Formal Verification of a System-on-Chip Using Computation Slicing

Alper Sen [1], Jayanta Bhadra [2], Vijay K. Garg [1] and Jacob A. Abraham [1]

[1] The University of Texas at Austin [2] Motorola Inc.

Contact email: sen@ece.utexas.edu

Abstract

Formal verification of Systems-on-Chips (SoCs) is an immense challenge to current industrial practice. Most existent formal verification techniques are extremely computation intensive and produce good results only when used on individual sub-components of SoCs. Without major modifications they are of little effectiveness in the SoC world. We attack the problem of SoC verification using an elegant abstraction mechanism, called computation slicing, and show that it enables effective temporal property verification on large designs. The technique targets a set of execution sequences, that is exhaustive with respect to an intended subset of system level properties, and automatically finds counter-example execution sequences in case of errors in the design. We have obtained exponential gains in reducing the global state space using a polynomial-time algorithm, and also applied a polynomial-time algorithm for checking global liveness and safety properties. We have successfully applied the technique to verify properties on two high level transaction based designs – the MSI cache coherence protocol and an admittedly academic SoC having a bus arbiter and a parameterizable number of devices connected to a PCI bus backbone.

1 Introduction

Modern day hardware systems often consist of individual IP blocks connected in meaningful arrangements carrying out independent as well as cooperative objectives. Such Systems-on-Chips (SoCs) are gradually becoming more and more diverse and complicated, thereby gaining popularity. Their enormous potential for re-using IP makes them economically viable solutions to many products out in the market today. The design of an SoC can be very large and complex. Therefore, verification of such systems is a hard but an important problem. Shorter design cycles increase the severity of the problem, elevating early bug detection to the highest priority. While reasonably effective techniques exist to tackle the problem of verification for individual sub-components, to the best of our knowledge, we are yet to see any report on formal verification of SoCs. This is because SoCs represent an exponential increase in state space as well as design complexity with respect to their sub-components. The dual problem of size and complexity makes existing techniques totally ineffective for SoC verification. In this paper, we address these issues using an abstraction technique that provides satisfactory amelioration in both areas.

As design objectives become larger and more complex, the natural design methodology is to decompose a large system into relatively independent pieces or components that are more manageable. Analysis of hardware designs is often limited to the analysis or testing of the components in isolation since analyzing an entire system is limited by its own size and complexity. As a result, although individual components can be effectively verified as stand alone modules embedded in carefully constructed environments, many bugs can be introduced when the pieces of a system are put together and the environmental assumptions used in designing each piece are not guaranteed by the actual environments [3]. These interface errors can be difficult to detect because the knowledge needed to effectively test for these errors is distributed among different design groups. These difficulties are exacerbated greatly for high level hardware designs because of their concurrency, i.e., design components consisting of multiple interacting threads. In such design descriptions the environmental assumptions have the added dimension of time and involve sequencing of events from different sources. The complexity of debugging concurrent programs has lead to interest in *model checking* approaches [5] where the state space of a program is often exhaustively (at least in an implicit fashion) searched for erroneous behaviors. Unfortunately, practical model checking techniques are limited to programs with a relatively small number of states or a small number of boolean variables, while many concurrent programs reside in megabytes of memory. Some verification of individual blocks have been achieved by *theorem proving* where conditions of system correctness have to be written in precise terms of mathematics as theorems and have to be manually proved using axioms and lemmas arrived at by examining the behavior of the system under verification [4]. Many individual IP blocks can be verified to some satisfaction by using these techniques. Many such verification obligations have to be achieved only by very experienced users with a very good domain knowledge (theorem proving), or by relatively automatic techniques (model checking) that do not achieve satisfactory verification on large designs like SoCs because of capacity limitations. Therefore, random and pseudo-random techniques for generating tests have been used for SoCs with some success [24, 1]. However, these techniques have not yet reported any major success with verification of large and complex SoC interfaces where coverage plays a major role. Although simulation is expected to remain the mainstay of SoC validation due to its simplicity and scalability, we choose model checking using abstractions on system level execution traces as we believe that this approach provides us with a judicious blend of automation and the ability to find bugs in large and complex interfaces of SoCs.

In this paper, we introduce an application of *computation slicing* [11, 22, 30, 28] to verification of high level hardware design descriptions. As opposed to *program slicing* [36], where the user is interested in generating a projection of a program with respect to a set of interesting variables, computation slicing generates program abstractions with respect to execution traces (also called computations). Normally, an execution trace can be defined as a directed graph and so specialized graph algorithms can be employed to find the abstractions for property checking [30, 28]. The inputs to our automatic abstraction generation algorithm are a set of execution traces and a temporal logic property (interchangeably called a temporal logic predicate) that is a subset of Computational Tree Logic (CTL) [5]. A slice, or an abstraction, defined with respect to a global predicate, is the computation with the least number of global states that contains all the global states of the original computation for which the property evaluates to true. The algorithm spends polynomial amount of time to compute an abstraction slice that has exponential reduction in the number of global states one needs to examine in order to validate the property [30, 28]. We also use a polynomial-time algorithm to verify the property on the slice. This allows an exponential overall speedup as one need not exhaustively examine the entire state space any more. Our technique judiciously marries two fundamental state space reduction tools – (a) non-determinism: the inherent non-determinism of a *partial order* execution captures possibly exponential number of total order executions, and (b) divide and conquer: analysis of exponentially abstracted partial order traces for property checking. Since the criterion for generating the abstraction is not a set of variables, but a temporal logic property, computation slicing is a better choice for verification of hard-to-verify properties on large systems, like the SoCs. Therefore, our technique is more attuned to property checking than program slicing, which is more useful for manufacturing test generation applications [35]. Also, the verification check in our technique is as strong as model checking because in the end the initial state(s) need to be in the set of global states obeying the temporal property under verification. However, unlike in classical model checking, where the entire state space has to be computed and traversed, in our technique we can abstract the state space of a partial order computation trace and verify properties in a smaller subset of global states. Even if model checking algorithms are used on partial order computation traces as in our case, the complexity will still be exponential unlike in our case. Therefore, great success can be achieved when our technique is employed in property checking for systems where it is hard to even build the entire state transition relation, let alone explicitly or implicitly traversing it in its entirety. Our initial experimental results show a lot of promise of the proposed technique with verification of complex SoCs.

