Efficient State Representation for Symbolic Simulation

Valeria Bertacco
Computer Systems Lab
Stanford University
Stanford, CA 94305
valeria@stanford.edu

Kunle Olukotun
Computer Systems Lab
Stanford University
Stanford, CA 94305
kunle@ogun.stanford.edu

ABSTRACT

Symbolic simulation is attracting increasing interest for the validation of digital circuits. It allows the verification engineer to explore all, or a major portion of the circuit's state space without having to design specific and time-consuming test stimuli. However, the complexity and unpredictable run-time behavior of symbolic simulation have limited its scope to small-to-medium circuits.

In this paper, we propose a novel approach to symbolic simulation that reduces the size of the BDDs of the state vector while maintaining an exact representation of the set of states visited. The method exploits the decomposition properties of Boolean functions. By restructuring the next-state functions in their disjoint support components, we gain a better insight in the role of each input variable. Consequently, we can simplify the next-state functions without significantly sacrificing the simulation accuracy. Our experimental results shows that this approach can be used in effectively reducing the memory requirements of symbolic simulation while surrendering only a small portion of the design's state space.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification, Simulation*; B.8 [**Hardware**]: Performance and Reliability

General Terms

Design, Verification, Performance, Theory

Keywords

Formal Verification, Symbolic Simulation, BDDs

1. INTRODUCTION

Validating the functionality of digital circuits and systems is an increasingly difficult task. This is due to the growing complexity of the designs that has not been accompanied by improvements in functional verification techniques.

Logic simulation is still the mainstream approach for the validation of large synchronous systems ([1, 2]) because of its scalability : CPU time is proportional to the design size and test length. Simulation is also flexible: Practical cycle-based simulators allow for circuits with multiple clocks and the ability to mix cycle-based and event-based simulation.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA

Unfortunately, the fraction of the design space which can be explored by simulation is miniscule, especially for large designs. Only one state and one input combination of the design under test are visited during each simulation cycle. Moreover the test stimuli must be hand crafted by the designer to cover those areas of the design that she wishes to validate.

On the other side of the verification spectrum, are symbolic exploration techniques. These methods have significantly increased the reach of formal verification by making it possible to analyze systems with many states. However practical designs are still out of the grasp of these techniques.

Symbolic methods attempt to validate a system by exploring and checking its behavior under all the possible input stimuli. The advantage of this approach is that a property can be verified for all the possible executions of a circuit and without the need for the designer to create the stimuli. However, these methods have not become mainstream solutions so far, mainly because their computational complexity is very high. As a consequence, they are only able to validate successfully designs only up to a few hundreds latches.

One symbolic exploration approach that has been used is *symbolic model checking*. The basic idea underlying this method is to use BDDs ([3]) to represent all the functions involved in the validation and the set of states that have been visited during the exploration. The primary limitation of this approach is that the BDDs that need to be constructed can grow extremely large, exhausting the memory resources of the simulation host machine and/or causing severe performance degration. In order to overcome this limitation, various solutions have been proposed that try to contain the size of the BDDs involved, for instance: [4, 5, 6].

An alternative approach is *symbolic simulation*. This me-thod verifies a set of scalar tests with a single symbolic vector. Symbolic functions (represented by BDD) are assigned to the inputs and propagated through the circuit to the outputs. (see Figure 1. below). This method is used in [7] and has the advantage that large input spaces can be covered *in parallel* with a single symbolic sweep of the circuit. Again, the bottleneck of the approach lies in the explosion of the BDD representations. Various techniques have been suggested to approximate the functions represented in order to contain the BDDs within reasonable limits: [8, 9].

In this paper, we present a novel technique for symbolic simulation that uses a new, parametric representation for the functions at the sequential elements of the circuit. This representation produces BDDs that are more compact (*i.e.*, have fewer nodes) than the original ones, while at the same time constitute an *exact* representation of the state of the circuit. It is generated dynamically during the simulation exploiting the properties of disjoint support decomposition of a Boolean function.

In the remainder of the paper we review the ideas of symbolic simulation and of disjoint support decomposition. We then present our new techniques for containing the size of the BDDs in symbolic simulation. We conclude by presenting experimental results and directions for future work.

