Cycle-based Symbolic Simulation of Gate Level Synchronous Circuits

Valeria Bertacco †　　Maurizio Damiani　　Stefano Quer *

Advanced Technology Group
Synopsys, Inc.
Mountain View, CA 94043

†Systems Science Group
Synopsys, Inc.
Palo Alto, CA 94035

Abstract

Symbolic methods are often considered the state-of-the-art technique for validating digital circuits. However, due to their complexity and unpredictable run-time behavior their potential is currently limited to medium–small circuits. Logic simulation, on the other hand, privileges capacity, it is nicely scalable, flexible, and has a predictable run-time behavior. Nevertheless, it typically visits only a small fraction of the state space: The discovery of bugs heavily relies on the expertise of the designer of the test stimuli.

In this paper we consider a symbolic simulation approach to the validation problem. Our objective is to trade-off between formal and numerical methods in order to simulate a circuit with a "very large number" of input combinations and sequences in parallel. We demonstrate larger capacity with respect to symbolic techniques and better efficiency with respect to cycle-based simulation. We show that it is possible to symbolically simulate very large trace sets in parallel (up to 80 symbolic inputs) for the largest ISCAS benchmark circuits, using only 96 *Mbyte of main memory.*

1 Introduction

The complexity of digital circuits and systems is making the validation of their functionality a daunting task. Sequential circuits, in particular, constitute a hard problem. Two approaches to attack circuit validation are symbolic search techniques and cycle-based simulation.

Search algorithms, e.g, breadth-first search, are a convenient way to visit the state diagram of a sequential circuit. Given enough time and memory, a search can terminate either upon finding an error or by completing the visit of all reachable states successfully. Current symbolic traversal tools are quite efficient for medium–small circuits, but they become impractical for circuits with more than one hundred latches. In those cases the BDDs involved in the computation often grow too large, image computation gets to expensive, or the circuit is sequentially too depth to be dealt with. It is also worth adding that, for large circuits, the BDDs of the next-state function may be very large as well. The solutions (exact or approximate) to these bottlenecks are still the subject of intense current research.

For all these reasons, *cycle-based simulation* is still the technology of choice for the validation of large synchronous systems. It is nicely scalable, as memory and time are proportional to the gate count, and it is flexible, as it can deal with multiple clocks and event-based approach simulation. Today's cycle-based simulators allow the simulation of fairly large systems (up to a few million gates) with an execution rate of approximately 10^8 2-input gates/second on a 100 MHz processor machine, or 100 states/second for a 1-million gate circuit. However, each run only proves the correctness of the design under test (DUT) for a single particular sequence of stimuli, and only one DUT state and input combination are visited per simulated clock cycle. The number of DUT states and input values visited is thus a very small fraction of the state space of the circuit. Moreover, the design of the input stimuli is left to the designer, and it is an obviously crucial task.

In this work, we consider a tradeoff between symbolic search and simulation. In our approach, at each clock cycle, the DUT inputs can assume constant values, as in simulation, or they can be free, as in symbolic search. A (possibly) minimal number of inputs is tied to constants in order to avoid an explicit representation of the next-state function, and to obtain an easy-to-represent frontier subset, while at the same time reach in each cycle a number of states (hopefully) larger than logic simulation.

For this work, we adopt a *parametric* representation of frontier sets. This representation can be constructed and manipulated very efficiently. The selection of which inputs to tie and to what value is based on the "ease of construction" of this representation. Alternatively, however, this selection can be left to the user or to the tool: By freeing inputs selectively, it is possible to symbolically simulate any "neighborhood" of the sequence generated by the test bench.

The parametric representation allows us also to avoid the computation and representation of the global next state functions of the circuit, thereby avoiding a lengthy simulation set-up time. Finally, no reached state set is main-

*Stefano Quer is also with the Politecnico di Torino, Dipartimento di Automatica ed Informatica, Turin, ITALY tained, so its representation is not a bottleneck.