In this paper, we provide (a) an introduction to computation slicing, (b) a systematic approach of how it can be used on high level transaction based descriptions of hardware for property verification on SoCs, and (c) experimental results showing the effectiveness of our technique on large scale transaction based hardware descriptions.

2 Related Work

Although the area of formal verification of SoCs is extremely important, it is relatively new. To the best of our knowledge there has not been much work published in this area. However, the area of validation of SoCs has been a relatively more popular area of research. Although the following techniques are not formal in nature this discussion will give the state of the art in SoC validation. (A) Directed test cases that were carefully hand generated by verification engineers that worked on individual IPs stopped being effective on SoCs because of their sheer size and complexity. Therefore, random and pseudo-random test generation techniques became popular [24, 1]. However, these techniques were largely troubled by the issue of coverage and effectiveness on complex interfaces. (B) The idea of self checking tests is to compute predicted results at the time of test generation. SpecMan [33], from Verisity Design Systems, QuickBench [32], from Forte Design Systems and Vera [16], from Synopsys use this approach. This approach has two major drawbacks – firstly, the observability is limited and secondly, most of the times information regarding relative timing of various transactions may be missing, making it hard to debug. Although some of the problems can be mitigated by a cycle-accurate model, but that approach entails a huge performance penalty. (C) Compiled testing paradigm for SoCs embeds all the requisite data of a test into the source of the test generator. The test generator thus has to be compiled for each test or a set of related tests. The main advantage of this technique is that if the test generator is compiled into the testbench dynamic behavior can be easily handled [26]. However, dynamic behavior was also achieved by a later data driven test generator as well [24].

Program slicing, originally proposed by Weiser [36], is a static program analysis technique to extract appropriate portions of programs relevant to a set of program variables. These portions are referred to as slices – artifacts that maintain exact information about the program's behavior projected onto the relevant segments of the original program. This technique has been widely studied and applied to a myriad of applications in software engineering such as debugging [8], testing [20], maintenance [9] and reuse [19]. Such a source-to-source transformation technique offers an opportunity to formulate a systematic approach to simplify a design described in a Hardware Description Language (HDL) for test generation without having to synthesize a prohibitively large design. However, most of these algorithms have been developed for sequential languages and cannot be directly applied to HDLs such as Verilog or VHDL, which allow concurrent constructs. Iwaihara *et al.* [15] suggested an approach to use program slicing for analyzing VHDL designs. An automated program slicing approach for VHDL was proposed by Clarke *et al.* [6], in which VHDL constructs were mapped onto constructs for C-like procedural languages. The primary focus of their work was formal verification. Vedula *et al.* [35] reported a technique where program slicing was used for the first time to automatically generate effective manufacturing tests from HDLs.

The problem of test generation on SoCs has been addressed using divide and conquer as well as Built In Self Test (BIST) and full or partial scan designs – either individually or in a judicious combination. However, for temporal property checking tackling the problem of state space explosion becomes of primary importance as the techniques that are effective for manufacturing test generation do not always work for property checking. This is because while structural information is important in the manufacturing test generation world, functional information is more important in the temporal property checking area.

The idea of using temporal logic for analyzing execution traces (also referred to as runtime verification) has recently been attracting a lot of attention. We first presented a temporal logic framework for partially ordered execution traces in [27] and a tool for runtime verification in [30]. Some other examples of using temporal logic for checking execution traces are the MaC tool [17], the JPaX tool [13], and the JMPaX tool [31]. The specification languages used in these tools are different than ours and we have polynomial-time complexity, whereas the complexity is exponential for the other tools.

Computation slicing was first proposed by Garg *et al.* [11] in order to generate abstractions of computations of distributed programs (finite execution traces). The central problem in predicate detection (interchangeably called property checking here) in a hardware description is the *state explosion problem* – the set of possible global states of a design consisting of n individual processes, each having at most k states, is of the order of $O(k^n)$. Strategies like symbolic state representation and partial order reduction have been proposed earlier to address the problem [12, 21, 25, 34]. The technique of computation slicing has been shown to be extremely useful for throwing away extraneous global states of the original computation (with respect to a given property) in an efficient manner, and focus on only the relevant global states [30, 28]. Although computation slicing has been used successfully on computations of distributed programs, we will apply it for the first time to solve the problem of formal verification of temporal properties in high level hardware descriptions. The next section deals with details of the technique of computation slicing.

3 Slicing Partial Order Traces

We can model an execution trace of a transaction level hardware description (interchangeably called a program here), which consists of events, in two ways. The first model imposes a total order (interleaving) of events. The second model imposes a partial order between events. A partial order is a reflexive, antisymmetric, and transitive binary relation. Traditionally, the former model has been used in testing, simulation and runtime verification. We use the latter approach which has several advantages over the former. First, it is a more faithful representation of concurrency [18], that is, only the events that have a causal dependency are ordered. Second, a partial order encodes possibly exponential number of total orders. This usually translates into better coverage in terms of testing and bug detection than in cases where a total order is used. Third, our partial order approach is applicable to both message passing and shared memory based transaction level hardware descriptions. For example, the partial orders in [18] and [31] are used for representing traces of message passing and shared memory programs, respectively. These aspects aid in modeling transaction level hardware descriptions in a very compact and efficient manner aiding in relatively easier debugging.

