Sequential Logic Synthesis Using Symbolic Bi-Decomposition

Victor N. Kravets
IBM TJ Watson Research Center
Yorktown Heights, NY
kravets@us.ibm.com

Alan Mishchenko
Department of EECS
University of California, Berkeley
alanmi@eecs.berkeley.edu

ABSTRACT
This paper uses under-approximation of unreachable states of a design to derive incomplete specification of combinational logic. The resulting incompletely-specified functions are decomposed to enhance the quality of technology-dependent synthesis. The decomposition choices are computed implicitly using novel formulation of symbolic bi-decomposition that is applied recursively to decompose logic in terms of simple primitives. The ability of BDDs to represent compactly certain exponentially large combinatorial sets helps us to implicitly enumerate and explore variety of decomposition choices improving quality of synthesized circuits. Benefits of the symbolic technique are demonstrated in sequential synthesis of publicly available benchmarks as well as on the realistic industrial designs.

1. INTRODUCTION AND MOTIVATION

Due to recent advances in verification technology [2] circuit synthesis of semiconductor designs no longer has to be limited to logic optimization of combinational blocks. Nowadays logic transformations may involve memory elements which change design's state encodings or its reachable state-space, and still be verified against its original description. In this paper we focus on a more conservative synthesis approach that changes sequential behavior of a design only in unreachable states, leaving its intended "reachable" behavior unchanged. Unreachable states are used to extract incomplete specification of combinational blocks, and are applied as *don't cares* during functional decomposition to improve circuit quality.

To implement combinational logic of a design we rely on a very simple, yet complete, form of functional decomposition commonly referred to as *bi-decomposition*. In general, for a given completely specified Boolean function $f(\mathbf{x})$ its bi-decomposition has form

$$f(\mathbf{x}) = h(g_1(\mathbf{x}_1), g_2(\mathbf{x}_2))$$

where h is an arbitrary 2-input Boolean function. This decomposition is not unique and its quality varies depending on selected subsets $\mathbf{x}_1$ and $\mathbf{x}_2$ that form possibly overlapping partition of $\mathbf{x}$. The problem of finding good bi-decomposition has been studied in [1,8,18,19].

The main contribution of this paper is symbolic formulation of bi-decomposition for incompletely specified functions. The bi-decomposition is used as main computational step in the prototype sequential synthesis tool, and is applied recursively to implement logic of combinational blocks whose incomplete specification is extracted from unreachable states of a design. Our symbolic formulation of bi-decomposition finds all feasible solutions and picks the best ones, without explicit enumeration.

Computation of variable partitions in our symbolic formulation of bi-decomposition favors implicit enumeration of decomposition subsets. They are represented compactly with a binary decision diagram (BDD) [4], and are selected based on optimization objective. Unlike previous approaches (e.g. [1,23]) that rely on BDDs, the decomposition is not checked explicitly for a variable partition, and is solved implicitly for all feasible partitions simultaneously utilizing fundamental property of BDDs to share partial computations across subproblems. Thus, no costly enumeration that requires separate and in independent decomposability checks is needed. The technique was also used to tune greedy bi-decomposition when handling larger functions.

To overcome limitations of explicit techniques authors in [15] proposed solution that uses a satisfiability solver [11]. Their approach is based on proving that a problem instance is unsatisfied. The unsatisfiable core is then used to greedily select partition of variables that induces bi-decomposition. Authors demonstrate the approach to be efficient in runtime, when determining existence of non-trivial decomposition. The experimental results on a selected benchmark set however, are primarily focused on the existence of decomposition and do not offer a qualitative synthesis data.

The problem of using unreachable states of a design to improve synthesis and verification quality has been studied before in various contexts. In general, these algorithms either avoid explicit computation of unreachable states, or first compute them in pre-optimization stage. Approaches that do not explicitly compute unreachable states are mostly limited to incremental structural changes of a circuit, and

978-3-9810801-5-5/DATE09 © 2009 EDAA

rely on ATPG environment or induction [12,7,9] to justify a change. In contrast, approaches that pre-compute subsets of unreachable states treat them as external don't cares [20] for resynthesis of combinational logic blocks [14,5]. In this paper we adopt the later approach as it offers more flexibility in logic re-implementation through functional decomposition.

