

Generator-based Verification

Yunshan Zhu
Advanced Technology Group
Synopsys, Inc.
Mountain View, CA
yunshan@synopsys.com

James H. Kukula
Advanced Technology Group
Synopsys, Inc.
Hillsboro, OR
kukula@synopsys.com

ABSTRACT

To prove system correctness, assumptions made in verifying a block must be cleared by verifying that the block's environment guarantees them. Conversely, guarantees enforced by a block may be used as assumptions for its environment. Block level interface specifications thus serve as both assumptions and guarantees in compositional verification. Traditionally, such specifications have been represented as monitors or checkers. In this paper, we propose an alternative representation using generators. Novel algorithms are presented for simulation and formal verification. We argue that for simulation, representation as a generator can be more efficient than as a checker – both asymptotically and practically. We also identify a subset of generators that can be efficiently handled using formal techniques. Experimental results are given for some benchmark examples and industrial case studies.

1. INTRODUCTION

Verifying a design generally involves deciding whether the design will guarantee that its outputs satisfy some set of properties, under the assumption that its inputs satisfy some other properties. The properties the design is to guarantee are often expressed as checkers, which “observe” its input/output behaviors and either “accept” or “reject” the behaviors. Checkers are also referred to as monitors or assertions. The input behavior that the design can assume is often expressed as a generator, which “produces” legal inputs to the design based on the design's outputs and internal generator states. The structure of the generator resembles a hardware design block. The generator block provides a general model for the legal environments in which the design is intended to operate.

While checkers and generators are used in different roles in design verification, they share an underlying semantics which defines a set of input/output traces [3]. A given set of traces can be expressed either as a generator or as a checker. Deciding which representation to use involves considering two criteria: ease of expression, and tool support for a particular form of representation (checker or generator) in a particular verification role (assumption or guarantee). Neither form of representation is universally easier for expressing traces. For instance, the correctness of a sorting circuit involves two conditions: the values in the output array should vary monotonically with the array indices, and should be a permutation of the sorter's input array. Monotonicity is more easily expressed in the checker representation, while it is easier to use a generator to define the set of input permutations that must contain the sorter's output. There does not seem to be any general rule for predicting which traces will be more easily expressed by which representation. Tool support for the various combinations of role

and representation is needed.

Compositional verification is a methodology where verification of a complete system is performed by separately verifying each of the blocks from which the system is constructed. These smaller verification sub-tasks can be much more effective with both simulation and formal tools, mainly because the exposed block interfaces give better controllability and observability. In compositional verification, one block's guarantee is used as a connected block's assumption, and vice versa. Consequently, over the set of block verifications involved in the compositional verification of a system, a single trace set representation will need to be used in both roles, as an assumption and as a guarantee. Thus the choice of representation for a trace set cannot be governed by tool support for the particular role that trace set will play in any one block's verification, because that trace set will serve the complementary role in some other block's verification.

We will consider support for checker and generator representations by two classes of verification tools, simulation and formal tools. Tables 1 and 2 summarize the approaches used by these tools to support these representations in the two different roles. In both classes of tools, the generator representation is most natural in the role of block assumption, and the checker representation in the guarantee role. Checkers are also easily incorporated as assumptions in formal verification, but to use a checker as an assumption for simulation is non-trivial. The problem, referred to as stimulus generation, has been studied in several recent works [13, 4, 9].

To use a generator as a guarantee is also non-trivial and is the focus of this paper. We study relevant algorithms for both simulation and formal verification. To simulate against a guarantee represented as a generator, one must test if the trace of a simulation run can be produced by the generator. We refer to the verification problem as trace checking. To develop efficient algorithms for trace checking, we classify generators into three groups: structurally sequentially deterministic (SSD), functionally sequentially deterministic (FSD) and sequentially non-deterministic (SND). More details on the generator representation will be presented in Section 2.

In Section 3, we present algorithms for trace checking these classes of generators. Trace checking the general SND class can be quite complex. However, by comparing our algorithms with those for stimulus generation, we show that trace checking of SSD generators can be both asymptotically and practically more efficient than stimulus generation.

Formal methods can be used to verify that every trace emitted by a design is in the set of traces defined by a generator, i.e. that the design guarantees the property represented by the generator. This comparison of a design against a generator is known as a refinement checking problem. In the general case where a guarantee generator

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICCAD '03, November 11-13, 2003, San Jose, California, USA.

Copyright 2003 ACM 1-58113-762-1/03/0011 ...\$5.00.

	assumption	guarantee
checker	stimulus generation	assertion
generator	testbench	trace checking

Table 1: Roles and Representations in Simulation

	assumption	guarantee
checker	constraint	assertion
generator	testbench	refinement checking

Table 2: Roles and Representations in Formal Verification

is sequentially non-deterministic, refinement checking is PSPACE complete with respect to the number of generator states [11]. Existing approaches of refinement checking require manual construction of refinement maps or witness functions [7, 6, 1]. In Section 4, we present an algorithms for refinement checking where the guarantee generator is restricted to be sequentially deterministic.

