date. The number of C source files that have either **.LIF** or **.T** files such that the C file is younger that either the **.LIF** or the **.T** file or both.
- Number of main program files analyzed, i.e., the number of main program files identified in the **SYSTEM** file.
- Number of main program files linked. This is the number of main program files identified in the **SYSTEM** file that also have **.LINK** and **.K** files.
- Number of duplicate procedures found. This is the number of procedures identified in the **SYSTEM** file as ambiguous (appearing in more than one file).
- 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 The **Exit** button does the following:

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

Run Analyzer The **Run Analyzer** button does the following:

1. Invokes the **Analyzer Control Panel**, passing the window id of the **Run Analyzer** button as a command line parameter.

2. When the **Run Analyzer** button receives a non-maskable event (i.e., **xsend** from **Analyzer Control Panel**) the displayed counts are updated.

Review History The **Review History** button pops-up a four item menu, and display the indicated file.

Last Analysis	**HISTORY-A**
Last Slice	**HISTORY-S**
This Session	**HISTORY**
All History	**HISTORY.LOG**

Run Slicer The **Run Slicer** button invokes the **Selection Control Panel**.

Help The **Help** button runs **helpu** with the file **unravel.help** as command line parameter.

Current Directory The following is done when the directory is changed:

1. Append the file **HISTORY** to **HISTORY.LOG**
2. Delete **HISTORY**
3. Initialize **HISTORY-S** to **No slices computed this session**
4. Initialize **HISTORY-A** to **No analysis done this session**
5. Initialize **HISTORY** to **UNRAVEL** *directory name* **current date and time**

4.4.2 Analyzer Control Panel

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

- Buttons to control file selection, running the *analyzer*, clearing analysis files and popping-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** parser.
- A list of *selected* source files from the current directory.
- A list of other 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, do 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 should be set to the foreground and background colors specified in the application resources **runningFG** and **runningBG**.

After all selected files have been analyzed, run **map**.

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

Help The **Help** button displays the file **analyzer.help**.

4.4.3 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 panel.

The **Help** button displays the file **select.help**.

HISTORY-S is updated with a message indicating the file to be linked before the linker is called. Any linker output (e.g., error messages) is appended to **HISTORY-S**.

4.4.4 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 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.

- The last line of the 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.

In addition to interaction with the user through the window interface, the slicer has the following inputs and outputs:

Command Line The slicer takes one command line argument, **file.c**, the name of a main program file.

Summary Counts The slicer looks for a **.K** file, **file.K**, that matches the main program file on the command line where **file** is the name of the main program file without any extension.

Linked .LIF File The slicer looks for a **.LINK** file, **file.LINK**, that matches the main program file on the command line where **file** is the name of the main program file without any extension.

Computed Slices Each slice that is computed is saved in **file.Y** as a criterion and set of flow graph node numbers. The file format is as follows:

1. Criterion (four integers): variable id number, procedure id number that variable is local to or zero if global, file id number containing statement and statement number.

2. Partial slice flag (integer): value 0 if slice computation was not interrupted, value 1 if interrupted.

3. File entries (one per file): consists of file id number followed by zero or more statement numbers, terminated by **-1**. The last file entry is followed by a file id of **-1** (i.e., slice entry is terminated by two **-1** entries, one to end the statements of the last file of the main program and one to mark end of files in the program for the slice).

HISTORY-S The slicer records information in the following format for each slice computed is placed in **HISTORY-S** to record the criterion, the slice size in flow graph nodes and the wall clock time to compute the slice:

`slice on` *var name* `(in` *procedure name*`) at line` *nnn* `in` *file name (nnn* `stmts in` *mm:ss)*

If the variable is global then the word *global* replaces the procedure name.

The **Slice Control Panel** buttons do the following:

Exit This button appends **HISTORY-S** to **HISTORY** and then exits.

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

1. Stops computation of the slice and displays partial results.

2. Marks the slice as partial and saves.

Help This button pops-up the panel help file, **u.help**.