The paper is organized as follows. In Section 2 we introduce preliminary constructs. Section 3 describes bi-decomposition existence requirements. They are used in Section 4 to formulate implicit computation of decomposition. Implementation details are described in Section 5. Experimental results are given in Section 6. Section 7 gives conclusions and possible directions for future work.

2. PRELIMENARY CONSTRUCTS

Basic constructs used by synthesis algorithms of the paper are introduced in this section.

2.1 "Less-than-or-equal" relation

Computational forms constructed in this paper rely on the partial order relation between Boolean functions. Given functions $f(\mathbf{x})$ and $g(\mathbf{x})$, $f(\mathbf{x}) \le g(\mathbf{x})$ indicates that $f(\mathbf{x})$ precedes $g(\mathbf{x})$ in the order. This "less-than-or-equal" relation between the two functions can be expressed by one of the following three equivalent forms:

$$[f(\mathbf{x}) \Rightarrow g(\mathbf{x})] \equiv [f(\mathbf{x}) \le g(\mathbf{x})] \equiv [\overline{f(\mathbf{x})} + g(\mathbf{x}) = 1]$$

The relation imposes *consistency* constraint on constructed computational forms.

It allows us represent incompletely specified Boolean functions in terms of intervals [3], defined as

$$[l(\mathbf{x}), u(\mathbf{x})] = \{f(\mathbf{x}) \mid l(\mathbf{x}) \le f(\mathbf{x}) \le u(\mathbf{x})\} .$$

An interval represents a set of completely specified functions using its two distinguished members $l(\mathbf{x})$ and $u(\mathbf{x})$, known as upper and lower bounds respectively. It is non-empty (or consistent) if and only if $l(\mathbf{x}) \le u(\mathbf{x})$ is satisfied.

Example 2.1 Consider interval $[\bar{x}y, x+y]$ which represents an incompletely specified function. It is composed of four completely specified functions: $\bar{x}y$, y, $x \oplus y$, and $x+y$. Each of them has a don't care set represented by function x. ❑

Application of existential quantification $\exists$ and universal quantification $\forall$ to lower and upper bounds of the interval enables convenient selection of its member functions that are *vacuous*, i.e. independent, in certain variables.

Example 2.2 Consider abstraction of x from the interval in Example 2.1: $[\exists x(\bar{x}y), \forall x(x+y)]$. The abstraction yields non-empty interval that is composed of a unique function that is vacuous in x: $[y,y]$. Abstraction of y however, results into empty interval since the relation between its lower and upper bounds is not satisfied: $[\bar{x}, x]$ is empty. ❑

2.2 Parameterized abstraction

To determine subsets of variables whose abstraction preserves consistency of a symbolic statement (or a formula) we use *parameterized abstraction* construct. It parameterizes computational form with a set of auxiliary variables **c** that are used to guide variable abstraction decisions. An assignment to **c** effects consistency of a computational form, and thereby determines feasibility of abstracting a corresponding variable subset.

We use the "if-then-else" operator $ITE(c, x, y)$ to encode effect of quantifying variable subsets from a formula. Defined as $cx + \bar{c}y$, the operator selects between variables x and y depending on value of c. As stated, it can be used to parameterize signal dependencies in a Boolean function. It can be also generalized to the selection between functions. In particular, $ITE(c, f(\mathbf{x}), \forall x f(\mathbf{x}))$ encodes a decision of universal quantification of x from $f(\mathbf{x})$; similarly for the existential quantification.

Example 2.3 We can parameterize effect of abstracting the $[\bar{x}y, x+y]$ interval variable x using the *ITE* operator and auxiliary variable $\boldsymbol{c}$ as

$$[ITE(c, \exists x(\bar{x}y), \bar{x}y), ITE(c, \forall x(x+y), x+y)]$$

or equivalently $[cy + \bar{c}\bar{x}y, cy + \bar{c}(x+y)]$. The parameterization can be repeated for variable y in a similar way. ❑

3. BI-DECOMPOSITION OF INCOMPLETELY SPECIFIED FUNCTIONS

This sections gives formal statement of bi-decomposition over 2-input decomposition primitives, namely *or* and *xor*.