Section 5 contains experimental results of trace checking and refinement checking on a number of benchmark examples as well as industrial designs. We conclude with a summary of contributions in section 6.

2. REPRESENTATION

2.1 Definitions

DEFINITION 2.1. Let V be a finite set of boolean variables $\{v_1, v_2, \dots, v_n\}$. A trace over V is an infinite sequence $\pi = \alpha_0 \alpha_1 \dots$, where each α_i is a complete assignment of variables in V at time i . For $D \subseteq V$, $\alpha_{\downarrow D}$ denotes the projection of assignment α to the subset of variables in D . We refer to V as the type of trace π .

We define checkers and generators as special types of automata as follows.

DEFINITION 2.2. A checker is a Mealy machine defined by the tuple $\tilde{M} = \langle S, s_0, I, v, f_v, f_s \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, $I = D_I \cup D_O$ is a finite set of input variables. D_I denotes design inputs and D_O denotes design outputs. $D_I \cap D_O = \emptyset$. v is an output variable. $f_v : S \times \Sigma_I \rightarrow \{0, 1\}$ is the output function, where Σ_I represents the set of all possible assignments of I . $f_s : S \times \Sigma_I \rightarrow S$ is the next state function. Both f_v and f_s are total functions.

When composing a guarantee checker with a DUT, D_I is connected to the corresponding DUT input signals and D_O to the corresponding DUT output signals. Both D_I and D_O are inputs to the checker. All inputs of a checker are connected to their corresponding design signals in a proper composition. By our definition, a checker is deterministic, i.e., there is a single next state and output for every current state and input combination.

In [10], an additional restriction is imposed on checkers. In particular, the valid signal may only depend on variables in $S \cup D_O$, but not in D_I . $f_v : S \times \Sigma_{D_O} \rightarrow \{0, 1\}$. The restriction was referred to as output separability. Such restricted checkers are sufficient to specify behaviors that can be modeled as Moore machines.

DEFINITION 2.3. Assume $\pi = \alpha_0 \alpha_1 \dots$ is an infinite trace of type $D_I \cup D_O$. π is accepted by $\tilde{M}$, or $\pi \models \tilde{M}$, iff there exists an infinite sequence of states $S = s_0 s_1 \dots$ such that (1) s_0 is the initial state; (2) $\forall i \ s_{i+1} = f_s(s_i, \alpha_i)$; (3) $\forall i \ f_v(s_i, \alpha_i) = 1$. We interpret

a checker as the set of all traces $I(\tilde{M})$ accepted by $\tilde{M}$, formally, $I(\tilde{M}) = \{\pi \mid \pi \models \tilde{M}\}$.

DEFINITION 2.4. A generator is a Mealy machine defined by the tuple $\tilde{M} = \langle S, s_0, I, O, f_O, f_S \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, $I = D_I \cup P$ is a finite set of input variables. D_I denotes the set of design inputs, and P denotes the set of parametric inputs. $O = D_O$ is a finite set of output variables. D_O denotes the set of design outputs. $I \cap O = \emptyset$. $f_O : S \times \Sigma_I \rightarrow \Sigma_O$ is the output function, where Σ_I and Σ_O represents the set of all possible assignments of I and O , respectively. $f_S : S \times \Sigma_I \rightarrow S$ is the next state function. Both f_O and f_S are total functions.

Parametric input variables P model non-determinism in a generator.

DEFINITION 2.5. Assume $\pi = \alpha_0 \alpha_1 \dots$ is an infinite trace of type $D_I \cup O$. π is generated by $\tilde{M}$, or $\pi \models \tilde{M}$, iff there exists an infinite sequence of states $S = s_0 s_1 \dots$ and an infinite trace $\sigma = \beta_0 \beta_1 \dots$ of type P , such that (1) s_0 is the initial state; (2) $\forall i \ s_{i+1} = f_S(s_i, \alpha_i \downarrow_{D_I}, \beta_i)$; (3) $\forall i \ \alpha_i \downarrow_{D_O} = f_O(s_i, \alpha_i \downarrow_{D_I}, \beta_i)$; We interpret a generator as the set of all traces generated by $\tilde{M}$, formally, $I(\tilde{M}) = \{\pi \mid \pi \models \tilde{M}\}$.

2.2 Special Cases

DEFINITION 2.6. A generator $\tilde{M}$ is sequentially deterministic if there is a single next state for each assignment of S , D_I and D_O , i.e.

$$\begin{aligned} f_O(s, d_I, p) &= f_O(s, d_I, p') \\ \rightarrow f_S(s, d_I, p) &= f_S(s, d_I, p') \end{aligned} \quad (1)$$

where s , d_I and d_O denote vectors of state variables, design input variables, and design output variables, respectively.

