

# Program Slicing of Hardware Description Languages<sup>\*</sup>

E. M. Clarke<sup>1,6</sup>, M. Fujita<sup>3</sup>, S. P. Rajan<sup>3</sup>, T. Reps<sup>4,7</sup>, S. Shankar<sup>1,5,6</sup>, and T. Teitelbaum<sup>2,4</sup>

<sup>1</sup> Carnegie Mellon University, Pittsburgh, PA, USA  
emc+@cs.cmu.edu

<sup>2</sup> Cornell University, Ithaca, NY, USA

<sup>3</sup> Fujitsu Labs of America, Sunnyvale, CA, USA  
{fujita,sree}@fla.fujitsu.com

<sup>4</sup> Grammatech, Inc., Ithaca, NY, USA  
{reps,tt}@grammatech.com

<sup>5</sup> Hunter College and the Graduate School, The City University of New York  
New York, NY, USA  
sshankar@roz.hunter.cuny.edu

<sup>6</sup> Verysys Design Automation, Inc., Fremont, CA, USA

<sup>7</sup> University of Wisconsin, Madison, WI, USA

**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 formal verification, design, simulation, and testing. Program slicing is a static program analysis technique that allows an analyst to automatically extract portions of programs relevant to the aspects being analyzed. We extend program slicing to HDLs, thus allowing for automatic program reduction to allow the user to focus on relevant code portions. We have implemented a VHDL slicing tool composed of a general inter-procedural slicer and a front-end that captures VHDL execution semantics. This paper provides an overview of program slicing, a discussion of how to slice VHDL programs, a description of the resulting tool, and a brief overview of some applications and experimental results.

## 1 Introduction

Hardware description languages (HDLs) are used today to describe circuits at all levels from conceptual system architecture to low-level circuit implementations

---

<sup>\*</sup> This research is supported in part by the Semiconductor Research Corporation (SRC) (Contract 97-DJ-294), the National Science Foundation (NSF) (Grants CCR-9505472, CCR-9625667, CCR-9619219), the Defense Advanced Research Projects Agency (DARPA) (Contract DABT63-96-C-0071), the United States-Israel Binational Science Foundation (Grant 96-00337), IBM, and the University of Wisconsin (Vilas Associate Award). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporting agencies.

suitable for synthesis. There are also several tools that apply model checking [1] to formally verify correctness of HDL designs (one such system for VHDL is described in [2]). 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.

HDL reduction is also useful for other design, simulation, and testing tasks since a major lack in current simulation methodologies is the need for structured design and analysis techniques. The use of automatic reduction techniques allows an analyst to focus on relevant code portions for further study. Moreover, with the increasing use of reusable libraries of existing code, reduction techniques are also useful to simplify the usage and/or modification of unstructured libraries.

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 [3] 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.

Program slicing results 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 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 operational 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 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 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 our techniques 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 some applications of slicing, and provides experimental

results that concretely illustrate the benefits of slicing in reducing state space size for model checking. We compare 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 [4,3,5,6,7,8]. (See [9,10] for surveys on slicing.) Slicing, and subsequent manipulation of slices, has applications in many software-engineering tasks, including program understanding, maintenance [11], debugging [12], testing [13,14], differencing [15,16], specialization [17], reuse [18], and merging [15].

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* [19,20]. 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 sets of source and target program points,  $S$  and  $T$ , the *chop* consists of all program points that could transmit the effect of executing a member of  $S$  to a member of  $T$ .

It is important to understand the distinction between two different but related “slicing problems”: the *closure slice* of a program  $P$  with respect to program point  $p$  and variable  $x$  identifies all statements and predicates of  $P$  that might affect the value of  $x$  at point  $p$ ; the *executable slice* of  $P$  with respect to  $p$  and  $x$  is a reduced program that computes the same sequence of values for  $x$  at  $p$  (i.e., at point  $p$  the behavior of the reduced program with respect to  $x$  is indistinguishable from that of  $P$ ). In *intraprocedural* slicing, an executable slice can be obtained from the closure slice; however, in *interprocedural* slicing, where a slice can cross the boundaries of procedure calls, an executable slice is harder to obtain since the closure slice might contain different subsets of a procedure’s parameters for different calls to the same procedure. However a closure slice can always be extended to an executable slice [21]. 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 most recently invoked call [7,8,20]. 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 [22,19].

