Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Program Slicing for VHDL*

E. M. Clarke[1], M. Fujita[6], S. P. Rajan[3], T. Reps[4,7], S. Shankar[5], T. Teitelbaum[2,4]

[1] Carnegie Mellon University, e-mail: `emc+@cs.cmu.edu`

[2] Cornell University

[3] Fujitsu Labs of America, e-mail: `sree@fla.fujitsu.com`

[4] Grammatech, Inc. e-mail: `tt@grammatech.com`

[5] Hunter College and the Graduate Center, City University of New York, e-mail: `subash.shankar@hunter.cuny.edu`

[6] University of Tokyo, e-mail: `fujita@ee.t.u-tokyo.ac.jp`

[7] University of Wisconsin, e-mail: `reps@cs.wisc.edu`

The date of receipt and acceptance will be inserted by the editor

Abstract. Hardware description languages (HDLs) are used today to describe circuits at all levels. In large HDL programs, there is a need for source code reduction techniques to address a myriad of problems in design, simulation, testing, and formal verification. Program slicing is a static program analysis technique that allows an analyst to automatically extract portions of programs relevant to the aspects being analyzed. Slicing is fundamentally based on data and control dependences between program points. However, program slicing algorithms have traditionally been designed for sequential languages, and the presence of concurrent constructs such as those found in HDLs complicates slicing considerably. In this paper, we develop the concepts needed for slicing VHDL. The techniques extend readily to other HDLs such as Verilog. Our techniques are based on a slicing-oriented VHDL execution semantics that augments traditional dependences with inter-procedural dependences between VHDL processes. Based on these concepts, we have developed an automatic VHDL slicing tool composed of a traditional slicer and a front-end that captures VHDL execution semantics. This paper discusses our techniques for VHDL slicing, the slicer tool, and slicing applications in design, simulation, testing, and formal verification of VHDL programs. A particular important application of HDL slic-

* This research was sponsored in part by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant Nos. CCR-9505472, CCR-9625667, and CCR-9619219, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

ing is source code reduction leading to reduced state spaces for model checking, and we also present empirical results for this.

Key words: hardware description languages, program slicing, VHDL, model checking, formal verification

1 Introduction

Hardware description languages (HDLs) are used today to describe circuits at all levels from conceptual system architecture to low-level circuit implementations suitable for synthesis. HDL source-code simulation is a common technique for analyzing the resulting descriptions and debugging the design before implementation. Consequently, there is a need for structured design and analysis techniques that can be integrated into existing source-code simulation methodologies. The need for structured development methodologies is becoming even more important with the increased use of reusable libraries of existing code, since it is difficult to use, modify, and/or maintain unstructured libraries. Thus, major needs for source-code simulation include support for testing, debugging, and maintenance of the simulations. There are also several tools that apply model checking ([6]) to formally verify correctness of HDL designs (one such system for VHDL is described in [7]). It is well recognized that the fundamental problem in model checking is state explosion, and there is consequently a need to reduce the size of HDL descriptions so that their corresponding models have fewer states. For many designs, it is not even possible to build the state-transition relation, and the need for HDL program reduction techniques is even more critical in these cases.

Several of these desiderata have close parallels in the software-engineering domain, where it is desirable to understand and manipulate large programs. This is difficult to do, partly because of the presence of large quantities of irrelevant code. *Program slicing* was defined by Weiser [28] to cope with these problems by performing automatic decomposition of programs based on data- and control-flow analysis. A *program slice* consists of those parts of a program that can potentially affect (or be affected by) a *slicing criterion* (i.e., a set of program points of interest to the user). The identification of program slices with respect to a slicing criterion allows the user to reduce the original program to one that is simpler but functionally equivalent with respect to the slicing criterion.

Results of program slicing in the software-engineering world suggest that the techniques can also be applied to HDLs to solve many of the problems mentioned above. However, most traditional program-slicing techniques are designed for sequential procedural programming languages, and the techniques are not directly applicable to HDLs, which have a fundamentally different computation paradigm. An HDL program is a non-halting reactive system composed of a set of concurrent processes, and many HDL constructs have no direct analogue in more traditional programming languages. In this paper we present an approach for slicing VHDL based on its execution semantics. Our approach is based on a mapping of VHDL constructs onto traditional programming language constructs, in a way that ensures that all traces of the VHDL program will also be valid traces in the corresponding sequential program. Corresponding to this approach, we have also implemented a VHDL slicing tool consisting of a VHDL front-end coupled with a language-independent toolset intended for inter-procedural slicing of sequential languages such as C. We have also applied the tool to some formal verification problems, and have achieved substantial state-space reductions.

The remainder of the paper is organized as follows: Section 2 presents requisite background material while Section 3 presents the methods we use for performing language-independent interprocedural slicing. Section 4 shows how we capture VHDL semantics for slicing. Section 5 describes the architecture and implementation of the VHDL slicing tool, and provides a walkthrough of a simple VHDL example. Section 6 lists many applications of slicing, and provides experimental results that concretely illustrate the benefits of slicing in reducing state space size for model checking. We compare and contrast our work with other approaches in Section 7.

Finally, Section 8 summarizes our conclusions and briefly discusses our future plans in this area.

2 Background

Slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions consist of elements that are not necessarily textually contiguous [27,28,22,8,13,14]. (See [26,4] for surveys on slicing.) Slicing, and subsequent manipulation of slices, has applications in many software-engineering tools, including tools for program understanding, maintenance [9], debugging [20], testing [2,1], differencing [12, 11], specialization [25], reuse [21], and merging [12].

In Weiser's terminology, a slicing criterion is a pair $\langle p, V \rangle$ where p is a program point and V is a subset of the program's variables. In his work, a program slice consists of all statements and predicates of the program that might affect the values of variables in V at point p. This is a more general kind of slice than is often needed. Rather than a slice taken with respect to a program point p and an arbitrary variable, one is often interested in a slice taken with respect to a variable x that is defined or used at p. A variable x is said to be *used* at program point p if its contents may be accessed at p, and x is said to be *defined* at p if the execution of p can potentially modify the value of x. The value of a variable y used at p is directly affected by assignments to y that reach p and by the loops and conditionals that enclose p. Similarly, the value of a variable x defined at p is directly affected by the values of the variables used at p and by the loops and conditionals that enclose p. When slicing a program that consists of a single monolithic procedure (which we will call *intra-procedural slicing*, a slice can be determined from the closure of the directly-affects relation. Ottenstein and Ottenstein [22] pointed out how well-suited *program dependence graphs* are for this kind of slicing. Once a program is represented by its program dependence graph, the slicing problem is simply a vertex-reachability problem, and thus slices may be computed in linear time.