Example: Consider an execution of a program. The partial order model of the resulting execution trace is shown in Figure 1(a). In the trace, there are two processes P_1 and P_2 with local integer variables x and y, respectively. The events are represented by circles. Process P_2 sends a message to process P_1 by executing event f_1 and process P_1 receives that message by executing event e_1. In each process an event is labeled with an ordered tuple – the value of the respective local variable immediately after the event is executed and a *vector clock*, which we will define later. For example, the value of x immediately after executing e_1 is 2. The first event on each process initializes the state of the process. Figure 1(b) contains the set of all reachable global states of the computation reachable from the initial state $\{e_0, f_0\}$. In the figure, we represent a global state as a tuple where each element is the last event that occurred on a process. Observe that $\{e_1, f_0\}$ is not a reachable global state because it depicts a situation where a message has been received from P_2 by P_1, that is e_1, but P_2 has not yet sent the message. By using a partial order representation, we are able to capture all possible interleavings of events, ten in total, rather than a single interleaving. One such inter-

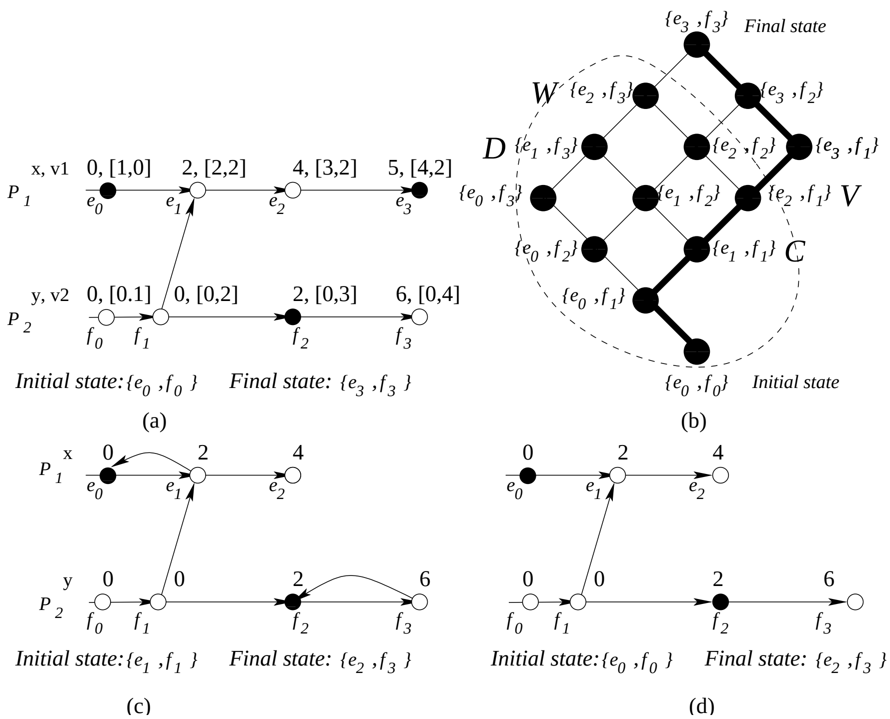

Figure 1: (a) A computation (b) its set of all reachable global states (c) its slice with respect to $(2 \leqslant x \leqslant 4) \wedge (y \neq 2)$ (d) its slice with respect to $\mathsf{EF}((2 \leqslant x \leqslant 4) \wedge (y \neq 2))$

leaving sequence is $\{e_0, f_0\}$, $\{e_0, f_1\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_2\}$, $\{e_3, f_3\}$ as shown in Figure 1(b) with thick lines. Therefore, we can obtain better coverage in terms of testing and debugging by capturing all interleavings. This coverage may translate into finding bugs that are not found using a single interleaving.

System Model: We assume a system consisting of processes denoted by $P_1, \ldots, P_n$. Examples of processes are a node sitting on a PCI bus or a cache in a cache coherence protocol. Processes execute events. Events on the same process are totally ordered. However, events on different processes are only partially ordered. In this paper, we relax the partial order restriction on the set of events and use directed graphs to model computations as well as slices.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote the set of vertices and edges, respectively. We define a *consistent global state* on directed graphs as a subset of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. Formally, C is a consistent global state of G, if $\forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) \wedge (f \in C) \Rightarrow (e \in C)$. We denote the set of consistent global states of a directed graph G by $\mathcal{C}(G)$, which forms a distributive lattice under the subset relation ($\subseteq$) [22]. We say that a state D is *reachable* from a state C if $C \subseteq D$. In the rest of the paper, unless otherwise stated, a global state or simply a state refers to a consistent global state.

We model an *execution trace* (or a *computation*) as a directed graph. We use event and vertex interchangeably. Figure 1 shows a computation and its lattice of global states. Only the frontier of the global states is displayed in the figure where the frontier is composed of events in a global state that occur last in process order.

How to generate partial order traces: For modeling message passing program traces, a partial order relation known as Lamport's happened-before relation [18] has been used. Lamport's *happened-before relation* is defined as the smallest transitive relation satisfying the following properties:

- if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and
- if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

A mechanism known as *vector clocks* has been used to represent the partial order relation on events in traces. A vector clock assigns timestamps to events such that the partial order relation between events can be determined by using the timestamps.