2. BACKGROUND

Let $\mathcal{B}$ denote the set $\{0,1\}$. A symbolic variable is a variable defined over $\mathcal{B}$. 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 $\mathcal{R}(F)$. The ith component of F will be denoted by $f_i : \mathcal{B}^m \rightarrow \mathcal{B}$. We use lower case for single output functions and upper case for multiple output functions.

The *1-cofactor* of a function f w.r.t. a variable v is the function $f_{v=1}$ obtained by substituting 1 for v in f. Similarly, the *0-cofactor* is obtained by substituting 0 for v in f.

We say that a function $f : \mathcal{B}^m \rightarrow \mathcal{B}$ **depends** on a variable x_i iff the Boolean difference $\partial f/\partial x_i = f_{x_i} \oplus f_{\overline{x_i}}$ is not the constant function 0. In the most general case when F is a multiple output function, we say that $F : \mathcal{B}^m \rightarrow \mathcal{B}^n$ depends on a variable x_i, if at least one of its components f_i depends on it.

The **support** of a logic function is the set of variables f depends on and it is indicated by $\mathcal{S}(f)$.Two functions f, g are termed **disjoint-support** if they share no support variables, *i.e.*, $\mathcal{S}(f) \cap \mathcal{S}(g) = \emptyset$. The size of a support set is indicated by $|\mathcal{S}(f)|$.

We assume functions to be represented by their Binary Decision Diagrams. We refer the reader to [3, 10] for a tutorial introduction to BDDs.

A **synchronous logic circuit** is defined by a 6-tuple:

- an ordered set (i_1, $\cdots$, i_m) and (o_1, $\ldots$, o_p) of Boolean input and output symbols,
- an ordered set (s_1, $\ldots$, s_n) of Boolean state symbols,
- next-state function $\delta : S \times I : \mathcal{B}^{n+m} \rightarrow S : \mathcal{B}^n$,
- output function $\lambda : S \times I : \mathcal{B}^{n+m} \rightarrow O : \mathcal{B}^p$,
- and an *initial assignment* S_0 of the state symbols.

The next section reviews the symbolic simulation algorithm, which applies to a synchronous logic circuit. Section 2.2 provides an overview of disjoint support decompositions as we will use them in presenting our approach.

2.1 Symbolic Simulation

Symbolic simulation refers to the iterative symbolic exploration of the state space of a synchronous circuit. The circuit is initialized at time step 0 with the initial assignment S_0 to the state symbols and with the set of variables $IN_{@0} = \{i_{1@0}, \cdots, i_{m@0}\}$ for the input symbols. At each time step k, the expression of the primary outputs and state variables is computed, in terms of the variables in $\{in_{@0}, \cdots, in_{@k}\}$. At the end of the time step, the vector of Boolean functions for the state symbols $S_{@k} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ represents all the states that can be visited by the circuit at step k . Figure 1 shows the flow just described using a time-unrolled version of the circuit.

Bugs are found by checking at every step that the function $OUT_{@k} : \{in_{@0}, \cdots, in_{@k}\} \rightarrow \mathcal{B}^p$ represents a set of legal values for the outputs of the circuit. When an illegal output combination is found, $out_{@k}$ reports all the possible input combinations that expose it. Theoretically the iteration can be repeated indefinitely, although typically the BDDs for $S_{@k}$ and $OUT_{@k}$ outgrow the memory resources available.

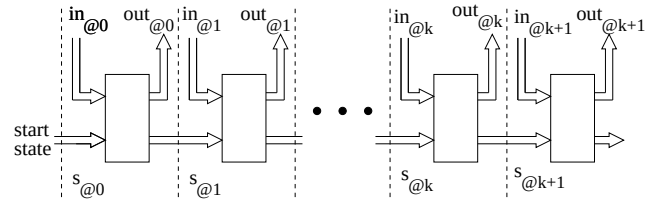

Figure 1: Iterative model of symbolic simulation

2.2 Disjoint Support Decompositions

The disjoint support decomposition of a scalar function $F : \mathcal{B}^m \rightarrow \mathcal{B}$, consists of finding other, simpler functions L and A_i such that:

$$F(x_1, \cdots, x_m) = L(A_1(x_1, \cdots, x_{A_1}), A_2(x_{A_1+1}, \cdots, x_{A_2}), ..)$$

with $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset$, $\forall i, j$.