There are two kinds of slices: a *backward slice* of a program with respect to a slicing criterion C is the set of all program elements that might affect (either directly or transitively) the values of the variables used at members of C; a *forward slice* with respect to C is the set of all program elements that might be affected by the computations performed at members of C.

A related operation is program *chopping* [17,24]. A chop answers questions of the form "Which program elements serve to transmit effects from a given source element s to a given target element t?". Given a set of source program points S and a set of target program points T, the *chop* consists of all program points that might be affected by assignments performed at S that can affect the values of variables used at T.

It is important to understand the distinction between two different but related "slicing problems":

Version (1) (Closure Slice) The slice of a program with respect to program point p and variable x identifies all statements and predicates of the program that might affect the value of x at point p.

Version (2) (Executable Slice) The slice of a program with respect to program point p and variable x produces a reduced program that computes the same sequence of values for x at p. That is, at point p the behavior of the reduced program with respect to variable x is indistinguishable from that of the original program.

In *intra*procedural slicing, a solution to Version (1) provides a solution to Version (2), since the "reduced program" required in Version (2) can be obtained by restricting the original program to just the statements and predicates found in the solution for Version (1); in *inter*procedural slicing, where a slice can cross the boundaries of procedure calls, it turns out that a solution to Version (1) does not necessarily provide a solution to Version (2) since a slice may contain different subsets of a procedure's parameters for different call instances of the same procedure [13]. However, a solution to Version (1) can always be extended to provide a solution to Version (2) [3]. Our system does closure slicing, with partial support for executable slicing.

A second major design issue is the type of interprocedural slicing. Some slicing and chopping 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 corresponding call [13,14, 24]. In contrast, 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 [29,17]. Precise algorithms are preferable because they return smaller slices. Precise slicing and chopping can be performed in polynomial time [13,14,24]. Our VHDL slicing tool supports precise interprocedural slicing and chopping.

3 Inter-Procedural Slicing

In intra-procedural slicing, slices were obtained by following chains of dependences in the directly-affects relation. Ottenstein and Ottenstein [22] noted that *procedure dependence graphs* (PDGs), which were originally devised for use in parallelizing and vectorizing compilers, are a convenient data structure for slicing. PDGs for the procedures in a program can be combined to form a *system dependence graph* (SDG) [13], upon which our inter-procedural slicing algorithms are based.

The PDG for a procedure is a directed graph whose vertices represent the individual statements and predicates of the procedure. Vertices are included for each of the following constructs:

- Each procedure has an entry vertex.
- Each formal parameter has a vertex representing its initialization from the corresponding actual parameter.
- Each assignment statement has a vertex.
- Each control-structure condition (e.g.,*if*) has a vertex.
- Each procedure call has a vertex.
- Each actual parameter to a procedure has a vertex representing the assignment of the argument expression to some implicit (generated) variable.
- Each procedure with a return value has a vertex representing the assignment of the return value to some generated name.
- Each formal parameter and local variable has a vertex representing its declaration.

A procedure's parameters may sometimes be implicit. If a procedure assigns to or uses a global variable x (either directly or transitively via a procedure call), x is treated as an "hidden" input parameter, thus giving rise to additional actual-in and formal-in vertices. Similarly, if a procedure assigns to a global variable x (either directly or transitively), x is treated as a "hidden" output parameter, thus giving rise to additional actual-out and formal-out vertices. Henceforth, the terms 'input parameters' and 'output parameters' will include these hidden parameters as well.

Denote the program code corresponding to a vertex V as $\#V$. PDG vertices are connected through the following types of edges:

- There is a *flow dependence edge* between two vertices v_1 and v_2 if there exists a program variable x such that v_1 can assign a value to x, v_2 can use the value in x, and there is an execution path in the program from $\#v_1$ to $\#v_2$ along which there is no assignment to x.

- There is a *control dependence edge* between a condition vertex v_c and a second vertex v if a path in the control-flow graph that follows one $\#v_c$ branch can bypass $\#v$, while the path that follows the other $\#v_c$ branch must pass through $\#v$.
- There is a *declaration edge* from the declaration vertex for a program variable, x, to each vertex that can reference x.
- There is a *summary edge* corresponding to each indirect dependence from a procedure call's actual parameters and its output(s). These edges are used to avoid recomputing these summary relationships, thus making inter-procedural slicing more efficient. They are actually computed after PDG construction.

Given PDGs for each procedure, a *system dependence graph* (SDG) is then constructed by connecting the PDGs appropriately using the following additional types of edges:

- There is a *call edge* from a procedure call vertex to the corresponding procedure entry vertex.
- There is a *parameter-in edge* between each actual-in parameter and the corresponding formal-in parameter.
- There is a *parameter-out edge* between each formal-out parameter and the corresponding actual-out parameter. Furthermore, there is a parameter-out edge between each procedure's output value vertex and the vertex for an implicit (generated) variable on the caller side designated to receive it.

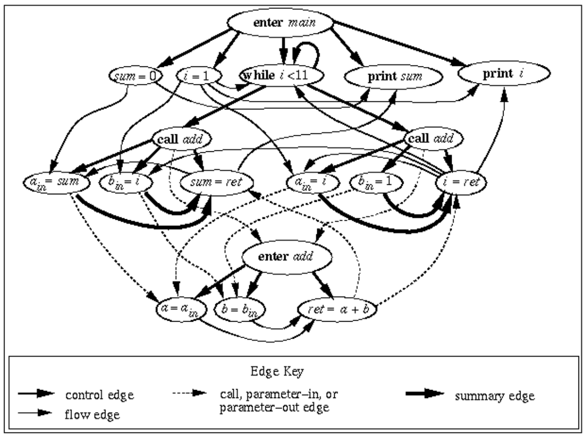


Fig. 1. Sample SDG

```
FUNCTION add(a,b)
   return(a+b);
FUNCTION main()
   sum <= 0;
   i <= 1;
   while (i < 11)
      sum <= add(sum,i);
      i <= add(i,1);
   print(sum);
   print(i);
```

Fig. 2. Sample Program

For example, Figure 1 illustrates an SDG for the small pseudocode program shown in Figure 2.

