Cycle-based Symbolic Simulation of Gate-level Synchronous Circuits

Valeria Bertacco[†] Maurizio Damiani[‡] Stefano Quer[‡1]

[†]Vera Group
Synopsys, Inc.
Palo Alto, CA 94303

[‡]Advanced Technology Group
Synopsys, Inc.
Mountain View, CA 94043

ABSTRACT

Symbolic methods are often considered the state-of-the-art technique for validating digital circuits. Due to their complexity and unpredictable run-time behavior, however, their potential is currently limited to small-to-medium circuits. Logic simulation privileges capacity, it is nicely scalable, flexible, and it has a predictable run-time behavior. For this reason, it is the common choice for validating large circuits. Simulation, however, 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 (over 100 symbolic inputs) for the largest ISCAS benchmark circuits, using 96Mbytes of 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 [1, 2, 3] (e.g, breadth-first search), are a convenient way to visit the state diagram of a sequential circuit. They require maintaining a *frontier* and a *reached* state set. A search step consists of computing the image of the frontier state set under all possible input combinations ("in parallel"). Newly discovered states form the new frontier, while the old frontier is merged into the set of reached states. Given enough time and memory, a search can terminate either upon finding an error or by completing the visit of all reachable states. Current symbolic traversal tools often become impractical for circuits with over one hundred latches, for many reasons: 1) the size of the BDDs involved in the computation to represent and maintain state sets grows too large; 2) the time for computing the new frontier set (*i.e.* image computation) gets too long; 3) the circuit is sequentially too deep; 4) the BDD of the next-state function (or relation) is too large. The solution (exact or approximate) to these bottlenecks is still the subject of intense current research. Eventually, symbolic traversal is not very informative from a design debugging standpoint: If a bug is found, it is nontrivial to construct an input trace that exposes it.

For these reasons, *cycle-based simulation* [4, 5, 6] is still the technology of choice for the validation of large synchronous systems. Logic simulation is nicely scalable. The memory image of a circuit is proportional to its gate count, and so is the time to propagate values from inputs to outputs. Moreover, it is flexible: Practical cycle-based simulators allow for circuits with multiple clocks and interface to event-based simulation [7]. Today's cycle-based simulators allow the simulation of large systems (up to a few million gates) with an execution rate of up to 10^8 2-input gates/second on a 100MHz CPU machine, or 100 states/second for a 1-million gate circuit.

Simulation, however, is not a satisfactory solution to the validation problem. Each run only proves the correctness of the design under test (DUT) for that particular sequence of stimuli. 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. The design of the input stimuli is left to the designer, and it is an obviously crucial task. Expensive emulation engines can also be used to speed up simulation and reach more states. The simulation set-up, however, often requires weeks of work.

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 this way, we : 1) avoid representing the full next-state function, and 2) obtain an easy-to-represent frontier subset. At the same time, we

[1]Stefano Quer is also with Politecnico di Torino, Dipartimento di Automatica ed Informatica, Turin, Italy

```
        SYMBOLICVERIFICATION (δ, λ, S0) {
1       Reached = To = From = New = {S0};
2       while (New ≠ ∅) {
3           CHECKOUTPUTS (∀i λ(New, i));
4           To = δ(From);
5           New = To ∩ ‾Reached‾;
6           Reached = Reached ∪ New;
7           From = BEST_BDD (New, Reached);
        } }
```

Figure 1. Forward traversal-based reachability analysis.

simulate many input combinations in parallel and (hopefully) reach a large number of states.

We adopted a *parametric* representation of frontier sets [8]. 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, 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 an input trace 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 maintained, 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 only a few inputs need be assigned a constant value. 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 PC.

2. PRELIMINARIES

Let B denote the set $\{0,1\}$. A logic function f is a mapping $f : B^m \rightarrow B^n$. The *range* of f is the set of n-tuples that can be asserted by f. It will be denoted by $Range(f)$. The ith component of f will be denoted by f_i. The support of f is the set of variables v for which $f(v = 0) \neq f(v = 1)$. it is denoted by $Supp(f)$. We assume functions to be represented by their BDDs [9, 10]. We indicate by $|f|$ the number of nodes of a BDD of f.

We indicate with (i_1, $\cdots$, i_m), (o_1, $\ldots$, o_0), and (s_1, $\ldots$, 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.