An exact solution to this problem has been proposed by Ashenhurst [11] and Curtis [12] using *decomposition charts*. This decomposition algorithm has exponential complexity on the number of variables in $\mathcal{S}(f)$. Ashenhurst also proved in [11] that there is a unique decomposition for a function once we pick a canonical representation for associative operations.

More recently, methods based on the BDD representation of a function have been suggested in [13, 14]. In particular, the method in [13] has complexity quadratic in the size of the BDD of the function to decompose, in the worst case.

The disjoint support decomposition can be applied recursively to each of the A_i components leading to a **block tree** representation of a Boolean function as in Figure 2, where each block represents a Boolean function with a single output and inputs that have pair wise disjoint support. The leaves of the tree are the input variables of the function.

Based on the naming convention in [13], each block can represent either an **associative operator** (AND/OR/XOR) or a **prime function** , that is, a complex function that cannot be decomposed any further with disjoint support inputs.

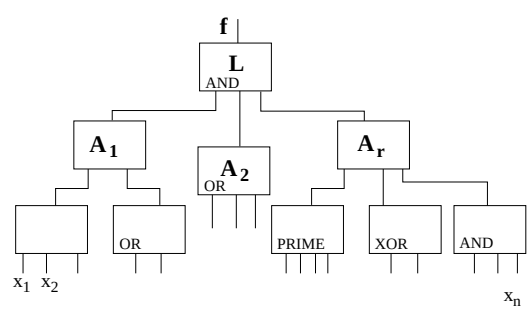

Figure 2: Block tree representation of a Boolean function

3. STATE FUNCTION RE-ENCODING

The method we propose in this paper is based on the observation that at each symbolic simulation step k, it is possible to substitute the state function $S_{@k} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ with a new function $DS_{@k}$ such that $\mathcal{R}(S_{@k}) = \mathcal{R}(DS_{@k})$ without affecting the results of the simulations; namely: 1) The set of outputs that can be generated by the circuit and 2) the set of states the circuit can reach at each cycle. If we can find a suitable function $DS_{@k}$ that also has a smaller BDD representation (*i.e.*, fewer BDD nodes), then we can control the size of the Boolean expression and improve on the performance of symbolic simulation.

We now present various transformations that accomplish this objective. The first technique applies to the state function as a whole, the other two are specific to the decomposition type of a block: either prime function or associative operator. For reasons of readability, the proofs of the theorems presented are reported in the Appendix.

3.1 Reduction at Free Points

By producing the disjoint support decomposition of each component of $S_{@k}$, we obtain a vector of block trees. While each element of the vector has a tree decomposition with no reconvergence, it is now possible that two or more elements intersect at some intermediate node of their trees. An example situation is produced in Figure 3. We call this structure a **decomposition graph**.

From now on, we use the terms *decomposition graph F* and *function F* interchangeably to refer to a multiple output function F. We also drop the subscript $_{@k}$ whenever referring to the state vector S at step k.

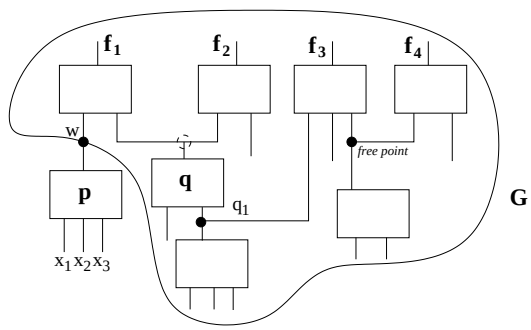

Figure 3: The decomposition of a vector of functions

Definition 1. A **free point** p in a decomposition graph of F is a function corresponding to an output of a block in the graph. It has the property that, if we substitute the sub-graph rooted at the point with a new input variable w, the new function G is disjoint support with the function rooted at p:

$$F(x_1, \cdots, x_m) = G(p, x_{p+1}, \cdots, x_m) \circ p(x_1, \cdots, x_p) \qquad (1)$$

and $\mathcal{S}(G) \cap \mathcal{S}(p) = \emptyset$.

Figure 3 shows three free points with darkened circles. Note that the output of p is a free point since none of the variables in the support of p appears in the support of other parts of the graph. On the other hand, the dashed circle at q is not a free point since, if we split the graph at that node, the functions G and q obtained would still share the input q_1. The following theorem shows that we can use free points to simplify the decomposition graph.