Precise algorithms are preferable because they return smaller slices. Precise slicing and chopping can be performed in polynomial time [7,8,20]. Our VHDL slicing tool supports precise interprocedural slicing and chopping.

### 3 Inter-procedural Slicing

The value of a variable  $x$  *defined* at  $p$  is directly affected by the values of the variables used at  $p$  and by the predicates that control how many times  $p$  is executed. Similarly, the value of a variable  $y$  *used* at  $p$  is directly affected by assignments to  $y$  that reach  $p$  and by the predicates that control how many times  $p$  is executed. Consequently, a slice can be obtained by following chains of dependences in the directly-affects relation. This observation is due to Ottenstein and Ottenstein [5], who noted that *procedure dependence graphs* (PDGs), which were originally devised for use in parallelizing and vectorizing compilers, are a convenient data structure for slicing.

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.

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 the truth of the condition of  $\#v_c$  controls whether or not  $\#v$  is executed.
- 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, for efficiency reasons. 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 parameter and the corresponding formal parameter.
- There is a *parameter-out edge* between each procedure output value vertex and the vertex for an implicit (generated) variable on the caller side designated to receive it.

Figure 1 illustrates a SDG for a small pseudocode program.

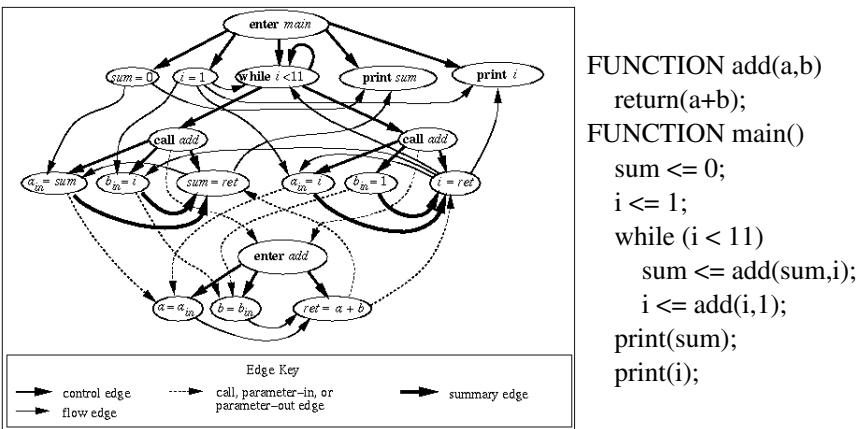


Fig. 1. Sample SDG

The complete algorithm for building a SDG from a program is:

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 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.

## 4 VHDL Slicing

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 operational semantics provided by the VHDL LRM [23]. Figure 2 lists the mapping between VHDL and traditional constructs that we use:

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

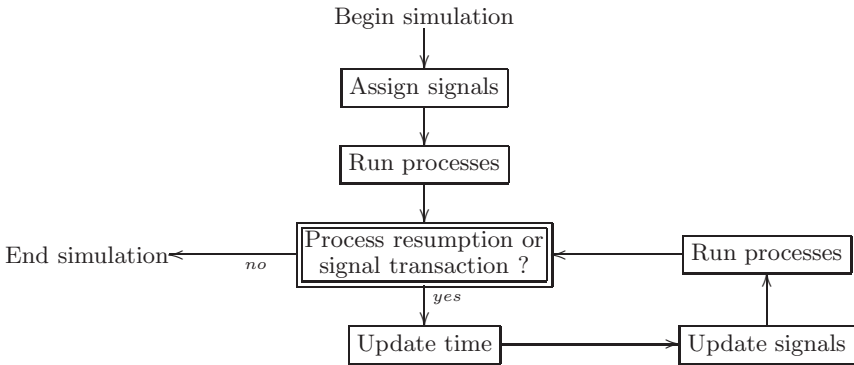
**Fig. 2.** Mapping of VHDL Constructs

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 3.

