Slicing SAL
CSL Technical Report

Vijay Ganesh, Hassen Saidi, Natarajan Shankar
SRI International, Menlo Park, CA
{vganesh,saidi,shankar}@csl.sri.com

October 7, 1999

Abstract

Model checking has been successfully applied to verify finite-state systems albeit ones with small state-space. But most interesting systems have very large or infinite state-spaces. Automatic Abstraction techniques can help alleviate the state-space explosion problem to some extent. Another complementary approach is the use of program slicing to automatically remove portions of the input transition system irrelevant to the property being verified. This may result in state-space reduction. The reduced state system, if finite, may then be more amenable to model checking.

In this paper we discuss application of slicing to the SAL intermediate language. SAL intermediate language (or just SAL) is a concurrent language designed so that popular programming languages can be converted to SAL and whole set of Abstraction, Program Analysis, Theorem Proving and Model Checking tools/techniques can be combined and methodologies defined to verify large state systems. We describe a novel algorithm for slicing SAL and report on its implementation. It is one of the few slicing algorithms which deal with concurrency. We also discuss methodologies for combining slicing and other techniques to enable verification of larger state systems, use of theorem proving techniques to refine slicing, and techniques to convert temporal formulae into slicing criteria.

1 Introduction

Model checking [CES83, QS82] has proved very useful in verifying relatively small finite state systems in a highly automated fashion. To enable highly automated verification of larger systems researchers have turned to automatic abstraction techniques [GS97, PH97, CU98, BLO98], methodologies to combine theorem proving and model checking [BBC$^+$96, ORR$^+$96, Saï97] and very recently to program analysis techniques. Automatic abstraction techniques help in making larger systems more tractable for model checking. Program analysis can complement automatic abstraction in many ways. e.g. they can help discover predicates as required by Predicate Abstraction [SS99]. More importantly certain Program Analyses like Slicing can help remove code irrelevant to the property under consideration from the input transition systems. Slicing can help reduce the state-space of the system resulting in a safe approximation,and remove irrelevant components of the input software thus easing the subsequent verification process.

Slicing was first reported by Mark Weiser [Wei84] as a technique to understand very large and complex programs. Slicing has been used extensively by the software engineering community to build debuggers and program understanding tools. Many slightly varying definitions of slicing exist [Tip95], but we shall use the following one. A static slice (henceforth referred to as a slice) of a program is defined as those parts of the program that can potentially affect a slicing criterion. A slicing criterion is (in our case) a set of program variables.[OL84] describes slicing as graph reachability on a Program Dependence Graph (PDG) of the program. [BH92] report on a slicing algorithm for arbitrary control flow.

In SAL, we envision combining Slicing with other techniques to verify large systems. We envisage slicing as a pre-processing step before transition systems are fed to the verification tools and conversely use verification techniques to improve slicing.

This report is organized as follows. In Section 2 we provide a brief description of SAL intermediate language. In Section 3 we provide definitions and finally describe the slicing algorithm. In section 4 we describe the methodologies for combining slicing with existing verification tools to verify larger designs. In section 5 we describe related work. In section 6 we describe future work. In this section we also discuss use of decision procedures to further refine slicing and issues related to conversion of temporal formulae into slicing criterion. In section 7 we present conclusions.

2 SAL Intermediate Language

SAL (Symbolic Analysis Laboratory) is a framework for combining different tools for abstraction, program analysis, theorem proving and model checking [BSS]. This framework aims to leverage the above techniques to bring greater automation in verifying larger/infinite state-space systems. The SAL intermediate language (henceforth referred to merely as SAL) is a concurrent language designed to describe both hardware and software systems as transition systems. This language serves as the target for translators that extract transition systems from specification and programming languages like Esterel, Java, Verilog etc. SAL is quite similar to SMV [McM92] and reactive modules [AH96]. Each SAL program consists of collection of modules. Modules may be composed synchronously or asynchronously. Each module consists of guarded transitions and initializations. Each module has input/output variables through which the module communicates with other modules or its environment. The module may also have local variables not visible outside the module. The Expression language of SAL is as rich as the expressions in Java or Verilog. SAL supports composite data types like arrays and records besides boolean, real, integer, natural, subrange etc..For the purposes of our discussions we need only concentrate on the 'guarded transitions' part of SAL. Please refer [BSS] for further information on SAL.