THEOREM 1. *Given a decomposition graph for a multiple output Boolean function $F(x_1, \cdots, x_m) : \mathcal{B}^m \to \mathcal{B}^n$, a free point $p(x_1, \cdots, x_p) : \mathcal{B}^p \to \mathcal{B}$ in it, and the function $G(p, x_{p+1}, \cdots, x_m) : \mathcal{B}^{m-p+1} \to \mathcal{B}^n$, obtained by substituting the function $p()$ with the new input variable p in the graph of F, $\mathcal{R}(F) = \mathcal{R}(G)$.*

Thus, we can substitute all the free points with new variables and generate a new state function G with a smaller representation. A simple traversal of the graph is sufficient to discover all the free points with maximal support, that is, all the free points whose support is not contained in any other free point of the decomposition graph. The transformation of free sub-graphs with new variables produces a new function G, with $|\mathcal{S}(G)| \leq |\mathcal{S}(F)|$.

Example 1. Consider the decomposition graph of Figure 4. Figure 4(a) shows all the free points of the graph. Only the circled free points are maximal. Figure 4(b) shows the new, reduced, function obtained. Note that we can re-use any input variable of a free subgraph as the new variable at the free point. □

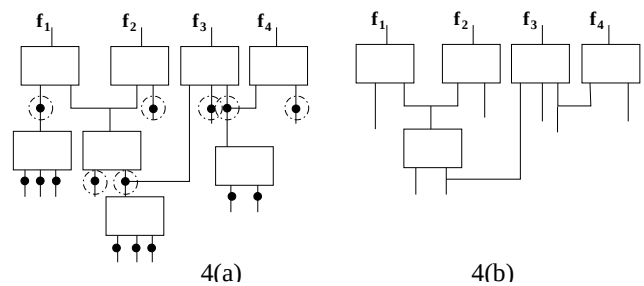

Figure 4: Free points reduction

3.2 Elimination of Prime functions

As mentioned in Section 2.2, each block of a decomposition is either termed a prime function or it is an associative operator. We found that, if a prime function satisfies certain conditions, we can remove it from the decomposition graph, along with all of its subgraph and substitute it with a fresh input variable.

THEOREM 2. *Given a prime function $r(r_1, \cdots, r_r)$ in a decomposition graph F, if all of its inputs, except at most one, are free points, than the decomposition graph G obtained by substituting the new variable r for function $r()$, $F(x_1, \cdots, x_m) = G(r, \cdots, x_m) \circ r(r_1, \cdots, r_r)$ is such that $\mathcal{R}(F) = \mathcal{R}(G)$.*

A possible structure of the graph F is represented in Figure 5(a): All the input to block r are free points, except for r_1. We can then remove the block r and substitute it with a new input variable obtaining the graph in Figure 5(b) without affecting the range of the function. Note that input variables r_2 and r_3 are not needed anymore.

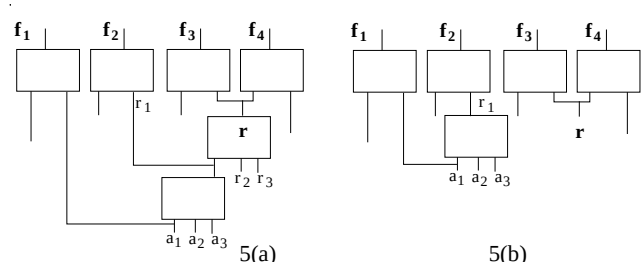

Figure 5. General case for prime function elimination: (a) before and (b) after

Example 2. The testbench s1196 from the IWLS suite contains the blocks reported in Figure 6 in its next state function at step 10 of symbolic simulation. The variables names are just indices corresponding to the variables in the support of the state function. Since the prime function has two inputs that are free points and only one input that has multiple fanout, we can completely eliminate this portion of the graph and just substitute it with an input variable. □

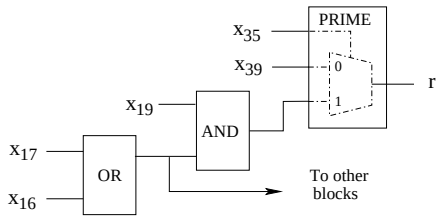

Figure 6: Prime elimination for Example 2.