We demonstrate on several benchmarks (all the larger ISCAS benchmarks, as well as other industrial designs) that with these techniques the number of circuit inputs that need be tied to a constant value is low. We show experimentally that symbolic simulation allows us to speed up logic simulation by a factor of over 10^3 for circuits that cannot be handled by current verification techniques, on a 96Mbyte Workstation.

2 Preliminaries

2.1 Notation

Let $\mathcal{B}$ denote the set $\{0,1\}$. A logic function f is a mapping $f : \mathcal{B}^m \rightarrow \mathcal{B}^n$. The *range* of a function f is the set of n-tuples that can be asserted by f. It will be denoted by $Range(f)$. The i^{th} component of f will be denoted by f_i. The support of a logic function f is the set of variables v for which $f_v \neq f_{\overline{v}}$, i.e., the left and right cofactors differ. We indicate it by $Supp(f)$.

We assume the reader is familiar with the concept of BDDs, and basic operations using BDDs. We use them to represent functions, relations and sets. Given a function a BDD representing a function f, we indicate with $|f|$ its number of nodes.

Given a synchronous logic circuit we indicate with (i_1, ..., i_m), (o_1, ..., o_0), and (s_1, ..., s_n), input, output, and state variables. Next state functions are indicated with $\delta(s,i)$, and output functions with $\lambda(s,i)$. S_0 is the initial state set.

2.2 Symbolic Search

Figure 1 shows the pseudo–code of a verification algorithm for a synchronous circuit, using a *symbolic forward traversal.*

```
SymbolicVerification (δ, λ, S0)
{
Reached = To = From = New = S0;
while (New ≠ ∅)
    {
    CheckOutputs (∀i λ(New, i));
    To = δ(From);
    New = To · ¬Reached;
    Reached = Reached + New;
    From = Best_BDD (New, Reached);
    }
}
```

Figure 1: Symbolic Verification based on Forward Traversal.

CheckOutputs represents a generic checker, that evaluates the correctness of the λ functions. The actual functionality of CheckOutputs depends on the application of the traversal routine. At each traversal step, function δ(`From`) computes the set of states directly reached from `From`. Visited states are accumulated in `Reached`. The termination condition is to reach a least fixed–point, i.e., visiting all states reachable from S_0. This condition is equivalent to testing the emptiness of `New` at each step. `New` sets can be subsequently used to find the sequence that exposed the fault.

2.3 Logic Simulation

Figure 2 shows the pseudo-code of a simple functional simulation loop. `T` contains the test vectors.

```
CycleBasedSimulation (δ, λ, S0, T)
{
From = S0;
while (IsNotEmpty (T))
    {
    t = NextVector (T);
    CheckOutputs (λ (From, t));
    To = δ (From, t);
    From = To;
    }
}
```

Figure 2: Cycle-based Simulation Approach.

In compiled code simulation [1], gates and combinational RTL components are mapped into machine instructions, while latches are mapped into memory locations. In interpreted simulation, the netlist is represented in memory as a simulation code. The memory occupation of the circuit is thus linear in the circuit size.

Recently, approaches based on a BDD representation of the netlist were proposed in [2, 3, 4].

The simulation may be "oblivious", in which case all gates are evaluated at each clock tick, or it can propagate only value changes across the netlist. Although there are typically many more gates than value changes, in practice, compiled-code oblivious simulation appears to produce the most compact circuit representation and fastest execution. Empirical measurements by several authors indicate an evaluation rate of *1 gate / processor clock cycle.* On a 100MHz processor, this rate allows the simulation of 100 test vectors per second, for a 1-million gate design.

2.4 Symbolic Simulation

Figure 3 shows the iterative model of a synchronous circuit.