There are six information display windows on the **Slice 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, it is 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, **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 from Table 4-4. 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 proc	Location of *procedure_name* in *file_name*
Call tree	Call tree of *procedure_name*

Table 4-4: Text Description Windows

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 sets of statements such as the statements that are members of a slice. 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.

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 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 main 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 list of procedure names 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 This selection pops-up a list of previously computed slices. The first entry in the list is **No Selection**. If a slice is selected, it becomes the *primary slice* and is displayed. The list is then popped-down.

Secondary This selection pops-up a list of previously computed slices. The first entry in the list is **No Selection**. If a slice is selected, it becomes the *secondary slice* and is displayed. The list is then popped-down.

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

The selection **Dice** highlights the statements of the *primary* slice that are not members of the *secondary* slice and updates the **Text Description Window**.

The selection **Dice S-P** highlights the statements of the *secondary* slice that are not members of the *primary* slice and updates the **Text Description Window**.

The selection **Intersection** highlights the statements in both the *primary* and *secondary* slice and updates the **Text Description Window**.

The selection **Union** highlights the statements in either the *primary* or *secondary* slice and updates the **Text Description Window**.

The selection **Clear** removes all highlighting and updates the **Text Description Window**.

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.

5 System Evaluation & Performance

Unravel was evaluated in the context of reviewing safety system software for quality. The evaluation considered the size of slices produced, time to compute slices and usability by a novice user.

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?

3. Is **unravel** usable by a novice user?

Program slicing can help automate two tasks performed by reviewers:

1. A *thread check* traces a variable chosen for evaluation through the software. This includes reviewing relevant sections of program source code that currently must be manually located.

2. Evaluation of functional diversity is accomplished by attempting to determine if two application functions share source code. If source code is shared by two diverse application functions, then the reviewer must carefully evaluate the shared code for errors. The concept of *functional diversity* is used to defend against *common mode failure* in digital systems.

Two examples of typical safety system code were used to test and refine **unravel**. Demonstration of **unravel** using these and other examples were given to software reviewers. The demonstrations provided useful results that resulted in improvements to the user interface and in the identification of features to be explained in more depth in the user manual or to be included in a later version of **unravel**.

5.1 Capability Analysis

The first example a simplified safety system was developed in three versions. One version was written to conform to safety system diversity requirements while the other two were deliberately seeded with common code. **Unravel** was able to verify and display the presence of the common code in the seeded versions and show the absence of common code in the diverse version. Only the conforming version of the example is used in the size and timing analysis.

The second example, a commercial sample of safety related code, presented a realistic evaluation for **unravel**. While the code was not ANSI C, **unravel** was used after a few simple changes brought the commercial code into ANSI compliance. The commercial code was used by a software reviewer to evaluate the utility of **unravel**.

Slice Size	Ex-1 Example		Commercial Example	
Size ≤ 1%	147	76%	155	37%
1% < Size ≤ 25%	26	14%	135	32%
25% < Size ≤ 50%	10	5%	129	31%
50% < Size	10	5%	0	0%
Total	193	100%	419	100%

Table 5-1: Slice Size Analysis

Table 5-1 presents an analysis of slice sizes for the two example programs. The sizes are clustered in terms of number of statements in a slice relative to the total program size. For each example, the number of slices in a category and the percentage of the total slices are given. The table shows that the user of **unravel** can expect **unravel** to eliminate a significant portion of code from consideration when using program slicing to extract a given computation for examination.

The reviewer directed **unravel** to compute six 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 perform and analyze string checks[†].

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

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

5.2 Timing Analysis

This section reports on empirical tests to estimate the time necessary for **unravel** to compute program slices for programs of 1000 lines, 10,000 lines and 100,000 lines. The tests are divided

[†] A *string check* is a method for software evaluation that includes locating all source code statements involved in some computation.

into three areas: analyzer, linker and slicer. The tests were performed on three sets of source code: a simplified safety system, a commercial example, and the **unravel** source code. Times are in seconds for the analyzer and linker except as noted. Times for the slicer are in minutes.