3.3 Removal of non-dominant variables

Under certain conditions, an input variable can be removed from the decomposition graph without affecting its range.

Example 3. Consider the following 3-outputs function: $f_1 = AND(b,e)$, $f_2 = AND(e, OR(a,b,d))$, $f_3 = XOR(a,c)$.

The range of this function is $\mathcal{B}^3 \setminus \{101, 100\}$. Notice that we can remove the variable a from the function, by assuming it has value 0, without changing the range spanned by $F : f_1 = AND(b,e)$, $f_2 = AND(e, OR(b,d))$, $f_3 = c$ has still range $\mathcal{B}^3 \setminus \{101, 100\}$.

We could do this simplification because the range of the function for $a = 1$ is a subset of the range for $a = 0$. □

Definition 2. An input variable of a decomposition graph has a **non-dominant value 0** iff it fans out only to blocks that are decomposed through OR or XOR associative operators. It has a **non-dominant value 1** iff it fans out only to blocks that are AND or XOR decompositions. Otherwise it does not have a non-dominant value.

Note in particular that a variable may have a non-dominant value 0 and a non-dominant value 1 simultaneously if it fans out only to XOR decompositions. The theorem below shows that in the most general case, a variable that fans out only to associative operators can be removed from the decomposition graph if it has a unique non-dominant value for the whole graph.

THEOREM 3. *If a decomposition graph F has an input variable v with non-dominant value $k \in \{0,1\}$, and each of the blocks (i.e., intermediate single-output functions) that have v in their fanin have at least one* free point *in their fanin, then:* $\mathcal{R}(F) = \mathcal{R}(F_{v=k})$

3.4 Approximating

Even when all of the previous techniques fail, we still want to maintain a compact representation of the state function $S_{@k}$, so that we can keep progressing in the simulation. When the state function exceeds a configurable threshold value, we choose a variable to set to a constant value. The variable with fanout to the maximum number of blocks is selected to maximize our ability to discover a reduced exact parametric representation at the next attempt. In choosing the variable, we only consider decomposition blocks that have input variables in their fanin, the intuition being that these blocks are closer to become free points.

We found experimentally that often, after eliminating a variable by setting it to constant as described, we discovered additional free points or variables with non-dominant values.

4. EXPERIMENTAL RESULTS

We built a symbolic simulator to experiment with our solution and linked it to the CUDD package [15]. At the end of each simulation step we performed the decomposition of the next state symbolic vector and the set of reductions described in Section 3. We set an upper limit of 2500 nodes for the state vector size. We run our simulation on the sequential circuits of the IWLS '89/'93 benchmark suite using a PC based on a 1.7Ghz Pentium processor equipped with 2GB of memory. The results are presented in Table 1.

For each circuit we report the number of primary inputs, primary outputs, sequential elements, combinational logic gates and the number of cycles we run the the simulation.

The next five columns report how many times we were able to apply our reductions of Section 3: **FP** is the number of free point substitutions, **PR** is the number of prime function eliminations, **VD** the number of non-dominant variables removals, **NL** counts the cases where no exact transformation could be applied, but the state vector was within the limit size and **SV** counts the times when we cofactored out a maximally shared variable as described in Section 3.4. Note that during a single simulation step we may apply more than one technique until we reduce the state vector within limits or until there is no exact parametrization possible. The values reported in Table 1 indicate that the conditions that allow condensing the state vector are frequently met in all the circuits.

We then estimated the number of simulation vectors that are run in parallel, as follows: At each simulation step, we produce a fresh set of input symbols at the primary inputs of the circuit. At the sequential inputs, the number of stimuli presented in parallel is equal to the cardinality of the $\mathcal{R}(NS)$. Thus the number of vectors that we simulate in parallel is: $\#vectors = 2^{\#In} \cdot \mathcal{R}(S_k)$. Column **Vectors** reports the log_2 *#vectors*, which corresponds to the equivalent number of input wires that are maintained symbolic at each cycle. The number of traces which we carry forward to the next cycle is given by: $\#traces = 2^{\#In - \#SV} \cdot \mathcal{R}(S_k)$. Again, we report the bit-equivalent value of the number of traces. In order to compute these values, we needed to compute the size of the reached set at every cycle: All the cases where **steps** is less than a 100 have been limited by this computation.