The complete algorithm for building an SDG from a program involves the following steps:

1. Build a Control Flow Graph (CFG) for each procedure in the program.
2. Build the call graph for the program.

3. Perform global variable analysis, turning global variables into hidden parameters of the procedures that reference or modify them.
4. Construct the PDGs by doing control-dependence and flow-dependence analysis.
5. Optionally compress the PDG so that each strongly connected region is represented by one node.
6. Bring together the PDGs and the call graph to form the SDG.
7. Compute summary edges for procedures that describe dependences between the inputs and the outputs of each procedure.

Then, slices and chops are computed by following the chains of dependences represented in the edges of the SDG.

cobegin {

S_{11}: assign z to x;

S_{12}: assign x to y;

}{

S_{21}: assign y to z;

}

Fig. 3. Example Concurrent Program

PROCESS BEGIN

S_{11}: x <= z;

S_{12}: y <= x;

END PROCESS

PROCESS BEGIN

S_{21}: z <= y;

END PROCESS

Fig. 4. Example VHDL Program

4 VHDL Slicing

Before discussing VHDL slicing, we first consider a pitfall that may result from a naive application of slicing to concurrent languages. Consider the concurrent program in Figure 3, and suppose that we augment the dependences of the previous section with a new type of dependence representing inter-process communication. In this example, there would be communication dependence edges from S_{21} to S_{11}, and from S_{12} to S_{21}. The assumption that dependences are transitive would lead to the conclusion that S_{11} is dependent on S_{12}. This is clearly false, thus showing that transitivity does not hold, and traditional slicing algorithms can not be applied directly [1].

Now consider an analogous VHDL program with 2 processes (Figure 4). Since VHDL processes are infinite, there should indeed be a dependence from S_{12} to S_{11} in this case. Since VHDL processes are infinite, we can con-

[1] In the interprocedural-slicing problem for programs written in sequential programming languages, dependence relationships are also non-transitive [13,14,23]. It is well understood how to cope with non-transitivity in this situation, whereas the problem of slicing concurrent programs has received attention only recently [18].

VHDL Construct	Traditional Construct
Procedure Function Process Concurrent Assignment	Procedure
Architecture variable	Local variable
Signal, Port	Global variable
Sequential Statement	Statement

Fig. 5. Mapping of VHDL Constructs

tinue to use the machinery of traditional slicing as long as additional dependences corresponding to inter-process communication are added. Thus, rather than creating an independent slicer built specifically for VHDL, our approach is to map VHDL constructs onto constructs for more traditional procedural languages (e.g. C, Ada), utilizing the semantics provided by the VHDL LRM [15]. Figure 5 lists the mapping between VHDL and traditional constructs that we use.

While many of these mappings may seem obvious, there are several major differences between VHDL and traditional programming languages which complicate the generation of the SDG. A VHDL program executes as a series of simulation cycles, as illustrated in Figure 6.

The VHDL computational paradigm differs fundamentally from traditional languages in three ways:

1. A VHDL program is a non-halting reactive system, rather than a collection of halting procedures.
2. A VHDL program is a concurrent composition of processes, where there is no explicit means for these processes to be called (in the manner of traditional procedures).
3. VHDL processes communicate through multiple-reader signals to which they are sensitive, instead of through parameters defined at a single procedure entry point.

VHDL procedures and functions are modeled in the traditional way. However, VHDL process models must capture the above differences, and we do this through three types of modifications:

- The constructed CFGs model the non-halting reactive nature of VHDL processes.
- The constructed PDGs capture an additional dependence corresponding to VHDL signal communication.
- An implicitly generated master "main" procedure controls process invocation, analogous to the event queue which controls VHDL simulator execution.

These mechanisms are described below. In all cases, the discussion only mentions processes, though it is to be understood that concurrent statements are treated as one-line processes.

4.1 Constructing the CFG

CFG construction for traditional languages is well understood, and the identical technique is used for VHDL procedures and functions. VHDL processes require some CFG modifications. We first consider processes with an

[h]

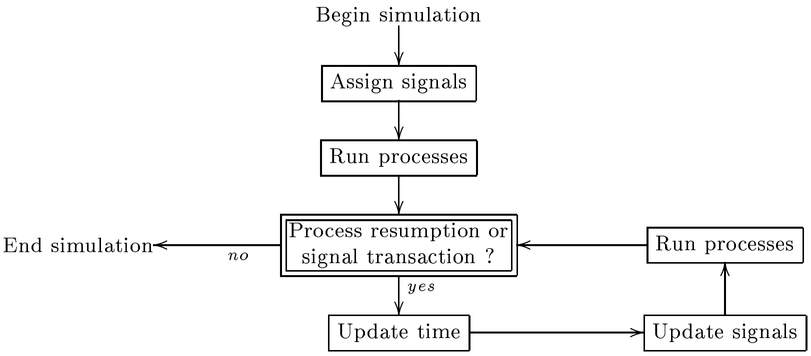


Fig. 6. Simplified VHDL simulation cycle

```
1 PROCESS BEGIN
2    WAIT ON x;
3    IF (y = '1')
4       THEN z <= x;
5    END IF;
6 END PROCESS;
```

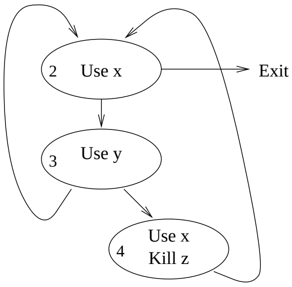


Fig. 7. Sample CFG (line numbers added for clarity)

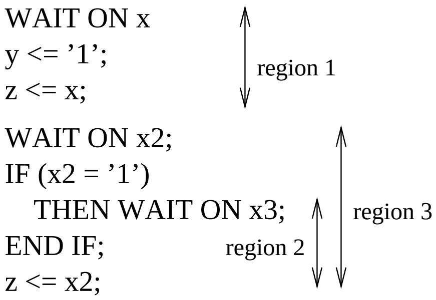


Fig. 8. Process regions in the presence of multiple wait statements

explicit sensitivity list or a single wait statement. The non-halting nature of processes is modeled simply by passing control from the end of the process back to its beginning. The wait statement provides the only complication. As suggested by Figure 6, from a wait statement, either control passes to the next statement or the simulation exits (in case the wait condition is never satisfied). This is simple to capture in the CFG by creating two corresponding child control-flow arcs from the wait statement. Figure 7 illustrates a CFG for a simple process.