Each Guarded transition (or Guarded Command) consists of a guard and an assignment part. The guard is a boolean expression in the current controlled (local and output) variables and current and next observed (input) variables. The assignment part is a list of equalities between the next of left hand side variable and a right hand side expression in current and next variables. Each assignment inside a guarded command can be executed simultaneously.

3 Slicing SAL

The input to the slicing algorithm consists of the slicing criterion and a SAL program which may be synchronously or asynchronously composed of multiple modules. The slicing criterion is merely a set of local/output variables of a subset of the modules in the input SAL program. The output of the slicing algorithm is another SAL program similarly composed of modules wherein irrelevant code from each module has been sliced out. For every input module there will be an output module, empty or otherwise. In a nutshell the slicing algorithm does a dependency analysis of each module and computes backward transitive closure of the dependencies. This transitive closure would take into return only a subset of all transitions in the module. We call these transitions as observable and the remaining transitions are called as τ or silent transitions. We replace silent transitions with skips.

The following definitions are necessary to describe the slicing algorithm. The algorithm is described in detail in section 4

3.1 Slice

A program slice consists of the parts of a program P that may potentially affect the slicing criterion at some point of interest in the program. In our case we do not explicitly specify any points of interest but put the restriction that the slice will behave as a projection of the original program with respect to the variables specified in the slicing criterion. Our algorithm handles arbitrary control flow.

```
module1: CONTEXT =
BEGIN
module1: MODULE =
BEGIN

OUTPUT a : boolean
INPUT  x : natural
LOCAL PC_1 : natural,y1 : natural,
             y2 : natural
INIT
 TRUE -->
    PC_1 = 1; //control variable
    y1 = 0;
    y2 = 0;
    a = true;
NEXT
 (PC_1 = 1) -->
     next(y1) = 1;
     next(PC_1) = 2;

 (PC_1 = 2) -->
     next(y1) = x + 1;
     next(a) = false;
     next(PC_1) = 3;

 (PC_1 = 3) -->
     next(a) = true;
     next(y2) = x + 1;
     next(PC_1) = 1;
```

MODULE-1

```
module2: CONTEXT =
BEGIN
module2: MODULE =
BEGIN

OUTPUT x : natural;
INPUT  a : boolean, z : natural
LOCAL  PC_2 : natural

INIT
 TRUE -->
    PC_2 = 1;//control variable
    x = 0;


NEXT
 (a AND PC_2 = 1) -->
    next(x)  = x + 1;
    next(PC_2) = 2;

 (a = false AND PC_2 = 2) -->
    next(x) = z + 1;
    next(PC_2) = 2;
```

MODULE-2

Figure 1: A sample SAL code with three modules. third module, which produces output z in turn an input to module-2, is not shown here

3.2 Slicing Criterion

The slicing criterion is defined as a set of output/local variables of some module(s), $S = \{v1, v2, \cdots\}$. The slicing criterion may not include variables input from the environment. Unlike the definition of slicing criterion in other algorithms, our definition does not have a reference to a state/node of the control flow graph(CFG) of the SAL program. In the context of SAL, which is primarily designed for reactive systems, it is not always sensible to slice backwards from an arbitrary node and ignore all susbsequent nodes from this arbitrary node. Reactive systems generally tend to be non-terminating and constantly reacting to the environment. This implies that these systems have a big outer loop and hence slicing from any arbitrary node may result in slicing the whole system. Hence to be most general, the slicing criterion does not refer to any particular node. Also, in SAL there is no general concept of an exit control node, hence we arbitrarily assume the node with highest value for the control variables to be the exit control node.

3.3 Program Dependence Graph

The Program Dependence Graph (PDG) of a program is defined as a Control Flow Graph (CFG) augmented with dependency edges. A CFG consists of nodes where each node represents a control state and edges represent flow of control in the program. Every CFG has a start node. A control node/state in the program is defined by the values of the control variables. Any variables in the program with a finite subrange can potentially act as a control variable. We require the user to mark the control variables in each module. Our slicing algorithm converts each module into a CFG then decorates it with dependency edges to construct the PDG.

