

CodeSurfer Technology Overview

Dependence Graphs and Program Slicing

Introduction

CodeSurfer, GrammaTech's program analysis, understanding, and inspection system for ANSI C, is based on *system dependence graphs*, a fundamental intermediate structure for representing programs. Slicing is a particular application of dependence graphs. Together they have come to be widely recognized as a centrally important technology in software engineering, with applications in program understanding, maintenance, debugging, testing, differencing, specialization, reuse, optimization, parallelization, and anomaly detection. Because they operate on the deep structure in programs rather than surface structures, they enable much more sophisticated and useful analysis capabilities than conventional tools. GrammaTech's implementation of components for program dependence graphs and slicing is probably the most advanced in existence.

This overview is structured as follows:

- Background and Terminology explains the basic concepts.
- Component Architecture summarizes GrammaTech's packaging of dependence graph and slicing technology, and illustrates the high-level architecture of a typical end-user application built with GrammaTech's components.
- Program Understanding Tool Walkthrough demonstrates CodeSurfer, a tool built from the components.
- Applications describes how dependence graphs and slicing can be used to help solve a wide range of software engineering problems.
- Component Details gives further implementation details.

Background and Terminology

Slicing and Chopping. A *backward slice* consists of all program points that affect a given point in the program. For example, the backward slice shown in Figure 1 indicates exactly which statements influence the output of variable *i*. A *forward slice* consists of all program points that are affected by a given point in the program. For example, the forward slice shown in Figure 2 indicates exactly which statements are influenced by the initialization of variable *sum*.

```
void main()
{
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

static int add(int a, int b)
{
    return(a+b);
}
```

Figure 1. Backward slice from `printf("i = %d\n", i);`

```
void main()
{
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

static int add(int a, int b)
{
    return(a+b);
}
```

Figure 2. Forward slice from `sum = 0;`

A *chop* [5, 6] consists of all program points affected between one point (the chop source) and another (the chop target). For example, the chop between the initialization of variable *sum* and the output of variable *i* is empty, reflecting the fact that there is no information flow between the source and target.

Dependence Graphs. The *system dependence graph* (SDG) representation of a program is illustrated in Figure 3 [2]. An SDG is a collection of procedure dependence graphs (PDGs) [3, 4], one for each procedure.

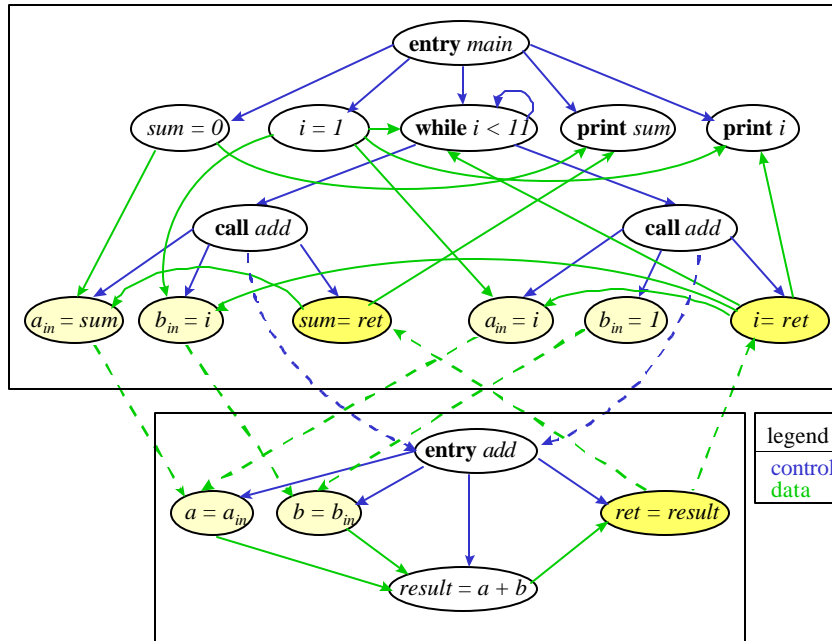


Figure 3. The sample program and its system dependence graph.

The vertices of a PDG represent the individual statements and predicates of the procedure. A call statement is represented by a call vertex and a collection of actual-in and actual-out vertices: there is an actual-in vertex for each actual parameter, and there is an actual-out vertex for each actual parameter that might be modified during the call. Similarly, procedure entry is represented by an entry vertex and a collection of formal-in and formal-out vertices. If a procedure references a global variable, it is treated as a “hidden” input parameter, and thus gives rise to additional actual-in and formal-in vertices. Similarly, if a procedure assigns to a global variable, it is treated as a “hidden” output parameter, and thus gives rise to additional actual-out and formal-out vertices. Each PDG also contains vertices for its local-variable and parameter declarations (not illustrated in Figure 3).