The situation is substantially more complicated when there are multiple wait statements in the process. Although the above approach still works in this case, the resulting slice may be substantially larger than needed. Since each wait statement corresponds to a point where a region of the process may be invoked, a forward slice that affects a wait statement needs to include only the portion of the process between the wait statement and the next wait statement (and similarly for backward slices). To model this, we partition each process into regions corresponding to the portion of the process between successive wait statements (see Figure 8 for an example).

Note that we only require that each end node of a region precedes a wait statement; there may be multiple end nodes, and regions may overlap in the presence of wait statements within branching control structures (though very few VHDL programs have such control structures in practice). Then, a procedure is created for each process region, and a CFG for each of the resulting procedures is created as usual. To capture context information between process regions within the same process, all objects local to the process (e.g., variables) are treated as global variables after renaming to avoid conflicts with other processes (recall that the SDG build algorithms treat global variables as hidden parameters). Thus, for a process with W wait statements, W+1 procedures are created, one starting at each of the wait statements and one starting at the beginning of the process.

4.2 PDG Modifications

In traditional languages, inter-procedure communication occurs through global variables and parameters explicitly passed from the calling procedure to a called procedure. In contrast, VHDL process communication occurs through signals, and a process (or process region) is invoked when it is at a wait statement w, and there is an event for a signal that w is sensitive to. This communication is captured through the notion of signal dependence (in addition to the dependence types listed in Section 3): A process region p is said to be *signal dependent* on statement s if s potentially assigns a value to a signal that p is sensitive to. Note that every edge in the transitive closure of the signal dependence relation forms a true VHDL dependence due to the infinite nature of VHDL processes. Rather than modeling this signal dependence explicitly in the PDG, we generate implicit procedure calls in the CFG every time a signal is potentially assigned. For example, every assignment to signal s is followed by implicit calls to every procedure (e.g. VHDL process region, concurrent assignment) that is sensitive to s.

4.3 The Master Process

The above changes do not handle the reactive nature of VHDL, since processes may also be invoked by events on input ports. For simplicity, the following discussion deals with processes, though the same arguments are also applicable to process regions. Consider a VHDL program $\Pi =\|_{i=1}^{n} P_i$, where the P_i's are the processes comprising the program (as before, other concurrent statements are treated as one-line processes for the purposes of this discussion). Partition Π into two disjoint sets Π_1, Π_2, where Π_1 is the set of processes that are sensitive to at least one input port (hence, $\Pi_2 = \Pi \setminus \Pi_1$). It is possible for Π to be run with an input testbench that invokes a given process $P \in \Pi_1$ infinitely often. Therefore, the SDG must conservatively include dependences for infinitely often calls to all processes $P \in \Pi_1$. In contrast, any non-initial invocations of a process $Q \in \Pi_2$ must occur after an assignment to a signal that Q is sensitive to, and such invocations are handled using the signal dependences discussed above. Given these two observations, a CFG for the master process comprising the following (pseudocode) steps can be constructed:

```
- - initial invocations of each process
for Q ∈ Π
    call Q
- - subsequent invocations of Π1 processes
while (true)
    for P ∈ Π1
        call P
```

Given PDGs for the master process and each procedure in the VHDL program, the SDG is constructed as usual.

4.4 Correctness

Our motivation for the VHDL mapping discussed above is captured in the following theorem: First, define a VHDL *process invocation trace* to be a sequence $T = \langle T_1, T_2, \ldots, T_i, \ldots\rangle$, where $T_i \in 2^{\Pi}$, and T_i is the set of processes that are invoked on simulation cycle i of the trace. Initialization occurs during simulation cycle 0.

Theorem 1. *Let the VHDL program $\Pi = \|_{i=1}^{n} P_i$ have a process invocation trace $T = \langle T_1, T_2, \ldots, T_i, \ldots\rangle$. Then, for any $P_j \in T_i$, either $P_j \in \Pi_1$ or there is a signal dependence from some statement in $P_k \in T_\ell$ to P_j for some $\ell < i$.*

As in previous sections, the theorem is stated only for processes for simplicity, though it is equally applicable to process regions. The correctness of the theorem can be seen to follow from VHDL operational semantics.

Proof. We only sketch the proof here, as the actual proof would need a tedious restatement of the VHDL LRM. Consider any T_i and suppose $P_j \in T_i$. Then, P_j is sensitive to some signal s which had an event scheduled for simulation cycle i. From the LRM, it can be shown that the event on s occurred in one of the following ways:

1. Signal s corresponds to an input port that was excited during simulation cycle i.
2. Signal s is the target of some signal assignment statement, α, that was executed on some prior simulation cycle.

In the first case, $P_j \in \Pi_1$. In the second case, there is a signal dependence from α to P_j where α is a statement in some $P_k \in T_\ell$ for some $\ell < i$.

From Theorem 1, we can conclude that the VHDL slicing algorithm is correct (given the correctness of traditional slicing), since any inter-process dependences will have corresponding call edges in the SDG, by construction. In the first case ($P_j \in \Pi_1$), these edges arise from the master process. In the second case (signal dependence from α to P_j), the slicer models the dependence as a call of the procedure corresponding to process P_j on each execution of α, thus generating the needed dependence edges.

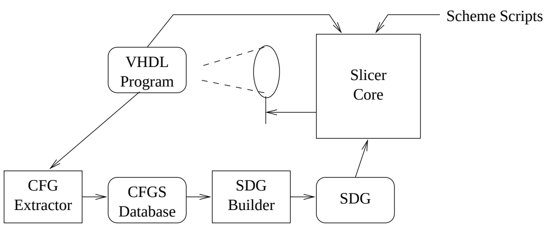


Fig. 9. VHDL Slicer Architecture

5 The Slicer

As mentioned earlier, the VHDL slicer is constructed using Codesurfer [10], a toolset developed and marketed by Grammatech, Inc. The toolset consists of reusable, multi-lingual components for building and operating on dependence-graph, call graph, and symbol-table representations of specifications and programs. The dependence graph is presented as an abstract data type, with interfaces in both C and Scheme. The toolset supports efficient implementations of powerful operations on program dependence graphs, such as precise interprocedural slicing and chopping.