A control node of the PDG corresponds to a guarded transition in SAL. The node contains the guard and assignments. One or more of the assignments will define the control variables for the next control state. Please see figure 2 for a PDG.

3.4 Dependency Edges

An edge between two nodes in a CFG is called a dependency edge if the two nodes are related by a dependency relation. We need to consider only data dependence and control dependence[Tip95]. There are many forms of data dependence like flow dependence, output dependence [Tip95] etc. but we need only flow dependence for our purposes. Our Slicing algorithm preserve the control flow of the module and hence we do not consider control dependence.//hassen:think about a counter-example in which we illustrate that if the control structure is not preserved then branching bisimulation cannot be achieved. As explained earlier,those transitions in the program which are not in the slice are replaced by skips. To collapse the skips we use branching bisimulation algorithm [**?**].

3.5 Data Dependence

The sets DEF(i) and REF(i) denote the sets of variables defined and referenced at CFG node i, respectively. We say a variable is defined when it is assigned. We say a control node j is data dependent on node i if there exists a variable x such that:

$x \in$ DEF(i)
$x \in$ REF(j)
there exists a path between from i to j without intervening definitions of x.

3.6 Control Dependence

Control Dependence is defined in terms of post-dominance. A node i in the CFG is post-dominated by a node j if all paths from i to the EXIT node pass through j. A node j is control dependent on a node i if:

There exists a path P from i to j such that any node in P is post-dominated by j
But i is not post-dominated by j

3.7 Special considerations for SAL

As described earlier a typical SAL program is composed of modules. Each module may modify certain output variables. These output variables may be fed back into any module(s) as input or can go back to the environment. A SAL program can be compared to a circuit composed of chips (modules). The slicing algorithm computes a backward transitive closure of the dependences.

The following two possibilities need consideration. An output variable may be completely ignored by the dependency analysis. In this case we do not consider the definitions of that output variable or its references any further. Certain definitions of an output variable may be captured by the dependency analysis and certain other may not. In this case we have to forcibly include every definition of this output variable. This has to be done to preserve the non-deterministic behavior of SAL programs. consider the following scenario in which there are two modules,module1 and module2, in a SAL program. The output variable,a, of module1 is fed into module2. The dependency analysis ignores a certain definition of a, $def(a)$, in module1. Let the control node corresponding to $def(a)$ be $C(a)$. If in the original program module1 stopped after the node $C(a)$, the value of the input to module2 would be defined by $def(a)$. If this definition is ignored in the sliced output then for the same scenario the sliced program would misbehave. Many other scenarios can be drawn up but the essential point is that all definitions of dependent output variables and their dependences need to be captured. The same argument does not apply to local variables since they do not interact outside of the module.

SAL, unlike java, has only certain types of dependences for the shared variables. The input/output ports of a module are implemented thru' shared input/output variables. An output variable is modified by only one module but can be read by multiple modules. In java, the same shared variable can be written by multiple threads resulting in the need for an explicit synchronization. This synchronization has to be carried over to the resulting slice. In SAL explicit constructs for synchronization do not exist and it is the responsibility of the user to provide synchronization through the use of control variables.

3.8 Unresolved input variables

After the dependency analysis of a module terminates, certain variables will still have dependences as yet un-resolved. These variables are always input variables to that module and will either be input from a module or from the environment. If input from a module we add it to the set of unresolved input variables.

3.9 Correctness Criterion for Slicing

A Slice is correct if the behavior, described in terms of process graphs, of the original program with respect a subset of variables used in the program and the behaviour of the slice with respect to the same set of variables are branching bisimilar. Process Graphs[GW96] define the behavior of a program in a run. Branching bisimilarity was first defined by [GW96] as a notion of bisimilarity more refined than weak bisimulation. Branching bisimilarity guarantees that CTL*-X properties of the original program will be preserved in the slice. This characteristic of branching bisimulation argument motivates us to base the correctness of our slicing algorithm on branching bisimulation. Every assignment in the original program may correspond to transition(s) in the process graph. By slicing certain assignments we are replacing these transition(s) in the process graph of the original program by tau or non-observable transitions. The process graph thus resulting is the behavior of the slice. The remaining transitions are called observable transitions. At the moment we are in the process of proving our slicing algorithm correct by arguing that the process graphs of the original program and the slice, resulting by applying our algorithm, are branching bisimilar.