3.1 Or decomposition

For a completely specified function $f(\mathbf{x})$, the decomposition of this type is described in terms of equation below:

$$f(\mathbf{x}) = g_1(\mathbf{x}_1) + g_2(\mathbf{x}_2) \quad (1)$$

When function is incompletely specified with interval $[l(\mathbf{x}), u(\mathbf{x})]$ we need to make sure that *or*-composition $g_1 + g_2$ is a member function of the interval.

Let $\underline{\mathbf{x}}_1$ and $\underline{\mathbf{x}}_2$ be signal subsets in which decomposition functions g_1 and g_2 are respectively vacuous, i.e. are functionally independent. (The underscore in $\underline{\mathbf{x}}_i$ indicates that the computed g_i is independent in these variables.) Vacuous in $\underline{\mathbf{x}}_1$ function g_1 must not exceed largest member $u(\mathbf{x})$ in all its minterm points, independent of $\underline{\mathbf{x}}_1$, i.e. relation $g_1(\mathbf{x}_1) \le \forall \underline{\mathbf{x}}_1 u(\mathbf{x})$ must be satisfied. Otherwise g_1 is either not contained in the interval or it is not independent of $\underline{\mathbf{x}}_1$. Similarly, $g_2(\mathbf{x}_2) \le \forall \underline{\mathbf{x}}_2 u(\mathbf{x})$ must hold. Thus, $\forall \underline{\mathbf{x}}_1 u(\mathbf{x})$ and $\forall \underline{\mathbf{x}}_2 u(\mathbf{x})$ give upper bounds on $g_1(\mathbf{x}_1)$ and $g_2(\mathbf{x}_2)$. To ensure that the selection of g_1 and g_2 is "large enough" the following must hold:

$$l(\mathbf{x}) \le \forall \underline{\mathbf{x}}_1 u(\mathbf{x}) + \forall \underline{\mathbf{x}}_2 u(\mathbf{x}) \quad (2)$$

The *or*-composition does not exceed u universally due to "reducing" effect of $\forall$ on u. Thus, we can determine exist-

ence of the bi-decomposition limiting check to relation in (2). This check provides necessary and sufficient condition for the existence of *or* decomposition, and is a re-statement of the result from [16].

And Decomposition. As indicated in [16], *and* decomposition of f can be obtained from *or* decomposition utilizing dual property of the two gates. For an incompletely specified function $[l, u]$ we can find complemented g_1 and g_2 by establishing *or* decomposability of the interval complement, derived as $\overline{[l, u]} = [\bar{u}, \bar{l}]$.

3.2 Xor decomposition

We first describe *xor*-decomposability condition for a completely specified function $f(\mathbf{x})$. The existence condition for the *xor* decomposition

$$f(\mathbf{x}) = g_1(\mathbf{x}_1) \oplus g_2(\mathbf{x}_2) \tag{3}$$

requires partitioning of $\mathbf{x}_1$ and $\mathbf{x}_2$ into finer subsets. Let $\underline{\mathbf{x}}_1$ and $\underline{\mathbf{x}}_2$ be subsets of variables in which g_1 and g_2 are respectively vacuous; and let $\mathbf{x}_3$ be a set of variables on which both decomposition functions are depending. We can then state necessary and sufficient condition for the existence of *xor* decomposition as follows:

Proposition 3.1 *Xor* bi-decomposition

$$f(\mathbf{x}) = g_1(\underline{\mathbf{x}}_2, \mathbf{x}_3) \oplus g_2(\underline{\mathbf{x}}_1, \mathbf{x}_3) \tag{4}$$

exists if and only if

$$f(\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3) \neq f(\underline{\mathbf{y}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3) \tag{5}$$

$$\Downarrow$$

$$\forall \underline{\mathbf{y}}_2 [f(\underline{\mathbf{x}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3) \neq f(\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3)] \ . \tag{6}$$

We derived conditions in the above proposition when analyzing library requirements for an advanced technology [13]. In [15] authors recently and independently stated analogous proposition in terms of the unsatisfiability problem. For that reason, we show correctness of the above proposition giving only an information-theoretical argument: For (4) to hold, it must be that all onset/offset minterms in f that cannot be distinguished by g_1 (relation (5)) must be distinguished by g_2 (relation (6)).