Figure 9 illustrates the architecture of the VHDL slicer. The CFG Extractor and SDG Builder perform the algorithm described in Section 3 and output the SDG as well as a map from PDG nodes to source-text references. The slicing and chopping algorithms are embedded in the slicer core.

The user interface for a program-slicing tool is through the source code: By maintaining appropriate maps between the underlying SDG on which slicing is performed and the source code, the user specifies the slicing criterion by selecting program elements in the display of the source code, and the slice itself is displayed by highlighting the appropriate parts of the program. Slices may be forward or backward, and unions of slices may be computed using the GUI. The toolset GUI also supports browsing of projects and project files, as well as navigation through dependence graphs, slices, and chops. The user may also write scripts for operating on the dependence graphs of his program; scripts are written in *STk* – an implementation of Scheme enhanced with the Tk graphical user-interface widgets. Data types internal to the slicer are lifted into *STk*, and the user interface is built up around them.

5.1 Tool Walkthrough

To give a feel for the interface and some capabilities of our tool, we use a simple 48-line VHDL description of a signal chopper (modified from the programs provided in [5]), corresponding to a circuit with 4 flip flops, 4 gates, and 3 inverters (Figure 10). Figure 11 illustrates the project view and file view user interfaces provided by the slicer [2]. Whereas the project view provides hierarchical summary information that can be browsed interactively (partially shown in the figure), the file view provides the actual text comprising the program.

Suppose we are only interested in the part of the circuit that is dependent on the `clkn` signal. This corresponds to a forward slice on the definition of clkn (which

[2] The screenshots reproduced here are gray-scale versions of the tool's color output, and thus suffer from some loss of clarity

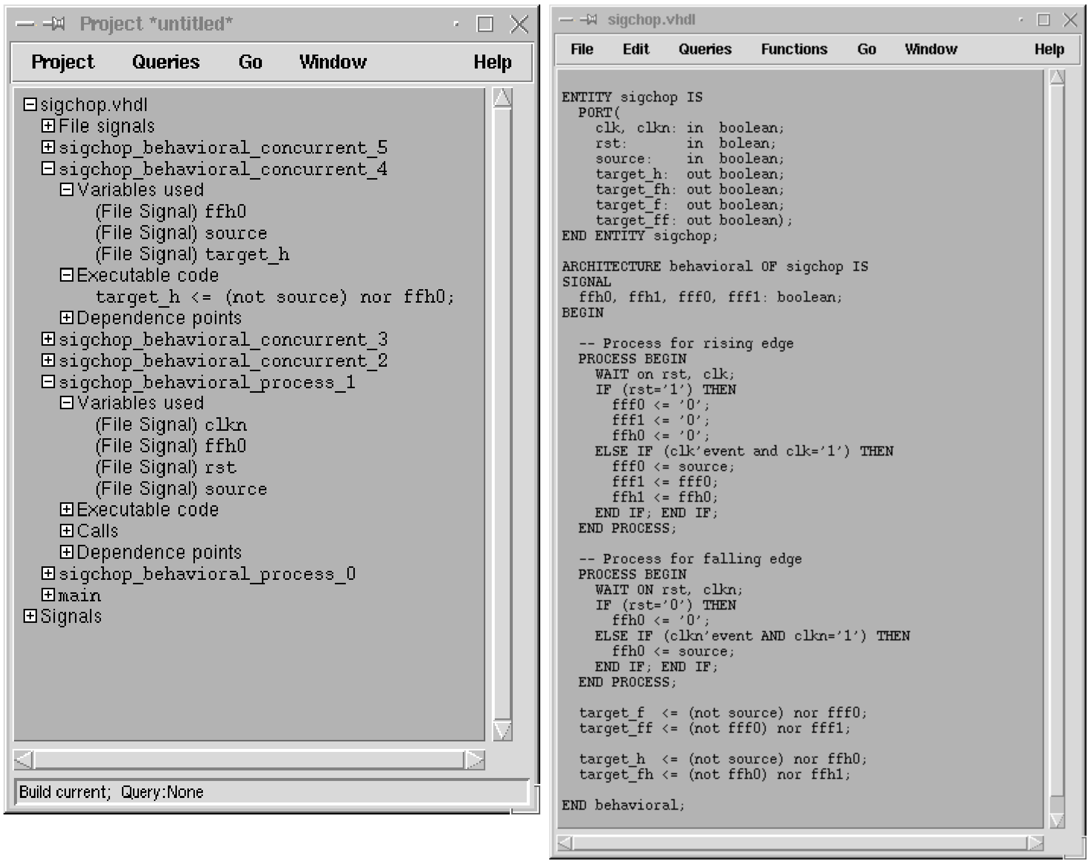


Fig. 11. Signal Chopper Project and File Views

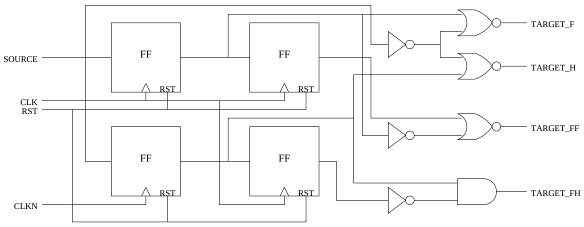


Fig. 10. Signal Chopper Circuit

was done by selecting the definition using the mouse). Figure 12 shows a file view of this slice. The resulting slice contains less than half of the original circuit. In large files, the colorbars to the right of the scrollbar allow the user to quickly scroll to the slice. Note that the forward slice excludes some subcomponents entirely (e.g., two nor gates) as well as portions of other subcomponents (e.g, the reset circuitry for a flip flop).

Now, suppose that we would like to analyze the behavior of the `target_f` output signal. A scan of the code reveals that the signal is assigned by only one [concurrent assignment] statement in the program. Selecting this statement, and performing a backward slice results in the program listed in Figure 13, which corresponds to the circuit of Figure 14. This slice is substantially smaller than the original circuit, thus allowing for much simpler analysis.