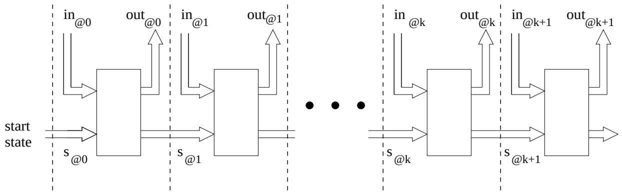

Figure 3: Symbolic Simulation Approach.

In symbolic simulation, at each time step k, the expression of the primary outputs and state variables is computed, in terms of the variables in $in_{@0}, \cdots, in_{@k}$. These expressions quickly become large and intractable, making the whole approach practically infeasible.

One obvious simplification consists of resorting to state variables. When computing the expression of δ at time $k+1$, instead of using the expressions of the state variables $s_{@k}$, one keeps track only of the possible configurations that these variables can assume, i.e., the simulation frontier at time k.

The simulation loop then reduces essentially to the symbolic analysis loop of Figure 1, without lines 6 and 7. BEST_BDD just returns the set `To`.

This approach, however, has still all the drawbacks of reachability analysis, except for the computation of the set `Reached`.

3 Our Approach

We based our methodology on the following observations. Consider the situation of Figure 3. Although the functions δs in general can be complex, in practice their components are often very simple (constants, copies of the input variables, or complements of input variables) once the state variables are replaced by constant values. Moreover, an input variable may be copied into several state variables: There are then *functional dependencies* among the various state bits [5, 6]. The function δ at time 1 may be simplified by taking advantage of these functional dependencies.

Even if δ is complex, in practice usually very few inputs need to be tie to constant values in order to simplify it drastically.

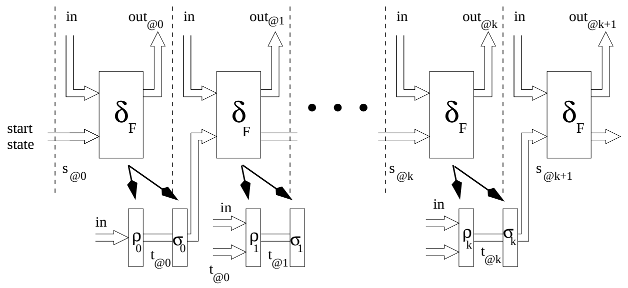

Figure 4: Improved Symbolic Simulation Approach.

Our approach is as follows. At each clock tick, in order to build a simplified δ, we try to take advantage of constant state bits, and functional dependencies among the various state bits. To do that, as indicated in Figure 4, we resort to a *parametric* representation of the reached state set. This representation is as follows. We introduce some *intermediate* variables t_i. δ_F will be a function of the primary inputs and of these intermediate variables.

After building its BDDs, δ_F is decomposed into two blocks, labeled ρ and σ in Figure 4. The outputs of ρ represent intermediate variables t_i for the next clock tick. The components of ρ should be "simple" functions (disjoint-support or data-transfer functions, see Section 3.1 for details) of primary inputs and intermediate variables at the previous clock tick. The function σ is the core of the parametric representation. Ideally, σ should be such that $Range(\sigma) = Range(\delta_F)$.

In practice, we settle for a σ such that $Range(\sigma)$ represents a "large" and easy-to-identify subset of $Range(\delta_F)$.

The functions ρ and σ are built *dynamically*, and change at each clock cycle. The number of components of ρ may change as well, but it is always smaller than (or equal to) the number of state variables.

At the next clock cycle, the BDD of $\delta_F(i, \sigma(t))$ is built and new intermediates identified. We replace each state variable directly with its expression in terms of intermediate variables, and compute directly the BDDs of $\delta_F(i, t)$. In general, δ_F is expected to be simpler than δ because of the fewer variables involved. The function λ_F is also represented in terms of the parameter variables. It is worth noting that, if a bug is discovered, it is possible to reconstruct the sequence that exposes the bug.

Figure 5 shows the described procedure.