5.2.1 Analyzer Timing

Table 5-2 presents the timing results for the simplified example. Results for the commercial code are presented in Table 5-3 and the **unravel** source code results are presented in Table 5-4. The first column is the file name. The next three columns represent three different measures of the file size. Column two, labeled *Lines*, is the number of source lines in the file; this is the number of lines the programmer sees when editing the file. The next column, labeled *CPP LOC*, is the number of lines in the file after expansion by the C preprocessor to insert any *include files*. This is the actual input that the slicing component of **unravel** receives. The last of the three columns, labeled *NCLOC (Non-Commentary Lines Of Code)*, is the size of the expanded file after comments and blank lines are removed. The column labeled *LIF* is the size in bytes of the analysis files. The data are sorted by *NCLOC*.

File	Lines	CPP LOC	NCLOC	Time	LIF
main.c	193	256	143	1	30,595
coolant.c	467	704	320	1	65,161
pressure.c	510	747	342	1	69,529
Total	1270	1707	805	3	165,285

Table 5-2: Analyzer Results for simplified Example

File	Lines	CPP LOC	NCLOC	Time	LIF
file1.c	545	282	475	1	19,424
file2.c	672	409	587	1	30,292
prog.c	707	1764	638	2	20,981
file3.c	668	1405	651	2	32,221
file4.c	552	1521	655	2	30,178
file5.c	888	1625	703	2	41,754
Total	4032	9006	3709	10	174,850

Table 5-3: Analyzer Results for Commercial Code

File	Ver	Lines	CPP LOC	NCLOC	Time	LIF
visit-filter.c	1.1	17	192	48	1	1,825
visit-ctrl.c	1.1	38	213	52	1	2,381
pss-driver.c	1.1	30	532	238	1	3,179
tsummary.c	1.1	198	815	251	1	9,341
summary.c	1.2	198	505	252	1	11,994
err.c	1.1	23	613	264	1	3,355
const.c	1.1	30	620	277	1	3,340
slice_driver.c	1.3	198	500	296	2	11,361
call-tree.c	1.1	94	613	302	1	8,684
sets.c	1.1	302	592	323	1	22,699
parser.c	1.6	141	815	390	1	12,204
history.c	1.1	227	669	411	4	16,487
mem_alloc.c	1.1	167	842	431	1	12,436
auto-slice.c	1.4	216	1,099	471	2	20,088
chain.c	1.3	375	1,025	540	1	27,163
map.c	1.4	614	1,355	711	2	42,176
pss.c	1.3	871	1,373	431	3	58,607
stmt.c	1.7	296	1,401	946	2	57,003
xpr.c	1.3	778	1,513	965	4	63,532
kgram.c	1.5	1,950	2,590	1,872	5	108,902
kscan.c	1.4	2,017	2,734	1,884	4	65,433
slice-load.c	1.5	1,390	2,349	1,351	5	105,134
slice.c	1.6	1,691	2,108	1,642	4	141,898
slink.c	1.3	1,176	1,999	1,131	3	98,524
sym_tab.c	1.3	1,359	1,999	1,279	4	95,638
helpu.c	1.1	76	12,815	5,646	15	8,970
MultiSlice.c	1.2	793	12,723	6,353	32	92,807
analyzer.c	1.2	1,216	14,980	6,673	25	66,713
select.c	1.3	601	14,576	6,208	20	36,968
u.c	1.5	1,383	14,742	6,839	32	86,455
unravel.c	1.4	709	14,089	6,210	59	35,904
Total		19,656	112,191	55,095	3:21	1,330,192

Table 5-4: Analyzer Results for Unravel

These data indicate that it is practical to run the **unravel** analyzer on programs of at least 100,000 lines of code. Since the definition of *lines of code* is often controversial, three file size measures are given. In this analysis we consider *NCLOC* the most reasonable measure of *lines of code* and the most accurate predictor of run time. For the simplified example and the commercial code, analyzer run time is about 3 or 4 seconds for a thousand lines of NCLOC, a rate of 250-330 lines per second. Since the analyzer does a single pass over the source code and linear performance is expected, 100,000 lines of code should be analyzed in about 6:40 minutes (at 250 lines per second). The 55,095 lines of **unravel** source code were analyzed in 3:21 minutes; this time agrees with the expected value.