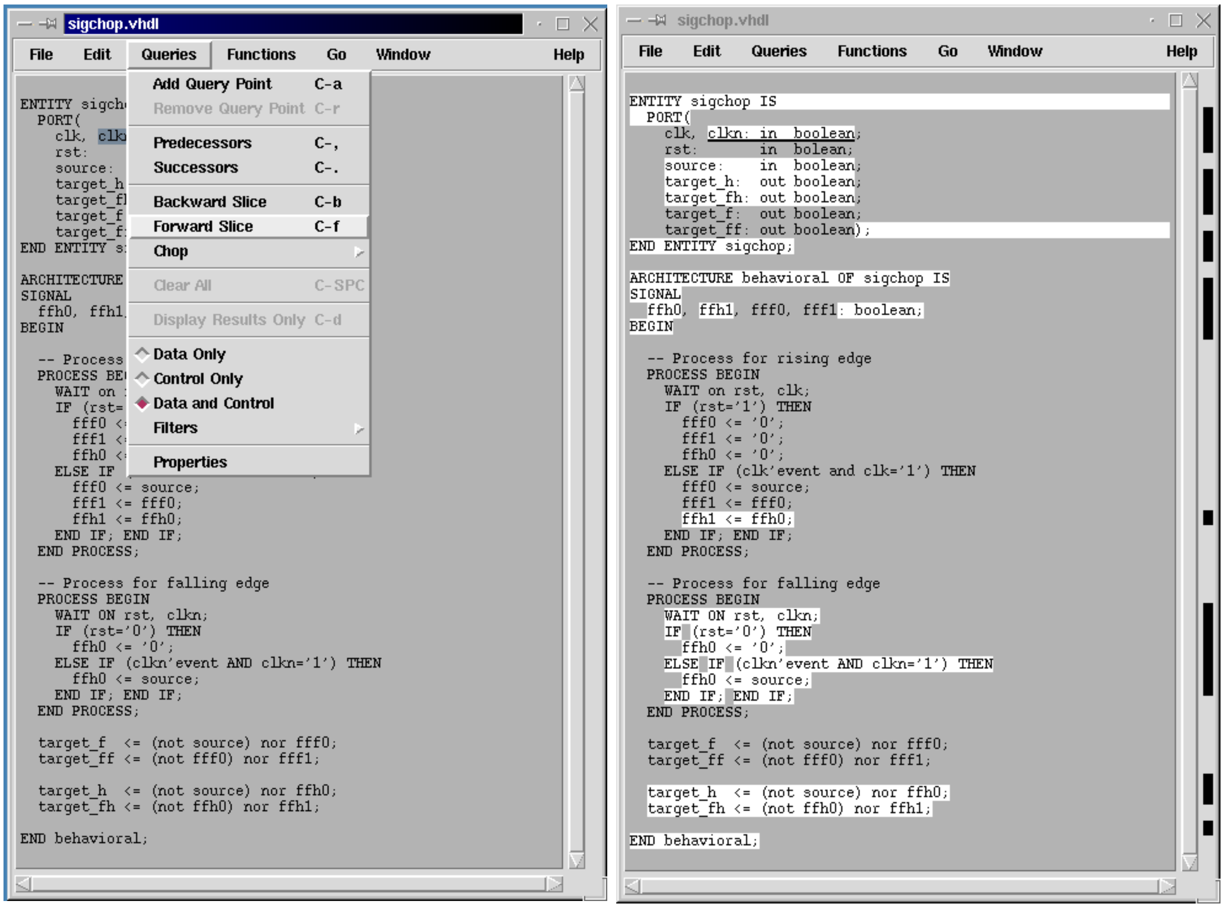


Fig. 12. Forward Slice on clkn

6 Applications

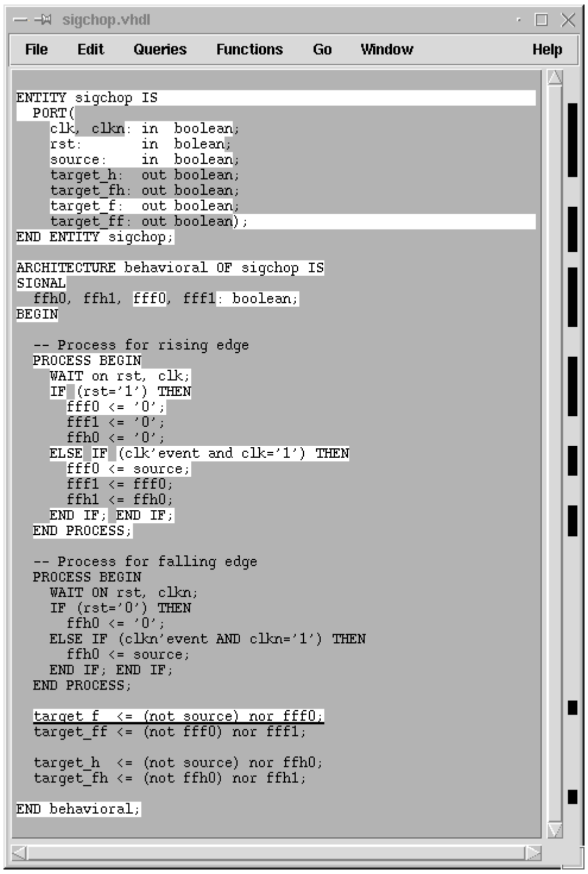
There are numerous applications of slicing in hardware simulation, design, testing, and formal verification, and we describe some of these applications in this section.

6.1 Design

Some questions that a slicer can help answer during design include:

– What part of the circuit is in the control path (and not the datapath)?
– What part of the circuit is responsible for one particular circuit function (in a multi-function circuit)?
– What part of the design is relevant to the actual function (and not the design-for-test and debug circuitry)?
– When reusing an existing IP design, how should it be modified to meet the new chip requirements?