Definition 1 (vector clock) *Given a computation G on n processes, a vector clock v is a map from $\mathsf{V}(G)$ to $\mathcal{N}^n$ (vectors of natural numbers) such that*

$$\forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) \iff e.v \leqslant f.v$$

where e.v is the vector assigned to the element e.

For example, given an n process message passing program, for every process j, there is a vector clock of size n, denoted by v_j. Initially, $v_j[i] = 0$, for $i \neq j$, and $v_j[j] = 1$. The vector clock is manipulated after each event in the process. A process increments its own component of the vector clock after each event. It also includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by taking a component wise maximum with the vector clock included in the message. A sample execution of the algorithm is given in Figure 1(a), where the vector clocks corresponding to all the states have been shown. Vector clock algorithms for shared memory programs also exist [31].

Computation Slice: The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [7]. Informally speaking, a computation slice (or a slice) is a concise representation of all those global states of the computation that satisfy a global predicate (or simply predicate or property). Formally,

Definition 2 (slice [22]) *A* slice *of a computation with respect to a predicate is a directed graph with the least number of global states that contains all global states of the given computation for which the predicate evaluates to true.*

We denote the slice of a computation G with respect to a predicate p by $\mathsf{slice}(G, p)$. Note that $G = \mathsf{slice}(G, \mathsf{true})$.

Figure 1(c) shows the slice of the computation in Figure 1(a) with respect to the predicate $(2 \leqslant x \leqslant 4) \wedge (y \neq 2)$, which is a conjunction of two local predicates from the processes P_1 and P_2, respectively. We say that a predicate is *local* if it only depends on variables of a single process. In the figure, if a local predicate is true for an event of a process we represent it using an empty circle, otherwise with a filled circle. To obtain the slice, we add an edge from the successor of a filled circle back to it, thereby increasing the number of incoming neighbors. As a result, from the thirteen consistent global states in Figure 1(b), this exercise eliminates nine – retaining states C, D, V, and W. In order to understand how we generate slices, we use the following theorem.

Theorem 1 ([10]) *Let G be a directed graph. Let H be obtained by adding edges to and removing vertices from G. Then $\mathcal{C}(H)$ is a sublattice of $\mathcal{C}(G)$. Conversely, every sublattice of $\mathcal{C}(G)$ is generated by some directed graph H obtained from G by adding edges and removing vertices.*

Theorem 1 gives a general procedure to compute slices for arbitrary predicates. It has been shown that the slice exists for all predicates, however it is, in general, intractable to compute the slice for an arbitrary predicate [11]. Our approach to computation slicing is based on exploiting the structure of the predicate itself. For this purpose, we focused on regular predicates.

Definition 3 (regular predicates [11]) *A predicate is regular if the set of consistent global states that satisfy the predicate forms a sublattice of the lattice of consistent global states. Equivalently, a predicate p is* regular *with respect to G if it is closed under set union and intersection, i.e., for all consistent global states C, D of G:*

$$(C\ satisfies\ p) \wedge (D\ satisfies\ p) \Rightarrow$$

$$(C \cup D)\ satisfies\ p \wedge (C \cap D)\ satisfies\ p$$

Some examples of regular predicates are conjunction of local predicates such as "P_i and P_j are in critical section", monotonic channel predicates such as "there are at least k messages in transit from P_i to P_j", and relational predicates such as "$x_1 - x_2 \leqslant 5$, where x_i is a monotonically non-decreasing integer variable on process i". A disjunctive predicate such as $x_1 \vee x_2 \vee x_3$ is not regular.

In the light of Theorem 1, since the set of global states satisfying a regular predicate forms a sublattice of the set of global states of the computation, we can obtain the slice for regular predicates from the given computation by adding edges and removing vertices. Furthermore, these slices are exact (also called *lean slices*) in the sense that they contain exactly the states that satisfy the predicate.

Although Theorem 1 gives a general procedure to compute slices for arbitrary predicates, it does not specify how to add edges or remove vertices. We developed efficient (polynomial in the number of processes or better) slicing algorithms for both non-temporal and temporal regular predicates and to compute slices with respect to boolean combination of regular predicates [11, 22, 28].

4 Slices for Non-Temporal Predicates

We present an algorithm to compute the slice for a non-temporal regular predicate from [23] after introducing a lemma. Non-temporal regular predicates are regular predicates that contain only boolean operators and no timing information.

Let G be a directed graph and let $G[e, f]$ denote the directed graph obtained by adding an edge from e to f in G. The following lemma shows which edges should be added to obtain the slice.

Lemma 2 ([23]) *There is an edge from an event e to an event f in $\mathsf{slice}(G, p)$ if and only if no consistent global state in $\mathcal{C}(G) \setminus \mathcal{C}(G[e, f])$ satisfies p.*

We can efficiently obtain a graph $\widehat{G}[e, f]$ from G by adding an edge from f to every initial event and removing vertex e and all its successors. Graph $\widehat{G}[e, f]$ is pertinent here because it can be shown that $\widehat{G}[e, f]$ has the same state space as $\mathcal{C}(G) \setminus \mathcal{C}(G[e, f])$.

Lemma 2 is useful provided it is possible to ascertain efficiently whether some consistent global state in $\widehat{G}[e, f]$ satisfies p. Let CONTC denote the problem of deciding whether some consistent global state of a given graph satisfies p. There are efficient CONTC algorithms for non-temporal regular predicates [23]. Figure 2 gives the algorithm for computing the slice for a non-temporal predicate.