The edges of a PDG represent the control and data dependences among the procedure’s statements and predicates. A vertex is *control* dependent on a predicate (shown in blue) if the predicate controls whether or not the vertex will be executed. An entry vertex is treated as a pseudo-predicate. A vertex is *data dependent* on an assignment (shown in green) if the value assigned can be referenced in the vertex. There is a dependence edge from each declaration vertex to each of its definitions and uses (not illustrated in Figure 3).

The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call vertex to an entry vertex) and by *parameter-in* and *parameter-out* edges (which represent parameter passing and return values, and which run from an actual-in vertex to the corresponding formal-in vertex, and from a formal-out vertex to all corresponding actual-out vertices, respectively).

Implementing Slicing and Chopping. Slices and chops are efficiently computable by reachability analysis in the program’s SDG. For example, the backwards slice from the `printf("i=%d\n", i);` statement in Figure 1, which contains all computations that affect the value printed, can be computed by traversing edges backwards in the SDG from the `print i` node.

Some slicing algorithms are *precise* in the sense that they track dependences transmitted through the program only along paths that reflect the fact that when a procedure call finishes, control returns to the site of the most recently invoked call. Other algorithms are *imprecise* in that they safely, but pessimistically, track dependences along paths that enter a procedure at one call site, but return to a different call site. Precise algorithms are preferable because they return smaller slices. For example, the precise backward slice from the `printf("i=%d\n", i);` statement in Figure 1 does not include the statement `sum=add(sum, 1);` even though it is reachable by traversing edges backwards from the `print i` node. Precise interprocedural slicing and chopping is one of the distinctive innovations of GrammaTech’s technology.

Program Specialization. Slicing allows one to isolate separate “program threads” -- semantically meaningful decompositions of programs -- where the “threads” consist of elements that are not necessarily contiguous. For example, Figure 4 shows a scaled down version of the UNIX word-count utility `wc`. It scans a file, counts the number of lines and characters in the file, and prints the results. The first slice is with respect to the statement that prints the number of characters in the file. The second slice is with respect to the statement that prints the number of lines in the file. The results are, respectively, a program that counts characters and ignores the number of lines, and a program that counts lines and otherwise ignores the number of characters.

Original program	Count characters only	Count lines only
<pre>void line_char_count (FILE *f) { int lines = 0; int chars; BOOL eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOL *bp, int *ip); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { lines = lines + 1; scan_line(f, &eof_flag, &n); chars = chars + n; } printf("lines = %d\n", lines); printf("chars = %d\n", chars); }</pre>	<pre>void char_count (FILE *f) { int lines = 0; int chars; BOOL eof_flag = FALSE; int n; extern void scan_line(FILE *f, BOOL *bp, int *ip); scan_line(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { lines = lines + 1; scan_line(f, &eof_flag, &n); chars = chars + n; } printf("lines = %d\n", lines); printf("chars = %d\n", chars); }</pre>	<pre>void line_count(FILE *f) { int lines = 0; int chars; BOOL eof_flag = FALSE; int n; extern void scan_line2(FILE *f, BOOL *bp, int *ip); scan_line2(f, &eof_flag, &n); chars = n; while (eof_flag == FALSE) { lines = lines + 1; scan_line2(f, &eof_flag, &n); chars = chars + n; } printf("lines = %d\n", lines); printf("chars = %d\n", chars); }</pre>

Figure 4. A scaled-down version of the UNIX word count utility and two of its slices. In each slice, the starting point is shown in italics, and the strikethrough regions indicate which program elements are to be removed from the original program. Items in boldface indicate where new names have been introduced.

Component Architecture

GammaTech’s dependence-graph and slicing technology is implemented as a collection of ANSI C components organized into the following modules:

- Front Ends
- SDG Builder
- Analysis Engine/Slicer
- Transformation Engine
- Scripting Languages
- GUI

The data flow between these modules in a typical end-user tool is shown in Figure 5.

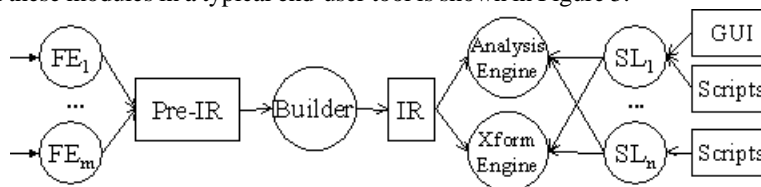


Figure 5. The architecture of the component toolkit

- Front ends for multiple languages (FE_i) emit a common preliminary intermediate representation (Pre-IR) designed to require minimal analysis to produce. The Pre-IR includes abstract syntax trees (ASTs), control flow graphs (CFGs), and symbol table information.
- The Builder performs extensive static analyses to produce the common intermediate representation (IR), which extends the Pre-IR with dependence graphs, i.e., the SDG.
- The Query Engine provides an API to the IR and a collection of built-in primitive analyses (e.g., slicing) that are efficiently implemented by computation over the IR. The API is lifted as an abstract data type to scripting-language interpreters (SL_i).
- The Transformation Engine supports generation of new code from old.