A checker, by our definition, is sequentially deterministic. Its next state is determined by the assignment of variables in S , D_I and D_O . The difference between a checker and a sequentially deterministic generator lies in their combinational representations of design outputs. A checker uses a relation to represent design outputs, i.e. $R(s, d_I, d_O) = 1$. A generator uses a function with parametric inputs, i.e. $d_O = f(d_I, s, p)$. An equivalent relation can be obtain by quantifying the parametric inputs, i.e. $\exists p. (d_O = f(d_I, s, p))$. Each representation can be more suitable in certain situations. The set of variable assignments d that satisfies a boolean formula B can be expressed a relation $B(d) = 1$. On the other hand, the set of composite natural numbers is more easily expressed as a function with parametric inputs $d = x \times y$.

2.3 Methodology

Figure 1 shows a simple system with two functional blocks, A and B. The verification task is to check that when the inputs to block A are driven by the outputs of block B, the outputs of A satisfy property Q. The simplest way to attack this problem is simply to connect B to A and search for an input sequence to B that causes the outputs of A to violate Q. A variety of methods can be used for this search, including simulation of random vectors or manually constructed sequences, or formal methods such as model checking. In any of these techniques, the size of the system can become an impediment to the effectiveness of the search. It can be very difficult to find inputs to B that cause it to emit sequences that will exercise A thoroughly and expose any possible violations of Q.

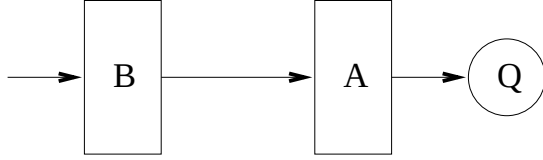


Figure 1: A simple system to be verified

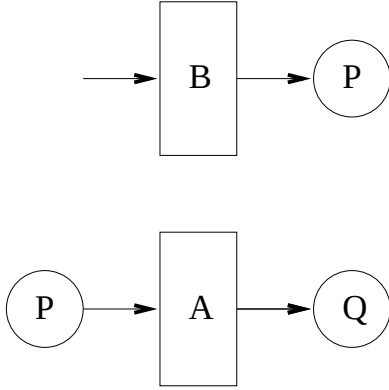


Figure 2: Compositional verification

Compositional verification enables much more effective searching for erroneous behaviors. To enable compositional verification, the system designer must provide additional block specifications in a form that verification tools can interpret. In Figure 2, property P has been introduced to specify the legal communication between blocks A and B . Block A is required to guarantee property A , under the assumption that its inputs obey property P . Block B is then required to guarantee property P . The system verification problem, to check that A and B connected will together guarantee Q , has thus been broken into two simpler block verification problems. This becomes a simple example of assume-guarantee reasoning[8, 7]: if A can guarantee Q assuming P , and B can guarantee P , then one can conclude that the system consisting of A and B will guarantee Q . Note that the converse is not true. It is possible to err in expressing P so that the entire system does guarantee Q , yet one or both of the block verifications fails. Compositional verification requires accurate block specification. What this buys is dramatically more effective block level searches for design errors.

In section 2.1 we defined two forms of specification, checkers and generators. Consider property P . In performing compositional verification of our simple example, one has a choice to express P in the form of a checker or in the form of a generator. Let us examine the consequences of this decision.

If P is expressed as a checker, the block verification of B becomes possible through conventional simulation methods. The sequence of outputs of B are simply fed into the checker inputs and the checker output is observed. If the output is ever observed to have a 0 value, a property violation will have been detected. Simulation of block A is a bit more challenging, though. At each simulation cycle, an input vector must be constructed for which the checker for P will output a 1 value. P may well have changing internal state, so this input vector construction problem is changing cycle by cycle. Previous works in the literature[13, 4, 9] have studied techniques to efficiently solve this stimulus generation problem.

If P is instead expressed as a generator, simulation of block A

becomes very straightforward. One simply drives the parametric inputs of the generator for P with directed or random stimulus, and observes the output of a checker for property Q . In this case, it is the simulation of block B that becomes more difficult. To verify the block simulation, one must check if the design output can be matched by the generator output. The difficulty here is to find out parametric input assignments that produces a matching generator output. The algorithms we develop in the next section address this problem of block verification when the property the block is required to guarantee is expressed as a generator. We will consider both simulation and formal verification techniques. Simulation generates a single trace, which then can be compared against the set of traces of the generator. In formal verification the block itself is analyzed to check the complete set of output traces it can emit.

