Software Inspection Using CodeSurfer

Paul Anderson, Tim Teitelbaum

I. Introduction

Software Inspection is a technique for detecting flaws in software before deployment. It was introduced by Fagan in 1976 [12], and since then its use has spread widely.

Despite the widespread adoption and success of software inspection, many software products continue to be released with large numbers of flaws. This can partly be attributed to the inherent complexity of software systems. The complexity of the software thwarts manual attempts to comprehend it.

Furthermore, the ideal situation for conducting software inspections in the field may often not be feasible. Time, geographical, or other constraints may mean that the original author of the code is not available to explain the structure of the code or describe its intended properties. Documentation may be misleading or even missing. General-purpose program understanding tools are crucial if code is to be inspected efficiently. However such tools until now have mostly operated on the surface-level syntactic features of the code.

Yet another difficulty is raised by the fact that safety or secuiry requirements of software may be extremely difficult to show using manual techniques. For example, regulatory authorities that specify standards for safety-critical programs such as the Federal Aviation Authority (FAA) or the Nuclear Regulatory Commision (NRC) sometimes require that programs involved in the control of components have specific properties such as "part A must be independent of part B". It is a difficult and error-prone process for a human to determine whether these properties hold for a program.

We believe that tools that allow reasoning about the deep structure of the code at a high level of detail will be extremely useful for doing software inspections. In this paper we describe how our own system—*CodeSurfer*[1]—provides access to and queries on the system-dependence graph representation of a program for the purposes of helping with software inspections.

GrammaTech, Inc., 317 N. Aurora St., Ithaca NY 14850. {paul,tt}@grammatech.com

Partially supported by DARPA contracts F30602-00-0080: Dependence Graphs for Information Assurance of Systems (OASIS), DAAH01-99-C-R192: Multi-Lingual Dependence-Graph Components for Software and Hardware Analysis, Design, and Specialization, and DAAH01-01-C-R129: Verification of Hierarchical Graph Structures (SBIR)

[1]CodeSurfer is a registered trademark of GrammaTech, Inc.

The remainder of the paper is structured as follows. Section II presents some basic material on dependence graphs. Section III describes CodeSurfer—our system for program understanding. Section IV describes how queries on the system dependence dependence graph can be used for software inspection. Section V describes using model checking techniques on the control-flow graph to reveal underlying flaws in the software. Section VI describes the ways in which CodeSurfer has been designed to be open and extensible. Section VII shows how this work relates to other work in software inspection. Finally, Section VIII concludes with a brief description of future work planned.

II. Dependence Graphs

Dependence graphs have applications in a wide range of activities, including parallelization [4], optimization [13], reverse engineering, program testing [2], and software assurance [17].

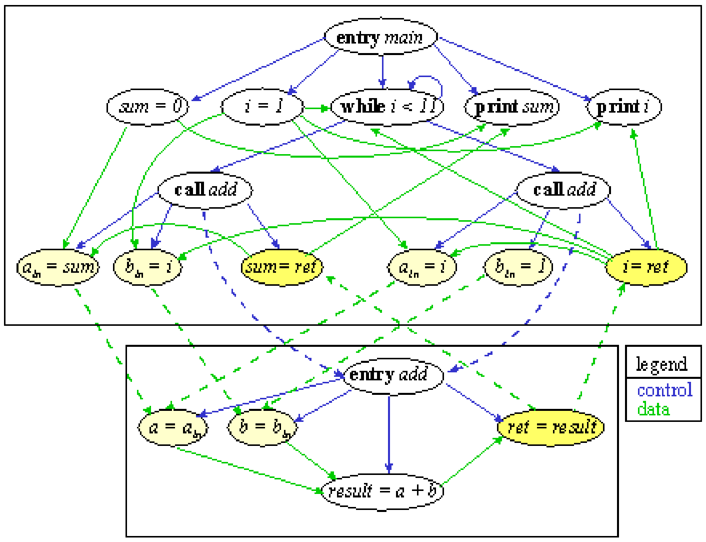
Figure 1 shows the dependence graph representation for a simple program with two procedures.

A *Program Dependence Graph* (PDG) [13] is a directed graph for a single procedure in a program. The vertices of the graph represent constructs such as expressions, call sites, parameters, and predicates. The edges between the vertices indicate either a data dependence or a control dependence. The data dependence edges are essentially data flow edges. For example, in Figure 1, there is a data dependence between the point `i=1` and the point `while (i < 11)` indicating that the value of `i` flows between those two points.

A control dependence edge between a source vertex and a destination vertex indicates that the result of executing the source vertex controls whether or not the destination vertex is reached. For example, in Figure 1, there is a control-dependence edge between the vertex representing the point `while (i < 11)` and the call site to the function `add`.