- Clients write scripts to perform complex analyses on the IR in terms of the API and primitive analyses supported by the Query Engine and the Transformation Engine.
- One particular script written in STk (Scheme with Tk widgets) implements an optional GUI.

CodeSurfer, a program understanding tool for ANSI C built with GrammaTech's dependence graph and slicing components, which is illustrated in the next section, has the architecture illustrated in Figure 5.

Program Understanding Tool Walkthrough

This section demos CodeSurfer, GrammaTech's program understanding tool [7]. A project is built by invoking *make* on the program's standard Makefile, with *CC* redefined to invoke CodeSurfer rather than the C compiler. This generates the CFGs for each source file, and the SDG for the project.

Figure 6 shows CodeSurfer's project viewer, which lists the source files, globals and statics, and include files of the project. This particular project, which is one of the SPEC95 benchmarks, contains two files. The area just below the menu bar displays information about the current status of the project. In the figure, the status area indicates that the SDG is up to date for this project, and that no operation has been invoked or is being displayed.

Clicking on one of the [+] icons reveals the list of functions contained in the file, and double-clicking on one of the function names opens a file viewer on the file.

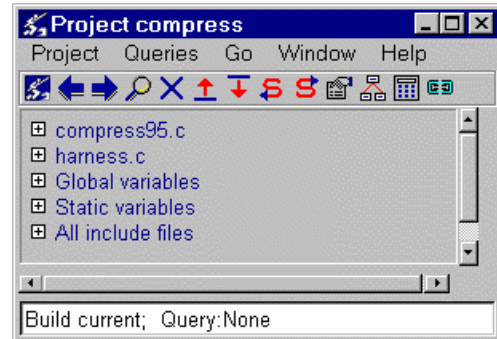


Figure 6. CodeSurfer's project viewer

Figure 7 shows a file viewer after double-clicking on function *decompress*. Text in *italic* indicates preprocessed areas: regions of text that do not correspond to nodes in the SDG (comments, #-directives, and excluded conditionally compiled code) and areas whose correspondence to the SDG is not direct (macro expansions).

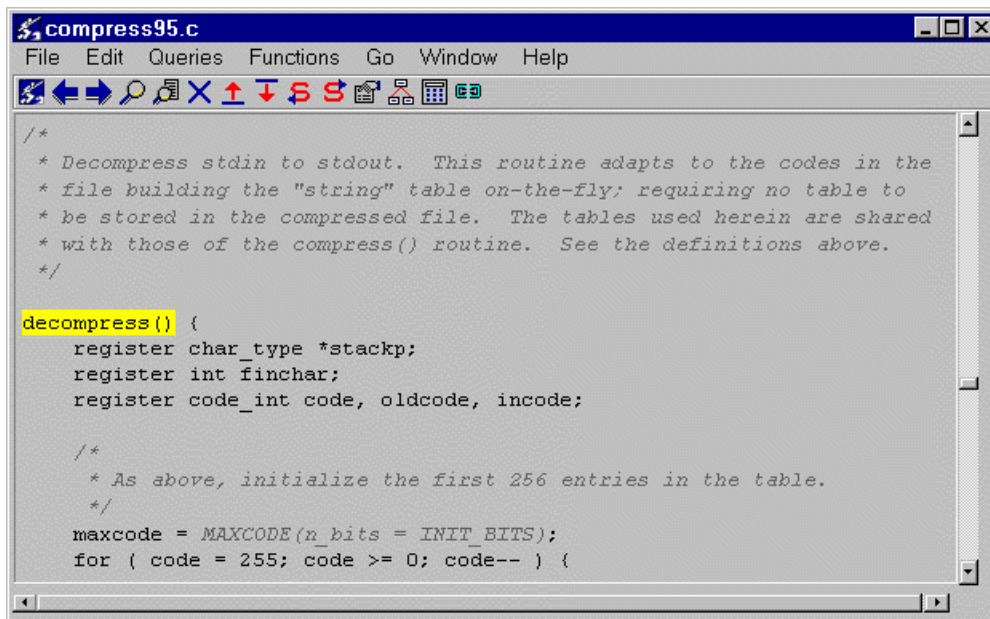


Figure 7. CodeSurfer's file viewer.

The user can select any text in the file and invoke commands to do forward slicing, backward slicing, chopping, or the display of immediate dependence successors or predecessors. The slice points must correspond to nodes in the SDG. If the user selects text that does not intersect with any PDG nodes, then a warning is issued. In general, text is displayed in a distinctive style for each property that holds at that point.

For example, say the user selects function `main` and does a forward slice. This will color every reachable statement in the system red. Figure 8 shows how some properties are displayed. Underlines indicate the query point (i.e., the point at which the slice was initiated) and red indicates the query results (i.e., the points in the slice).