5.2.2 Linker Timing

The **unravel** linker has two main components: **map** which scans the analysis files for **main** procedures and **slink** which merges the **LIF** files of a program into a single **LINK** file. The **map** component of the linker ran in less than 2 seconds on the **unravel** source files (55,000 lines) and should run in less than 5 seconds on 100,000 lines of code.

The timing results of linking each main program are presented in Table 5-5. The column labeled *Program* contains the program name. The source files that must be linked together for the program are given under *Files*. The *Link Time* column gives the time in seconds for the linker to run on each program. Two programs for **unravel** are typical. The program *u* represents 17,000 NCLOC and is linked at the rate of 1,700 lines per second. The program **parser** represents about 7,500 NCLOC and is linked at the rate of about 950 lines per second. Since the linker is a single pass algorithm, and a linear timing relationship is expected, the linker should be able to link 100,000 lines of code in about 2 minutes.

5.2.3 Slicer Timing

To evaluate the times to compute program slices, criteria were automatically generated by slicing on the last statement of the **main** procedure for each global variable and the last statement of the procedure where a local variable is declared for each local variable. Usually the timing results produce clusters of slices with similar times. It should be noted that we use these examples as performance benchmarks as **unravel** evolves. These results are for **unravel version 2.1**, different results are obtained as improvements are made to the slicing algorithm or as bugs are fixed.

The example code provided 193 slicing criteria. Most slices (183) were completed in under 1 second. The remaining ten slices were completed in less than ten seconds.

The commercial example provided 419 slicing criteria. The slicing times divided into four clusters, 354 slices were completed in under 1 minute. Fifty-eight slices required more than 1 minute but under 5 minutes. Six slices required between 35 minutes to one hour. One slice took 3 hours and 9 minutes.

Program	Files	Link Time	Link Size
main	main.c coolant.c pressure.c	1	160,341
prog	prog.c file1.c file2.c file3.c file4.c file5.c	3	148,479
visit-filter	visit-filter.c	1	1,343
visit-ctrl	visit-ctrl.c	1	1,901
unravel	unravel.c	1	33,104
u	u.c MultiSlice.c slice.c sets.c history.c slice-load.c pss.c	10	480,886
tsummary	tsummary.c	1	8,760
summary	summary.c	1	11,396
slink	slink.c	1	96,898
slice_driver	slice_driver.c slice.c slice-load.c sets.c pss.c	6	307,815
select	select.c	1	33,956
pss-driver	parser.c sym_tab.c mem_alloc.c xpr.c chain.c stmt.c kgram.c kscan.c err.c	8	417,953
map	map.c	1	40,940
helpu	helpu.c	1	6,277
call-tree	call-tree.c	5	305,616
analyzer	analyzer.c	1	63,800

Table 5-5: Linker Results

Cluster	Time	N Slices	Percent
1	< 1	104	36
2	$1 < t < 12$	113	40
3	$23 < t < 25$	15	5
4	$45 < t < 60$	13	5
5	$60 < t$	40	14

Table 5-6: Slicer Results for Unravel Code

The **unravel** source code generated 834 slicing criteria, 155 global variables and 679 local variables. Slices were computed for 285 of the criteria, the 155 global variables and 130 local variables. Table 5-6 presents the results for each cluster size. Times are in minutes.