Input: (1) a directed graph G, (2) a predicate p, and (3) an algorithm to evaluate CONTC(H, p) for an arbitrary directed graph H

Output: the slice of G with respect to p

$K := G$;
for every pair of events (e, f) do
    if not(CONTC$(\widehat{G}[e, f], p)$) then
        add an edge from e to f in K;
            // $K := K[e, f]$
    endif;
endfor;
output K;

Figure 2: An algorithm to compute the slice for a non-temporal predicate

The complexity of the algorithm is a small multiple of the complexity of an algorithm for CONTC. In particular, the multiple is $|E|^2$, where E is the set of vertices in G. The complexity of CONTC in the case of non-temporal regular predicates is $O(n|E|)$, where n is the number of processes. In [11], we present a more efficient slicing algorithm for non-temporal regular predicates with $O(n^2|E|)$ complexity.

There are no known efficient algorithms for solving CONTC for temporal regular predicates, i.e., regular predicates containing timing information. Therefore, we cannot use the above slicing algorithm to compute the slice for such predicates. Next, we will discuss a temporal logic for specification of hardware predicates. In Section 6, we will present our slicing algorithms for temporal regular predicates.

5 Specification of Hardware

The concept of slicing is useful for detecting temporal logic predicates since it enables us to reason only on the part of the global state space that could potentially affect the predicate. Many specifications of hardware are temporal in nature because designers are interested in properties related to the sequence of states during an execution rather than just the initial and final states. We can specify both *safety* and *liveness* properties of a system using temporal logic. A safety property specifies that something bad will never happen, whereas a liveness property specifies that something good will eventually happen. For example, the safety property in a cache coherence protocol, "no two cache lines are in the exclusive state at the same time", is a temporal property, as is the liveness property in a bus protocol, "all requests for the bus are eventually acknowledged". We use the *branching* temporal logic CTL [5] to write specifications. This logic allows us to express many alternative futures from every instant of time. Each *path* is a sequence of global states ending at the final state, where a successor of each global state on the path is obtained by the execution of a single event from a single process. We interpret CTL on the finite distributive lattice of global states of a computation.

Basic temporal operators of this logic include EF, AF, EG, and AG. There are two path quantifiers: A denotes *for all paths* and E denotes *for some path*. There are two linear temporal operators that are of interest to us: G, that is the *always* operator, and F, that is the *eventually* operator. We define some of these operators here, the rest may be found elsewhere [5].

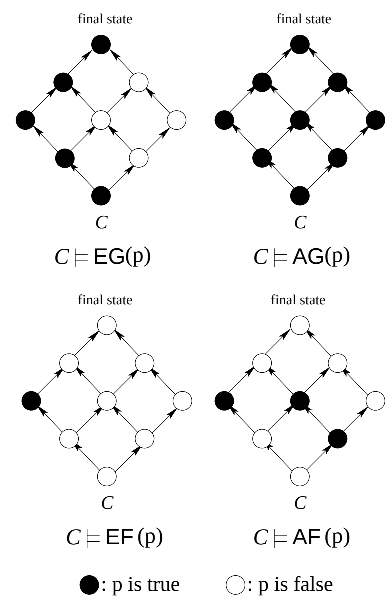

Figure 3: Basic CTL operators

Given a global state, a predicate is evaluated with respect to the values of variables resulting after executing all events in the state. If a predicate p evaluates to true for a global state C, we say that C satisfies p (symbolically, $C \models p$). We say that temporal predicate EF(p) (resp. EG(p)) holds at a global state C if and only if *for some path* starting from C and ending at the final state, the predicate p *eventually* (resp. *always*) holds on the path. We say that temporal predicate AF(p) (resp. AG(p)) holds at a global state C if and only if *for all paths* starting from C, the predicate p *eventually* (resp. *always*) holds on all the paths. There can be arbitrary nesting of temporal operators. For example, a nested temporal predicate AG(EF$(reset)$) states that reset is possible from every state. Figure 3 illustrates the afore mentioned temporal predicates on a lattice of global states where C

is the initial state.

6 Slicing and Checking for Temporal Predicates

In this section we present slicing algorithms for temporal regular predicates from [28]. It has been proved that only a subset of CTL temporal operators are regular [28]. Those are – EF, AG, and EG collectively referred to as *Regular CTL* (RCTL). Most typical safety and liveness properties of hardware can be specified in RCTL. We developed a property checking system that uses slicing of temporal predicates in RCTL. For the sake of brevity, we provide slicing algorithms for temporal operators AG and EG in Figure 4. The slicing algorithm for EF will be explained in the next section with a simple example.

From Theorem 1, it is known that a slice may contain *additional edges*. Since the set of states that satisfy $\mathsf{AG}(p)$ and $\mathsf{EG}(p)$ are a subset of states that satisfy p, the slice with respect to $\mathsf{AG}(p)$ and $\mathsf{EG}(p)$ can be obtained by adding even more edges to the slice with respect to p. The algorithms in Figure 4 show which edges should be added.

Both algorithms take a computation and its slice with respect to a non-temporal regular predicate p as inputs. The output of each algorithm is a slice for the temporal predicate. We make the following observation to obtain the $\mathsf{AG}(p)$ algorithm.

Lemma 3 ([28]) *For each additional edge (e, f) and global state D in $\mathsf{slice}(G, p)$, if D does not include vertex e then D does not satisfy $\mathsf{AG}(p)$.*

To see this, first observe that the global state that contains f but not e (denote this state by V) is not a consistent global state of $\mathsf{slice}(G, p)$, which implies that V does not satisfy p. Second, D does not include either e or f but V includes f so $D \subset V$, i.e., V is reachable from D. Finally, since a state that does not satisfy p is reachable from D, D does not satisfy $\mathsf{AG}(p)$. Therefore, for any additional edge (e, f) in $\mathsf{slice}(G, p)$, we add an edge from e to all the initial vertices (Algorithm AG Step 2), thereby increasing the incoming neighbors of the initial vertices. This procedure results in all consistent global states of the output graph including e.

