Program Slicing for Design Automation: An Automatic Technique for Speeding-up Hardware Design, Simulation, Testing, and Verification[⋆] [⋆⋆] [⋆⋆⋆]

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

[1] Carnegie Mellon University,
Pittsburgh, PA, USA
FAX: (412) 268 5576
e-mail: {sshankar, emc}+@cs.cmu.edu

[2] Fujitsu Labs of America
Sunnyvale, CA, USA
e-mail: {fujita,sree}@fla.fujitsu.com

[3] University of Wisconsin
Madison, WI, USA
e-mail: reps@cs.wisc.edu

[4] Cornell University
Ithaca, NY, USA
e-mail: tt@cs.cornell.edu

[5] Grammatech, Inc.
Ithaca, NY, USA
e-mail: {reps, tt}@grammatech.com

⋆ Topic Numbers: T5.1, M1.3, M1.2, T1.2, T2.2

⋆⋆ All appropriate organizational approvals for the publication of this paper have been obtained. If accepted, the author(s) will prepare the final manuscript in time for inclusion in the Conference Proceedings and will present the paper at the Conference.

⋆⋆⋆ This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, 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.

† Presenter, Contact Person: (412) 268 2828

Program Slicing for Design Automation: An Automatic Technique for Speeding-up Hardware Design, Simulation, Testing, and Verification

Abstract. Program slicing is a static program analysis technique that allows an analyst to automatically extract portions of programs relevant to the aspects being analyzed. This paper discusses a tool that applies slicing to hardware description languages. The resulting tool provides assistance in various hardware design, simulation, testing, and formal verification tasks, as described here.

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. However, a major lack in current source-code-simulation methodologies is the need for structured design and analysis techniques that can be applied to the simulation process. Moreover, these issues are becoming more important with the increased use of reusable libraries of existing code, since it is difficult to use and/or modify 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 to formally verify correctness of HDL designs. It is well recognized that the fundamental problem in model checking is state explosion, and there is thus also a need to reduce the size of the HDL descriptions so that the corresponding models have fewer states. For many designs, it is not even possible to build the state transition relation, and HDL program reduction techniques are thus needed.

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 [1] 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.

In this paper, we apply program slicing to HDLs to support simulation and formal verification of VHDL programs. The only related work that we are aware of is [2], which discussed a number of issues and applications related to VHDL slicing. To the best of our knowledge, we are the first to have implemented a tool for slicing VHDL programs. We also identify several new classes of slicing applications. Finally, we have applied the tool to some significant programs; this experience, which is reported on in this paper, supplies concrete evidence of the benefits that can be attained from HDL slicing.

The remainder of the paper is organized as follows: Section 2 presents requisite background material. Section 3 outlines three applications of program slicing for HDLs. Section 4 discusses how to extend slicing to VHDL. Section 5 presents an overview of the implemented VHDL slicing tool. Section 6 presents a walk-through of a simple VHDL example and then presents concrete evidence of the benefits that can be attained through slicing VHDL. Section 7 discusses the relationship of slicing to other reduction techniques that are used in model checking. We conclude with a brief discussion of the results and our future plans in Section 8.

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 [1, 3, 4]. (See [5, 6] for surveys on slicing.) Slicing, and subsequent manipulation of slices, has applications in many software-engineering applications, including program understanding , maintenance, debugging, testing, differencing, specialization, reuse, and merging.

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.

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; 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 [3], who noted that *program 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 program is a directed graph whose nodes represent the program's individual statements and predicates. PDG edges represent the control and flow dependences among the procedure's statements and predicates [7].

A related operation is program *chopping* [8, 9]. A chop is a kind of "filtered" slice that answers questions of the form "Which program elements serve to transmit effects from a given source element s to a given target element t?"

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

Version (1) 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) 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 by 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) [4]; however, a solution to Version (1) can always be extended to provide a solution to Version (2) [10].