SYMBOLICSIMULATION (δ, λ, S_0, T)
{
$\sigma = S_0$;
while (ISNOTEMPTY (T))
{
$\{\delta_F, \lambda_F\}(inputs, intermediates)$ = SIMULATE(σ);
CHECKOUTPUTS (λ_F);
$\{\rho, \sigma\}$ = DECOMPOSE (δ_F, T);
}
}

Figure 5: Pseudo-code for the Improved Symbolic Simulation Approach.

Procedure SIMULATE substitutes latch output variables with their expressions in σ. It then simulates symbolically the combinational portion of the circuit and returns the arrays of BDDs δ_F and λ_F.

DECOMPOSE is detailed in Figure 6.

DECOMPOSE (δ_F, T)
{
C = FINDCOMPLEXVARIABLES(δ_F);
δ_F = ASSIGNANDCOFACTOR(δ_F, C, T);
F = FINDSHAREDVARIABLES(δ_F);
δ_F = ASSIGNANDCOFACTOR(δ_F, F, T);
$\{\rho, \sigma\}$ = REWRITE(δ_F);
}

Figure 6: Pseudo-code for the function DECOMPOSE.

It basically performs two main operations. First, it makes sure that the parameterization of $Range(\delta)$ can be carried out in linear time. If this is not the case, it identifies variables for assignment, and cofactors δ accordingly. The actual constant values are provided by the test bench input `T`. It then decomposes δ_F into σ and ρ. Functions FINDCOMPLEXVARIABLES, FINDSHAREDVARIABLES, and REWRITE are described in Sections 3.1, 3.2, and 3.3.

3.1 Function FINDCOMPLEXVARIABLES

We address first the problem of identifying quickly a function σ such that $Range(\sigma)$ is a "large" subset of $Range(\delta_F)$. This description requires the following definitions.

Definition 1 *A variable x is termed* **simple** *if there is a component $\delta_{F,i}$ of δ_F such that $Supp(\delta_{F,i}) = \{x\}$.*

Given a function δ_F, let S denote the set of simple variables. A component $\delta_{F,i}$ is termed **simple** *if $Supp(\delta_{F,i}) \subseteq S$.*

Definition 2 *Let again S denote the set of simple variables. A non-constant function $\delta_{F,i}$ is termed* **complex** *if $Supp(\delta_{F,i}) \cap S \neq \phi$* **and** *$Supp(\delta_{F,i}) \cap \overline{S} \neq \phi$. For a complex function $\delta_{F,i}$, a variable belonging to $Supp(\delta_{F,i}) \cap \overline{S}$ is also termed* **complex**. *A variable or function is* **unbound** *if it is neither simple nor complex.*

Definition 3 *Two components $\delta_{F,i}, \delta_{F,j}$ of δ_F are termed* **equivalent** *if they are unbound and $\delta_{F,i} = \delta_{F,j}$ or $\delta_{F,i} = \delta'_{F,j}$.*

Definition 4 *Given an equivalence class ϵ of functions, we indicate with $Supp(\epsilon)$ the set of variables belonging to the support of any function in ϵ. A variable $x \in Supp(\delta_F)$ is said to be* **bound** *if it belongs only to the support of a* **single** *equivalence class of δ_F. It is termed* **shared** *otherwise.*

Let us suppose now that the components of δ_F are only: 1) Constants, 2) functions of a single variable, or 3) functions of variables also appearing as single variables in other components, i.e., simple functions. In this case, as the following example shows, it is easy to construct an exact parametric description for $Range(\delta_F)$, with the minimum number of variables.

Example 1 *Let us suppose to have*

$$\delta_F(x, y) = (x, x', y, 0, f(x, y), g(x, y), y')$$

An obvious parametric description in this case is:

$$\sigma = (t_0, t'_0, t_1, 0, f(t_0, t_1), g(t_0, t_1), t'_1)$$

It is immediate to see that it uses the minimum number of variables. Moreover, notice that in this case ρ represents just a data-transfer operation $t_0 = x, t_1 = y$.