2.1. Symbolic search

Fig. 1 shows the pseudo–code of a verification algorithm for a synchronous circuit, using a *symbolic forward traversal*. CHECKOUTPUTS represents a generic checker. It evaluates the correctness of λ. Its actual functionality depends on the final application of the traversal routine. At each traversal

```
        CYCLEBASEDSIMULATION (δ, λ, S0, T){
1       From = S0;
2       while (ISNOTEMPTY (T)) {
4           t = NEXTVECTOR (T);
5           CHECKOUTPUTS (λ (From, t));
6           To = δ (From, t);
7           From = To;
        } }
```

Figure 2. Cycle-based simulation approach.

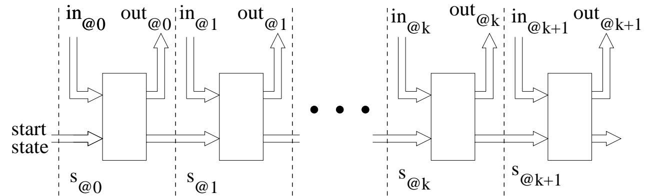

Figure 3. Symbolic simulation approach.

step, δ(From) determines the states To reached from the set From. Set New contains the To states that have not yet been visited. Reached states accumulate in Reached. After the first step, function BEST_BDD [11] selects a subset From with a simple BDD representation. From ranges from New to Reached. The code terminates when no more New states are reached.

2.2. Logic simulation

Fig. 2 shows the pseudo-code of a simple functional simulation loop. T contains the test vectors. In compiled code simulation [4], gates and combinational RTL components are mapped to machine instructions, while latches are mapped to memory locations: The netlist is effectively compiled into a program.

In interpreted simulation, the netlist is a graph structure in main memory. Each node contains type (AND, OR , ...), fanin and / or fanout information. The simulation code is independent from the netlist structure. It visits each node of the graph and computes the node output from the inputs, according to the node type. In either compiled-code or interpreted simulation, δ is still represented by the circuit's netlist. The memory occupation of the circuit is thus linear in the circuit size. Approaches based on a BDD representation of the netlist were proposed in [12, 13, 14].

In "oblivious" simulation all gates are evaluated at each clock tick. Alternatively, only value changes across the netlist are propagated. Although there are typically many more gates than value changes, in practice, the additional data structure requirements and checks appear to favor oblivious simulation. Today, compiled-code oblivious simulation appears to produce the most compact circuit representation and fastest execution.

2.3. Symbolic simulation

Fig. (3) shows the iterative model of a synchronous circuit. In symbolic simulation, at each time step k, the expression of the primary outputs and state variables is computed, in terms

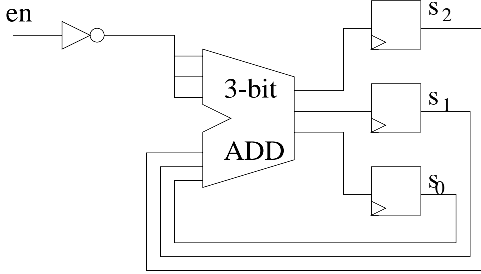

Figure 4. a 3-bit down counter with enable.

of the variables in $in_{@0}, \cdots, in_{@k}$. Example (1) illustrates this construction for a 3-bit down counter.

Example 1 *Fig.(4) shows a 3-bit* down *counter with enable. Outputs coincide with the state variables. Starting from state* 0*, the expressions of the outputs are*

$$\begin{array}{lcll} time = 0 & : & out_{@0} = & (0,0,0); \\ time = 1 & : & out_{@1} = & (en_{@0}, en_{@0}, en_{@0}); \\ time = 2 & : & out_{@2} = & (en_{@0} + en_{@1}, en_{@0} + en_{@1}, \\ & & & en_{@0} \oplus en_{@1}); \\ time = 3 & : & \cdots & \end{array}$$

Notice that the state variables do not appear in the output expressions.

If a bug is found at some time k the expression of `out` can provide the entire set of input sequences of length k that expose it. Unfortunately, however, 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). A symbolic simulation loop is then essentially the forward reachability analysis loop of Fig. (1), without lines 5, 6. BEST_BDD just returns To. It also inherits all the drawbacks of reachability analysis, except for the computation of Reached. In particular, one looses information on how a certain state is reached. This makes debugging more complex.