Some slicing and chopping algorithms are *precise* in the sense that they track dependencies transmitted through the program only along paths on which a procedure call returns to the site of the most recently invoked call [4, 11, 9]. 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 [1, 8]. Precise algorithms are preferable because they return smaller slices. Precise slicing and chopping can be performed in polynomial time [4, 11, 9]. Our VHDL slicing tool supports precise interprocedural slicing and chopping.

The user interface for a program-slicing tool can be through the source code: By maintaining appropriate maps between the source code and the underlying structures on which slicing is performed, the user can specify the slicing criterion by selecting program elements in the display of the source code, and the slice itself can be displayed by highlighting the appropriate parts of the program. Thus, program slicing is superior to methods that reason on extremely low-level program representations (e.g., netlists), because a backwards transformation from the lower level to the source code is not needed.

3 Applications of Program Slicing for HDLs

This section briefly surveys three applications of program slicing for HDLs.

First, the ability to slice programs written in an HDL enhances the capabilities of the simulation environment. 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. For the same reasons, slicing assists in developing test plans. Simulation maintenance is enhanced in many ways. If a description is updated by modifying a set of statements, a forward slice from that set results in the portion of the description that is potentially changed. If this slice includes code from other hardware units, this may alert the designer to potential interactions that may result in the introduction of new bugs due to the update. When modifying hardware, slicing can be used to do intelligent version differencing. Forward slices starting from the textual differences between the two versions result in the code corresponding to portions of hardware that might be affected by the version change. When the modification involves the addition of a new module, this forward slicing identifies the modules and statements that must be involved in regression testing.

Second, slicing of HDLs also allows for improved designs. Modern designs typically include circuitry that performs the main function, as well as other circuitry for process-oriented functions (e.g., debugging, design-for-test). The extra circuitry code often dwarfs the functional code, and in this case, an architectural designer can slice out the test circuitry to focus attention on the parts of interest. Similarly, slicing can be used to isolate data and control aspects of a circuit so that a designer can concentrate on one of these. Finally, slicing can provide assistance for intelligent test-pattern generation. An interesting example of using slicing for hardware design using IP macros is given in Section 6.2.

Third, slicing can help with formal verification, which is becoming an increasingly important component of the hardware design cycle. A common approach is to extract a model corresponding to the HDL program, and apply temporal logic model checking ([12]) on a CTL (Computation Tree Logic) specification (one such system for VHDL is described in [13]). It is well recognized that the most serious problem in model checking is state explosion. If the slicing criteria is the set of variables in the specification, a backward slice can be used to create a reduced VHDL program that is guaranteed to be equivalent to the original program with respect to the specification. Thus, slicing can dramatically reduce the size of the VHDL program and alleviate the state-explosion problem, and an example is presented in Section 6.4.

4 Slicing VHDL Programs

A VHDL program executes as a series of simulation cycles, as illustrated in Figure 1.

As mentioned in Section 2, slicing algorithms for traditional procedural programming languages produce slices by analyzing a dependence graph whose nodes corresponds to source statements or control predicates of the program and whose edges represents dependencies among the statements and predicates.

However, computing dependence graphs for VHDL is significantly more complex than for sequential languages because the basic VHDL execution model is fundamentally different 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 set of concurrent processes, where there is no explicit means for these procedures to be called (in the manner of traditional procedures).

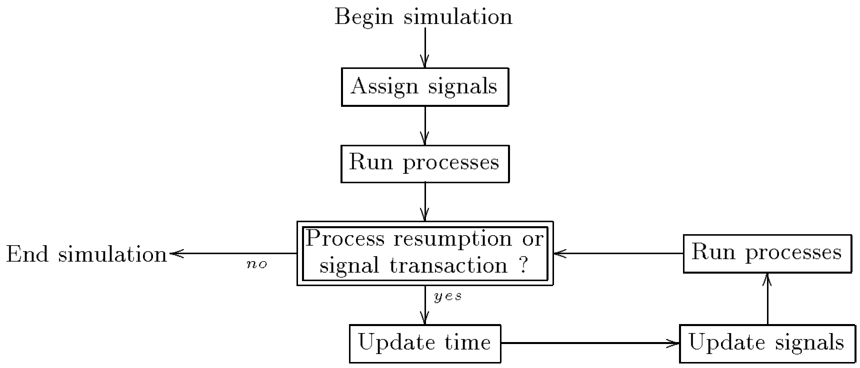

Fig. 1. Simplified VHDL simulation cycle

3. VHDL processes communicate through multiple-reader signals to which they are sensitive, instead of through parameters defined at a single procedure entry point.