5.3 Analysis Summary

The software review process as currently implemented is a manual process that is slow, tedious, and prone to human errors. With **unravel**, once a software reviewer has identified a variable for further investigation, the reviewer directs **unravel** to compute a program slice on the variable. Instead of examining the entire program, only the statements in the slice need to be examined by the reviewer. By speeding up the process of locating relevant code for examination by the reviewer, a larger sample of a commercial product can be inspected with greater confidence that some relevant section of source code has not been missed.

Once two computations that could be vulnerable to *common mode failure* have been identified, program slices can be computed to find statements relevant to each computation. Functional diversity of the two computations can be evaluated by intersecting slices to show any statements in common. Source program statements that have potential to cause common mode failure would be present in the intersection of the program slices. Without any tool, a software reviewer evaluates the software until it is determined that there is no common code, or that the common code present will not compromise the mission of the safety critical software.

The analyzer and linker components can process source code of up to 100,000 lines of code in less than 10 minutes. The linear behavior of the analyzer and linker leads to stable run time performance. The slicer component does not use a linear algorithm, but rather uses a quadratic algorithm that can have significant run time variability. The slicer performed well on the simplified example. Larger programs, such as **parser** with 7,500 lines, have slices (14%) that

can take longer than one hour. It should be noted that there is potential for significant algorithm improvement. The slicer makes repeated passes over the program until no more changes occur. The slicer is sensitive to the order in which program statements are analyzed. For example, after one small change in the slicer code that controls the order of visiting nodes during the slice computation, the longest time to compute a slice on the commercial code dropped from 10 hours to 3 hours. Other areas that can be improved include loop analysis and procedure calls.

6 References

1. ANSI. American National Standard for Information Systems - Programming Language - C. Technical Report ANSI X3.159-189/FIPS PUB 160, American National Standards Institute, 1430 Broadway New York, New York 10018, December 1989.

2. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352-357, July 1984.

3. 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.

4. J. R. Lyle and M. D. Weiser. Experiments in slicing-based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*. Ablex Publishing Corporation, Noewood, New Jersey, 1986.

5. K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751-761, August 1991.

6. M. Weiser. Programmers Use Slicing When Debugging. *CACM*, 25(7):446-452, July 1982.

7. S. Horwitz, J. Prins, T. Reps. Integrating Non-Interfering Versions of Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345-387, July 1989.

8. J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use In Optimization. *ACM Transactions on Programming Languages*, 9(3):319-349, July 1987.

9. FIPS PUB 151-2, "Portable Operating System Interface (POSIX)-System Application Program Interface [C Language]," U.S. Department of Commerce/National Institute of Standards and Technology, 1993 May 12.

10. FIPS PUB 158-1, "The User Interface Component of the Application Portability Profile," U.S. Department of Commerce/National Institute of Standards and Technology, 1993 October 8.