4 The Slicing Algorithm

Roughly speaking the slicing algorithm computes a PDG for each module and does a backward transitive closure of the dependences computed by the dependency analysis. When the dependency analysis is complete for a module a few un-resolved input variables may remain. A variable is said to be unresolved when its dependences have not yet been resolved. The slicer then finds out if these input variables are output of another module or input from the environment. If it is an output of another module, input into the current module, then the slicer slices this new module by computing a transitive closure of the dependences with respect to the input variable (output for this module). the slicer stops when all such input variables have been resolved. The backward transitive closure is also called as the cone of influence. Since a SAL module in general does not have an exit node we are forced to assume that the node with the largest value for the control variables is the exit node. A notion of exit node is essential to compute backward transitive closure of the dependences. We have given the slicing algorithm in pseudo code below.

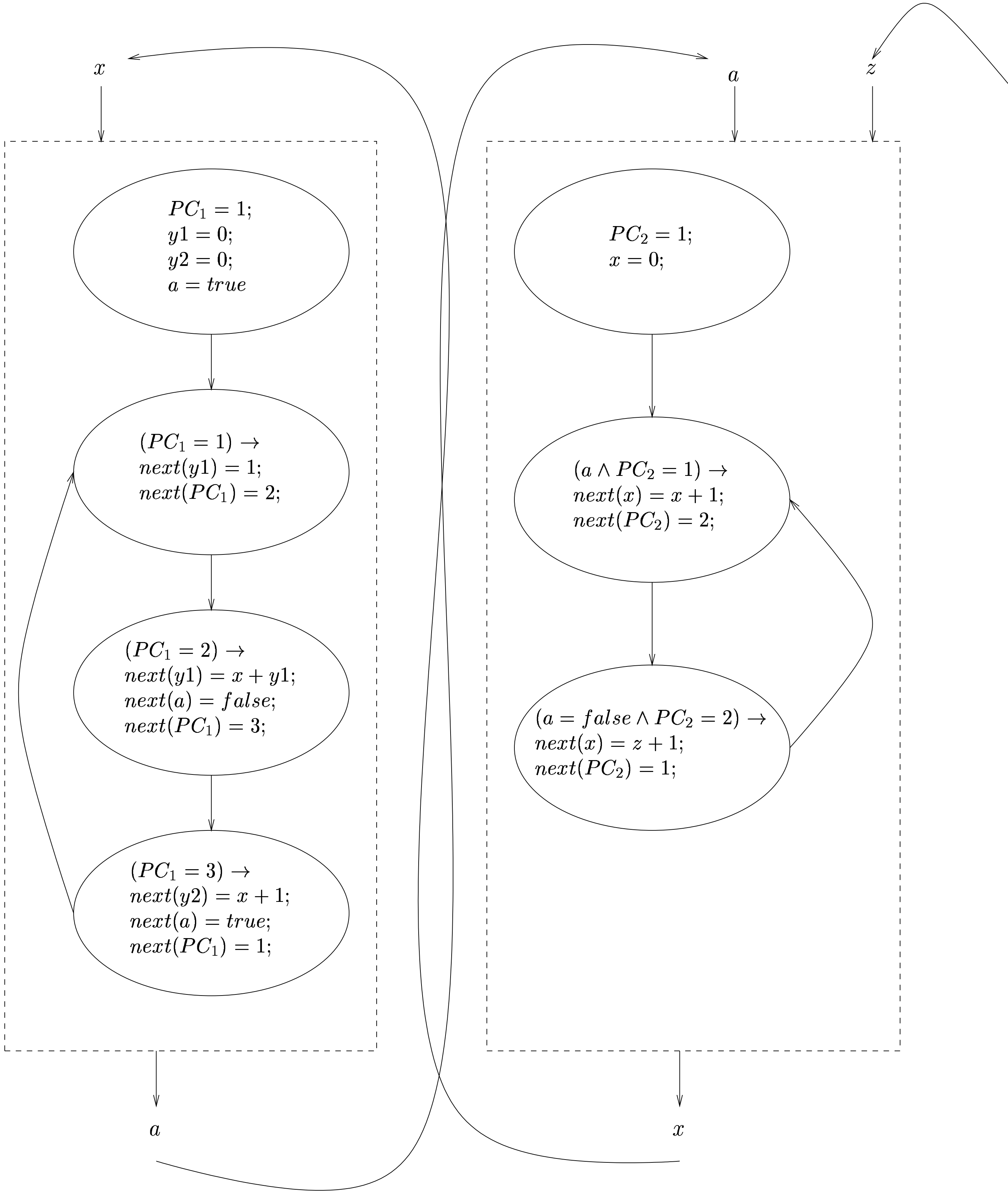
x
a
z
PC_1 = 1; y1 = 0; y2 = 0; a = true
(PC_1 = 1) → next(y1) = 1; next(PC_1) = 2;
(PC_1 = 2) → next(y1) = x + y1; next(a) = false; next(PC_1) = 3;
(PC_1 = 3) → next(y2) = x + 1; next(a) = true; next(PC_1) = 1;
PC_2 = 1; x = 0;
(a ∧ PC_2 = 1) → next(x) = x + 1; next(PC_2) = 2;
(a = false ∧ PC_2 = 2) → next(x) = z + 1; next(PC_2) = 1;
a
x