With these trace checking and refinement checking algorithms, tool support becomes available for every combination of role and representation. Thus the decision of which representation to use need not be constrained by tool support limitations. Ease of expression for a particular trace set remains a criterion. Efficient tool processing is also a criterion. A checker or generator is often written in RTL Verilog or some assertion language. Using a checker as a guarantee and a generator as an assumption can take advantage of RTL simulation, and is therefore very efficient. Both stimulus generation and trace checking typically involve formal reasoning and are often slower. To use a checker as an assumption, a simulator must interact with stimulus generation each cycle to obtain legal input stimulus. It is observed that simulation generation time represents a significant portion of total simulation time, sometimes more than 50 percent. A generator as guarantee, however, can be used offline – a simulator can save the simulation trace in a dump file (e.g. VCD) to be checked later against the generator. This does not affect overall cpu time, but unburdening simulation lets it more quickly hit other bugs and free limited resources such as licenses and memory. Thus it seems from a tool efficiency perspective, the generator representation has an overall efficiency advantage.

3. TRACE CHECKING

The trace checking problem is defined as follows. Given a trace $\pi = \alpha_0\alpha_1 \dots$ and a generator $\vec{M}$, check whether π can be generated by $\vec{M}$, or mathematically, $\pi \models \vec{M}$.

Trace checking problems arise in practice when a generator is used as a guarantee in simulation. To be compatible with well-established theories on temporal logic [5], we have defined traces to be infinite. Any simulation runs, however long, must terminate and therefore will produce only finite traces. In this paper, we focus on safety properties, for which a simple solution exists. A finite trace satisfies a generator iff it is a prefix of a satisfying infinite trace. The algorithms presented in this section will decide whether the finite prefix can indeed be generated. If it can, the existence of an infinite extension is trivial – just simulate the generator with arbitrary input combinations and use the corresponding input/output assignments to extend the finite trace. To check the acceptance of a finite trace by a checker is more complex and the absence of dead-end states needs to be verified. Detailed discussion on this subject is out of the scope of this paper.

3.1 Sequentially Deterministic

In this subsection, we present trace checking algorithms for sequentially deterministic generators. We further divide the class into structurally sequentially deterministic (SSD) generators and func-

```

procedure trace-check-ssd( $\vec{M}$ ,  $\pi$ ) {
   $R \leftarrow \exists p (d_O = f_O(s, d_I, p))$ ;
   $state \leftarrow s_0$ ;
  for ( $i = 0$ ;  $i < |\pi|$ ;  $i++$ ) {
     $v_I \leftarrow \alpha_{i \downarrow D_I}$ ;
     $v_O \leftarrow \alpha_{i \downarrow D_O}$ ;
    if ( $R(state, v_I, v_O) = \emptyset$ )
      return fail;
     $state \leftarrow \hat{f}_S(state, v_I, v_O)$ ;
  }
  return pass;
}

```

Figure 3: Trace Checking – SSD

tionally sequentially deterministic (FSD) generators.

3.1.1 SSD

For SSD generators, the next state function f_S can be expressed as a composition of a function $\hat{f}_S : S \times \Sigma_{D_I} \times \Sigma_O \rightarrow S$ and $f_O : S \times \Sigma_{D_I} \times \Sigma_P \rightarrow \Sigma_O$. Namely,

$$f_S(s, d_I, p) = \hat{f}_S(s, d_I, f_O(s, d_I, p))$$

Such decomposition, if it exists, significantly simplifies the trace checking problem. In particular, $\hat{f}_S$ can be used to efficiently compute the next state of an SSD generator. A trace checking algorithm is showed in Figure 3. $|\pi|$ denotes the length of input trace π ; v_I and v_O denote assignments of design input variables (d_I) and design output variables (d_O), respectively. The relation R between s , d_I and d_O is represented as a BDD [2].

Figure 4 shows an example of SSD generator in Verilog. In an SSD generator, next state functions do not directly depend on parametric inputs. Design inputs, design outputs and current states should form a cut in the fanin logic of every next state assignment. The cut can be computed by syntactical analysis of an SSD generator. Figure 4 is a special case where next state ($state$) depends only on design inputs (di).

3.1.2 Complexity Analysis

We now analyze the complexity of the SSD trace checking algorithm. There is a one-time cost to compute BDD R by quantifying parametric inputs p . For each simulation cycle, there is a cost to verify the correctness of design outputs and compute the next state of the generator. Design outputs are verified by evaluating $R(state, v_I, v_O)$ and checking for its emptiness. The cost of the operation is $O(|S| + |D_I| + |D_O|)$, where $|S|$, $|D_I|$ and $|D_O|$ denote the numbers of state variables, design inputs and design outputs, respectively. Note that the complexity is independent of the size of BDD R , which in the worst case is exponential in the total number of variables. The next state is computed by evaluating $\hat{f}_S(state, v_I, v_O)$. The cost is linear in the size of generator representation. In practice, if a generator is written in RTL, its next state can be computed using RTL simulation.