Suppose δ_F consists only of simple and complex functions. By assigning a value to complex variables, some complex functions may become simple as shown by the following example.

Example 2 *Consider*

$$\delta_F(p, q, r, x, y) = (x, y, x + y + p + q, p + xq).$$

$\delta_{F,0}$ and $\delta_{F,1}$ are simple. $\delta_{F,2}$ and $\delta_{F,3}$ are complex, as variables p and q are complex. By assigning a value to p and q, complex components become simple and δ_F can have a simple parametric representation.

Simple and complex variables (and functions) can be identified in a two-pass scan of the BDDs of δ_F. Figure 7 shows the pseudocode. Initially, functions and variables are labeled unbound. One pass finds the support of each component of δ_F and identifies simple variables. This is accomplished in the first `foreach` loop and takes time proportional to $|\delta_F|$. The second `foreach` loop identifies complex variables and places them in the set C. Function FunctionType acts as previously described.

```
FindComplexVariables (δ_F)
{
S = C = ∅;
foreach (δ_F,i ∈ δ_F) {
    if (|Supp(δ_F,i)| == 1) {
        S = FunctionType (δ_F,i, Simple);
        S = S ∪ Supp(δ_F,i);
    }
}
foreach (δ_F,i ∈ δ_F) {
    if (Supp(δ_F,i) ∩ S ≠ ∅) {
        Temp = Supp(δ_F,i) ∩ S̄;
        if (Temp ≠ ∅) {
            C = FunctionType (δ_F,i, Complex);
            C = C ∪ Temp;
        }
    }
}
return(C);
}
```

Figure 7: Identifying Simple and Complex Variables.

3.2 Function FindSharedVariables

After the identification and removal of complex variables, the components of δ_F are labeled as either simple or unbound. The supports of unbound functions have no variables in S.

We now examine unbound components. The simplest case occurs when one unbound component has support disjoint from all other components. This case is described by the following example.

Example 3 *In the following equation:*

$$\delta_F = (f(p, q), x, y, g(x, y)).$$

the first component is unbound and has a disjoint support. In this case, the value of $f(p, q)$ is independent from that of the others. Hence, the component can be replaced by an independent intermediate variable:

$$\sigma = (t_0, t_1, t_2, g(t_1, t_2)).$$

and the function ρ is $t_0 = f(p, q), t_1 = x, t_2 = y$.

This case can be generalized further as the following example shows.

Example 4 *Consider the situation shown in the following equation*

$$\delta_F = (f(p, q), f'(p, q), x, y).$$

Clearly, the first and second component of δ_F can be replaced by t_0, t'_0, respectively.

Definition 4 partitions the set of unbound functions in δ_F into equivalence classes. These classes can be discovered in a single scan of the array δ_F.

Consider assigning a value to all shared variables. The support of each equivalence class will then contain only bound variables, and each equivalence class can then be replaced by an independent parameter.

Example 5 *Consider the situation where*

$$\delta_F = (x+y+z, (x+y+z)', z'w, z'w).$$

By assigning value 0 to z, the components of δ_F become:

$$\delta_F = (x+y, (x+y)', w, w).$$

A parametric representation of $Range(\delta_F)$ is then given by

$$\sigma = (t_0, t_0', t_1, t_1),$$

where $t_0 = x+y, t_1 = w$

Figure 8 illustrates the algorithm for finding shared variables.

```
FindSharedVariables (δ_F)
{
Shared = EqvClasses = ∅;
foreach(δ_F,i ∈ δ_F) {
    if(FunctionType(δ_F,i) == Unbound) {
        Class = FindOrMakeNewClass(δ_F,i);
        EqvClasses = EqvClasses ∪ {Class} ;
        Tag(Supp(δ_F,i), Class);
    }
}
foreach(Class ∈ EqvClasses) {
    foreach (x ∈ Supp(Class)) {
        if (Tag(x) ≠ Class)
            Shared = Shared ∪ {x};
    }
}
return(Shared);
}
```

Figure 8: Identifying Bound and Shared Variables.