We represent VHDL processes as procedures, and capture the first two differences in the PDG, with slight modifications. Note that this requires each process to have $W+1$ entry points where W is the number of wait statements in the process. To capture the third difference, we introduce the notion of signal dependence: A process p is said to be *signal dependent* on statement s if s assigns a value to a signal that p is sensitive to. Each signal dependence is associated with a process entry point corresponding to the process statement which is sensitive to the signal (e.g., a wait statement). Note that there may be more than one signal dependence between a program and a statement. Dependence graphs with flow-, control-, and signal-dependence edges account for flow of control between processes, as well as information communicated between processes.

5 The Slicer

The VHDL slicing tool is constructed using CodeSurfer, a toolset 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 C and Scheme interfaces. Figure 2 illustrates the architecture of the VHDL slicer. The CFG Extractor analyzes a VHDL program, and produces corresponding control-flow graphs (CFGs), one for each VHDL process, procedure, and function. The SDG Builder takes the CFGs and produces the corresponding system-dependence-graph (SDG) representation of the program. The Slicer Core provides the fundamental dependence-graph operations on the SDG, a GUI for invoking them and displaying results in VHDL source text, and a script interpreter for processing end-user analyses and extensions written in Scheme.

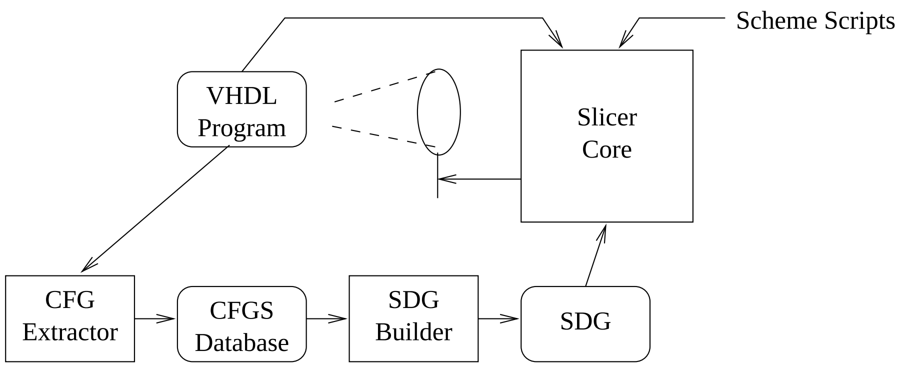

Fig. 2. VHDL Slicer Architecture

An SDG consists of one or more PDGs, where PDG vertices represent the individual statements and predicates of the procedure. A call statement is represented by a call vertex and a collection of actual-in and actual-out vertices. Similarly, procedure entry is represented by an entry vertex and a collection of formal-in and formal-out vertices. References (or assignments) to global variables or signals give rise to additional vertices for these "hidden" input (or output) parameters. Each PDG also contains vertices for its local-variable and parameter declarations.

The edges of a PDG represent the control and data dependences among the procedure's statements and predicates. A vertex, V, is said to be *control dependent* on a predicate, p, if p controls whether or not V will be executed. An entry vertex is treated as a pseudo-predicate. A vertex V is *flow dependent* on an assignment if the value assigned can be referenced in V. There is a dependence edge from each declaration vertex to each of its definitions and uses.