Figure 5 outlines an algorithm of stimulus generation for comparison. Let us suppose that a checker $\vec{M}$ and a generator $\vec{M}$ specify a set of traces using the same state encoding. In this case, the two BDDs (R) of stimulus generation and trace checking are equal. In each simulation cycle, stimulus generation obtains an assignment v_I of design input signals d_I from a simulator. The cofactor operation to compute the set R' of legal assignments of design outputs

```

module oneBitError(clk, rst, di, nd, do);
input clk, rst;
input [7:0] di;
input [2:0] nd;
output [7:0] do;
reg [7:0] state;
reg [7:0] do;

always @(posedge clk or posedge rst) begin
  if (rst)
    state <= 0;
  else
    state <= di;
end

always @(state or nd) begin
  do = state;
  do[nd] = ~do[nd];
end
endmodule

```

Figure 4: An SSD generator example

d_O is $O(|R|)$. If all variables d_O are ordered at the bottom of BDD R , the cofactor R' can be computed in $O(|S| + |D_I|)$. Such variable orderings, however, often lead to BDD size explosion. Interleaving bits of a data path is usually required for a compact BDD representation.

To pick an arbitrary assignment v_O from R' is $O(|D_O|)$. It is desirable in practice to generate v_O with certain probability distributions. One can bias individual bits of d_O by traversing the BDD R' and assigning weights to its BDD nodes. The time complexity of this algorithm is $O(|R'|)$. This algorithm assumes independence among individual bits and cannot achieve all possible word level distributions. What we have described is essentially the SimGen algorithm[13], having the same complexity. SimGen achieves more practical speed by avoiding BDD operations, operating instead directly on the fixed BDD R .

We now compare the costs of trace checking and stimulus generation. In the both approaches, there is one-time cost to compute BDD R and the memory usage is dominated by computing BDD R and storing R during simulation. The difference is in simulation time cost. For trace checking, per cycle cost is $O(|S| + |D_I| + |D_O|)$; For stimulus generation, the cost is $O(|R|)$. The size of BDD R is in the worst case exponential in the number of variables. Therefore, per cycle time cost of stimulus generation can be much higher than that of trace checking. It is important to minimize per cycle simulation overhead, as a thorough simulation can require thousands or even millions of cycles.

3.1.3 FSD

In some cases, a generator obeys equation (1), but its next state function cannot be expressed as a composition $f_S = \hat{f}_S \circ f_O$. We call such generators functionally sequentially deterministic (FSD). The condition defined by Equation (1) can be verified by evaluating the expression using BDD operations. The evaluation involves building BDDs for the next state function and can be expensive. Alternatively, the condition can be checked dynamically as a part of trace checking. An algorithm for FSD trace checking is showed in Figure 6. The cofactor operation returns a set of parametric in-

```

procedure stimulus-generation( $\vec{M}$ ) {
   $R \leftarrow (f_v(s, d_I, d_O) = 1)$ ;
   $state \leftarrow s_0$ ;
  for ( $i = 0$ ;  $i < \text{SimCycles}$ ;  $i++$ ) {
    read( $v_I$ );
     $R'(v_O) \leftarrow \text{cofactor}(R, \{v_I, state\})$ ;
     $v_O \leftarrow \text{random-pick}(R')$ ;
     $state \leftarrow f_S(v_I, v_O, state)$ ;
    write( $v_O$ );
  }
}

```

Figure 5: Stimulus Generation

```

procedure trace-check-fsd( $\vec{M}, \pi$ ) {
   $R \leftarrow (d_O = f_o(s, d_I, p))$ ;
   $state \leftarrow s_0$ ;
  for ( $i = 0$ ;  $i < |\pi|$ ;  $i++$ ) {
     $v_I \leftarrow \alpha_{i \downarrow D_I}$ ;
     $v_O \leftarrow \alpha_{i \downarrow D_O}$ ;
     $P \leftarrow \text{cofactor}(R, \{state, v_I, v_O\})$ ;
    if ( $P = \emptyset$ )
      return fail;
     $state \leftarrow f_S(state, v_I, p)$ ; // pick any  $p \in P$ 
    if (CHECK-FSD) {
       $S \leftarrow \text{image}(state, v_I, P)$ ;
      if ( $|S| > 1$ )
        return notFSD;
    }
  }
  return pass;
}

```

Figure 6: Trace Checking – FSD

puts assignments P that can produce generator outputs v_O . With the CHECK-FSD option turned on, the algorithm computes the set of next states by image computation while constraining design inputs to be v_I and parametric inputs to be in P , and it checks that the set is a singleton, i.e. all parametric inputs in P lead to a single next state. In fact, the FSD algorithm is sound with a somewhat more general condition – Equation (1) only has to hold for the specified design inputs from a trace and reached states.