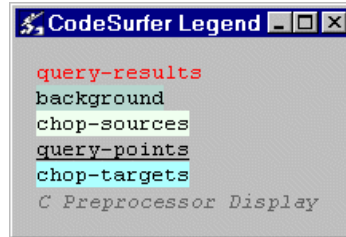


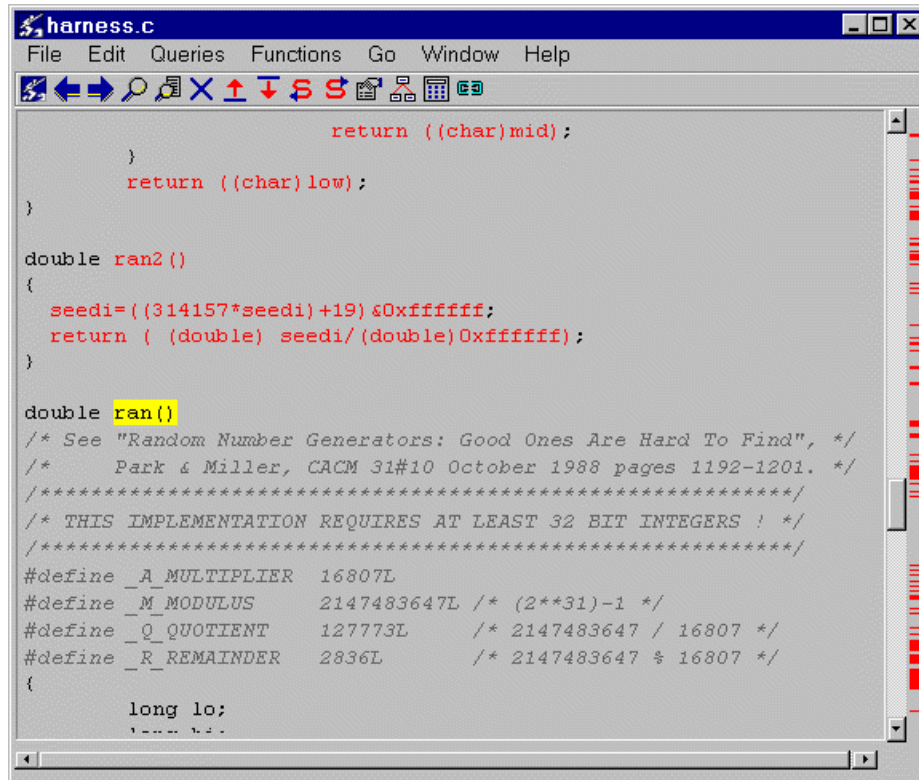
Figure 8. Color coding of text properties.

Figure 9 shows the project window after doing this slice. The status area has been updated to show the kind of operation being displayed, and the number of vertices in the PDG selected by that operation. Evidently, six functions (shown in black) are unreachable. The red bars at the right are explained below in connection with Figure 10.



Figure 9. Slice results displayed in the project viewer.

Double clicking function `ran` opens a file viewer on file `harness.c` and scrolls to the definition of `ran`.



```

return ((char)mid);
    }
    return ((char)low);
}

double ran2()
{
    seedi=(314157*seedi)+19)&0xfffff;
    return ((double) seedi/(double)0xfffff);
}

double ran()
/* See "Random Number Generators: Good Ones Are Hard To Find", */
/*   Park & Miller, CACM 31#10 October 1988 pages 1192-1201. */
/*****
/* THIS IMPLEMENTATION REQUIRES AT LEAST 32 BIT INTEGERS ! */
/*****/
#define _A_MULTIPLIER 16807L
#define _M_MODULUS 2147483647L /* (2**31)-1 */
#define _Q_QUOTIENT 127773L /* 2147483647 / 16807 */
#define _R_REMAINDER 2836L /* 2147483647 % 16807 */
{
    long lo;
    .....

```

Figure 10. Slice results displayed in the file viewer.

Program elements that are in the slice result are shown in red. The tick marks in the *summary bar* on the right indicate where slice results are located in the file. The length of the summary bar represents the length of the file. A tick mark half way down the summary bar indicates a slice result half way through the file. Large empty spaces in the summary bar indicate either dead code, comments, code conditionally compiled out, or commented out code. For example, the first part of function `ran` is not red because it does not correspond to any vertices in the SDG. The next part of function `ran` is not red because it is not reachable from `main`.