Appendix A: LIF Format

```
#ifndef _lif_h
#define _lif_h
#define LIF_H_SCCS_ID "  @(#)lif.h 1.8  5/23/94 "
/*
*****************************************************************
*                                                               *
*         L I F    F O R M A T                                  *
*                                                               *
*  id   - variable id                                           *
*  node - source program statement (or fragment)                *
*  pid  - procedure id                                          *
*  name - variable or procedure name                            *
*  level- indirection level                                     *
*  addr - address number                                        *
*  chain- chain number of pointer chain on a node               *
*  field- sequence number of field in chain                     *
*  fid  - field offset in struct                                *
*  offset number of fields in a declared structure variable     *
*  A    - is an array                                           *
*  P    - is a pointer                                          *
*  S    - static object                                         *
*  X    - is declared extern                                    *
*  G    - is a goto statement                                   *
*  B    - is a break statement                                  *
*  C    - is a continue statement                               *
*  R    - returns an expression value                           *
*                                                               *
*****************************************************************
*/

# define LIF_PROC_START  1  /*  1(node,pid,name)             */
# define LIF_PROC_END    2  /*  2(node[,S][,R])              */
# define LIF_FORMAL_ID   3  /*  3(id,name[,A][,P])           */
# define LIF_LOCAL_ID    4  /*  4(id,name[,S][,P][,X][,A])   */
# define LIF_GLOBAL_ID   5  /*  5(id,name[,S][,P][,X][,A])   */
# define LIF_FILE        6  /*  6(file_id,file_name)         */
# define LIF_REF         7  /*  7(node,id[,level])           */
# define LIF_DEF         8  /*  8(node,id[,level])           */
# define LIF_GREF        9  /*  9(node,id[,level])           */
# define LIF_GDEF       10  /* 10(node,id[,level])           */
# define LIF_CALL_START 11  /* 11(node,pid)                  */
# define LIF_ACTUAL_SEP 12  /* 12    *** void ***            */
# define LIF_CALL_END   13  /* 13    *** void ***            */
# define LIF_RETURN     14  /* 14(node,1|0)                  */
# define LIF_GOTO       15  /* 15(node,G|B|C)                */
# define LIF_SUCC       16  /* 16(from_node,to_node)         */
```

```
# define LIF_REQUIRES   17  /* 17(required_node,node[,to node])*/
# define LIF_SOURCE_MAP 18  /*18(node,fr_l,fr_c,to_l,to_c)     */
# define LIF_CHAIN      19  /* 19(chain,id)                     */
# define LIF_GCHAIN     20  /* 20(chain,id)                     */
# define LIF_FIELD      21  /* 21(chain,field,fid,name)         */
# define LIF_CREF       22  /* 22(node,chain)                   */
# define LIF_CDEF       23  /* 23(node,chain)                   */
# define LIF_AREF       24  /* 24(node,addr)                    */
# define LIF_ADDRESS    25  /* 25(addr,pid,id)                  */
# define LIF_STRUCT     26  /* 26(pid,id,offset)                */

#endif /* _lif_h */
```

Appendix B: YACC Grammar

B.1 Expressions

```
%%

primary_expr
        : identifier
    | CONSTANT
    | string_literal_list
    | '(' exprXX ')'
    ;
string_literal_list
    : STRING_LITERAL
    | string_literal_list STRING_LITERAL
    ;

postfix_expr
    : primary_expr
    | postfix_expr '[' exprXX ']'
    | postfix_expr '(' ')'
    | postfix_expr '(' argument_expr_list ')'
    | postfix_expr '.' identifier
    | postfix_expr PTR_OP identifier
    | postfix_expr INC_OP
    | postfix_expr DEC_OP
    ;

argument_expr_list
    : assignment_expr
    | argument_expr_list ',' assignment_expr
    ;

unary_expr
    : postfix_expr
    | INC_OP unary_expr
    | DEC_OP unary_expr
    | unary_operator cast_expr
    | SIZEOF unary_expr
    | SIZEOF '(' type_name ')'
    ;

unary_operator : '&' | '*' | '+' | '-' | '~' | '!'
    ;

binary_operator : '&' | '*' | '+' | '-' | '~' |
    '!' | '/' | '%' | '<<' | '>>' | '<' | '>' |
    '<=' | '>=' | '==' | '!=' | '^' | '&&' | '||'
    ;

cast_expr
    : unary_expr
    | '(' type_name ')' cast_expr
    ;

binary_expr
    : cast_expr
    | binary_expr binary_operator cast_expr
    ;

conditional_expr
    : binary_expr
    | binary_expr '?' expr ':' conditional_expr
    ;

assignment_expr
    : conditional_expr
    |   unary_expr   assignment_operator
assignment_expr
    ;

assignment_operator
    : '=' | MUL_ASSIGN | DIV_ASSIGN
    | MOD_ASSIGN
    | SUB_ASSIGN | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | XOR_ASSIGN | OR_ASSIGN
    | ADD_ASSIGN | AND_ASSIGN
    ;


expr
```

```
    : exprXX
    ;

exprXX
    : assignment_expr
    | expr ',' assignment_expr
    ;

constant_expr
    : conditional_expr
    ;
```