3.1.4 Optimizations

So far, we have proposed to use BDDs to implement trace checking algorithms. In particular, relation R is represented as a BDD with s , d_I and d_O as variables. The advantage of this static approach is that per cycle cost of trace checking is minimized. In some cases, however, computing the BDD may be too expensive in either time or memory usage. Let us re-examine the algorithm in Figure 3. The essential operation on R is an emptiness check after substitution of state, design input and output variables. Rather than building a BDD for R , we can represent R as a boolean expression. At each cycle, we substitute the variables in R with their assigned values and obtain a much simplified expression. Now there are two choices. We can either build a BDD for the simplified expression and test for its emptiness or use a SAT solver to test the unsatisfiability of the expression. Both operations are much cheaper than

```

procedure trace-check-snd( $\vec{M}, \pi$ ) {
   $S \leftarrow \{s_0\}$ ;
  for ( $i = 0$ ;  $i < |\pi|$ ;  $i++$ ) {
     $v_I \leftarrow \alpha_{i \downarrow D_I}$ ;
     $v_O \leftarrow \alpha_{i \downarrow D_O}$ ;
    // image computation
     $S' \leftarrow \exists s \exists p (s' = f_S(s, v_I, p) \wedge s \in S \wedge v_O = f_O(s, v_I, p))$ ;
    if ( $S' = \emptyset$ )
      return fail;
    else
       $S \leftarrow S'$ ;
  }
  return pass;
}

```

Figure 7: Trace Checking – SND

computing a BDD for R . Unfortunately, the operations must be repeated for each cycle of the simulation. This analysis also applies to the FSD algorithm in Figure 6 if CHECK-FSD option is disabled.

3.2 Sequentially Non-Deterministic

A generator is sequentially non-deterministic (SND) if legal parametric inputs can lead to distinct next states, each of which may later *pass* or *fail* the trace under test. In this case, a set of “accepting” generator states must be maintained for each simulation cycle, and trace checking is essentially reachability analysis with input/output constraints. Figure 7 shows an algorithm for this general case of generators.

4. REFINEMENT CHECKING

We consider formal refinement checking as a comparison of two Mealy machines, a module machine $\vec{M}^m = \langle S^m, s_0^m, D_I, O, f_O^m, f_S^m \rangle$, and a generator machine $\vec{M}^g = \langle S^g, s_0^g, I, O, f_O^g, f_S^g \rangle$, where $I = D_I \cup P$, i.e. the generator has some extra parametric inputs to model non-determinism. The general refinement checking problem is to verify whether every input/output trace of the module can be observed in the generator (after projecting out the parametric inputs). The special case of sequentially deterministic generators permits a more efficient solution than the general case. In this special case we can explore the state space of the product machine, considering only transitions for which the output of the generator matches the output of the module, i.e. we can form the constrained product machine transition relation

$$\begin{aligned}
T(s_p^m, s_p^g, s_n^m, s_n^g) = & \exists p \exists d_I [(f_O^m(s_p^m, d_I) = f_O^g(s_p^g, d_I, p)) \\
& \wedge (s_n^m = f_S^m(s_p^m, d_I)) \\
& \wedge (s_n^g = f_S^g(s_p^g, d_I, p))]
\end{aligned} \tag{2}$$

At each state of the product machine reachable by way of these transitions, there must be a generator output corresponding to every module output, i.e. the product state must satisfy

$$v(s^m, s^g) = \forall d_I \exists p [f_O^m(s^m, d_I) = f_O^g(s^g, d_I, p)] \tag{3}$$

Thus the special case of refinement checking for sequentially deterministic generators can be reduced to a classical model checking problem of a safety property, checking that the initial state (s_0^m, s_0^g) satisfies the CTL property $\text{AG}v$ in the transition system defined by T .