The algorithm for $\mathsf{EG}(p)$ is similar to that of $\mathsf{AG}(p)$, however, this time for every additional edge (e, f) that generates a strongly connected component in $\mathsf{slice}(G, p)$, we add an edge to all the initial vertices. For any state D in $\mathsf{slice}(G, p)$ that does not include the component, we cannot construct a path in G that starts from D. This is because any global state upon including a vertex from a strongly connected component, has to include all the vertices in the component. Therefore, we add an edge from e to all the initial vertices (Algorithm EG Step 2).

The complexity analysis is similar for both algorithms. In Step 2 we can add edges for each additional edge in $\mathsf{slice}(G, p)$. There are $O(n|E|)$ such edges when the skeletal representation of a slice is used [22]. Therefore, the overall complexity of both algorithms is $O(n|E|)$, which is linear in the number of processes.

Algorithm AG

Input:	A computation G and $\mathsf{slice}(G, p)$
Output:	$\mathsf{slice}(G, \mathsf{AG}(p))$
1.	Let K be $\mathsf{slice}(G, p)$
2.	For each pair of vertices (e, f) in $\mathsf{slice}(G, p)$ such that, (i) $(e, f) \notin \mathsf{E}(G)$, and (ii) $(e, f) \in \mathsf{E}(\mathsf{slice}(G, p))$ add an edge from vertex e to the initial vertex on each process in K
3.	**output** K

Algorithm EG

Input:	A computation G and $\mathsf{slice}(G, p)$
Output:	$\mathsf{slice}(G, \mathsf{EG}(p))$
1.	Let K be $\mathsf{slice}(G, p)$
2.	For each pair of vertices (e, f) in $\mathsf{slice}(G, p)$ such that, (i) $(e, f) \notin \mathsf{E}(G)$, and (ii) Both (e, f) and $(f, e) \in \mathsf{E}(\mathsf{slice}(G, p))$ add an edge from vertex e to the initial vertex on each process in K
3.	**output** K

Figure 4: Algorithm for generating a slice with respect to $\mathsf{AG}(p)$ and $\mathsf{EG}(p)$

Temporal Property Checking:

The *property checking (predicate detection)* problem is to decide whether the initial global state of a computation satisfies a predicate.

Figure 5 displays our property checking algorithm that uses slicing algorithms explained earlier. The complexity of property checking for RCTL is dominated by the complexity of computing the slice with respect to a non-temporal regular predicate, which has $O(n^2|E|)$ complexity [11, 22]. Therefore, the overall complexity of property checking for RCTL is polynomial in n, that is, $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p.

Input: A computation G and an RCTL property p
Output: Property is satisfied or not
1. Recursively process p from inside to outside while applying temporal and boolean operators to compute slices
2. **If** $\mathsf{initial}(G) \neq \mathsf{initial}(\mathsf{slice}(G, p))$ **then**
3.   **return** false and a counter-example
   else
4.   **return** true

Figure 5: Property Checking using Slicing

7 Simple Example

To illustrate property checking using computation slicing as shown in Figure 5, consider the computation in Figure 1(a) again. Let $p = (2 \leqslant x \leqslant 4) \ \wedge \ (y \neq 2)$, and suppose we want to detect $\mathsf{EF}(p)$, that is, whether there exists a global state that satisfies p. Without computation slicing, we are forced to examine all global states of the computation, thirteen in total, to decide whether the computation satisfies the predicate. Alternatively, we can compute the slice of the computation with respect to the regular predicate $\mathsf{EF}(p)$ and use this slice for predicate detection. For this purpose, first (Figure 5, start of Step 1), we compute the slice with respect to p as explained in Section 3. The slice is shown in Figure 1(c). The slice contains only four states C, D, V and W and has much fewer states than the computation itself – exponentially smaller in many cases – resulting in substantial savings. Using the slice in Figure 1(c), we can obtain the last state that satisfies p in the computation, which is denoted by W. We also know from the definition of $\mathsf{EF}(p)$ that every global state of the computation that can reach W satisfies $\mathsf{EF}(p)$, e.g., states enclosed in the dashed ellipse in Figure 1(b). Therefore, applying this observation we can compute the slice with respect to $\mathsf{EF}(p)$ as shown in Figure 1(d) (Figure 5, end of Step 1). Note that the slice has less vertices than the computation and the slice and the computation have the same global states up-to W. Finally, we check (Figure 5, Step 2) whether the initial state of the computation is the same as the initial state of the slice. If the answer is yes (Figure 5, Step 4), then the predicate is satisfied, otherwise not, and a counter-example is returned (Figure 5, Step 3). In this case the predicate p is satisfied.

8 Experimental Results

We ran experiments with our tool Partial Order Trace Analyzer (POTA) for checking execution traces of programs. The overall structure of POTA architecture is shown in Figure 6. The tool contains three modules; analyzer, translator, and instrumentor.

The *instrumentation* module inserts code at the appropriate places in the program to be monitored. The instrumented program is such that it outputs the values of variables relevant to the predicate in question and keeps a vector clock. Figure 1 shows such an output trace where x and y are the relevant variables for the predicate in question. The translator module translates execution traces into Promela [14] (SPIN input language). This module enables comparison with SPIN on execution traces. Since we are working with programs which exhibit a lot of parallelism and independence, partial order reduction techniques can take advantage of these properties of programs. The SPIN model checker contains implementation of partial order reduction techniques. In all fairness, SPIN is designed for checking correctness of programs and not traces. However, to the best of our knowledge it is the best program verification tool we can use for our trace model. The use of computation slicing abstraction technique for temporal logic property verification is the most significant aspect of POTA and constitutes the analyzer module.