A *System Dependence Graph* (SDG) is a directed graph consisting of interconnected PDGs [18], one per procedure in the program. Interprocedural control-dependence edges connect procedure call sites to the entry points of the called procedure. Interprocedural data-dependence edges represent the flow of data between actual parameters and formal parameters (and return values).

Non-local variables such as globals, file statics, and variables accessed indirectly through pointers are handled by



```
void main()
{
  int sum, i;
  sum = 0;
  i = 1;
  while (i<11) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  print(sum);
  print(i);
}

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

Fig. 1. *The System Dependence Graph for the small program shown on the right. Each function is represented as a collection of program points (shown in ovals) connected by edges (shown as arrows). There are different kinds of program points, e.g.,* entry, call-site, *etc.*

modelling the program as if it used only local variables. Each non-local variable used in a function, either directly or indirectly, is treated as a "hidden" input parameter, and thus gives rise to additional program points. These serve as the function's local working copy of the non-local variable.

The PDG/SDG-based representation subsumes the notion of call graphs and data flow graphs. Numerous additional intermediate program representations are generated in the course of constructing a high-precision SDG for a program. These include the following:

• The program's Abstract Syntax Tree (AST), with symbol table and full type information.
• The Control-Flow Graphs (CFG) and post-dominance graph.
• The Points-to Graph. This is a directed graph with vertices corresponding to variables (and structure fields, array elements and procedures), and edges indicating what values can point to which locations [22]. The pointer analysis algorithms used are those of Andersen [1], Steensgaard [35] and Das [8].
• Variable def/use information. The set of all variables whose values are taken or modified at all points in the program.
• The Call Multi-Graph. This includes calls made indirectly through function pointers variables.
• PDGs in which references to non-local variables of a procedure are modeled by turning such variables into extra (hidden) parameters.

A. Dependence Graph Queries

A number of queries on the dependence graphs are defined. The *backward slice* from a program point P includes all points that may influence whether control reaches P, and all points that may influence the values of the variables used at P when control gets there. The *forward slice* from P includes all program points affected by the computation or conditional test at P [37].

A program *chop* between a set of source program points S and a set of target program points T reveals how S can affect the state of the program at T [30].

These query algorithms can not be implemented using simple graph reachability — they must only return results that correspond to feasible executions of the program. A path that enters a procedure through a call site can only exit the procedure by going back to the call site from whence it came. We refer to queries on the dependence graph as being *precise interprocedural* if they follow this regime.

Precise interprocedural queries are implemented in CodeSurfer using *context-free language reachability* [29].

In order to do context-free language reachability on a graph, the edges in the graph are labelled with symbols. A *valid path* is one where the labels on the edges spell out a sentence in a context-free grammar.

It is a simple matter to construct a context-free grammar that models the call-return paths that correspond to the valid execution of a program. Let each call site in the program be given a unique index ranging from 1 through

N.

Let each interprocedural edge leaving from call site i be labelled $(_i$, and each interprocedural edge returning to call site i be labelled $)_i$. Let all other edges be labelled x.

The grammar that gives rise to a precise interprocedural path in the SDG is one where the parentheses are matched. The following grammar specifies paths that are completely balanced by calls and returns:

$$\begin{array}{rcll} \textit{matched} & \rightarrow & \textit{matched} \;\; \textit{matched} & \\ & \mid & (_i \;\; \textit{matched} \;\;)_i & 1 <= i <= N \\ & \mid & x & \\ & \mid & \epsilon & \end{array}$$

A grammar that can be used to compute a slice can be written as follows:

$$\begin{array}{rcll} \textit{realizable} & \rightarrow & \textit{matched} \;\; \textit{realizable} & \\ & \mid & (_i \;\; \textit{realizable} & 1 <= i <= N \\ & \mid & \epsilon & \end{array}$$

Note that the starting point for a slice can be in a procedure F called by a procedure G. The grammar given above allows the path to proceed into F without having to return back to G.

Context-free language reachability is $O(n^3)$ in the number of edges in the graph. However, a preprocessing step computes *summary edges* that summarize the transitive dependence at call sites. This step, although also $O(n^3)$ in the number of edges allows precise interprocedural queries to be computed later in linear time [19].

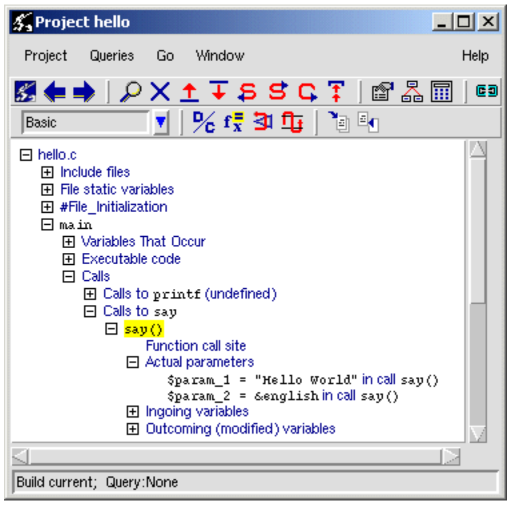
Note that unstructured inter-procedural control flow (such as that induced by throwing exceptions) can be modeled this way as described in [34].

III. CodeSurfer

CodeSurfer is a static analysis tool designed to support advanced program understanding based on the dependence-graph representation of a program. CodeSurfer is thus named because it allows surfing of programs akin to surfing the world-wide web.

CodeSurfer computes all of the above intermediate forms, and the entire system dependence graph for a program in advance. The dependence-graph queries discussed above are all implemented as primitive operations on the graph. All CodeSurfer operations operate by accessing these data structures directly, or by invoking the built-in dependence graph queries.

A number of viewers allow the user to access this information in a user-friendly manner. These viewers are connected by hypertext links. Some of the viewers are described below.