Figures 12 through 16 illustrate the utility of backward slicing. The user has selected the loop test `code>=0` in function `decompress` and invoked the backward-slice operation. The loop test is data dependent on the two statements that can assign the value of `code` being tested: `code=255` and `code--`. The loop test is also control dependent on the entry to procedure `decompress`: unless `decompress` gets called, the test will not be executed.

```

compress95.c
File Edit Queries Functions Go Window Help
/*
 * Decompress stdin to stdout.  This routine adapts to the codes in the
 * file building the "string" table on-the-fly, requiring no table to
 * be stored in the compressed file.  The tables used herein are shared
 * with those of the compress() routine.  See the definitions above.
 */

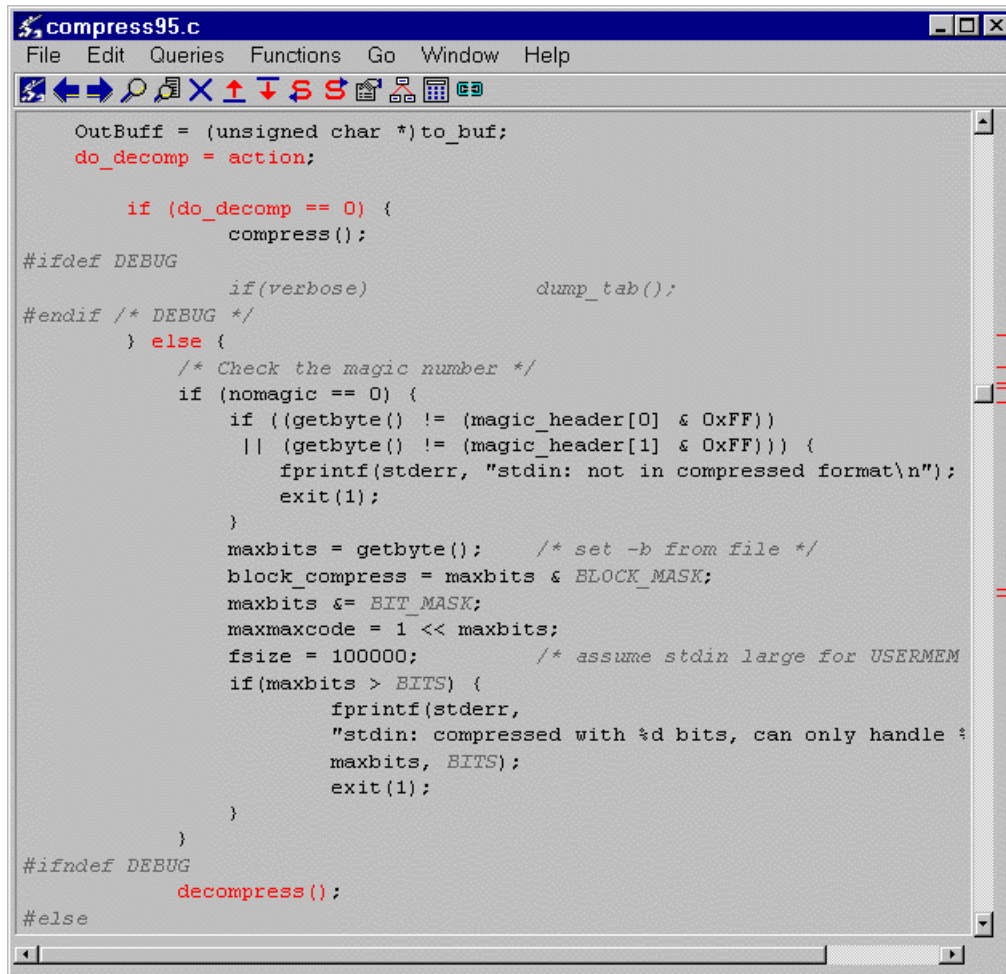
decompress() {
    register char_type *stackp;
    register int finchar;
    register code_int code, oldcode, incode;

    /*
     * As above, initialize the first 256 entries in the table.
     */
    maxcode = MAXCODE(n_bits = INIT_BITS);
    for ( code = 255; code >= 0; code-- ) {
        tab_prefixof(code) = 0;
        tab_suffixof(code) = (char_type)code;
    }
    free_ent = ((block_compress) ? FIRST : 256 );
}

```

Figure 12. Surfing the backward slice from within function `decompress`.