To the best of our knowledge, there did not exist tools that implement efficient algorithms (polynomial in the number of processes) to detect predicates that contain typical safety and liveness properties of hardware.

All experiments were performed on a 1.4 Ghz Pentium 4 machine running Linux. We restricted the memory usage to 512MB, but did not set a time limit. The two performance metrics we measured are running time and memory usage. In the case of slicing both metrics also include the overhead of computing the slice. Further experimental results can be obtained from the website [29].

As experimental testbeds, we chose PCI bus and MSI cache coherence protocols, and an SoC based on a PCI backbone. This way we could stretch the verification algorithms by incrementing both the complexity and the number of processes in the systems, which also demonstrates the effectiveness of the algorithms for very large state spaces and complex designs. The protocol IPs are taken from Texas97 benchmarks [2]. We do not consider ISCAS benchmarks because meaningful safety and liveness properties are not available for them.

The MSI (Modified Shared Invalid) cache coherence protocol is a protocol to maintain data consistency among a number of caches connected to a central directory structure in a multi-processor system. The protocol is a directory based scheme in which individual processes snoop on all other processors' activities over a shared directory. The property we checked on the MSI protocol is the safety property, "two caches cannot be in the modified state simultaneously". Symbolically, $\mathsf{EF}(modified_i \wedge modified_j)$, where i and j are cache identifiers. In the experiments, we drastically increased the number of processor caches from 3 to 120. The related time and memory usage is displayed in Figure 7. However, SPIN failed to complete verification for number of caches beyond 10. Our technique successfully and efficiently found bugs in the implementation with very large number of caches connected to the MSI.

The PCI Local Bus is a high performance bus with multiplexed address and data lines. It is used as an interconnect mechanism between peripheral controller components or add-in boards and processor/memory systems. Our example contains an arbiter, and a parameterizable number of nodes that can act either as a master or a target machine.

In the SoC based on the PCI backbone, we have a parameterizable number of devices that are nondeterministically sending requests to the arbiter for accessing the bus as a master and interacting with each other. The safety property that we verified is "there cannot be two masters of the bus". Symbolically, $\mathsf{EF}(master_i \wedge master_j)$, where i and j are device identifiers. The liveness property that we checked

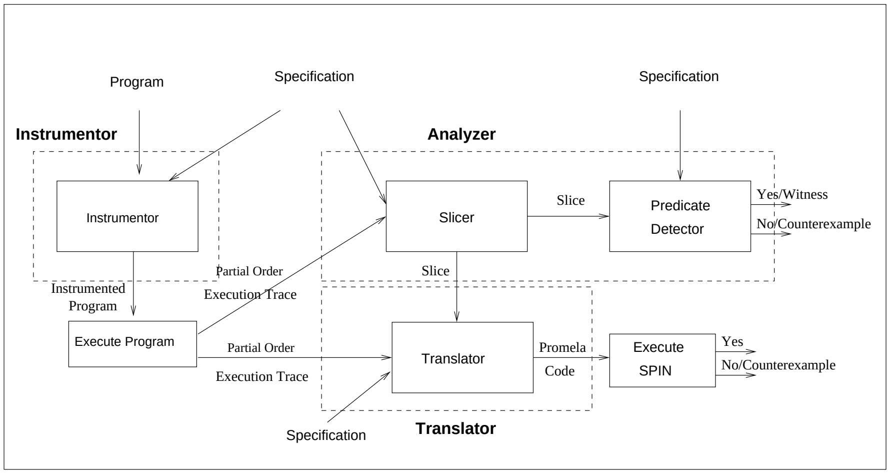

Figure 6: Overview of POTA Architecture

is that "all requests for the bus would be eventually be processed". The negation of the property is $\mathsf{EF}(request_i \wedge \mathsf{EG}(\neg granted_i))$, where i is a device identifier. For this experiment, we have varied the number of nodes from 10 to 100 for the safety property and from 10 to 200 for the liveness property. The reason for the less number of nodes in the safety property experiments is that the property size is larger in this case (that is, every possible combination of nodes i and j exists in the property). The related time and memory usage is displayed in Figure 8 and Figure 9 for safety and liveness properties, respectively. SPIN again failed to complete verification for number of nodes beyond 20 and 14, respectively.

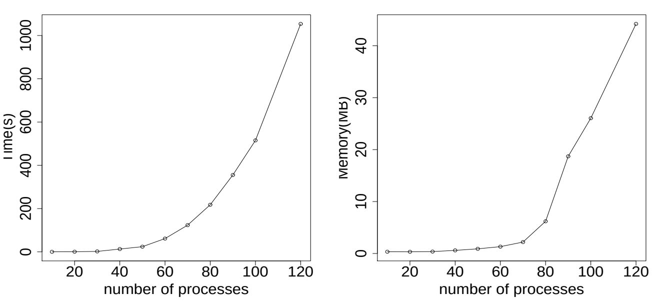

Figure 7: MSI verification results with POTA, SPIN runs out of memory for > 10 processes