3. OUR APPROACH

We based our method on the following observations. Consider the situation of Fig. (3). Although δ in general can be complex, in practice at time 0 its components are often very simple (constants , copies of an input, or complement of an input), because 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 [15, 16]. We use these functional dependencies to obtain a simplified representation of δ at time 1.

In practice, we never build explicitly δ. Rather, at each clock tick k, we build a simplified version δ_F of δ. We use the functional dependencies among the components of δ_F at time k

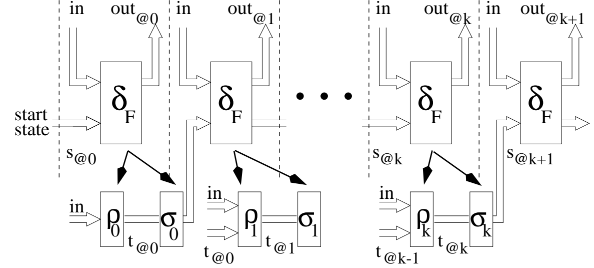

Figure 5. Our symbolic simulation approach.

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 6. Pseudo-code of our approach .

to build a version of δ_F at time $k+1$. If, in spite of our efforts, δ_F becomes "complex", few inputs are tied to constant values in order to simplify it.

3.1. Using functional dependencies.

We discover and exploit functional dependencies using a *parametric* representation of the reached state set [8]. Fig. (5) illustrates the approach. We introduce some *intermediate* variables t_i. At a generic clock tick k, we inspect the BDDs of δ_F and build a function $\sigma(t_i)$ such that

$$Range(\sigma) = Range(\delta_F). \qquad (1)$$

In practice, we will settle for a σ such that 1) the number of parameter variables t_i is small, and 2) $Range(\sigma)$ is a "large" and easily identifiable subset of $Range(\delta_F)$. Section(3.2) provides the details on σ and its construction. The BDD of $\delta_F(i_1, \cdots, i_m, \sigma)$ is then built, and a new σ constructed. Notice that state variables are effectively replaced by these intermediate variables.

In addition, we build a second mapping ρ. This second mapping expresses each t_i as a function of inputs and intermediates at the previous tick. Also ρ should be "simple", for the following reason. Suppose a bug is discovered at time k. There is then an assignment of primary inputs and intermediates at time k that exposes the bug. We need to be able to map the assignment of intermediates to an assignment of inputs and intermediates at time $k-1$, and then iteratively back to primary inputs at time $k-2, \cdots, 0$.

Fig. 6 shows the proposed 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 shown in Fig. 7. It performs two main operations. First, it makes sure that $Range(\delta_F)$ can be parameterized in linear time. If this is not

```
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);
    {ρ,σ} = REWRITE(δ_F);
}
```

Figure 7. Pseudo-code for the function DECOMPOSE**.**

the case, it identifies variables for assignment, and cofactors δ_F accordingly. The actual constant values are provided by the test bench input T. It then decomposes δ_F into σ and ρ. Functions FINDCOMPLEXVARIABLES, FINDSHARED-VARIABLES, and REWRITE are described in Sections 3.2 and 3.3, respectively.

3.2. Identifying intermediate variables

We show here a way to identify quickly a function σ such that $Range(\sigma)$ is a "large" subset of $Range(\delta_F)$. This 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* $Supp(\varepsilon)$ *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.*

Suppose first 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 (that is, simple functions). One such case would be, for example,

$$\delta_F(x,y) = (x, x', y, 0, f(x,y), g(x,y), y') \quad (2)$$

An exact parametric description is obtained by replacing x, y with two parameters:

$$\sigma = (t_0, t'_0, t_1, 0, f(t_0,t_1), g(t_0,t_1), t'_1) \quad (3)$$

Notice that ρ is just a data-transfer: $t_0 = x; t_1 = y$.