For an incompletely specified function $[l(\mathbf{x}), u(\mathbf{x})]$ the consistency constrain (5) $\Rightarrow$ (6) of Proposition 3.1 changes to:

$$[l(\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3) \neq l(\underline{\mathbf{y}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3)] \wedge [h(\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3) \neq h(\underline{\mathbf{y}}_1, \underline{\mathbf{x}}_2, \mathbf{x}_3)]$$

$$\Downarrow$$

$$\forall \underline{\mathbf{y}}_2 [[l(\underline{\mathbf{x}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3) \neq h(\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3)] \vee [h(\underline{\mathbf{x}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3) \neq l(\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3)]]$$

The above statement extends containment relation (5) $\Rightarrow$ (6) by reducing lower bound (5) and increasing upper bound (6) as much as possible. The relation provides the condition for *xor* bi-decomposition of incompletely specified functions, previously unsolved problem [15].

4. PARAMETERIZED DECOMPOSITION

The Section 3 decomposition checks assume that the $\mathbf{x}_1$ and $\mathbf{x}_2$ subsets are pre-determined. Finding such feasible subsets however, may not be straight forward and depending on the objectives could potentially require an exponential search if performed explicitly. Our solution to the problem is to perform the search implicitly, formulating the problem symbolically and solving it by leveraging the capability of binary-decision diagrams to compactly represent certain combinatorial subsets.

Or Parameterization. To parameterize a set of variables the operator is applied iteratively, one variable at a time:

$U \leftarrow u$;
for each $x \in \mathbf{x}$ **do**
  $U \leftarrow ITE(c_x, U, \forall x U)$;

This gives function $U(\mathbf{c}, \mathbf{x})$. It encodes the effect of abstracting all variable subsets from u, where variable x is abstracted iff $c_x = 0$.

The parameterized function $U(\mathbf{c}, \mathbf{x})$ can be used in equation (2) to encode possible supports to g_1 and g_2 in terms of the decision variables $\mathbf{c}_1$ and $\mathbf{c}_2$:

$$l(\mathbf{x}) \leq U_1(\mathbf{x}, \mathbf{c}_1) + U_2(\mathbf{x}, \mathbf{c}_2) \tag{7}$$

For any feasible assignment to $\mathbf{c}_1$ and $\mathbf{c}_2$, the above relation must hold universally, irrespective of values on $\mathbf{x}$. Thus, computational form

$$Bi(\mathbf{c}_1, \mathbf{c}_2) \equiv \forall \mathbf{x} [\overline{l(\mathbf{x})} + U_1(\mathbf{x}, \mathbf{c}_1) + U_2(\mathbf{x}, \mathbf{c}_2)] \tag{8}$$

yields a characteristic function of all feasible supports for g_1 and g_2.

We illustrate potentially scalable nature of BDDs to handle computation in (6) decomposing multiplexer function for its various support sizes:

Mux width		*Bi* computation		Best partition	
control	data	BDD size	time, sec	$(\lvert\mathbf{x}_1\rvert, \lvert\mathbf{x}_2\rvert)$	# choices
2	4	23	0.00	(4,4)	6
3	8	43	0.01	(7,7)	70
4	16	79	0.09	(12,12)	12870
5	32	147	1.35	(21,21)	6E8.0
6	64	279	20.56	(38,38)	1.8E18

The above table gives results of the computation in terms multiplexer widths, BDD size and time required to compute Bi, and the best support sizes of g_1 and g_2. As the table suggests, the amount of resources required in computation grows moderately for smaller problem instances, and is tolerable even for a larger function. More detailed discussion on selecting best $\mathbf{x}_1$ and $\mathbf{x}_2$ is given later, in Section 5.

Xor Parameterization. To simplify presentation we compute characteristic function of all feasible support partitions for a completely specified function. As before, encoding of possible supports for g_1 and g_2, is performed using two