Figure 2: Control Flow Graph of the SAL example in Figure 1. Statements followed by right arrow are guards. Each assignment in a CFG node is executed simultaneously, independently of other assignments in the same node.

```
Function Slice($P$ : program, $S$ : set of variables in the slicing criterion) : program
Declare
          $L$ : list of CFGs for each Module in $P$
          $G$ : Control Flow Graph (CFG) corresponding to a SAL Module.
          $D$ : Program Dependence Graph (PDG) corresponding to $G$.
          $Q$ : program /*sliced output*/
Begin
          For Each Module in the SAL program
          Begin
                $G$ := compute a CFG;
                $D$ := build a PDG on top of $G$ using dependency analysis;
                insert $D$ into $L$;
                return PDG2Slice($L$,$S$);
          End.
End.

Function PDG2Slice($L$ : list of CFGs for each Module in $P$, $S$ : slicing criterion) : program
Declare
          $sliceVars$ : Working set of variables to slice with.
          $prunedVars$ : Variables removed from the $sliceVars$ set.
          $listOfNodes$ : List of module's CFG/PDG nodes.
          $initNode$ : Initialization node of CFG/PDG.
          $exitNode$ : Node with largest value for control variables.
Begin /*initially all assignments are in the deleted mode.*/
          $sliceVars$ := $S$;
          $prunedVars$:= {}; /*empty set*/
          For Each var, $s$, in $sliceVars$ && not in $prunedVars$
          Begin
                $m$ := module where the $s$ is an output/local var;
                $listOfNodes$ := $m$'s PDG nodes;
                $initNode$ := initialization node of PDG;
                $exitNode$ := node with the largest control values;
                $workingNode$ := $exitNode$;
                do
                Begin
                      traverseDependencyEdges($workingNode$, $s$, $L$);
                      $found$ := searchForLargestControlNode($listOfNodes$);
                      $workingNode$ := $found$;
                      While($initNode$ != $found$ && $initNode$ has not been visited);
                      remove $s$ from $sliceVars$ and add it to $prunedVars$;
                End.
          End.
End.
```

4.1 Termination arguments for the Slicing Algorithm

The For-loop of the function `Slice()` will terminate because any SAL program can have only finite number of Modules. The `PDG2Slice()` function has an outer For-loop and an inner do-while loop. The functions `traverseDependencyEdges()` and `searchForLargestControlNode()` should themselves terminate for `PDG2Slice()` to terminate. The `traverseDependencyEdges()` computes a fixed point or the backward transitive closure of the dependences. Since the number of lines in a module are finite, this function should terminate. While executing `traverseDependencyEdges()` the control nodes visited are marked. The function `searchForLargestControlNode()` searches for that unvisited control Node which has the largest values for the control variables. Since the number of control states is finite (because there are only finite number of control variables each of finite subrange type), and since the do-while loop forces `traverseDependencyEdges()` to visit all nodes, eventually the `searchForLargestControlNode()` will return the $initNode$ and the do-while loop will terminate. The outer For-loop will terminate because there are only finite number of variables in the SAL program. But the $sliceVars$ set does not remain constant. Unresolved input variables get added to it and a variable which has been used for slicing gets deleted from the set. Deleted variables are stored in $prunedVars$ set. By maintaining the $prunedVars$ set we ensure that non-termination due to addition of the same variable again and again to the $sliceVars$ set, is prevented. //add some info on time complexity.