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, without any explicit means for these processes to be invoked (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:



**Fig. 3.** Simplified VHDL simulation cycle

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

These mechanisms are described below. Although the discussion only mentions processes, concurrent statements are to be treated analogously.

#### 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 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 3, 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 4 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 procedure still works, 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 portions of processes between successive wait statements. More precisely, for each wait statement  $w$  in a process  $p$ , let  $T$  be the set of statements

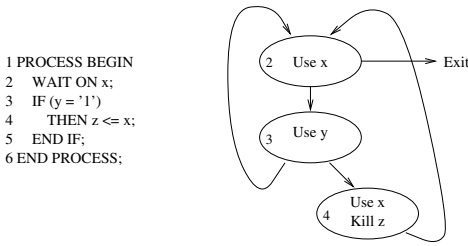


Fig. 4. Sample CFG

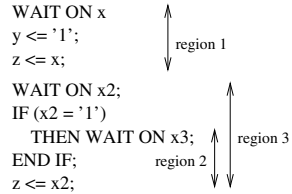


Fig. 5. Process regions in the presence of multiple wait statements

in some process execution trace starting at  $w$  and proceeding until another wait statement  $w'$  with no intervening wait statements ( $w' \notin T$ ). Let  $\mathcal{T}$  be the set of all such traces. Then, there is a *process region* corresponding to  $w$  which includes all the statements in  $\mathcal{T}$ . Figure 5 illustrates a simple 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 for each process region is created, 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 region) is invoked when it is at a wait statement  $w$ , and there is an event on 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  $s$  assigns a value to a signal that  $p$  is sensitive to. 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  $\Pi = \parallel_{i=1}^n P_i$ , 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  $\Pi$  into two disjoint sets  $\Pi_1, \Pi_2$ , where  $\Pi_1$  is the set of processes that are sensitive to at least one input port (hence,  $\Pi_2 = \Pi \setminus \Pi_1$ ). It is clearly not possible to determine *a priori* whether a process  $P \in \Pi_1$  is invoked in the simulation (after its initial invocation). 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:

```

for  $Q \in \Pi$            – initial invocations of each process
  call  $Q$ 
while (true)         – subsequent invocations of  $\Pi_1$  processes
  for  $P \in \Pi_1$ 
    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  $T = \langle T_1, T_2, \dots, T_i, \dots \rangle$ , where  $T_i \in 2^{\Pi}$ , and  $T_i$  is the set of processes that are invoked on simulation cycle  $i$  of the trace.

**Theorem 1** *Let the VHDL program  $\Pi$  have a process invocation trace  $T = \langle T_1, T_2, \dots, T_i, \dots \rangle$ . Then, for any  $P_j \in T_i$ , either 1)  $P_j \in \Pi_1$  or 2) there is a signal dependence from some statement in  $P_k \in T_\ell$  to  $P_j$  for some  $\ell < i$ .*

The theorem can be seen to follow from VHDL operational semantics.

Since the two cases of the theorem are captured using the master process and our notion of signal dependence, any inter-process dependences will have corresponding call edges in the SDG, by construction, and correctness of our VHDL slicing semantics thus follows from the theorem.

## 5 The Slicer

Figure 6 illustrates the architecture of the VHDL slicer. The tool implements all the issues discussed above except process regions. The CFG Extractor and

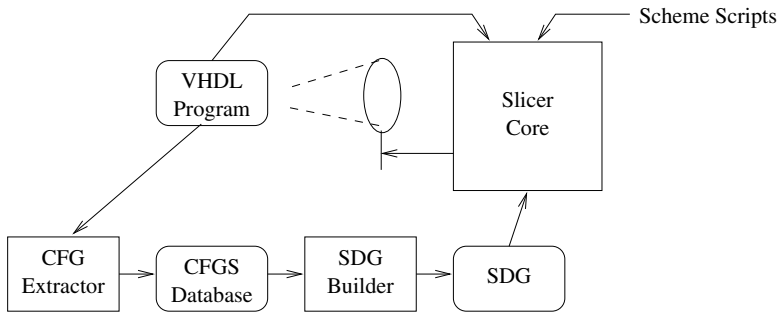


Fig. 6. VHDL Slicer Architecture

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 slicer user interface 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.

### 5.1 Tool Walkthrough

To give a feel for the interface and some capabilities of our tool, we use a simple VHDL program, consisting of 1 D flip-flop and 2 logic functions (Figures 7(a),(b))<sup>1</sup>. The project view provides hierarchical summary information that is interactively viewable (partially shown in the figure), while the file view provides the actual text comprising the program.