```
hello.c
  Include files
  File static variables
  #File_Initialization
  main
    Variables That Occur
    Executable code
    Calls
      Calls to printf (undefined)
      Calls to say
        say()
          Function call site
          Actual parameters
            $param_1 = "Hello World" in call say()
            $param_2 = &english in call say()
          Ingoing variables
          Outcoming (modified) variables
```

Build current; Query:None

Fig. 2. *A Project Viewer for a small program. This shows how CodeSurfer organizes the target code in terms of its program points.*

• The *Project Viewer* shows the program organized hierarchically by file, then by function. Figure 2 shows a screen shot of the CodeSurfer project viewer.

• The *File Viewer* displays the source file. Tokens that give rise to vertices in the dependence graph are hypertext links in the file viewer. Figure 4 shows a file viewer for a small program.

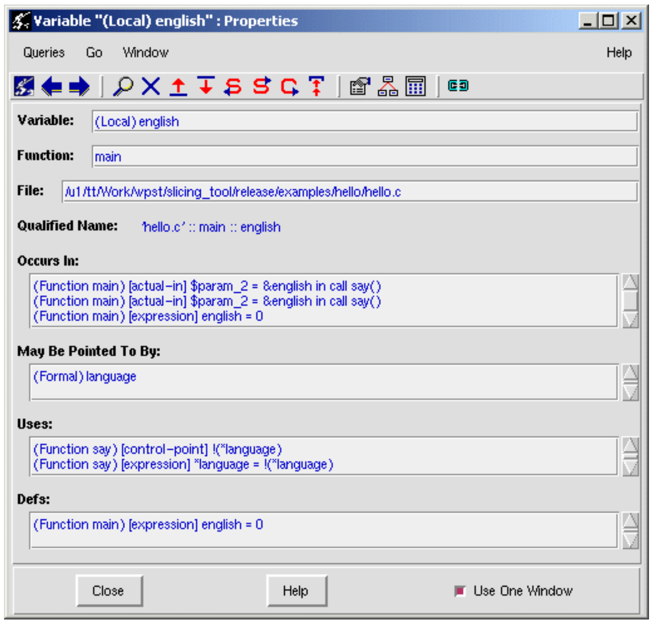
• The *Call Graph Viewer* shows the call graph for the program. In this view, the edges are hyperlinks to all the call sites.

• *Property Sheets* are available for most program elements. For example, the property sheet for a variable will show where the variable occurs, where its value is used, where its value is assigned, where it may point to, and what other variables may point to it. Figure 3 shows a property sheet for a variable.

• The *Finder* allows searching through the program for occurrences of strings, or for particular functions or variables. For variables, the user can request declarations, occurrences, uses, and assignments. Attention can be restricted to globals, file statics, function statics, formal parameters, and/or locals. All variables that point to, or that are pointed to by, a given variable can be shown.

• The *Set Calculator* allows direct manipulation of the sets of points in the program. It provides a palette of logical set operations including as union, intersection and difference.

CodeSurfer provides a number of queries on the system dependence graphs that can be used for program understanding, or for finding flaws in the program. The next two sections describe these queries.



Variable: (Local) english

Function: main

File: /u1/tt/Work/wpst/slicing_tool/release/examples/hello/hello.c

Qualified Name: 'hello.c' :: main :: english

Occurs In:

```
(Function main) [actual-in] $param_2 = &english in call say()
(Function main) [actual-in] $param_2 = &english in call say()
(Function main) [expression] english = 0
```

May Be Pointed To By:

```
(Formal) language
```

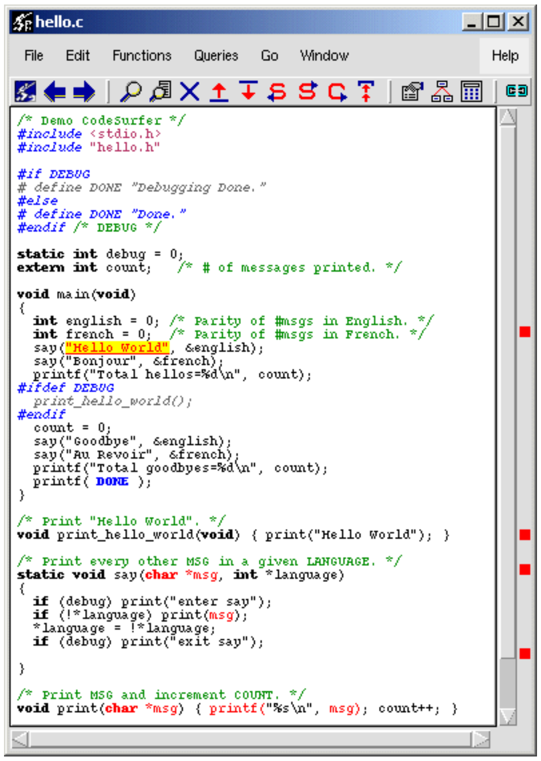
Uses:

```
(Function say) [control-point] !(*language)
(Function say) [expression] *language = !(*language)
```

Defs:

```
(Function main) [expression] english = 0
```

Fig. 3. *A Property Sheet for a variable in the program. Most items shown are themselves hypertext links to other viewers. For example, selecting one of the Uses entries will navigate to a File Viewer and position it at that point.*



```
/* Demo CodeSurfer */
#include <stdio.h>
#include "hello.h"

#if DEBUG
# define DONE "Debugging Done."
#else
# define DONE "Done."
#endif /* DEBUG */

static int debug = 0;
extern int count;   /* # of messages printed. */

void main(void)
{
  int english = 0; /* Parity of #msgs in English. */
  int french = 0;  /* Parity of #msgs in French. */
  say("Hello World", &english);
  say("Bonjour", &french);
  printf("Total hellos=%d\n", count);
#ifdef DEBUG
  print_hello_world();
#endif
  count = 0;
  say("Goodbye", &english);
  say("Au Revoir", &french);
  printf("Total goodbyes=%d\n", count);
  printf( DONE );
}

/* Print "Hello World". */
void print_hello_world(void) { print("Hello World"); }

/* Print every other MSG in a given LANGUAGE. */
static void say(char *msg, int *language)
{
  if (debug) print("enter say");
  if (!*language) print(msg);
  *language = !*language;
  if (debug) print("exit say");

}

/* Print MSG and increment COUNT. */
void print(char *msg) { printf("%s\n", msg); count++; }
```

Fig. 4. *A File Viewer for a small program. In this example, the user has selected the first parameter to the first call to the procedure* **say***, and has invoked a forward slice query. All points that depend on this parameter are shown in red*

IV. Queries for Software Inspection

Many of the features of CodeSurfer have been designed for program understanding, and as such are useful for detailed software inspection. This section describes some of the queries and their application to software inspection.

A. Variable Usage Information

Each point in the program may access some variables or modify some variables, each possibly through pointers. In order to compute the data dependence graph, the set of variables used and defined at each program point are first computed and associated with the vertex that represents that program point.

This information is easily accessed by the user. For example, in Figure 3 shows the property sheet for the variable named `english`. The **Defs:** section of the property sheet indicates that the variable is only assigned to in one place—the expression `english = 0`. The **Uses:** part shows that the value of the variable has its value taken at two points, and that both of them are through a pointer.

Furthermore, the set of variables that can be used and/or modified for each procedure, either directly or transitively through a callee, is also computed. This can be used to answer questions of the form "Can global variable G be modified if function F is called". The user can view this information directly through the Project Viewer.

B. Predecessors/Successors

It is natural for a user attempting to understand a program to ask "How could variable x have gotten its value here?", or alternatively "Where is the value generated at this point used?". The predecessor and successor operations provide the answer to these questions. These queries can be posed for the control dependences, the data dependences, or both. A program point's *data predecessors* are the points where the variables used at that point may have gotten their values. The *data successors* are the points where the variables that were modified at that point are used.

The predecessors and successors queries are implemented using the context-free language reachability algorithm on the dependence graph as described above in Section II-A.

The fact that the query is done directly on the dependence graph guarantees that the result will be correct with respect to the data flow properties of the program. For example consider the example in figure 5. If the predecessors query is invoked from line 5, lines 3 and 4 will be in the result. Line 1 will *not* be in the result because the value of `x` assigned on line 1 can never reach the use of `x` on line

5, because there is an assignment that kills it on line 3. Similarly the assignment to `w` on line 2 can never reach line 5 because of the kill on line 4.