Clicking on the tick marks earlier in the file scrolls to other program points in the slice. Alternatively, a dependence link can be selected from a pop up or property sheet and you can surf there. Figure 13 shows the part of the file that includes the call to function `decompress`, which is contained in the else-branch of a conditional statement that tests variable `do_decomp`. It also shows that `do_decomp` gets its value from `action`.



```

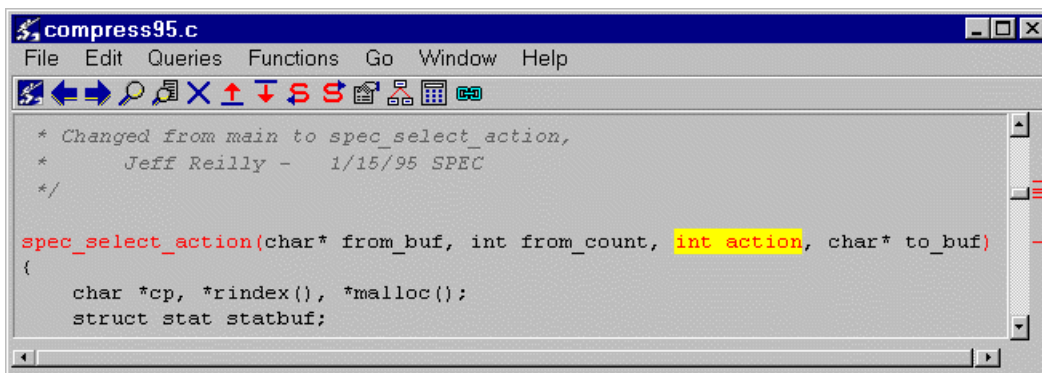
compress95.c
File Edit Queries Functions Go Window Help
OutBuff = (unsigned char *)to_buf;
do_decomp = action;

    if (do_decomp == 0) {
        compress();
#ifdef DEBUG
        if(verbose)            dump_tab();
#endif /* DEBUG */
    } else {
        /* Check the magic number */
        if (nomagic == 0) {
            if ((getbyte() != (magic_header[0] & 0xFF))
                || (getbyte() != (magic_header[1] & 0xFF))) {
                fprintf(stderr, "stdin: not in compressed format\n");
                exit(1);
            }
            maxbits = getbyte(); /* set -b from file */
            block_compress = maxbits & BLOCK_MASK;
            maxbits &= BIT_MASK;
            maxmaxcode = 1 << maxbits;
            fsize = 100000; /* assume stdin large for USERMEM */
            if(maxbits > BITS) {
                fprintf(stderr,
                    "stdin: compressed with %d bits, can only handle %d\n",
                    maxbits, BITS);
                exit(1);
            }
        }
#ifdef DEBUG
        decompress();
#endif
    }
#endif /* DEBUG */
} else

```

Figure 13. Surfing the backward slice from within function `decompress`, continued.

Selecting a dependence link from a pop up at `do_decomp=action;` navigates to the place where `action` gets its value. This is the header of function `spec_select_action`, where formal parameter `action` is declared, as illustrated in Figure 14.



```

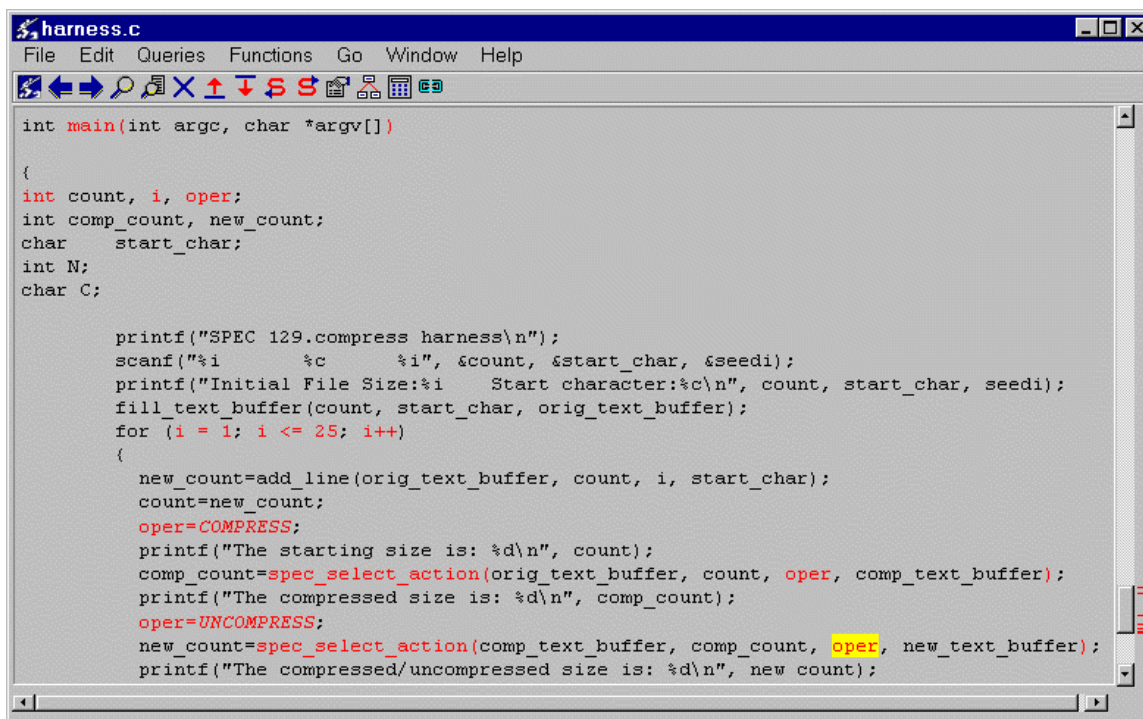
compress95.c
File Edit Queries Functions Go Window Help
* Changed from main to spec_select_action,
*   Jeff Reilly - 1/15/95 SPEC
*/

spec_select_action(char* from_buf, int from_count, int action, char* to_buf)
{
    char *cp, *rindex(), *malloc();
    struct stat statbuf;

```

Figure 14. Surfing the backward slice from within function `decompress`, continued.