3.3 Function Rewrite

Procedure Rewrite generates the actual ρ and σ in the following way. For a circuit with m inputs $(i_1, \cdots, i_m)$ and n state variables $(s_1, \cdots, s_n)$, exactly n intermediates t_i are introduced. Some of them, however, may end up unused. The BDD ordering of the intermediates reflects that of the state variables: If variable s_j has rank k, then variable t_j has rank k.

Once complex and shared variables are removed, the components of δ_F are either simple variables, or simple functions, or functions bound to equivalence classes. The components of σ are then obtained by *replacing* each simple variable and equivalence class by an intermediate t_i. The BDD of simple functions must be re-written in terms of the new intermediates t_i. To make this re-writing simple (i.e., linear in the BDD size), a dynamic replacement procedure is established, as follows. A *replacement table* with $m+n$ entries is kept. The k^{th} entry of the table represents the k^{th} variable in the BDD ranking, from the top. The entries are visited in order. The first variable that appears as a simple variable is replaced by t_1, the second one by t_2, and so on.

Each equivalence class ϵ_i is then assigned one of the still unassigned t_k.

4 Experimental Results

We implemented a prototype symbolic simulator and tested it on a Pentium II 150MHz Workstation with 96 Mbytes of memory, running Linux. The simulator is based on interpretive code, and it is oblivious. We tested it on the largest circuits in the ISCAS'89 and ISCAS'89–addendum suite, plus two medium-size commercial designs. Each 2-input gate takes 16 bytes of memory. A proprietary BDD package is used.

We evaluated the performance of the simulator by running it for 1000 cycles on each benchmark circuit. Complex and shared variables are assigned random values.

Table 1 reports the relevant circuit metrics and summarizes the experimental results.

For each circuit we report the number of primary inputs `#PI`, primary outputs `#PO`, memory elements `#FF` (Flip-Flops), and gates `#G`. Column `Constants` reports the number of state bits (components of δ_F) that are constant. The rest is formed by simple, complex, and unbound functions. In some circuits, the presence of counters makes this number high.

Column `Assigned` reports the average number of complex plus shared variables that are assigned at each clock cycle by function Decompose, in order to reduce all functions to simple or bound.

`Intermediates` reports the average number of intermediate variables appearing in σ. The number of states visited at each clock cycle is thus $2^{\texttt{Intermediates}}$. The support of δ_F at the next clock cycle has then size `Intermediates + #PI`. Because some inputs are bound to constants, the effective number of free inputs is then `Intermediates + #PI - Assigned`. This number is summarized in column `Free`. $2^{\texttt{Free}}$ represents the effective degree of parallelism in the computation of the next-state function. The parallelism in computing the output function, however, is given by `Intermediates + #PI`.

Column `CPU Time` reports the CPU time (in seconds) spent for 1000 cycles of simulation. The data indicate a processing rate of about 3×10^4 gates/sec, and are fairly uniform for the various circuits. This is apparently 3×10^3 times slowly than cycle-based simulation. Notice, however, that in each symbolic simulation cycle, $2^{\texttt{Free}}$ simulations are implicitly executed in parallel. For $s3384$, for instance, the number of vectors simulated in parallel is 2^{35}, resulting in an actual speedup of over 10^4.

The number of variables that we need to assign varies significantly, the least fortunate case being $s9234$, where the components of δ_F seem to be all unbound and distinct most of the time. In the case of $s3384$, instead, all components are either constant or disjoint-support.