Figure 7(c) shows a file view of the executable statements in the forward slice on the program point `t1 <= t0 AND not(a);`. As expected, the slice includes the flip-flop but not any input circuitry. Figure 7(d) shows a project view of the backward slice on the same program point as above. This time, the slice excludes the flip-flop. In large files, the colorbars to the right of the scrollbar allow the user to quickly scroll to the slice.

<sup>1</sup> The screenshots reproduced here are dithered monochrome versions of the color tool output, and thus suffer from some loss of clarity here

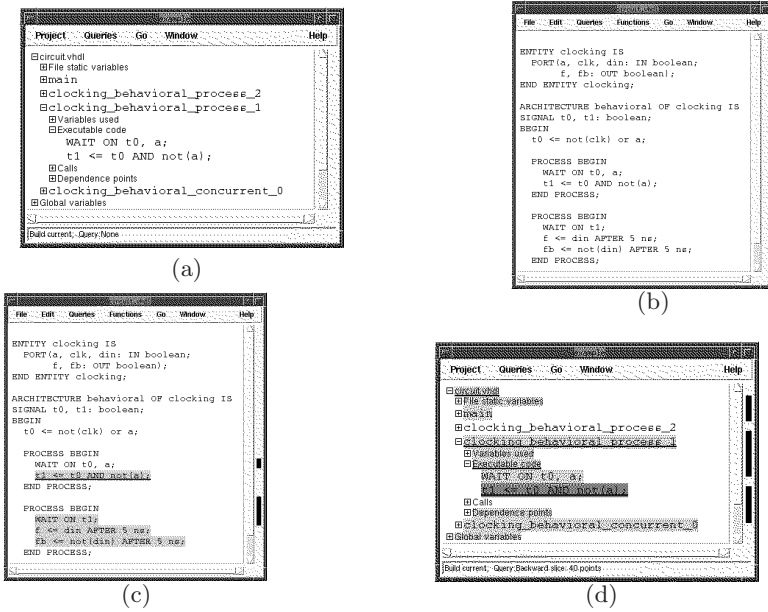


Fig. 7. Example (a) Project View, (b) File View, (c) Forward Slice, (d) Backward Slice

## 6 Applications

HDL slicing is useful in model checking to prove circuit correctness. The major problem in model checking is state space explosion. A backward slice on the set of statements assigning values to variables in the temporal specification results in a program subset consisting of only the statements that can potentially affect the correctness of the specification. Figure 8 illustrates the state space reduction that was achieved in the verification of the controller logic for a RISC processor, using the model checker described in [2] and sliced with respect to two different CTL specifications (all builds and slicing operations needed negligible time).

	Processes	Concurrent Statements	Total States	Reachable States
Original	7	18	$1.8 \times 10^{47}$	$2.5 \times 10^{37}$
Sliced (safety)	4	3	$2.0 \times 10^{31}$	$2.8 \times 10^{21}$
Original	7	18	$1.8 \times 10^{47}$	$6.5 \times 10^{39}$
Sliced (liveness)	4	1	$3.1 \times 10^{29}$	$1.1 \times 10^{22}$

Fig. 8. Benefits of Slicing for Formal Verification

The benefits of slicing vary widely depending on the nature of the circuit and the property being verified, and the above circuit was selected to be a typical one. However, an interesting aspect of slicing-based program reduction is a reverse-scalability effect. Since smaller VHDL programs tend to have fewer irrelevant components, we have observed the benefits of slicing to improve (percentage-wise) as programs grow in size.

There are numerous other applications of slicing in hardware design, simulation, and testing. The reader is referred to [24] for a more detailed description of the applications of slicing, but we briefly list some of them in this section. Sample questions that a slicer can assist an engineer in answering include:

- How do you specialize (or modify) an existing IP design for reuse?
- What part of the design is relevant to the actual function (and not the design-for-test and debug circuitry)?
- What part of the circuit is in the control path (and not the datapath)?
- 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?
- 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?

## 7 Related Work

The only other application of program slicing to HDLs that we are aware of is by [25], which discusses a number of issues and applications related to VHDL slicing (a resulting system that implements some of these is discussed in [26]). Our approach differs since it captures VHDL operational semantics within existing procedural frameworks so that the benefits of existing precise slicing technology can better be exploited. We believe that using such an approach will enable us to cover a larger subset of VHDL, and get finer slices. Also, to the best of our knowledge, we are the first to have used HDL slicing for formal verification.

A tool for slicing Promela, the input language for the Spin model checker, is currently being constructed ([27]). However, the concurrency issues dealt with there are different from those in VHDL.

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.

Another 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 ([28]).

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*), though the former is difficult.

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 formal 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 that the model size is a function of 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.