Suppose now that δ_F consists only of simple and complex functions. By assigning a value to complex variables, some complex variable may become simple:

Example 2 *Consider*

$$\delta_F(p,q,r,x,y) = (x, y, x+y+p+q, p+xq). \quad (4)$$

$\delta_{F,0}$ *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.*

```
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 8. Identifying Simple and Complex Variables.

Simple and complex variables (and functions) are identified in a two-pass scan of the BDDs of δ_F. Fig. (8) shows the pseudocode. Initially, functions and variables are labeled Unbound. The first foreach loop finds the support of each component of δ_F and identifies Simple variables. This takes $O(|\delta_F|)$ time. The second foreach loop identifies complex variables and places them in C.

After complex variables are identified and removed, the components of δ_F are labeled as either simple or unbound. Unbound functions have no support variables in S. We now examine unbound functions. The simplest case occurs when one unbound function has support disjoint from all other components. For example, in Eq. (5) below:

$$\delta_F = (f(p,q), x, y, g(x,y)). \quad (5)$$

the first component is unbound and has support disjoint from all others. The component can be replaced by an independent intermediate variable: $\sigma = (t_0, t_1, t_2, g(t_1,t_2))$ where $t_0 = f(p,q); t_1 = x; t_2 = y$.

Consider the more general situation:

$$\delta_F = (f(p,q), f'(p,q), x, y). \quad (6)$$

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 contain only bound variables, so each class can be replaced by an independent parameter.

Example 3 *Consider*

$$\delta_F = (x+y+z, x'y'z', z'w, z'w). \quad (7)$$

By assigning $z = 0$*, the components of* δ_F *become:*

$$\delta_F = (x+y, (x+y)', w, w). \quad (8)$$

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

$$\sigma = (t_0, t'_0, t_1, t_1), \quad (9)$$