The PDGs are connected together to form the SDG by *call* edges (which represent procedure calls, and run from a call vertex to an entry vertex) and by *parameter-in* (and *parameter-out*) edges (which represent parameter passing and return values, by connecting an actual-in (out) vertex to the corresponding formal-in (out) vertex). There are call edges for explicit procedure/function calls as well as implicit calls induced by VHDL signal dependences between processes.

The process of building a system dependence graph from the CFGs of the VHDL program involves the following steps:

- Build the call graph.
- Perform global variable analysis, turning global variables into hidden parameters of the procedures that reference or modify them.
- Construct the PDGs by doing control-dependence and flow-dependence analysis.
- Optionally compress the PDG so that each strongly connected region is represented by one node.

- Bring together the PDGs and the call graph to form the SDG.
- Compute summary edges for procedures that describe dependences between the inputs and the outputs of each procedure.
- Output the SDG and a map from PDG nodes to source-text references.

The GUI accesses the SDG representation not only to compute slices, but also to allow the user to browse the program's source code. The user is presented with a display of the VHDL program, and may select one or more program components as slicing criteria by pointing to and selecting program elements on the screen. The GUI then highlights the elements of the slice. Slices may be forward or backward, and unions of slices may be computed using the graphical user interface.

The GUI also supports browsing of projects and project files, as well as navigation through the dependence graphs, slices, and chops. The VDHL text is, in effect, hyperlinked by its underlying SDG representation, which users "surf" interactively by following links presented in popup menus. Scheme scripts allow users to write their own "crawlers" that navigate the VHDL dependence web to perform custom analyses.

6 Example Applications of Slicing

Section 3 listed many benefits of slicing in design, simulation, testing, and formal verification. This section provides a walk-through of a simple VHDL example, along with some examples that concretely illustrate the benefits of slicing.

6.1 Walk-Through

In this section we attempt to give a feel for the interface and capabilities of our tool. The tool interface consists of a project viewer and a file viewer. Consider a simple VHDL program, consisting of 1 D flip-flop and 2 logic functions (Figures 3, 4) [1]. 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 5 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. In large files, the colorbars to the right of the scrollbar allow the user to quickly scroll to the slice.

Figure 6 shows a project view of the backward slice on the same program point as above. This time, the slice excludes the flip-flop.

[1] The screenshots reproduced here are dithered monochrome versions of the color tool output, and thus suffer from some loss of clarity here

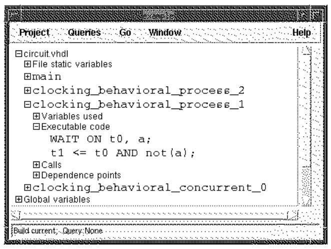

Fig. 3. Example Project Viewer view

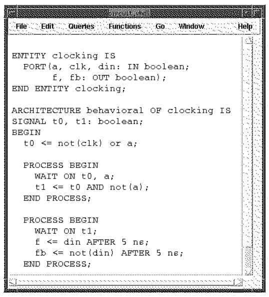

Fig. 4. Example File Viewer view

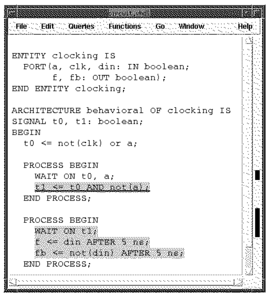

Fig. 5. Forward slice on `t1 <= t0 AND not(a);`

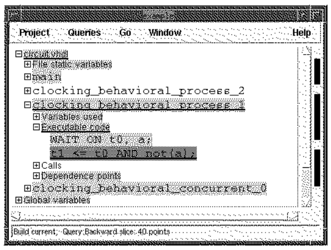

Fig. 6. Backward slice on `t1 <= t0 AND not(a);`

6.2 LSI IP Macro Reuse

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 are needed to meet the new chips' requirements (e.g., interface with other chip components, additional or unneccessary functionality in the reused designs). The efficiency of these modifications 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). In order to modify IP macros, users have to understand RTL code internals.