Most circuits perform multiple functions, and it is desirable to identify the circuit portions that perform a function of interest, so that the designer's attention is focused appropriately. Slicing provides a mechanism to automatically perform this isolation. In many circuits, non-functional (e.g., design-for-test and debug) circuitry often dwarf the functional circuitry, and slicing allows the



```
sigchop.vhdl
File  Edit  Queries  Functions  Go  Window  Help

ENTITY sigchop IS
  PORT(
    clk, clkn: in  boolean;
    rst:       in  bolean;
    source:    in  boolean;
    target_h:  out boolean;
    target_fh: out boolean;
    target_f:  out boolean;
    target_ff: out boolean);
END ENTITY sigchop;

ARCHITECTURE behavioral OF sigchop IS
SIGNAL
  ffh0, ffh1, fff0, fff1: boolean;
BEGIN

  -- Process for rising edge
  PROCESS BEGIN
    WAIT on rst, clk;
    IF (rst='1') THEN
      fff0 <= '0';
      fff1 <= '0';
      ffh0 <= '0';
    ELSE IF (clk'event and clk='1') THEN
      fff0 <= source;
      fff1 <= fff0;
      ffh1 <= ffh0;
    END IF; END IF;
  END PROCESS;

  -- Process for falling edge
  PROCESS BEGIN
    WAIT ON rst, clkn;
    IF (rst='0') THEN
      ffh0 <= '0';
    ELSE IF (clkn'event AND clkn='1') THEN
      ffh0 <= source;
    END IF; END IF;
  END PROCESS;

  target_f  <= (not source) nor fff0;
  target_ff <= (not fff0) nor fff1;

  target_h  <= (not source) nor ffh0;
  target_fh <= (not ffh0) nor ffh1;

END behavioral;
```

Fig. 13. Backward slice on target_f <= (not source) nor fff0;

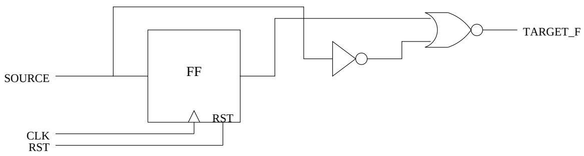


Fig. 14. Circuit Resulting after Backward Slice

high-level architectural designer to ignore these lower-level circuit details.

Another use of slicing occurs when reusing IP Macros. In order to effectively and efficiently design system LSIs (system-on-a-chip LSIs), it is desirable to reuse existing designs as much as possible, as it is unrealistically time-consuming to design the entire LSI chip from scratch. An increasingly popular methodology is to build chips from existing design components registered as IP macros. However, slight modifications to the IP macros may be needed to meet the new chips' requirements (e.g., interface with other chip components, add functionality or remove unnecessary functionality in the reused designs). The ability to make such modifications efficiently and correctly is a key issue in system LSI design. Modifiable IP macros are mostly given in terms of VHDL/Verilog synthesizable RTL descriptions (so called soft-macros), and users must understand RTL code internals in order to modify IP macros.

There are several levels of modification that are made to IP macros when they are reused, and slicing can assist these as follows:

– Changing polarities of inputs or outputs or adding encoders/decoders to inputs or outputs of the IP macros: Slicing (using the appropriate inputs/outputs as criteria) provides assistance to designers who need to determine where to make these modifications efficiently.
– Deleting functionality from an IP macro: Slicing can be directly applied to extract reusable code portions. For example, we obtain a 2-channel ATM from a 4-channel ATM by slicing with respect to two ATM output ports. Similarly, we slice an MPEG encoder/decoder with respect to the **P** frame outputs to obtain a simpler encoder/decoder similar to a motion JPEG encoder/decoder.
– Adding new functionality to an IP macro: A common way of adding new functionality is to describe

it in a new VHDL/Verilog module, and connect it with the existing macro through multiplexors. However, this is notoriously inefficient since no code is shared even if the modules are similar. Slicing can be used to determine macro portions related to the new functionality, thus minimizing the code added. If the difference between the original code and the new code is small, we can even expect similar delay/area performance from the circuits. This predictability is very important in hardware designs.

For example, consider high-speed communication chips. As network data rates increase, communication-chip macros designed earlier for slower data rates can be reused by adding/modifying functionality associated with data-/cell- processing portions of the communication chip. Slicing can be used to identify the portions of the VHDL/Verilog code associated with the data-/cell- processing functionality and reuse those portions.

Moreover, if appropriate process techniques are applied in logic synthesis, then we can use VHDL slicing before RTL code generation and still modify gate-level circuits for the original IP macros efficiently. We can thus modify hard-macros, i.e., macros that already have layout information (placement and routing).

6.2 Simulation

Some questions that a slicer can help answer during simulation include:

- What code portions can potentially cause an unexpected signal value found in simulation?
- What portions of the circuit can be affected by changing a particular code segment?
- What potentially harmful interactions with other modules can result from changing a particular code segment?

When debugging a simulation, designers routinely trace backward (in time and program state) from a point where a signal holds an unexpected value. If the slicing criteria is the set of statements that potentially assign the value to the signal, a backward slice is precisely the subset of the program that the designer must trace through, and the remainder of the program may safely be ignored.

If a code module is updated, a forward slice on the updated portion identifies the portions of the circuit that can potentially be affected by this update. Moreover, if that slice includes code from other hardware units, the designer may be alerted to unexpected potential interactions that may result in the introduction of new bugs due to the update.

6.3 Testing

Testing issues for which a slicer can provide assistance include:

- What execution paths are covered by a given test vector?

– What part of the circuit must be retested if a certain code segment is changed (i.e., regression testing)?
– What portion of a circuit is controllable from a given set of input ports?
– What portion of a circuit is observable from a given set of output ports?
– What portion of a circuit is testable (i.e., both controllable and observable) from a given set of input and output ports?

Test-plan generation is a complicated and time-consuming task. Part of this task involves the identification of execution paths that are covered by a given test vector. Slicing provides assistance in identifying these paths. For the same reason, slicing assists in determining which test vectors need to be rerun if a certain code segment is changed.

Another major research area in testing involves the use of metrics to quantify how testable a circuit is, if only some subset of inputs and outputs is available during test. A forward slice from the input subset provides an upper bound on circuit controllability, while a backward slice from the output subset provides an upper bound on circuit observability. The intersection of these two slices can be used to obtain an upper bound on circuit testability. However, the program chop from the input subset to the output subset, which in general is smaller than the intersection of the forward and backward slices [24], can be used to obtain an even tighter bound on circuit testability. In all four cases, dependence-graph operations provide metrics that can be used when constructing test plans.

6.4 Formal Verification

HDL slicing is particularly useful when applying model checking to prove circuit correctness. The major problem in model checking is state space explosion. Given a temporal logic specification, let the support of the specification be the set of variables/signals in it. Then a backward slice on the set of statements assigning values to elements of the support results in a subset of the program consisting of only the statements that can potentially affect the correctness of the specification.