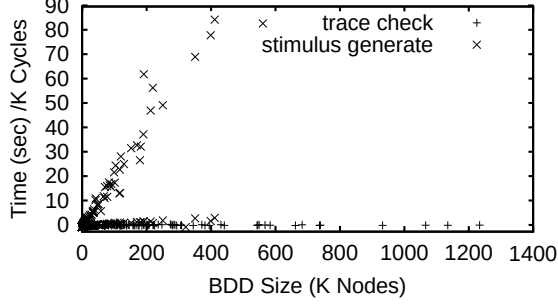


Figure 8: Per Cycle Cost Comparison

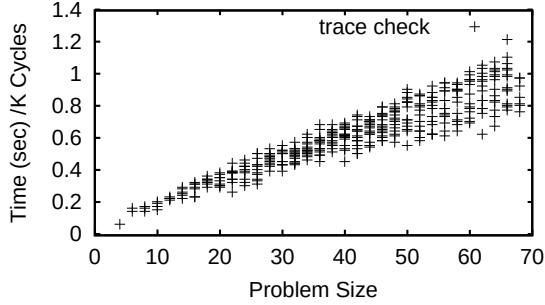


Figure 9: Trace Checking Cost

This approach to refinement checking actually handles a somewhat broader class of generators than just sequentially deterministic. In a sequentially deterministic generator there will be only one generator next state s_n^g corresponding to a product machine present state (s_p^m, s_p^g) and design input d_i . But our refinement check will allow several corresponding next states, as long as each of them satisfies AGv. While we can thus verify refinement in a somewhat broader class of generators than the sequentially deterministic class, this is still much narrower than refinement in general, where an arbitrary non-empty subset of corresponding generator next states needs to exist that can carry forward the correspondence.

5. EXPERIMENTS

We report here some experimental results on our implementation of trace checking and refinement checking. A SUN Ultra-Enterprise workstation with 336 MHz CPUs and 7.2 GB memory is used to for the experiments. Each job is executed on a single processor and the CPU time is reported.

5.1 Examples

Due to the absence of standard benchmarks for trace checking, we built some examples for our experimental study. Some interesting results are highlighted here.

Our first set of examples is based on multipliers. In particular, we specify the relation $c = a \times b$ as both a checker and a generator. Design input D_I contains the binary vector a , and design output D_O contains the binary vectors b and c . The widths of the vectors are varied for the experiments. Results are collected for trace checking and stimulus generation. In both cases, BDDs for R is computed prior to simulation runs. The data for stimulus generation is based on our implementation of the techniques in [13].

Size	SEQs	InitT sec	InitM MB	BDD Nodes	SSD sec
10	49	1.81	12.23	2684	0.91
20	90	15.95	15.42	9716	1.25
30	130	73.56	18.75	14555	1.80
40	171	25.46	18.72	10982	2.14
50	211	196.04	31.57	31088	2.49
60	251	566.27	34.34	97971	3.03
70	292	393.02	28.82	46476	3.58
80	332	3520.81	62.67	79351	4.25
81	336	2078.54	64.50	428190	4.23
82	340	>5000	N/A	N/A	N/A

Table 3: Permutation

Size	SEQs	InitT sec	InitM MB	BDD Nodes	SSD sec
50	260	0.51	12.12	305	2.49
100	510	6.57	14.33	545	4.61
150	760	21.89	25.16	833	7.04
200	1010	58.41	43.88	5333	9.29
250	1260	245.05	51.30	8471	11.49
300	1510	189.56	50.44	10253	14.52

Table 4: Pipe

Figure 8 compares per cycle cost of stimulus generation and trace checking. Note that per cycle cost of stimulus generation increases proportionally to the size of the BDD R . Figure 9 shows that per cycle cost of trace checking is linear in terms of the problem size, which is defined as the total width of a, b and c . It is well known that BDDs for multiplication are exponential in the width of the operands.

In the second set of examples, generators are used to define permutation. A generator collects input data for n cycles, and in the next n cycles, outputs a permutation of the input data. All permutations can be generated by controlling the parametric inputs. Input data is 4 bits wide. Results for different problem size n are showed in Table 3. We found it easier to express permutation as a generator than as a checker. An arbitrary permutation can be decomposed into a series of bi-permutation. The parametric inputs of the generator specify the bi-permutations.

The third set of examples extend Figure 4 to model 3-clock delayed pipes with single bit errors. Results are listed in Table 4 for pipes with different widths.

In Tables 3 and 4, Column 1 lists the problem size. Column 2 lists the number of sequential elements (SEQs) or state variables. Column 3 (InitT) and 4 (InitM) show time and memory usage for computing BDD R , respectively; BDD variable reordering is used. Column 5 lists the number of BDD nodes; Column 6 lists trace checking time using SSD algorithm; SSD memory usage is dominated by BDD R computation. Random traces are generated by simulating a generator with random inputs and removing the parametric variables. Trace checking time is collected for 1000 cycles.

In all three set of examples, SSD trace checking time grows linearly with the problem size and is independent of the BDD size. Note the variation in BDD sizes in Table 3 caused by BDD variable reordering. We also tested the SND trace checking algorithm from Figure 7. For the permutation examples, it takes 519.4 seconds to complete trace checking of problem size 7 and cannot finish problem size 8 within 5000 seconds. For the pipe examples, SND

algorithm takes 1007.7 seconds to complete size 50 and 3960 seconds to complete size 90. SND algorithms is slow because image computation occurs at every cycle of simulation.