sets of auxiliary variables $\mathbf{c}_1$ and $\mathbf{c}_2$. Using $\mathbf{c}_1$, relation (5) is transformed into $f(\mathbf{x}) \neq F_1(\mathbf{x}, \mathbf{y}, \mathbf{c}_1)$, where F is derived from $f(\mathbf{x})$ replacing each of its variables x_i with $ITE(c_{1i}, x_i, y_i)$. Similarly, part $f(\underline{\mathbf{x}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3)$ from (6) is parameterized with $\mathbf{c}_2$ to construct $F'_2(\mathbf{x}, \mathbf{y}, \mathbf{c}_2)$. It encodes selection of vacuous variables for g_2. The last component $f(\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \mathbf{x}_3)$ is transformed into $F''_2(\mathbf{x}, \mathbf{y}, \mathbf{c}_1, \mathbf{c}_2)$, replacing each variable in $f(\mathbf{x})$ with $ITE(c_{1i} \cdot c_{2i}, x_i, y_i)$. Universally abstracting $\mathbf{x}$ and $\mathbf{y}$ variables gives representation of all feasible supports for g_1 and g_2:

$$Bi(\mathbf{c}_1, \mathbf{c}_2) \equiv \forall \mathbf{x}, \mathbf{y}[(f \neq F_1) \Rightarrow (F'_2 \neq F''_2)]$$

We compare implicit computation of decomposition choices to a greedy algorithm for the *xor* decomposition, used by authors in [16,22]. Starting from a seed partition, the algorithm greedily extends support subsets calling *xor* decomposability check in its inner loop. Although efficient in general, the check has potentially formidable runtime. The profile of its behavior on a 16-bit adder is given in the table below; it is compared against our implicit computation:

Output			Time, sec	
sum bit	# inputs	best part.	implicit	[16] check
s2	7	(2,5)	0.01	0.00
s4	11	(2,9)	0.06	0.13
s6	15	(2,13)	0.12	4.44
s8	19	(2,17)	0.13	71.05
⋮	⋮	⋮	⋮	⋮
s16	33	(2,31)	0.42	time out

For a subset of sum-bit functions the table lists runtime for both techniques. (The *best part.* column gives data generated by our implicit enumeration of feasible partitions.) Although not typical, it is interesting that where a rather efficient greedy check times out after an hour, an implicit exhaustive computation takes only 0.42 sec.

In general, we can use best partition produced by the exhaustive implicit computation to evaluate and tune greedy algorithm, or to improve some other approximate technique.

5. IMPLEMENTATION DETAILS OF SEQUENTIAL SYNTHESIS

This section describes a sequential synthesis flow that first extracts incompletely specified logic accounting for unreachable states in a design, and then uses bi-decomposition to synthesize technology-independent circuit.

5.1 Extraction of incompletely specified logic

Unreachable states of a design form don't cares for the combinational logic. Due to the complexity of computing unreachable states even in designs of modest size, incompletely specified combinational logic is extracted with respect to an approximation of unreachable states. Unlike other partitioning approaches that try to produce a good approximation of unreachable states in reasonable time [10, 17], our objective is to compute a good approximation with respect to support of individual functions. A similar approach to approximate unreachable states using induction was proposed in [6].

We perform state-space exploration with forward reachability analysis for overlapping subsets of registers. These subsets are selected using structural dependence of next-state and primary outputs on the design latches. The selection tries to create partitions maximizing accuracy of reachability analysis for present-state signals $supp\_ps(f)$ output function f. In particular, the partitioning tries to meet following goals:

- for each function f, present-state inputs $supp\_ps(f)$ are represented in at least one partition
- each partition selects additional logic to maximize accuracy of reachability analysis.

After completing reachability analysis for a partition, incomplete specification of signals that depend on its latches becomes available in the form of a function interval.

5.2 Exploring decomposition choices

The characteristic function Bi gives all feasible supports for decomposition functions. Since the provided variety of choices could be very large, we restrict them to a subset of desired solutions. The restriction targets minimization and balanced selection of supports in decomposition functions. It is achieved symbolically, as described below.

Let T^n_k be a function representing combinatorial subsets $\binom{n}{k}$. This function has compact representation in terms of BDDs. Assume that $n = |\mathbf{x}|$ and k denotes support size of a function. For a desired support size $k_1 = |\mathbf{x}_1|$ of g_1, and $k_2 = |\mathbf{x}_2|$ of g_2 we can then determine existence of a particular decomposition constraining Bi with a corresponding solution space:

$$Bi(\mathbf{c}_1, \mathbf{c}_2) \cdot T^n_{k_1}(\mathbf{c}_1) \cdot T^n_{k_2}(\mathbf{c}_2)$$

If the resulting function is not empty, then desired decomposition exists. To target balanced decomposition of a function we seek feasible k_1 and k_2 minimizing $max(k_1, k_2)$. Minimization of both k_1 and k_2 balances supports $\mathbf{x}_1$ and $\mathbf{x}_2$, and favors their disjoint selection. (This is in contrast to [15], where different cost measures are used for each of the objectives.)