4.2 Implementation details

Our slicer has been written in Java and is integrated into the SAL parser. The slicer accepts SAL programs and the slicing criterion from the command line. Each guarded transition in a module forms a node in the control flow graph. Each control node is determined by the values of the control variables defined by the previous guarded transition. To construct the PDG, we first determine definitions or values which reach a particular control node. This is determined by the standard algorithm for reaching definitions given in the dragon book. Once the reaching definitions are determined a subsequent pass over the flow graph, determines the dependency edges. For every module a PDG is constructed. The final pass executes the PDG2Slice() converting every PDG to a sliced module. It is here that unresolved input variables are propogated from one module to another until a fixed point is reached.

5 Methodologies for using slicing for verification

Slicing is essentially a syntactic transformation and hence is best used as a preprocessor. We believe that slicing will be most useful in aiding more powerful semantic transformation like Property preserving Abstractions and in converting real programs written in languages like Java, Verilog etc. to a more verifiable or model checkable form. Slicing cannot replace semantic transformations like Abstractions. Consider a bakery algorithm written in Verilog. It is typical that the critical region has some code irrelevant to the mutual exclusion property which needs to be verified. Slicing can easily remove this irrelevant code leaving a skeleton Verilog code, which now is more amenable to verification. Engineers tend not to design their programs to be verifiable. Slicing can aid real existing code and future programs to become more amenable to existing verification tools/techniques.

For instance most chip designs tend to have a lot of extra circuitry for test and debug purposes[CFR$^+$97]. This extra circuitry/code is generally irrelevant to the property that needs verification. Slicing can help remove these modules in the code thus easing the job of abstraction and invariant generation. The abstractor and invariant generator will need to process fewer states.

One of the fundamental problems traditionally faced with slicing is that aliasing problems cause lowering in the quality of the slice. Aliasing problems are caused mainly by pointers and also by arrays. Aliasing has been studied extensively by the compiler community though very few good solutions exist. Aliasing forces the dependence analysis to be more conservative thus leading to addition of those lines in the slice which could otherwise have been omitted. Languages (most hardware languages) without pointers will therefore benefit most from Slicing. Other ways of tackling aliasing problems include use of decision procedures to resolve aliases, and use of dynamic or quasi-static slicing instead of purely static slicing.

6 Related Work

[Che93] was one of the first to propose an algorithm for slicing concurrent languages. [Che93] has not provided a correctness proof for the algorithm. Other algorithms for slicing concurrent languages include ones by [DH99] and [Kri98]. [MT] have reused existing slicing algorithm by [Che93] to illustrate application of slicing to model checking. [CFR$^+$97] also report on how slicing can be useful for simulation, model checking etc. [DH99] report on building a slicer for Java and have so far done the most comprehensive work on slicing for verification. We believe that by building a slicer for an intermediate language, we can easily retarget our tool-set for different language unlike dwyer et al. Also dwyer et al.

base the correctness of the their slicer on weak bisimulation. Weak Bisimulation cannot preserve all CTL*-X properties unlike branching bisimulation [GW96]. We believe that by ensuring branching bisimilarity between the input program and the sliced output we can use the slicer for many more types of properties. Our slicing algorithm is one of the few reported which handle concurrency.

7 Future and Ongoing Work

We have built a slicing tool based on the algorithm presented in this paper. We have tested this tool on some simple examples. At the moment one of the ongoing efforts is to prove the correctness of our slicing algorithm based on a branching bisimulation argument between the process graphs of the input program and its slice. We plan to build a Verilog to SAL translator to test the effectivness of the slicer in a real setting. Another ongoing effort is to improve the quality of the slicer by incorporating decision procedures and invariant generator.