5.2 Case Study

We verify an implementation of PCI-Express endpoint unit using compositional reasoning. Interfaces between various blocks are specified. In most cases, specifications can be represented as both checkers and SSD generators. We use a generator to specify the interface between Link Transmit (LT) and Physical Transmit (PT). The generator defines legal packets that can be transferred from LT to PT. The generator has five flip-flops for state encoding and five bits each for input and output signals. The same specification can also be represented as a checker. We choose a generator because it gives us better control of input stimulus when used as an assumption to simulate the PT module. To ensure system correctness, we need to make sure that the specification covers all packets that LT can send out. In other words, the generator must be verified as a guarantee of the LT block. We apply SSD trace checking to simulate the LT block against the generator. The LT fanin cone has 258 flip-flops. Ten thousand cycles of simulation took 8.05 seconds, with 2.01 seconds spent on trace checking.

5.3 Formal Checking

As discussed in section 4, refinement checking for sequentially deterministic generators reduces to a safety property check. A wide variety of algorithms have been explored for such checks. We used the abstraction refinement approach of Wang et al. [12]. Our implementation of the algorithm requires the transition system and property to be provided in the form of an RTL circuit. A BDD can be computed for property v of equation (3). A circuit for v can then be synthesized from this BDD, in the form of a multiplexor network.

Building a circuit for the transition system T of equation (2) is a bit more challenging. What is required is that the parametric inputs p of the generator be constrained so that the module and generator outputs correspond at every cycle. In a simulation-based methodology, this would form a classical constrained stimulus generation problem. For our formal refinement checking implementation we used a simpler method. We added an auxiliary state bit to the product machine, initially assigned to 1 and remaining 1 just as long as the input constraint is satisfied. This bit is then used to qualify the predicate v , so a property violating state is one where $v = 0$, reached by a path on which the module and generator outputs have corresponded at all previous time steps.

We compared the PCI-Express LT block to the generator we wrote to define the legal outputs of that block. This refinement check completed in 7.1 cpu seconds.

For a more complex refinement checking problem, we construct a parameterized tag allocator block. Its inputs are an n -bit tag word together with a request type $\in \{ \text{ALLOC}, \text{FREE}, \text{NONE} \}$. The outputs are an n -bit tag word and a validity indicator. Each of the 2^n tag values has an evolving status, either free or in use. When the tag allocator receives an ALLOC request, it emits one of the currently free tag values and changes the status of that tag to in use. If no free tag is available, an invalid request is indicated. When a FREE request is received, the tag value received is reverted to free status. If that tag value was already free, an invalid request is indicated.

We built two versions of this tag allocator. The deterministic version keeps the free tag values on a stack, and always returns the top value in response to an ALLOC request. The non-deterministic

version can choose any of the currently free tag values to return. The non-deterministic version was constructed to be sequentially deterministic, so our refinement checker could be used to verify that all the traces of the deterministic block are included in those of the non-deterministic block. For $n = 2$ this check was performed in 308 cpu seconds. This check requires analysis of the complete product machine - any simple abstraction will introduce false negatives. Since the sizes of the blocks grow exponentially with n , the analysis time grows very quickly with n . The $n = 3$ instance takes over 24 hours of cpu time to complete.

6. CONCLUSION

We have introduced here the concept of sequentially deterministic generators. This subset of generators allow efficient trace checking and refinement checking algorithms. With these new algorithms, a generator can now be used as either an assumption or a guarantee in block verifications. In a compositional verification methodology, such dual usage of a representation is required. By analyzing trace checking and stimulus generation algorithms, we showed that, for simulation, a generator representation has an overall efficiency advantage over a checker representation. The analysis is supported by our experimental results.

Future work remains in both methodology studies and algorithm development. A thorough comparison on the ease of expression between a generator and a checker representation would be extremely valuable. Further optimizations of both trace checking and refinement checking algorithms need to be investigated.

7. REFERENCES

- [1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.
- [2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [3] D. Dill. *Trace Theory For Automatic Hierarchical Verification Of Speed-independent Circuits*. ACN Distinguished Dissertations. MIT Press, Cambridge MA, 1989.
- [4] J. H. Kukula and T. R. Shiple. Building circuits from relations. In *Proc. of the Computer Aided Verification Conf.*, pages 112–123, July 2000.
- [5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [6] K. McMillan. <http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/>.
- [7] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. 9th International Computer Aided Verification Conference*, pages 24–35, 1997.
- [8] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and systems*, 16(3):843–871, may 1994.
- [9] K. Shimizu and D. Dill. Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification. In *Proc. of the Design Automation Conf.*, pages 801–806, June 2002.
- [10] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353, 2000.

- [11] L. J. StockMeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. of the 5th Annual Symposium on Theory of Computing*, pages 1–9. ACM Press, 1973.
- [12] D. Wang, P.-H. Ho, J. Long, J. H. Kukula, Y. Zhu, H.-K. T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Design Automation Conference*, pages 35–40, 2001.
- [13] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 584–589, Nov. 1999.