```
1:    x = 100;
2:    w = x;
3:    x = 1;
4:    w = 10;
5:    z = x + w;
```

Fig. 5. A simple program fragment used to illustrate the built-in queries. The underlining indicates the starting point for the queries discussed in the text.

The fact that the the query is done using context-free language reachability guarantees that paths that do not correspond to valid paths through the program are not considered.

There are variants on these queries to narrow down the set of starting points for a query. *Point mode* is the default mode. As described above, a point mode query from line 5 yields lines 3 and 4.

Point-and-variable mode allows the user to restrict the set of starting points to those involving a set of variables. When invoked the user is prompted for the set of variables to be considered. For example, a point-and- variable mode predecessors query starting at line 5 in Figure 5 with respect to variable `x` yields line 3.

In *variable* mode the input is a set of variables and is independent of a starting point. In the example in Figure 5, a variable mode predecessors query with respect to variable `x` yields lines 1 and 2.

C. Slicing

A backward slice with respect to a set of starting points S answers the question "What points in the program does S depend on?". The control dependence edges are used to determine how control could have reached S, and the data edges are used to determine how the variables used at S were computed. A forward slice with respect to a set of starting points S answers the question "What points in the program depend on S".

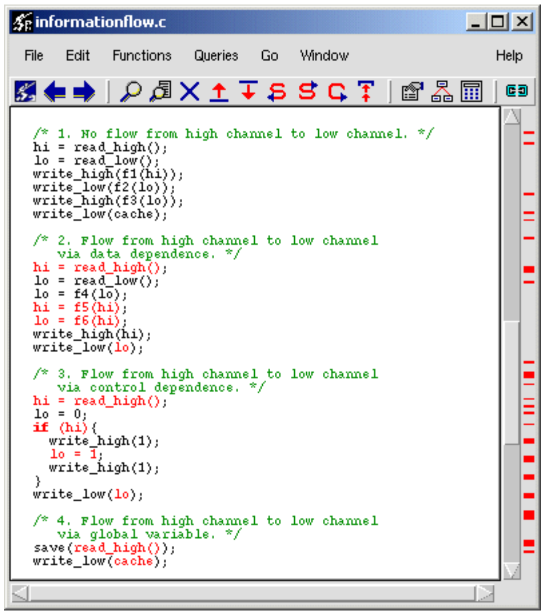
Like the predecessor and successor operations, the slice operations have point and variable modes.

Slices are best used with care. Our experience is that the slicing operations are generally not used much for program understanding as they often to deliver too much information to be easily comprehended.

Figure 4 shows a CodeSurfer File Viewer where a forward slice has been invoked.

D. Chopping

A chop is a point-to-point reachability query in the graph. It answers the question "How does execution of