Column `Peak Memory` indicates the total memory consumption of the simulation in Mbytes. This includes representing the netlist and all the BDDs. The consumption is very modest in all cases, except for two circuits, for which over 50Mbytes of memory are required. Memory consumption seems to correlate with the value of column `Free`. As `Free` approaches the number of total inputs and state for the circuit, the size of δ_F can grow close to that of δ. Fur-

Circuit	#PI	#PO	#FF	#G	Constants	Assigned	Intermediates	Free	CPU Time	Peak Memory
s1269	18	10	37	560	23	13	2	7	1.24	1.1
s1423	17	5	74	657	22	4	7	20	3.28	0.8
s1512	29	21	57	767	48	17	1	13	2.72	0.7
s3271	26	14	116	1559	58	25	1	2	6.27	0.9
s3330	40	73	132	1717	49	6	22	56	13.08	1.1
s3384	43	26	183	1660	166	25	17	25	12.11	12.1
s4863	49	16	104	1327	16	22	27	54	48.32	32.6
s5378	35	49	179	2779	119	21	11	25	14.89	9.8
s6669	83	55	239	3026	133	21	60	122	44.90	68.3
s9234	38	39	211	5808	211	10	0	28	25.53	1.6
s13207	31	121	669	8652	558	0	6	37	45.23	2.1
s15850	14	87	597	10384	516	1	3	16	43.14	2.1
s35932	35	320	1728	12204	873	34	1	2	407.10	5.6
s38417	28	106	1636	24033	1115	6	21	43	988.80	52.1
s38584	38	204	1426	35762	1037	5	2	35	294.10	3.6
dmac	44	149	328	5926	102	5	102	141	91.80	23.0
matmult	37	97	836	9660	526	7	32	62	604.10	18.4

Table 1: Average number of intermediate variables and free variable for each simulation cycle.

ther work should include strategies to prevent this potential blow-up.

Several ISCAS benchmarks seem to contain “highly sequential” components (such as counters). If the state bits of a counter take constant value at some point in time i.e., they are represented by constants, then also the at the next clock cycle they will be represented by constants. The last two circuits are more data-path intensive: They contain several large data-transfer or arithmetic operations. It is easier in these cases to assign state bits independently. Hence the larger number of parameter variables.

5 Conclusions and Future Work

This paper presents an approach to perform *symbolic simulations* of synchronous circuits at the gate level. The approach is built around the numerical cycle-based methodology, and it is based on the quick re-writing of frontier sets in terms of Boolean parameters. The technique allows the designer to selectively “freeing” some (if not all) circuit inputs, and to deal with more than one state and many input combinations at a time. The equivalent execution rate is boosted by a factor of over 10^4 over cycle-based simulation for larger circuits. Moreover, it seems to increase as the circuit size increases.

There are several tradeoffs that need to be explored further. For example, since the execution of a gate is non-trivial, the tradeoff point may be more towards an event-based execution. Moreover, improving the designer's confidence in the technique is obviously the main topic of future work. To obtain that, larger and larger trace sets must be considered in parallel. We are therefore investigating ways of increasing the number of free variables in each symbolic simulation cycle.

References

[1] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. Hss- a high-speed simulator. *IEEE Trans. on CAD/ICAS*, pages 601–617, July 1987.

[2] P. C. McGeer, K. L. McMillan, A. Saldanha, A.L. Sangiovanni-Vincentelli, and P. Scaglia. Fast discrete function evaluation using decision diagrams. In *Proc. ICCAD*, pages 402–407, Nov 1995.

[3] P. Ashar and S. Malik. Fast functional simulation using branching programs. In *Proc. ICCAD*, pages 402–407, Nov 1995.

[4] Y. Luo, T. Wongsonegoro, and A. Aziz. Hybrid Techniques for Fast Functional Simulation. In *Proc. IEEE/ACM DAC'98*, pages 664–667, San Francisco, California, June 1998.

[5] A. J. Hu and D. L. Dill. Reducing bdd size by exploiting functional dependencies. In *Proc. DAC*, pages 266–271, June 1993.

[6] C.A.J. van Eijk and J. A. G. Jess. Exploiting functional dependencies in fsm verification. In *Proc. EDAC*, pages 9–14, February 1996.