To complete decomposition of a function we need to find decomposition functions g_1 and g_2. For the *or* decomposition possible functions g_1 and g_2 can be deduced directly from the corresponding existence condition (2), universally quantifying our variables in which g_1 and g_2 are vacuous. To construct *xor* decomposition functions we use algorithm from [16].

5.3 Synthesis algorithm

Our synthesis algorithm selectively re-implements functions

```
create latch partitions of a design;
selectively collapse logic;
while (more logic to decompose) {
  select a signal and its function f(x);
  retrieve unreachable states u(x);
  abstract vars from interval  [f·ū, f+u] ;
  apply bi-decomposition to interval;
}
```

Fig. 1. Logic re-decomposition loop.

of circuit signals relying on bi-decomposition of extracted incompletely specified logic. The pseudocode code in Figure 1 captures general flow of the algorithm; it is described below.

The algorithm first creates overlapping partitions of a design. These partitions are formed according to Section 5.1, and are typically limited to 100 latches. Additional connectivity cost measures are used to control size of a partition. For each partition computation of unreachable states is delayed until being requested by a function that depends on its present-state signals. BDDs for computed reachable states are then stored in a separate node-space for each partition. When retrieving unreachable states for a given support, their conjunctive approximation is brought together to a common node-space.

To re-decompose logic of a design the algorithm first creates functional representation for selected signals in terms of their cone inputs, or in terms of other intermediate signals. The decision on whether to select a signal is driven by an assessed impact of bi-decomposition on circuit quality: if it has potential to improve variable partition, logic sharing, or timing over existing circuit structure, then signal is added to a list of re-decomposition candidates.

The logic of candidate signals is processed in topological order until it is fully implemented with simple primitives. This processing constitutes main loop of the algorithm. After a signal and its function $f(\mathbf{x})$ in the loop is selected, a set of unreachable states $u(\mathbf{x})$ is retrieved. This set is derived from reachability information of partitions that $f(\mathbf{x})$ depends on.

Before applying bi-decomposition to the incompletely specified function $[f(\mathbf{x}) \cdot \overline{u(\mathbf{x})}, f(\mathbf{x}) + u(\mathbf{x})]$, the algorithm tries to abstract some of the interval variables while keeping it consistent; this eliminates redundant inputs. The bi-decomposition is then applied targeting potential logic sharing and balanced partition of $\mathbf{x}$, as described in Section 5.2. From a generated set of choices, partition that best-improves timing and logic sharing is selected. Figure 2 illustrates bi-decomposition that benefits from logic sharing. The transformation re-uses logic of g_1 which was present in the network but was not in the fanin of f.

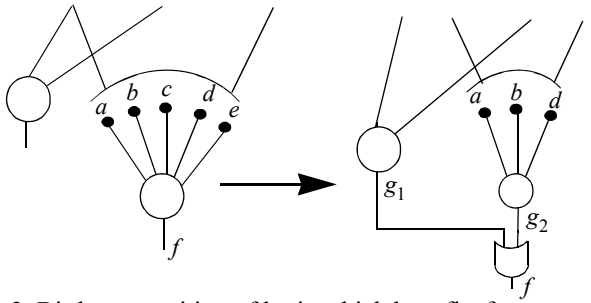

Fig. 2. Bi-decomposition of logic which benefits from re-using existing for g_1 logic.

6. EXPERIMENTAL EVALUATION

From a suite of publicly available benchmarks we selected a subset of sequential circuits and assessed effect of unreachable states on bi-decomposition. Three types of bi-decomposition were applied to functions of their output and next-state logic: *or*, *and*, and *xor*. They are evaluated in terms of their ability to reduce maximum support of functions g_1 and g_2. Experiments with and without reachable state-space analysis were performed.