- The precise interprocedural slicing technique used is based on “matched-parenthesis reachability” [7,8,29], which is more involved than the ordinary graph reachability used by pre-encoding COI. As mentioned in Section 2, not all SDG paths are possible execution paths, since paths in which calls and returns are mismatched can be excluded. Although the problem of determining the feasibility of a given SDG path is in general undecidable, such

mismatched paths can be excluded using a balanced parenthesis language. Ordinary reachability is an example of CFL-reachability in which the CFL is the regular language  $e^*$  (where all edges are labeled with  $e$ ), while this balanced parenthesis condition can not be expressed either with a regular language or with a linear-CFL. Thus, a pre-encoding COI requires a complete elaboration and unfolding of function calls to achieve the same effect, which results in greater design size.

## 8 Conclusions

In this paper, we have shown how to extend traditional slicing techniques to VHDL, using an approach based on capturing VHDL operational semantics with 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 four lines. First, we are enhancing the class of supportable HDL (both VHDL and Verilog) constructs. Second, 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. Third, 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 to generate slicing criteria from CTL specifications for use in formal verification.

## References

1. 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. 299
2. 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. 299, 308
3. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984. 299, 300
4. M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979. 300
5. 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. 300, 301
6. 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. 300

7. 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. **300, 301, 310**
8. 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. **300, 301, 310**
9. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1994. **300**
10. D. Binkley and K. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computers, Vol. 43*. Academic Press, San Diego, CA, 1996. **300**
11. 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. **300**
12. J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the First Conference on Empirical Studies of Programming*, June 1986. **300**
13. 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. **300**
14. S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *ACM Symposium on Principles of Programming Languages*, pages 384–396, 1993. **300**
15. 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. **300**
16. 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. **300**
17. 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. **300**
18. J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994. **300**
19. 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. **300**
20. 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. **300, 301**
21. D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2:31–45, 1993. **300**
22. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984. **300**
23. IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987. Std 1076-1987. **303**
24. E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. Technical Report CMU-CS-99-103, Carnegie Mellon University, 1999. **309**

25. 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. 309
26. S. Ichinose, M. Iwaihara, and H. Yasuura. Program slicing on VHDL descriptions and its evaluation. Technical report, Kyushu University, 1998. 309
27. L. Millett and T. Teitelbaum. Slicing promela and its applications to protocol understanding and analysis. In *4th International SPIN Workshop*, pages 75–83, 1998. 309
28. Robert P. Kurshan. *“Computer-Aided Verification of Coordinating Processes”*. Princeton University Press, 1994. 310
29. T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97: Int. Logic Programming Symposium*, pages 5–19, Cambridge, MA, 1997. M.I.T. 310