B.2 Declarations

```
declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list
    ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier
declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF | EXTERN | STATIC
    | AUTO | REGISTER
    ;

type_specifier
    : CHAR | SHORT | INT | LONG
    | SIGNED | UNSIGNED
    | DOUBLE | CONST | VOLATILE
    | FLOAT  | TYPE_NAME
    | struct_or_union_specifier
    | enum_specifier | VOID
    ;

struct_or_union_specifier
    : struct_or_union identifier
        '{' struct_declaration_list '}'
    | struct_or_union
        '{' struct_declaration_list '}'
    | struct_or_union identifier
    ;

struct_or_union
    : STRUCT | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : type_specifier_list struct_declarator_list
    ';'
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expr
    | declarator ':' constant_expr
    ;


enum_specifier
    : ENUM '{' enumerator_list '}'
```

```
    | ENUM identifier '{' enumerator_list '}'
    | ENUM identifier
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    | enumerator_list ','
    ;

enumerator
    : identifier
    | identifier '=' constant_expr
    ;

declarator
    : declarator2
    | pointer declarator2
    ;

parms_next : /* empty */
    ;

declarator2
    : identifier
    | '(' declarator ')'
    | declarator2 '[' ']'
    | declarator2 '[' constant_expr ']'
    | declarator2 parms_next '(' ')'
    | declarator2 parms_next
        '(' parameter_type_list ')'
    | declarator2 parms_next
        '(' parameter_identifier_list ')'
    ;

pointer
    : '*'
    | '*' type_specifier_list
    | '*' pointer
    | '*' type_specifier_list pointer
    ;

type_specifier_list
    : type_specifier
    | type_specifier_list type_specifier
    ;

parameter_identifier_list
    : identifier_list
    ;

identifier_list
    : identifier
    | identifier_list ',' identifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list ',' '...'
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : type_specifier_list declarator
    |   REGISTER    type_specifier_list
declarator
    | type_name
    ;

type_name
    : type_specifier_list
    | type_specifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | abstract_declarator2
    | pointer abstract_declarator2
    ;


abstract_declarator2
    : '(' abstract_declarator ')'
```

```
    | '[' ']'
    | '[' constant_expr ']'
    | abstract_declarator2 '[' ']'
    | abstract_declarator2 '[' constant_expr ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | abstract_declarator2 parms_next '(' ')'
    | abstract_declarator2 parms_next
            '(' parameter_type_list ')'
    ;

initializer
    : assignment_expr
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | initializer_list ',' initializer
    ;
```

B.3 Statements

```
statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : identifier ':' statement
    | CASE constant_expr ':' statement
    | DEFAULT ':' statement
    ;



decl_end
    : declaration_list
    ;

decl_start
    : /* empty */
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' decl_start decl_end '}'
    | '{' decl_start decl_end statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : ';'
    | expr ';'
    ;

selection_statement
    : IF '(' expr ')' statement
    | IF '(' expr ')' statement ELSE statement
    | SWITCH '(' expr ')' statement
    ;
oexpr : /* optional */
    | expr
    ;



iteration_statement
    : WHILE '(' expr ')' statement
    | DO statement WHILE '(' expr ')' ';'
    | FOR '(' oexpr ';' oexpr ';' oexpr ')'
        statement
```

```
    ;

jump_statement
    : GOTO identifier ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expr ';'
    ;
```

B.4 External Objects

```
program :
        | file
        ;

file
    : external_definition
    | file external_definition
    ;

external_definition
    : function_definition
    | declaration
    ;

function_start
    : /* empty */
    ;

function_definition
    : declarator function_start ";"
    | declarator function_start function_body
    | declaration_specifiers declarator
            function_start function_body
    ;


function_body
    : compound_statement
    | declaration_list compound_statement
    ;
```

```
identifier
    : IDENTIFIER
    ;

%%
```