```
/* 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);
```

Fig. 6. *The result of doing a chop shows how information can flow from one set of points to another. In this example, several points light up as being places where a security policy is being violated.*

the program points at A affect the execution of B?". This query can be used to determine the information flow between points in the program, or to show that two parts of the program are independent.

As mentioned previously, some regulatory agencies have software requirements that specify independence of components. In cases where the components are in the same program, a chop can be used to determine whether the software satisfies these requirements.

For example, the NRL network pump is a device that connects a high security network to a low security network [20]. The security requirement is that data can be transferred from the low side to the high side, but that no information can be allowed to flow from high to low. The exception is that communication acknowledgements *are* allowed to flow from high to low.

This property can be tested using the chop operation. The *sources* of the chop operation will be program points where data is read from the port connected to the high-security network. The *targets* of the chop operation will be the points where the data is written to the low channel. If the chop query returns the empty set, then this shows that the security property holds. If not, then the result is the set of points through which the property is violated.

Figure 6 shows the result of doing a chop on a mockup of the NRL pump. The points in red show those places involved in a violation of the security policy. There is no flow in the first example. The second example is a blatant violation. It is useful to point out that the third example is more subtle. Fresh high data, which is read into `hi`, is used to change `lo` conditionally. The value of `lo` is then written to the low security port. Information is communicated from high to low via control dependence; low receives a 1 if and only if the high input was non-zero. Therefore a single bit of information has leaked. If high were 0-1 valued, this would be perfect information.

D.1 Red/Black separation

One form of independence property common in security applications is known as red/black separation. Values stored in a set of private variables (the red set) must not be allowed to flow to any of the set of public variables (the black set). This is a subtly different requirement than that expressed for the NRL pump. Here, the requirement is expressed in terms of the program's *variables* as opposed to *program points.*

CodeSurfer's *variable* mode can be used to help explore red/black separation properties. A variable-mode chop between a source set of variables V_S and a target set of variables V_T shows all ways in which information can flow between variables. This can be used to determine if a program conforms to the desired red/black separation policies.

V. Model Checking

Model checking is a technique widely used in digital hardware design to check properties of digital circuits [6]. We are adapting these techniques for doing model checking on programs with the goal of discovering programming flaws in systems. A full technical description of the model checker is beyond the scope of this paper. For a description, see [11]. Here we give a brief outline of the approach and describe how it can be used to answer questions that may be raised in detailed software inspections.

In model checking of digital circuits, model checkers operate on a graph where the vertices are the states of the circuit and edges represent state transitions. In contrast, the model-checking approach operates on the program's control-flow graph. Thus it can be thought of as checking the program in terms of all the possible paths through the program.

Unlike digital circuits, where the state space is "flat", the state space for a program's control-flow graph is constrained by the fact that when a call to a function finishes, control can only pass back to the point of call. This is the precisely the same issue that prevents straightforward graph reachability from being used to perform slicing queries, as described in Section II-A. In this case we use a space-efficient version of an algorithm due to Burkart and Steffen [3].

The model checking algorithm is capable of checking formulae in the full modal mu-calculus [21]. Translators can be written to convert formulae in higher-level logics such as Fair CTL into the mu-calculus.

The user interface to the model checker is neither of these logics, but instead a set of prepackaged or "canned" assertions about the behavior of the program in terms of valid execution paths, each of which is parameterisable. When invoked, the model checker evaluates the formula with its parameters, and if the assertion fails, the user is allowed to browse a counter-example in terms of a path through the code.

The parameters to the assertions are atomic propositions that can be specified in terms of the vertex being visited. For example, one query is "There exists a path where X holds until Y". When invoked, the user is prompted to specify the propositions X and Y. These can be specified in terms of a set of predefined functions, or in terms of an arbitrary Scheme function. This function of course has access to the full system dependence graph, so sophisticated queries can be specified.

This mechanism can be put to use for posing queries useful in software inspection. One question that might be asked about a security-sensitive program is whether it contains a backdoor security vulnerability. If the application is a *login* program, then no user should be allowed access without having first gone through an authentication process. First the user would identify the points where the user is given access. These will typically be calls to a function. In the login program they might be calls to `exec()`. Let this set of points be named X. The user would then identify the points at which the authentication of the user is confirmed. Let these points be called Y. The canned query "No path goes through X without going through Y". The resulting query will thus be "No path goes through (all calls to `exec()`) without going through (a call to `authenticate()`)".

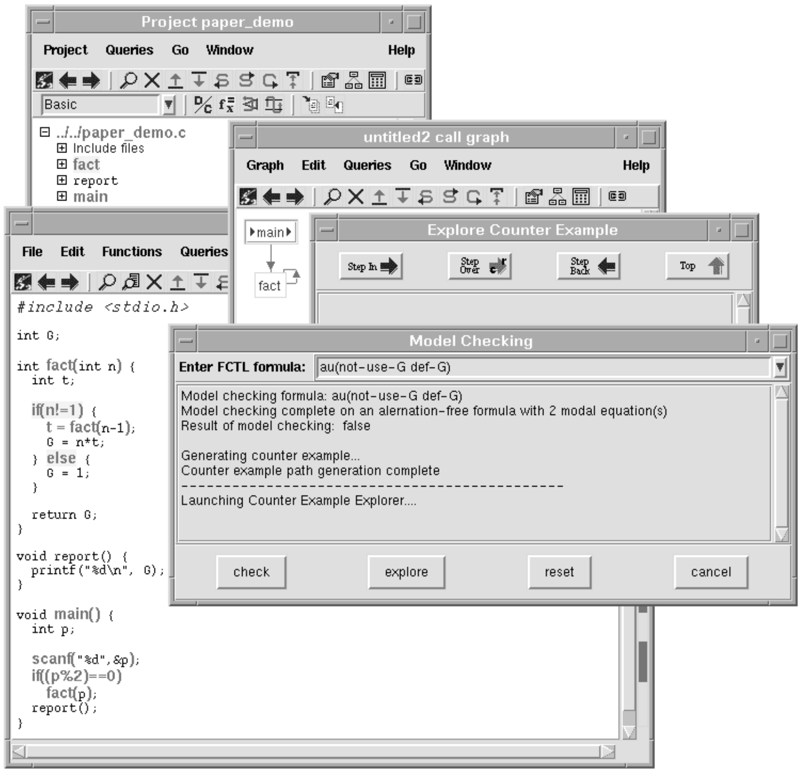
Figure 7 shows a screen shot of CodeSurfer with the model checking interface being used to find possible sources of errors in a program.

VI. Openness and Extensibility

CodeSurfer has been designed to be open and extensible where possible in order to foster users who wish to integrate with other tools, and to encourage users to build tools as add-ons to the system. The following sections describe the various ways in which CodeSurfer can be enhanced or extended.

A. Language.

The language-specific front-end to CodeSurfer produces an intermediate form that consists of a control-flow graph