It is clear from the table that we can achieve a high level of parallelism, in the order of tens of bits per cycle, while keeping a low memory profile.

We compared our results with a plain symbolic simulator that limits the size of the state functions by simply cofactoring out as many variables as needed at the end of each cycle: column **plain** reports the number of variables evaluated to constant by this simulator and should be compared to column **SV**. Most often our approach allows us to evaluate to constant fewer variables than a straightforward approach. It's worth noting that there are cases like *s1196* where both simulators do not need to perform any approximation, but the usage of our tecnique allowed us to mantain simpler BDDs and complete the simulation faster.

For the last two columns, we re-run the tests without calculating the size of the state space reached at every cycle and we report the memory profile in MBytes and the execution time needed for running the specified simulations.

5. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for constructing a parametric representation of the state vector during symbolic simulation. The construction exploits the property of disjoint support decomposition of the symbolic state vector. It applies transformations that preserve the range of the vector function while reducing the size of its BDD representation.

Our experiments indicate that the conditions required to perform these transformations often exist. The results show that these transformations are a valuable tool for improving the performance of symbolic simulation while maintaining the quality of its results.

Circuit	In	Out	FF	Gates	steps	FP	PR	VD	NL	SV	Vectors	Traces	plain	Mem	Sec
s1196	14	14	18	790	100	99	5	0	1	0	25.33	25.33	0	22.7	16.60
s1423	17	5	74	830	100	9	0	1	21	144	37.60	35.92	710	35.5	108.95
s298	3	6	14	197	100	16	0	0	20	117	10.35	9.31	45	16.7	67.95
s641	35	24	19	436	100	52	0	21	6	548	43.94	40.71	629	254.0	740.0
s713	35	23	19	480	100	40	0	14	10	531	43.98	41.06	651	154.0	470.4
s953	16	23	29	658	100	59	0	1	3	527	22.95	20.09	745	45.1	106.4
clma	382	82	33	24482	60	38	0	0	21	1	390.56	390.53	1	50.2	28.51
mm4a	7	4	12	310	100	0	0	0	82	40	16.10	15.80	88	30.7	31.11
s38417	38	304	1426	20281	2	2	0	0	0	0	inf	inf	15	33.4	0.05
s38584	38	304	1426	20281	2	1	0	0	1	0	inf	inf	29	42.2	0.06
s5378	35	49	163	3232	2	2	0	0	0	0	54.0	54.0	0	11.3	164.4
s9234	36	39	135	3019	7	1	0	5	1	0	63.49	63.49	0	18.7	350.5
sbc	40	56	27	1143	100	184	0	214	4	932	51.05	46.61	877	66.1	1123.5

Table 1: Experimental results

We are currently working on finding additional configurations that lead to exact transformations and on improving the quality of the fall-back action (measured as ratio between number of BDD nodes eliminated to number of elements of the state set lost) when no exact transformation can be found.

Acknowledgments

The authors would like to thank Maurizio Damiani and David Dill for their valuable suggestions during the development of this work. This research has been supported by Georgia Tech Marco contract #B-12-D00-S5.

6. REFERENCES

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

[2] C. Hansen. Hardware logic simulation by compilation. In *DAC, Proceedings of the Design Automation Conference*, pages 712–715, June 1987.

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

[4] I. Moon, J. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *DAC, Proceedings of the Design Automation Conference*, pages 23–28, June 2000.

[5] K. Ravi and F. Somenzi. High density reachability analysis. In *Proc. ICCAD*, pages 154–158, November 1995.

[6] G. Cabodi, P. Camurati, L. Lavagno, and S.Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *DAC, Proceedings of the Design Automation Conference*, pages 728–733, June 1997.

[7] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. Cosmos: a compiled simulator for mos circuits. In *DAC, Proceedings of the Design Automation Conference*, pages 9–16, June 1987.

[8] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In *Proc. ICCAD*, pages 580–583, November 1999.

[9] V. Bertacco, M. Damiani, and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *DAC, Proceedings of the Design Automation Conference*, pages 391–396, June 1999.

[10] R. Bryant. Symbolic boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[11] R. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, Part I 29, pages 74–116, 1957.

[12] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[13] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *Proc. ICCAD*, pages 78–82, November 1997.