```
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 ∪= { x };
    } }
    return(Shared);   }
```

Figure 9. Identifying Bound and Shared Variables.

where $t_0 = x + y; t_1 = w$.

Fig. (9) illustrates the algorithm for finding shared variables.

3.3. The REWRITE function.

REWRITE generates ρ, σ as follows. 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 t_i 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 will have 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 kth entry of the table represents the kth 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 symbolic simulator and tested it on a PC based on a 150MHz Pentium with 96 Mbytes of memory, running Linux. The simulator is interpretive and oblivious. We tested it on the largest circuits in the ISCAS'89 [17] and ISCAS'89–addendum suite, plus two medium-size commercial designs. Each 2-input gate takes 16 bytes of memory. A proprietary BDD package was used.

We evaluated 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, and gates # G.

The following measures are important for our purposes : 1) the average size of the support of δ_F, and 2) the average number of states reached at each simulated clock tick.

Interms reports the average number of intermediate variables appearing in σ. The average number of states visited at each clock tick is 2^{Interms}. The support of δ_F has size Interms + #PI . This support can be used for the detection of bugs in the output of the DUT. This number gives us the parallelism in the computation of δ_F and λ_F. It is reported in column In-δ_F.

Column Assd reports the average number of (input + intermediate) variables assigned by DECOMPOSE, during the construction of σ. The parallelism in computing the function σ is thus given implicitly by Interms + #PI - Assigned. This number is reported in Free. The actual number of parallel traces is 2^{Free}.

Column Memory indicates the total memory consumption of the simulation in Mbytes. This includes the netlist and the BDDs.

Column CPU reports the time for 1000 simulation cycles. Column CPU-sim indicates the time spent for 1000 cycles of compiled-code simulation. Finally, column efficiency contains the ratio $2^{\mathsf{Free}} \times (\mathsf{CPU\text{-}sim}/\mathsf{CPU})$. It represents the number of symbolic simulations executed in the time spent in one numerical simulation.

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 (that is, they are represented by constants), then also the at the next clock tick 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

We presented an approach towards a *symbolic simulation* of synchronous circuits. The approach is based on the quick rewriting of frontier sets in terms of Boolean parameters. It allows the designer to construct a symbolic simulation around a numerical cycle-based one, by selectively "freeing" some (if not all) of the circuit inputs. This approach allows us to deal with more than one state and many input combinations at a time. The equivalent execution rate is boosted by a large factor over cycle-based simulation for the larger circuits. Moreover, it seems to increase as the circuit size increases.

Several tradeoffs need be explored further. For instance, since the execution of a gate is non-trivial, event-based execution may outperform a cycle-based one.

Counters and sequencers occur several times in the benchmarks. A large number of constant bits of course lowers the

Circuit	#PI	#PO	#FF	#G	Interms	In-δ_F	Ass.d	Free	Mem.	CPU	CPU-sim	Effic.
prolog	36	73	136	1845	28.96	64.96	23.98	40.98	0.36	4.29	0.22	1.11×10^{11}
s1269	18	10	37	771	0.76	18.76	13.18	5.58	0.37	1.85	0.07	1.80
s1423	17	5	74	830	1.00	18.00	13.07	4.93	0.05	2.09	0.12	1.76
s1512	29	21	57	990	3.20	32.20	16.50	15.69	0.08	1.78	0.13	3.87×10^{3}
s3271	26	14	116	2166	6.38	32.38	25.89	6.49	0.55	17.76	0.20	1.01
s3330	40	73	132	2020	28.93	68.93	23.89	45.03	0.57	4.44	0.23	1.86×10^{12}
s3384	43	26	183	1734	32.79	75.79	40.70	35.09	0.65	5.83	0.25	1.57×10^{9}
s4863	49	16	104	2492	2.91	51.91	41.86	10.05	0.15	5.94	0.24	4.30×10^{1}
s5378	35	49	179	3973	12.89	47.89	30.93	16.96	0.64	7.96	0.31	4.95×10^{3}
s6669	83	55	239	3272	75.36	158.36	76.58	81.78	36.77	947.61	0.52	2.28×10^{21}
s9234.1	36	39	211	6585	17.96	53.96	19.65	34.31	0.24	11.25	0.34	6.42×10^{8}
s13207	31	121	669	9539	14.41	45.41	4.64	40.77	0.61	21.09	0.78	6.94×10^{10}
s13207.1	62	152	638	9539	57.30	119.30	13.52	105.78	1.36	34.29	0.87	1.76×10^{30}
s15850	14	87	597	11316	4.39	18.39	2.82	15.57	0.50	21.54	0.78	1.76×10^{3}
s15850.1	77	150	534	11316	17.19	94.19	55.73	38.46	1.75	100.01	0.85	3.22×10^{9}
s35932	35	320	1728	23085	1.00	36.00	35.00	1.00	0.96	56.59	2.19	7.75×10^{-2}
s38417	28	106	1636	27648	46.90	74.90	8.19	66.71	2.89	80.73	2.70	4.05×10^{18}
s38584	12	278	1452	24619	6.36	18.36	5.94	12.42	10.25	316.08	2.16	3.74×10^{1}
s38584.1	38	204	1426	24619	7.51	45.51	24.44	21.07	22.40	1248.26	1.88	3.31×10^{3}
dmac	44	149	328	5926	82.43	126.43	65.39	61.04	23.08	91.80	0.45	1.16×10^{16}
matmult	37	97	836	9660	32.65	69.65	17.45	52.20	18.43	604.10	0.86	7.36×10^{12}

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

parallelism of simulation.

Improving the designer's confidence in the technique is the main topic of future work: Larger and larger trace sets must be considered in parallel. Therefore, investigating ways of increasing the number of free variables in each symbolic simulation cycle is important.

References

[1] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Lecture Notes in Computer Science 407, Springer Verlag*, pages 365–373, Berlin, Germany, 1989.

[2] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proc. ICCAD*, pages 130–133, November 1990.

[3] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on CAD*, 13(4):401–424, April 1994.

[4] 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.

[5] C. Hansen. Hardware logic simulation by compilation. In *Proc. DAC*, pages 712–715, June 1987.

[6] L.T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. Ssim: A software levelized compiled-code simulator. In *Proc. DAC*, June 1987.

[7] C.J. DeVane. Efficient circuit partitioning to extend cycle simulation beyond synchronous circuits. In *Proc. ICCAD*, pages 154–161, nov 1997.

[8] P. Jain and G. Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *IEEE Trans. on CAD/ICAS*, 13:1005–1015, August 1994.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[10] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary–Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[11] H. Cho, G. Hachtel, S. Jeong, B. Plessier, E. Shwarz, and F. Somenzi. Atpg aspects of fsm verification. In *Proc. ICCAD*, pages 134–137, November 1990.

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

[13] P. Ashar and S. Malik. Fast Functional Simulation using Branching Programs. In *Proc. ICCAD*, pages 408–412, San Jose, California, November 1995.

[14] 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.

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

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

[17] F. Brglez, D. Bryan, and K. Koźmiński. Combinatorial Profiles of Sequential Benchmark Circuits. In *Proc. IEEE ISCAS'89*, pages 1929–1934, May 1989.