Invariant generation used in combination with decision procedures can refine the control structure of the parallel composition of the modules in a SAL program. In [?], it is shown that this combination allows us to discover that certain transitions may not be executed in a certain order. Consider the example in figure 1. The variable `y` is modified in module 1 only when the variable `a` is true. In module 2 variable `x` is modified with an input variable `z` only when variable `a` is false. Variable `x` forms the input to module 1, hence, the slice of module 2 will include all assignmenents to `x` and their dependencies. The dependencies include the variable `z` that is an output of module 3. Invariant generation techniques allows us to determine that the output value of `x` that affects variable `y1` is the value generated by the assignment `next(x)=x+1` that is executed only when `a` is true. Therefore the assignment `next(x)=z+1` will not affect the value of `y1`. The slicer can then safely remove module 3.

Many safety properties refer to values of program or control variables explicitly. This fact can be leveraged to do quasi-static or dynamic slicing. In quasi-static[Tip95] slicing certain program variables are assumed to retain an initial value under all runs of the program. The values specified in the property can be used initialize these variables and then do a quasi-static slicing. Quasi-static Slicing will return a much higher quality slice than purely static slicing. Another issue which needs further exploration is the conversion of the property specification in temporal into a slicing criterion.

8 Conclusions

We have presented a concurrent slicing algorithm for SAL. We have discussed the use of slicing to verification and presented possible limitations of slicing. We have also discussed methodologies for using slicing in conjunction with abstraction and model checking. We have implemented a prototype tool based on our algorithm and experimented with a few examples. We are at present working on a proof of correctness for our algorithm. We also plan to use decision procedures to improve the quality of the slice and improve the existing strategies to extract slicing criterion from property specifications.

9 Acknowledgements

I wish to thank Ashish Tiwari and prof. Joseph Sifakis for taking time to hold discussions with me during the early stages of this work. I thank Patrick Lincoln and prof. John Mitchell for giving me an opportunity to work on this project.

References

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[BBC^{+}96] Nikolaj Bjorner, Anca Browne, Eddie Chang, Michael Colon, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomas Uribe. Step: Deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *LNCS*, pages 415–418. Springer Verlag, August 1996.

[BH92] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. In *First International Workshop on Automated and Algorithmic Debugging*, pages 223–240. vol 749 of Lecture Notes in Computer Science, Springer-Verlag, 1992.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[BSS] The SAL Group: UC Berkeley, SRI, and Stanford. Sal intermediate language. First draft on SAL.

[CES83] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification: a practical approach. In *10th ACM Symposium on Principles of Programming Languages (POPL83)*, 1983. Complete version published in ACM TOPLAS, 8(2):244–263, April 1986.

[CFR$^+$97] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing for design automation: An automatic technique for speeding-up hardware design, simulation, testing and verification. 1997.

[Che93] J. Cheng. Slicing concurrent programs, a graph theoretical approach. In *First International Workshop on Automated and Algorithmic Debugging*, pages 223–240. vol 749 of Lecture Notes in Computer Science, Springer-Verlag, 1993.

[CU98] Michael Colon and Thomas Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[DH99] Matthew B. Dwyer and John Hathcliff. Slicing software for model construction. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, January,1999.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[GW96] Rob J. Van Glabeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. In *Journal of The ACM*, pages 553–600. ACM, 1996.

[Kri98] Jens Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7):35–42, July 1998.

[McM92] K. L. McMillan. Symbolic model checking. Phd thesis, Carnegie Mellon University, July 1992.

[MT] Lynette I. Millet and Tim Teiterlbaum. Slicing promela and its application to model checking, simulation, and protocol understanding. Department of Computer Science, Cornell University.

[OL84] K.J. Ottenstein and L.M.Ottenstein. The program dependence graph in a software development environment. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM SIGPLAN Notices 19(5), May, 1984.

[ORR^{+}96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[PH97] A. Pardo and G.D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[QS82] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems is cesar. In *International Symposium on Programming, LNCS 137*, pages 337 – 351. Springer Verlag, 1982.

[Saï97] Hassen Saïdi. The Invariant-Checker : Automated deductive verification of reactive systems. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'97*. Springer Verlag, 1997. `www.csl.sri.com/~saidi`.

[SS99] Hassen Saidi and Natarajan Shankar. Abstract and model check while you prove. In *Computer Aided Verification*, 1999.

[Tip95] Frank Tip. A survey of program slicing techniques. In *Journal of programming languages,3:121-189*, 1995.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.