[14] Fpga design by generalized functional decomposition. In T. Sasao, editor, *Logic Synthesis and Optimization*, chapter 11. Kluwer Academic, 1993.

[15] CUDD-2.3.1. *http://vlsi.Colorado.edu/~fabio*.

[16] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Workshop on Automatic Verification Methods for FSM*, volume 407 of *LNCS*, pages 365–373, June 1989.

APPENDIX

Proof of Theorem 1. Consider the function $F(x_1, \cdots x_m)$ and compute its range by splitting on the input variables [16]:

$$\mathcal{R}(F) = \mathcal{R}(F_{x_1=0}) \cup \mathcal{R}(F_{x_1=1})$$

By applying this equation recursively over all the variables $(x_1, \cdots x_p)$ in the support of p, we obtain:

$$\mathcal{R}(F) = \bigcup_{(i_1,\cdots,i_p)\in\mathcal{B}^p} \mathcal{R}(F_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p}) \tag{2}$$

Using Equation (1): $F_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p} = G_{p=i_w}$ where $i_w = p(i_1, \cdots, i_p) \in \{0,1\}$ since p evaluates to a constant. Substituting in Eq.(2) we finally obtain: $\mathcal{R}(F) = \bigcup_{i_w\in\{0,1\}} \mathcal{R}(G_{p=i_w}) = \mathcal{R}(G)$ □

Proof of Theorem 2. We distinguish two cases:

1. All the inputs of the prime block are free points. Then the output of the free block is also a free point and the theorem reduces to the hypothesis of Theorem 1.

2. The prime block r has one input that it is not a free-point, say r_1, without loss of generality. All the other inputs to the prime function: $(r_2, \cdots, r_r)$ are still free points and we can assume that have been reduced to input variables by Theorem 1. In the most general case, r_1 is a single output function of other input variables that are in the support of both G and r: $\mathcal{S}(r_1) = (a_1, \cdots, a_p)$. The function F has then the form:

$$\begin{aligned} F(a_1 \cdots a_p, \cdots r_r, \cdots x_m) &= G(r, a_1 \cdots a_p, r_1, \cdots x_m) \\ &\circ r(r_1, \cdots, r_r) \circ r_1(a_1, \cdots, a_p) \end{aligned} \tag{3}$$

Let's proceed again by computing the $\mathcal{R}(F)$ by recursively splitting on the input variables:

$$\mathcal{R}(F) = \bigcup_{(i_1,\cdots,i_p)\in\mathcal{B}^p} \mathcal{R}(F_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p}) \tag{4}$$

For each different assignment $(i_1, \cdots, i_p)$, r_1 evaluates to a constant value: $i_r = \boldsymbol{r}_1(i_1, \cdots, i_p) \in \{0,1\}$. Substituting the expansion of F as in Eq.(3):

$$F_{a_1=i_1,\cdots,a_p=i_p} = G_{a_1=i_1,\cdots,a_p=i_p,r_1=i_{r_1}} \circ r_{r_1=i_{r_1}} \tag{5}$$

Note that we cannot drop the cofactors w.r.t. the a_i in G because r_1 is not a free point and thus its inputs fan out to other nodes of the graph.

Now, the function $r_{r_1=i_{r_1}}(r_2, \cdots, r_r)$ is a free point and as such it can be substituted by a new free variable r. We show now that it is not possible that $r_{r_1=i_{r_1}}(r_2, \cdots, r_r)$ reduces to a constant for any value of i_{r_1}. Infact, if that was the case r could be expressed as $r = r_1 \otimes r_{res}(r_2, \cdots, r_r)$, where $\otimes$ is either AND or OR and $\mathcal{S}(r_1) \cap \mathcal{S}(r_{res}) = \emptyset$, and r would then have a disjoint support decomposition through an associative operator and would not be a prime function.

By carrying on the substitution $r = r_{r_1=i_{r_1}}(r_2, \cdots, r_r)$, Eq. (5) reduces to: $F_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p} = G_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p}$ which substituted into Eq. (4) proves the theorem. □

Proof of Theorem 3. For a generic function F, we have:

$$\mathcal{R}(F) = \mathcal{R}(F_{v=k}) \cup \mathcal{R}(F_{v=\bar{k}}) \tag{6}$$

We now show that under the conditions specified:

$$\mathcal{R}(F_{v=\bar{k}}) \subseteq \mathcal{R}(F_{v=k}) \tag{7}$$

and Eq.(6) reduces to $\mathcal{R}(F) = \mathcal{R}(F_{v=k})$.