And yet again selecting one of the dependence links from the pop up at the definition of function `spec_select_action` navigates to its caller(s), as illustrated in Figure 15. Actual parameter `oper`, corresponding to formal parameter `action`, is in the slice, as are the definitions of `oper`.



```

harness.c
File Edit Queries Functions Go Window Help
int main(int argc, char *argv[])
{
  int count, i, oper;
  int comp_count, new_count;
  char start_char;
  int N;
  char C;

  printf("SPEC 129.compress harness\n");
  scanf("%i %c %i", &count, &start_char, &seedi);
  printf("Initial File Size:%i Start character:%c\n", count, start_char, seedi);
  fill_text_buffer(count, start_char, orig_text_buffer);
  for (i = 1; i <= 25; i++)
  {
    new_count=add_line(orig_text_buffer, count, i, start_char);
    count=new_count;
    oper=COMPRESS;
    printf("The starting size is: %d\n", count);
    comp_count=spec_select_action(orig_text_buffer, count, oper, comp_text_buffer);
    printf("The compressed size is: %d\n", comp_count);
    oper=UNCOMPRESS;
    new_count=spec_select_action(comp_text_buffer, comp_count, oper, new_text_buffer);
    printf("The compressed/uncompressed size is: %d\n", new count);
  }
}

```

Figure 15. Surfing the backward slice from within function `decompress`, continued.

Figure 15 shows one of the approximations made by slicing: both calls to `spec_select_action` are in the slice even though only the second can result in a call to `decompress`. However, discovering this would require constant propagation along paths, something slicing does not do.

Finally, Figure 16 shows the same slice in the project viewer.

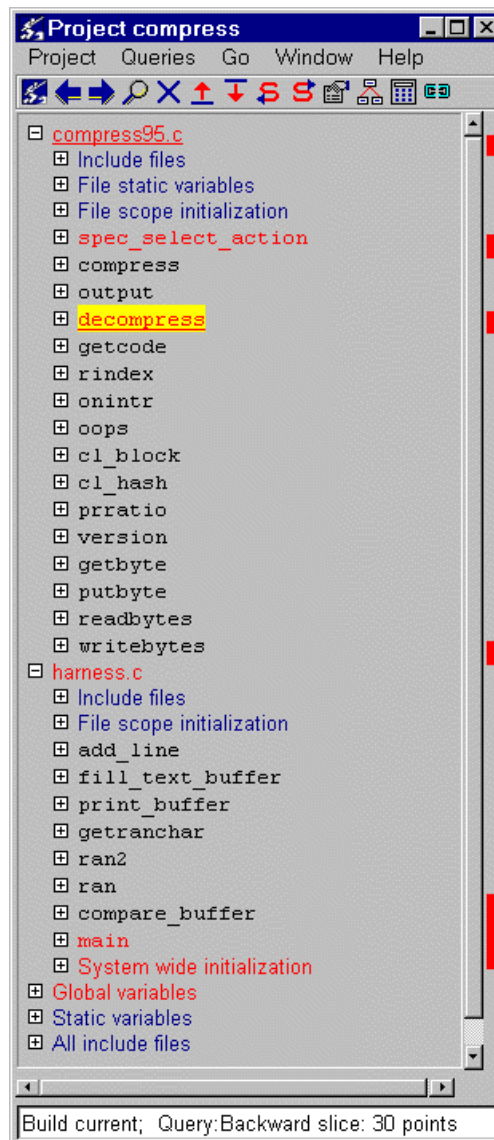


Figure 16. Backward slice shown in project viewer.

Applications

Slicing is a broadly applicable static program-analysis technique. Applications of slicing include program understanding, debugging, testing, parallelization, re-engineering, maintenance, etc. Here are some examples of how slicing helps with these tasks:

- **Program Understanding.** Slicing can be used to help engineers understand code. For example, a backward slice from a point in the program identifies all parts of the code that contribute to that point. A forward slice identifies all parts of the code that can be affected by the modification to the code at the slice point.
- **Program Restructuring.** Slicing isolates “computational threads”, which helps in identifying logical components. For example, the slices in the program in Figure 4 identify the threads that compute the number of characters and lines independently. The threads can be extracted and either replaced or used to create new programs.
- **Program Differencing.** Slices can be compared to identify semantic changes, not just textual or syntactic differences.
- **Testing (and Retesting).** Suppose a proposed program modification only changes the value of variable v at program point p . If the forward slice with respect to v and p is disjoint from the coverage of regression test t , then test t does not have to be rerun. Suppose a coverage tool reveals that a use of variable v at program point p has not been tested. What input data is required in order to cover p ? The answer lies in the backward slice of v with respect to p .
- **Program Specialization and Reuse.** Executable slices can be thought of specialized programs. Slices can be used to make code reuse more efficient. Instead of *reusing* an entire package, a slice can be used to identify only those parts that are really needed.
- **Safety/Security Validation.** *Slicing* and chopping support software inspections for validation of safety-critical and security-critical software. As illustrated below in Figure 17, the chop from a high-security input to a low-security output reveals improper information flows from high to low.