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 assists designers in determining where to make modifications.
- 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: We could describe the new functionality in a VHDL/Verilog module and connect it with the existing macro through multiplexors. However, this is 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 added code. If the difference between the original code and the new code is small, we can even expect similar delay/areas 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.

Furthermore, by applying Engineering Change (EC) techniques in logic synthesis as well as program slicing in RTL code generation, we can 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).

6.3 Testability

One goal of hardware design is testability, and slicing can be used to provide a quantitative metric of testability. Suppose the circuit is intended to be tested through a set of input and output signals $(\mathcal{I}, \mathcal{O})$. Then, define the circuit's *testable region* to be the program portions which are in the program chop with respect to source $\mathcal{I}$ and target $\mathcal{O}$. This provides a metric on the testability of the circuit. Since most current testability measures are intra-process, the above chop provides a superior testability metric. Note, however, that the metric is a measure of *potential* testability (i.e., an upper bound on testability).

We applied this technique to a controller consisting of 900 lines of VHDL. The inputs include reset, clock, condition code, coprocesser, and various load/store control signals. The outputs include various signals controlling the operation of the program counter, address and data registers, instruction decode/execute circuitry, and a bus interface. Using the slicer, we determined that the circuits testability was unaffected by the coprocessor inputs.

6.4 Formal Verification

The major problem in formal verification using model checking is state space explosion. Given the VHDL code for the control logic of a commercial RISC processor, we attempted to verify a simple safety property, both with and without slicing. The sliced program was constructed by taking a backward slice from the set of statements assigning values to variables/signals in the support of the property, and then manually editing the program to handle minor syntactic matters. Figure 7 illustrates the results, using the VHDL model checker described in [13]. The resulting model is clearly substantially smaller and simpler to verify.

	SDG Nodes	Processes	Concurrent Statements	Total States	Reachable States
Original	867	7	18	$1.8X10^{47}$	$2.5X10^{37}$
Sliced	408	4	3	$2.0X10^{31}$	$2.8X10^{21}$

Fig. 7. Benefits of Slicing for Formal Verification

We expect similar results from other industrial-size programs; interestingly, the smaller programs that we sliced tended to have smaller benefits. This scalability effect can be attributed to the lack of irrelevant code in typical smaller programs since they are usually artificially constructed.

7 Relationship to Model-Checking Optimizations

In this section, we compare slicing with two reduction techniques that are used in model checking: the *cone of influence* reduction (COI) and *localization* reduction ([14]).

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 set of equations that express the transition relation, and then using graph reachability to determine what parts of the specification are relevant to the variable of interest.

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 approaches and slicing are worth noting (the first 3 apply only if these are done as post-encoding operations):

- Slicing is done on the source code, and can thus be considered a pre-encoding technique as opposed to the other techniques, which are normally performed after encoding. However, 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.
- 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.
- The formula for model size includes terms that are functions of 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". With slicing, we can select the set of all statements that potentially assign non-zero 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 in CodeSurfer is based on "matched-parenthesis reachability" [4, 11, 15], which is more involved than the ordinary graph reachability used by COI. Although the problem of determining the feasibility of a given SDG path is in general undecidable, paths in which calls and returns are mismatched (as discussed in Section 2) 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, our program slicing technique is strictly more powerful than COI.

8 Conclusions and Future Work

This paper presented a tool for slicing VHDL programs, and showed the benefits of slicing to a number of application areas including design, simulation, testing, and formal verification. We are currently pursuing further research along four lines. First, we are enhancing the supported VHDL subset. There are no theoretical limitations, and we expect to be able to slice almost all of elaborated VHDL soon. 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 dependencies. 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. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
2. 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.
3. 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.
4. 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.
5. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, 1994.
6. D. Binkley and K. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computers, Vol. 43*. Academic Press, San Diego, CA, 1996.

7. 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.
8. 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.
9. 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.
10. D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2:31–45, 1993.
11. 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.
12. 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.
13. 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.
14. Robert P. Kurshan. *"Computer-Aided Verification of Coordinating Processes"*. Princeton University Press, 1994.
15. 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.