```
#include <stdio.h>

int G;

int fact(int n) {
  int t;

  if(n!=1) {
    t = fact(n-1);
    G = n*t;
  } else {
    G = 1;
  }

  return G;
}

void report() {
  printf("%d\n", G);
}

void main() {
  int p;

  scanf("%d",&p);
  if((p%2)==0)
    fact(p);
  report();
}
```

Enter FCTL formula: au(not-use-G def-G)

```
Model checking formula: au(not-use-G def-G)
Model checking complete on an alernation-free formula with 2 modal equation(s)
Result of model checking:  false

Generating counter example...
Counter example path generation complete
---------------------------------------------
Launching Counter Example Explorer....
```

Fig. 7. *The model checker has been used to make an assertion about the program which has failed. The GUI provides a means for browsing a path in the program counter-example. The interface for browsing the path is similar to a debugging tool being used to step through the execution of a program*

annotated with variable usage information. As the front end is a separate executable, users can replace it with a front end for another language as long as programs in that language can be expressed in terms of the control-flow graph intermediate form. This has been done by CodeSurfer users for Jovial, Verilog, a subset of VHDL, and Promela—the language for the SPIN model checker [25].

B. Pointer Analysis.

CodeSurfer provides a choice of several pointer analysis algorithms, each of which offers different choices for precision and scalability. The pre-packaged algorithms include those of Andersen [1], Steensgaard [35] and Das [8]. As new pointer analysis algorithms are constantly being developed, the tool was designed so that new algorithms could be easily incorporated. As with the language-specific front end, the pointer analysis algorithms are packaged as separate executables, so they can be easily replaced by a user if necessary.

C. Scripting Language.

The CodeSurfer executable itself is written as a Scheme interpreter extended with the ability to create and manipulate system dependence graphs. The scheme interpreter is based on the STk implementation from Erick Gallesio at the University of Nice [15]. STk is fully integrated with the Tk widget set—the entire CodeSurfer GUI is written in Scheme using these widgets.

D. API to the Dependence Graph

The extensions to the Scheme interpreter introduce several new primitive types and provide a range of operations on them. These correspond to the underlying dependence graph data structures. For example, one type is the PDG; the operation (`pdg-vertices` *G*) returns the set of vertices associated with the program dependence graph *G*. Thus users can extend the system with new kinds of queries, or even new GUI elements. The model-checking application described in section V above is just such an extension.

E. Import/Export formats

CodeSurfer provides the ability to export arbitrary sets of program points to files in a range of different formats. Additionally, some facilities are available for importing file formats and converting them to sets of program points. This mechanism facilitates integration with other tools. The set of standard formats is currently small, but growing. It currently includes *grep* format and *PureCoverage* format. A GXL filter [16] is planned for the future. This feature is also open and extensible. The user can define (in Scheme) new functions for converting sets of program points to external file formats and back again.

F. Set Calculator Operations

The set calculator provides the ability to manipulate sets of program points. The operations in the set calculator can be extended, again using Scheme, by an end user.

VII. Relationship with other work

Many Software Inspection Tools focus on groupware for the management of the software inspection process. These tools include ICICLE [33], ASSIST [23], Suite [9] A comparison of such tools can be found in [24]. Few tools have been for detailed fine-grain inspection of software, although ICICLE does allow users to run `lint` on the C source files during the inspection.

Dunsmore [10] argues for a greater role of comprehension in the software inspection. There are many tools solely for program understanding [31], [36], [28], but we believe there are none that bring so much static analysis information to the user.

CodeSurfer has some commonality with tools for reverse engineering. These include the DMS Software Reengineering Toolkit [32], Datrix at Bell Canada [5], and the Portable Bookshelf [14]. However, unlike these systems, CodeSurfer makes no attempt to recover architectural information—its analysis is limited to creating a fine-grain system dependence graph. CodeSurfer does not have a general purpose meta-query system in the sense of DMS. Instead it makes do with basic context-free language graph reachability queries that are customizable programmatically. CodeSurfer does not have the ability to transform the program in the style of DMS, or TXL [7].

Other tools that provide a similar level of static analysis as CodeSurfer include other software re-engineering tools such as Refine [27] and Discover [26].