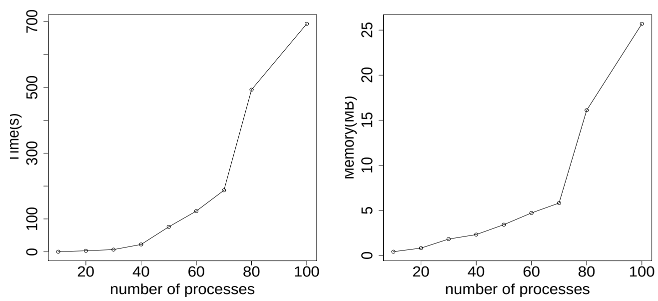

Figure 8: SoC safety verification results with POTA, SPIN runs out of memory for > 20 processes

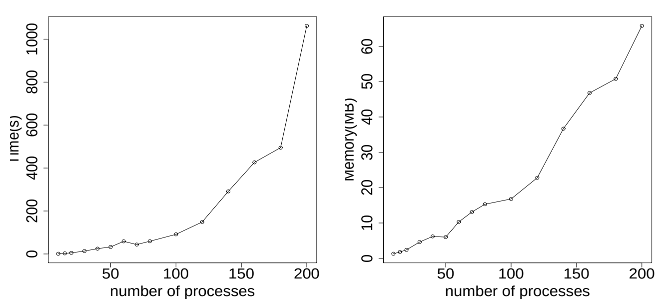

Figure 9: SoC liveness verification results with POTA, SPIN runs out of memory for > 14 processes

9 Conclusions and Future Work

For large and complex designs, temporal logic slicing has proved to be a very effective technique for formal verification. We have demonstrated that it can find bugs, which cannot be found using other techniques. The true power of our technique comes from the exponential reduction of state spaces using polynomial-time slicing abstraction and polynomial-time property detection. Our initial experimental results show a lot of promise of the proposed technique with verification of complex SoCs.

In the future we plan to enhance our technique by (a) exploring coverage metrics for execution traces in order to enhance the effectiveness of our approach, (b) incorporating multiple clock hardware designs, and (c) developing hierarchical slicing abstraction techniques for property checking.

References

[1] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwatrzburd. Verification of the IBM RISC System/6000 by a dynamic pseudorandom test program generator. In *IBM Systems Journal*, volume 30, 1991.

[2] Texas97 Verification Benchmarks. Examples of HW Verification using VIS. 1997. Available from VIS website.

[3] J. Bhadra and N. Krishnamurthy. Automatic Generation of Design Constraints in Verifying High Performance Embedded Dynamic Circuits. In *International Test Conference (ITC)*, 2002.

[4] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979.

[5] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.

[6] E. M. Clarke, M. Fujita, P. S. Rajan, T. Reps, S. Shankar, , and T. Teitelbaum. Program Slicing of Hardware Description Languages . In *Proc. Conf. on Correct Hardware Design and Verif. Method*, 1999.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[8] Y. Deng, S. Kothari, and Y. Namara. Program Slice Browser . In *Proc. of the Intl. Workshop on Program Comprehension*, 2001.

[9] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. In *IEEE Trans. on Software Engineering*, volume 17, 1991.

[10] V. K. Garg. Algorithmic Combinatorics based on Slicing Posets. In *Proceedings of the 22nd Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Kanpur, India, 2002.

[11] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.

[12] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[13] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop Runtime Verification (RV)*, volume 55 of *ENTCS*, 2001.

[14] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[15] S. Ichinose, M. Iwaihara, and H. Yasuura. Program Slicing on VHDL Descriptions and Its Evaluation. In *IEICE Trans. Fund.*, volume E81-A, 1988.

[16] Synopsys Inc. Synopsys VERA Datashee. on the Synopsys website.

[17] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Proceedings of the 1st International Workshop Runtime Verification (RV)*, volume 55 of *ENTCS*, 2001.

[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[19] F. Lanubile and G. Visaggio. Extracting Reusable Functions by Flow Graph-Based Program Slicing. In *IEEE Trans. on Software Engg*, volume 23, 1997.

[20] J. R. Lyle and K. B. Gallagher. A Program Decomposition Scheme with Applications to Software Modification and Testing. In *Proc. of the Hawaii Intl. Conf. on System Sciences*, volume 2, 1989.

[21] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[22] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[23] N.Mittal, A. Sen, V. K. Garg, and R. Atreya. Finding Satisfying Global States: All for One and One for All. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[24] M. H. Nodine, M. Bose, K. G. Davis, and W. R. Jurasz. An effective test generation methodology for System-on-Chips. In *Workshop on Microprocessor Test and Verification (MTV)*, 2002.

[25] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV)*, pages 409–423, Berlin, Heidelberg, 1993.

[26] S. Cox and M. Glasser and W. Grundmann and C. Norris Ip and W. Paulsen and J. L. Pierce and J. Rose and D. Shea and K. Whiting. Creating a C++ library for Transaction-Based Test Bench Authoring. In *Forum in Design Languages*, 2001.

[27] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, April 2002.

[28] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, December 2003.

[29] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA). 2003. http://maple.ece.utexas.edu/~sen/POTA.html.

[30] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proceedings of the 3rd International Workshop Runtime Verification (RV)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, 2003.

[31] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2003.

[32] Forte Design System. QuickBench Verification Suite Overview. on the Forte website.

[33] Verisity Design Systems. Spec-Based Verification - A New Methodology for Functional Verification of Systems/ASICs. Whitepaper on the Verisity website.

[34] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 3rd International Conference on Computer-Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Berlin, Germany, 1991.

[35] V. M. Vedula, J. Bhadra, J. A. Abraham, , and R. Tupuri. A Hierarchical Test Generation Approach Using Program Slicing Techniques on Hardware Description Language. In *Journal of Electronic Testing and Test Automation (JETTA)*, 2003.

[36] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.