Figure 15 illustrates the state space reductions that were achieved in the verification of some circuits, using the model checker described in [7]. Two circuits were verified: the controller logic for a RISC processor, and the control circuitry for an electronic camera. The figure provides the reductions in state-space and code size that were achieved by slicing with respect to the specification, as described above.

7 Related Work

The only other application of program slicing to HDLs that we are aware of is [16], which discusses a number of issues and applications related to VHDL slicing. However, the work presented there is motivational in nature, and we are not aware of any resulting automatable ap-

Circuit	Version	Processes	Concurrent Statements	Total States	Reachable States
RISC Processor	Original	7	18	$1.8X10^{47}$	$2.5X10^{37}$
	Sliced (SAFE)	4	3	$8.1X10^{31}$	$1.1X10^{22}$
RISC Processor	Original	7	18	$1.8X10^{47}$	$6.5X10^{39}$
	Sliced (INV)	4	1	$3.1X10^{29}$	$1.1X10^{22}$
Electronic Camera	Original	12	6	$2.2X10^{43}$	$7.1X10^{32}$
	Sliced	5	4	$9.0X10^{15}$	$3.5X10^{13}$

Fig. 15. Benefits of Slicing for Formal Verification

proaches. To the best of our knowledge, we are the first to have implemented a tool to automatically slice VHDL programs.

SDG-like structures form the basis of many gate-level test-generation algorithms. However, our approach works at the VHDL source level, thus avoiding the heavy complexity of synthesis. Moreover, many applications of slicing described in Section 6 make sense only at the VHDL source level.

A third area of related work occurs in the model-checking domain, where state-space size is reduced using the *cone of influence* reduction (COI) or *localization* reduction ([19]).

COI can be expressed as a fixpoint computation that constructs the set of state variables that can potentially affect the value of a variable in the CTL specification (i.e., the set of variables in the *cone of influence* of the variable of interest). Alternatively, COI can be thought of as building a dependence graph for the program, and then using graph reachability to determine what parts of the specification are relevant to the variable of interest. The actual dependence graph may be either on the VHDL source-code (*pre-encoding*) or on the set of equations that represent the transition function (*post-encoding*).

The localization reduction performs a related function. Intuitively, it works by conservatively abstracting system components and verifying a localized version of the specification. If the localized version is not verifiable, the abstractions are iteratively relaxed by adding more components, until the specification is eventually provable. Added components are in the specification's COI.

Several differences between these two reductions and slicing are worth noting (the first 3 apply only if the reductions are done as post-encoding operations):

– In HDL verification, the difficulty often lies in model-generation rather than model checking, and it is sometimes not even possible to build the model. Any post-encoding method obviously does not help in such cases.

– The model generation process often does some translation of the VHDL program into a restricted VHDL subset, and it is thus difficult or impossible to trace back to statements in the original program. Most of the design, simulation, and testing applications mentioned in this paper are consequently not possible using a post-encoding technique.
– One of the variables on which the size of the model depends is the size of the input program (e.g., the bits needed to represent the program counters). Post-encoding reductions cannot reduce this overhead in general.
– Slicing permits more complex reductions of programs to be specified than is possible using COI. For example, suppose the specification is of the form "Signal x is always false. In verification, we are primarily interested in ensuring that counterexamples in the original program are also in the slice. Thus, we can select the set of all statements that potentially assign non-false values to x as the slicing criterion, and perform a backward slice with respect to these statements to produce the desired reduced program. In the most general case, the structure of the specification can be analyzed to determine the appropriate combination of forward and backward slices that result in an equivalent program.

In some cases, slicing also has a disadvantage compared to the postencoding versions of the other reductions mentioned above, since it is applied at the level of granularity of the source rather than bits.

8 Conclusions

In this paper, we have shown how to extend traditional slicing techniques to VHDL, using an approach based on capturing the operational semantics of VHDL in traditional constructs. We have implemented a tool for automatic slicing, and the paper listed many applications of the tool along with some experimental results showing the state space reduction achievable in model checking. We are currently pursuing further research along three lines. First, we are investigating techniques to achieve more precise slices, by capturing VHDL semantics more accurately in the SDGs. The current SDGs are conservative in allowing for more dependences than actually exist, and more inter-cycle analysis of VHDL can remove some of these dependences. Second, we are working on developing slicing techniques for general concurrent languages, since the techniques described here extend readily to other concurrent languages. Finally, we are developing a theoretical basis for slicing with respect to CTL specifications for use in formal verification. The goal here is to achieve finer slices than possible using the approach outlined in this paper.

References

1. S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *ACM Symposium on Principles of Programming Languages*, pages 384–396, 1993.
2. D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the 1992 Conference on Software Maintenance* (Orlando, FL, November 9-12, 1992), pages 41–50, 1992.
3. D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2:31–45, 1993.
4. D. Binkley and K. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computers, Vol. 43*. Academic Press, San Diego, CA, 1996.
5. K.C. Change. *Digital Systems Design with VHDL and Synthesis*. IEEE Press, 1999.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
7. D. Déharbe, S. Shankar, and E.M. Clarke. Model checking VHDL with CV. In *Formal Methods in Computer Aided Design (FMCAD)*, page to appear, 1997.
8. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 3(9):319–349, 1987.
9. K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, SE-17(8):751–761, August 1991.
10. Grammatech, Inc. *Codesurfer Reference Manual*. Ithaca, NY 14850, 1998.
11. S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 234–245, 1990.
12. S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.
13. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
14. 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.
15. IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987. Std 1076-1987.
16. M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. In *Asian Pacific Conference on Hardware Description Languages (APCHDL)*, pages 132–139, 1996.
17. D. Jackson and E.J. Rollins. A new model of program dependences for reverse engineering. *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994), *ACM SIGSOFT Software Engineering Notes*, 19, December 1994.

18. J. Krinke. Static slicing of threaded programs. In *PASTE*, pages 35–42, 1998.

19. Robert P. Kurshan. *"Computer-Aided Verification of Coordinating Processes"*. Princeton University Press, 1994.

20. J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the First Conference on Empirical Studies of Programming*, June 1986.

21. J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

22. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, New York, NY, 1984. ACM Press.

23. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:701–726, November/December 1998.

24. 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.

25. T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proc. of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429, Schloss Dagstuhl, Wadern, Germany, February 1996. Springer-Verlag.

26. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1994.

27. M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, 1979.

28. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

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