VIII. Conclusion and Future Work

We have described a tool for inspecting and manipulating the dependence-graph representation of a program for the purposes of program understanding. We propose that such a tool will be of use for doing formal software inspections. We have described the means by which the system answers queries about the data flow properties of the program using context-free language graph reachability. We have described using a model checker to answer questions about possible paths through the program.

Work on CodeSurfer is continuing under several research contracts. There are two main thrusts in the development of CodeSurfer. The first is to improve the scalability of the system. This will be achieved partly by improving the efficiency of the pointer analysis algorithms without sacrificing precision, and partly by using demand-driven techniques to reduce the up-front cost of building the dependence graph. The other thrust is to extend the domain of applications for the system. We are currently studying applying the technology to software assurance, and to program testing problems.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19).
[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Symp. on Princ. of Prog. Lang.*, pages 384–396, 1993.
[3] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.
[4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, 1986.
[5] Bell Canada. http://www.iro.umontreal.ca/labs/gelo/datrix.
[6] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.
[7] J. Cordy, I. Carmichael, and R. Halliday. The txl programming language, 1995 1995.
[8] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PDLI'00*, Vancouver, BC, 2000.
[9] J. Drake, V. Mashayekhi, J. Riedl, and W. Tsai. A distributed collaborative software inspection tool: Design, prototype, and early trial. Technical Report TR-91-30, University of Minnesota, August 1991.
[10] A. Dunsmore. Comprehension and visualisation of object-oriented code for inspections. Technical Report EFoCS-33-98, Computer Science Department, University of Strathclyde, 1998.
[11] James Ezick, David W. Richardson, and Tim Teitelbaum. Practical model checking and example generation for context-free processes. Submitted to the Workshop on Software Model Checking, Paris, France, July 23 2001.
[12] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
[13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.
[14] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.
[15] Erick Gallesio. STk Home Page. http://kaolin.unice.fr/STk/.
[16] Richard C. Holt and Andreas Winter. A Short Introduction to the GXL Software Exchange Format. In *WCRE 2000: Working Conference on Reverse Engineering*, Brisbane, Australia, Nov 6 2000.

[17] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411. ACM, New York, May 1992.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.

[19] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New York, NY, December 1994. ACM Press. Available at "http://www.cs.wisc.edu/wpis/papers/fse94.ps".

[20] M.H. Kang, I.S. Moskowitz, and D.C. Lee. A network pump. Technical report, Naval Research Laboratory, 1997. http://www.itd.nrl.navy.mil/ITD/5540/-publications/CHACS/1997/1997kang-ACSAC97.ps.

[21] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[22] W. Landi, B. Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. Technical Report DCS-TR-336, Rutgers University, May 1998.

[23] F. Macdonald. *Computer-Supported Software Inspection*. PhD thesis, Dept. Computer Science. University of Strathclyde, 1998.

[24] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. In *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering (CASE-95)*, Toronto, Canada, July 1995.

[25] Lynn Millett and Tim Teitelbaum. Slicing promela and its applications to model checking, simulation, and protocol understanding. In *SPIN workshop.*, 1998.

[26] MKS. MKS Home Page. http://www.mks.com.

[27] Reasoning, Inc. Reasoning home page. http://www.reasoning.com.

[28] Red Hat Software. The Source-Navigator IDE. http://sources.redhat.com/sourcenav/.

[29] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November 1998. Special issue on program slicing.

[30] T. Reps and G. Rosay. Precise interprocedural chopping. *SIGSOFT 95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (Washington, DC, October 10-13, 1995)*, ACM SIGSOFT Software Engineering Notes*, 20(4), 1995.

[31] Scientific Toolworks, Inc. Understand. http://www.scitools.com/cpp.html.

[32] Inc. Semantic Designs. http://www.semdesigns.com/Products/-DMS/DMSToolkit.html.

[33] V. Sembugamoorthy and L. Brothers. ICICLE: Intelligent code inspection in a c language environment. In *The 14th Annual Computer Software and Applications Conference*, pages 146–154, October 1990.

[34] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineering*, pages 432–441, 1999.

[35] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.

[36] Upspring Software. CodeRover Browser for C/C++. http://www.upspringsoftware.com/products/-coderover/browser_cpp.html.

[37] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.