Let's consider first the case where $k = 0$ and let's label each of the functions that have v in their fanin $x(v, p, x_1 \cdots)$, $y(v, q, y_1 \cdots)$, $w(v, r, w_1 \cdots)$ ... where p, q, r ... are the free points in each of them and x_i, y_i, w_i ... are other variables the functions depend on. The x,y,w,... functions can only be OR or XOR decompositions by hypothesis.

We can then express F using the composition of these functions: $F = G(x, y, w \cdots, x_1 \cdots y_1 \cdots) \circ x(v, p, x_1..) \circ y(v, q, y_1..) \cdots$

Note that, in general, x_i, y_i, w_i ... are also in the fanin of G. Let's now compute the two cofactor of F w.r.t. v:

$$F_{v=0} = G(x, y, w \cdots, x_1 \cdots w_w \cdots) \circ x(0, p, x_1..) \circ \cdots$$
$$F_{v=1} = G(x, y, w \cdots, x_1 \cdots w_w \cdots) \circ x(1, p, x_1..) \circ \cdots$$

In order to show the inclusion of the ranges of Eq. (7), we are going to represent each range as a union of ranges by cofactoring the variables in the support of x, y, w, ... one function at a time starting with $x()$:

$$\mathcal{R}(F_{v=0}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(G(x \cdots x_1 \cdots) \circ x(0 \cdots)_{x_1=i_{x_1},\cdots})$$
$$\mathcal{R}(F_{v=1}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(G(x \cdots x_1 \cdots) \circ x(1, \cdots)_{x_1=i_{x_1},\cdots})$$

We distinguish two cases for each x,y,w,... function:

1) **x is a OR decomposition**. When all the $(x_1, \cdots, x_x)$ are zero, for $F_{v=1}$, x evaluates to the constant value 1. For $F_{v=0}$, $x = p$. In all the other cases x evaluates to 1. By grouping all the component ranges so that to distinguish the special case from all the others , we can simplify the expressions:

$$\mathcal{R}(F_{v=0}) = \mathcal{R}(G(p, \cdots 0 \cdots)) \cup \bigcup_{(x_1,\cdots,x_x) \neq \mathbf{0}} \mathcal{R}(G(1, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(F_{v=1}) = \mathcal{R}(G(1, \cdots 0 \cdots)) \cup \bigcup_{(x_1,\cdots,x_x) \neq \mathbf{0}} \mathcal{R}(G(1, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$

It can be easily seen that the first reange for $F_{v=1}$ is a subset of the corresponding range for $F_{v=0}$, while the rest of the expression is identical.

2) **x is an XOR decomposition**. In $F_{v=1}$, $x = XNOR(p, x_1, \cdots, x_x)$. For $F_{v=0}$, $x = XOR(p, x_1, \cdots, x_x)$. We can again group all the component ranges so that to distinguish the cases where $XOR(x_1, \cdots, x_x) = 0$ from the ones where $XOR(x_1, \cdots, x_x) = 1$:

$$\mathcal{R}(F_{v=0}) = \bigcup_{XOR(x_1,\cdots,x_x)=0} \mathcal{R}(G(p, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \cup \bigcup_{XOR(x_1,\cdots,x_x)=1} \mathcal{R}(G(\bar{p}, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(F_{v=1}) = \bigcup_{XOR(x_1,\cdots,x_x)=0} \mathcal{R}(G(\bar{p}, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \cup \bigcup_{XOR(x_1,\cdots,x_x)=1} \mathcal{R}(G(p, y, \cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$

And it can be observed that the two components of each expression match. It follows: $\mathcal{R}(F_{v=0}) = \mathcal{R}(F_{v=1})$.

This procedure can be applied recursively for each of the other functions y, ..., by computing and grouping all the cofactors for the sets of input variables $(y_1 \cdots y_y)$, ...

For the case where $k = 1$, the functions x,y,w, ... can now only be AND or XOR decompositions. The proof can be obtained by substituting AND for OR and 1 for 0 in the proof just discussed. Finally for the case where the input variable v has both a non- dominant value 0 and 1, we can just use any of the two value- specific proofs. □