The dependence-graph representation of programs can be used in many other software-engineering and software-engineering *applications*:

- **Anomaly detection.** The dependence graphs can be used to aid discovery of suspect code, e.g., possible uses of uninitialized variables, dereferencing of *null* pointers, dead code, etc.
- **Feature extraction.** Implementations of *different* program features are often interwoven. Set-theoretic operations on program slices can be used to identify code that is unique to a given feature.
- **Design and architecture recovery.** The structure of complex systems can be discovered by analysis (e.g., graph reduction) of control and flow *dependences*.
- **Cliché recognition.** Uses of a given programming cliché can be identified by pattern matching in the dependence graph. Such uses of clichés are candidates for *replacement* by applications of a common abstraction.
- **Optimization.** Slicing a class library from the client’s uses can eliminate both dead code and dead instance variables, leading to reduced program *footprints*.
- **Automatic differentiation.** Automatic differentiation systems transform a program that computes a numerical function, say $F(x)$, into a related *program* that computes the derivative, $F'(x)$. Slicing can be used to generate more efficient derivative-computing programs.

```

informationflow.c
File Edit Queries Functions Go Window Help
/* 1. No flow from high channel to low channel. */
hi = read_high();
lo = read_low();
write_high(f1(hi));
write_low(f2(lo));
write_high(f3(lo));
write_low(cache);

/* 2. Flow from high channel to low channel
via data dependence. */
hi = read_high();
lo = read_low();
lo = f4(lo);
hi = f5(hi);
lo = f6(hi);
write_high(hi);
write_low(lo);

/* 3. Flow from high channel to low channel
via control dependence. */
hi = read_high();
lo = 0;
if (hi){
    write_high(1);
    lo = 1;
    write_high(1);
}
write_low(lo);

/* 4. Flow from high channel to low channel
via global variable. */
save(read_high());
write_low(cache);

```

Figure 17. A chop between a high-security input (the return value of procedure `read_high`) and a low-security output (the parameter to procedure `write_low`) reveals the possible information flows.

Component Details

This section describes the dependence graph and slicing components in slightly greater detail.

The ANSI C front end consists of

- A customized C preprocessor that maintains accurate source references.
- A parser that outputs the AST.
- A control-flow analyzer that outputs the CFG, with each node annotated with def, use, and possible-kill sets. This is an initial approximation without pointer analysis.
- Files that relate items in the various derived program representations to locations in the source code (not shown in Figure 5).

The SDG builder produces the PDGs for the procedures and the SDG for the program in the following phases:

- Builds an initial approximation of the call graph (without indirect-procedure-call analysis).
- Performs global variable analysis, again without pointer analysis.
- Does indirect-procedure-call analysis. This expands each site of an indirect procedure call to a set of possible calls.
- Does pointer analysis. This is used to augment the def, use, and possible-kill sets of each CFG node, as well as the global-variable modification and use sets for each procedure.
- Constructs the PDG by doing control-dependence and flow-dependence analysis
- Optionally compresses the PDG so that each strongly connected region is represented by one node.

- Brings together the PDGs and the call graph to form the SDG.
- Computes summary edges for procedures that describe dependences between the inputs and the outputs of each procedure.
- Outputs the PDG for each file (with summary edges), an SDG file that indexes the PDG files, and a map from PDG nodes to source-text references.

The Analysis Engine/Slicer provides the following services:

- Forward and backward slicing from a set of starting points.
- Chopping between set of sources and a set of targets.

The User Interface provides:

- Browsing of projects and project files, with slices and chops highlighted in multiple views.
- Hypertext navigation through the dependence graphs, slices and chops.

References

1. K.J. Ottenstein and L.M. Ottenstein. *The program dependence graph in a software development environment*. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984, pp 177-184.
2. T. Reps, S. Horwitz, and D. Binkley. *Interprocedural slicing of computer programs using dependence graphs*. Patent No. 5161216. USA. 1992, Wisconsin Alumni Research Foundation.
3. J. Ferrante, K. Ottenstein, and J. Warren. *The program dependence graph and its use in optimization*. ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3(July) 1987, pp. 319-349.
4. J. Beck and D. Eichmann. *Program and interface slicing for reverse engineering*. In Proceedings of the Working Conference on Reverse Engineering, Baltimore, MD. IEEE Computer Society Press, 1993, pp. 134-143.
5. D. Jackson and E.J. Rollins. *Chopping: A generalisation of slicing*. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.
6. T. Reps and G. Rosay. *Precise interprocedural chopping*. in Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995.
7. CodeSurfer User Guide and Reference: <http://www.grammatech.com/csrf-doc/manual.html>

Acknowledgements

This work was partially supported by ONR contract N00014-97-C-0072 and DARPA/ITO contract DAAH01-99-C-R192.

GrammaTech, Inc.

One Hopkins Place
Ithaca, NY 14850
(607) 273-7340 (voice)
(607) 273-8752 (fax)

<http://www.grammatech.com>

admin@grammatech.com

3/1/00