The experimental results are given in Table 1. The table first lists circuit name, along with its corresponding number of inputs/outputs and latches. Each circuit was structurally pre-processed to remove cloned, dead and constant latches. The *#dec* column gives number of functions for which non-trivial decomposition was identified. The average ratio between maximum support sizes of g_1 and g_2, and support size of the function being decomposed is given in *avg. reduct.* column. Note that the reduction of less than 0.5 (as in s713 and s838) indicates that both g_1 and g_2 tend to be vacuous in some of the variables.

The results are collected for two experiments: with and without state-space information. The log_2 of computed reachable states is also listed in the table. Computed average reduction ratios suggest that decomposability of a function improves as the number of unreachable states gets larger. The unreachable states did not contribute much to s5378

Table 1: Application of bi-decomposition to functions of next-state and output logic (without and with state analysis).

Name	Original circuit		No states		With states		
	inp/outp	latches	#dec.	avg. reduct	log_2 states	#dec.	avg. reduct
s344	10/11	15	18	0.781	12	18	0.634
s526	3/6	21	21	0.775	14	21	0.556
s713	36/23	19	40	0.652	11	40	0.453
s838	36/2	32	33	0.540	5	33	0.088
s953	17/23	29	29	0.607	13	29	0.565
s1269	18/10	37	39	0.672	31	39	0.671
s5378	36/49	163	145	0.609	125	145	0.603
s9234	36/39	145	97	0.754	141	97	0.774
Average reduction:				0.673			0.54

Table 2: Results of applying bi-decomposition in synthesis of industrial circuits.

Name	Original circuit			Pre-processed		Fig. 1 algor.	
	inp/outp	latches	ands	area	delay	area	delay
seq4	108/202	253	1845	3638	44.8	2921	41.9
seq5	66/12	93	925	1951	47.2	1807	41.6
seq6	183/74	142	811	1578	34.9	1487	36.0
seq7	173/116	423	3173	6435	52.4	5348	48.3
seq8	140/23	201	2922	6183	50.1	5427	48.8
seq9	212/124	353	3896	8250	56.0	6938	45.2
Average reduction:						0.88	0.94

largely because its logic is highly decomposable even in the absence of state-space information. The runtime to compute reachable states for each of the circuits did not exceed one minute, requiring at most few seconds for circuits with 32 or less latches. Computation of bi-decomposition for was limited to one minute per circuit.

We evaluate our Figure 1 algorithm synthesizing technology-independent netlists for a set of macro-blocks of a high-performance industrial design. Results of the netlists optimized with bi-decomposition are given in Table 2. First four columns list general parameters of each circuit, including number of gates it has in its `and/inv` expansion. The circuits were first pre-processed using our in-house tool, by optimizing it against publicly available mcnc.genlib library.

An implementation of the Figure 1 algorithm was then applied to improve each of the circuits. Columns *Pre-processed* and *Fig. 1 algor.* compare area (which corresponds to the number of literals) and delay (estimated with a load-dependent model) of mapped netlists before and after running our algorithm. The additional area and timing savings are due to the algorithm, with the average area and delay reductions of 0.88 and 0.94 respectively. We attribute these gains to the algorithm's ability implicitly explore reach arsenal of decomposition choices during bi-decomposition. Optimization of each circuit was completed within four minutes of runtime.

7. CONCLUSIONS AND FUTURE WORK

Extraction of incompletely specified logic using under-approximation of unreachable states in sequential designs offers valuable opportunity for reducing the circuit complexity. We developed a novel formulation of symbolic bi-decomposition and showed that the extracted logic has better implementation, with substantial area and delay improvements. The introduced symbolic bi-decomposition computes decomposition choices implicitly, and enables their efficient subsetting using BDDs. Selecting best decomposition patterns during synthesis, we improved circuit quality of publicly available and realistic industrial design. We are currently working on ways to further maximize logic sharing through bi-decomposition, and to apply it in a re-synthesis loop of well-optimized designs.

8. REFERENCES

[1] R. L. Ashenhurst. The decomposition of switching functions. *Ann. Computation Lab.*, Harvard University, vol. 29, pages 74-116, 1959.

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Proc. ICCD*, Oct. 2006, pp. 259-266.

[3] F. M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, Boston, 1990.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE TC*, C-35(6):677–691, August 1986.

[5] M. L. Case, A. Mishchenko, and R. K. Brayton. Inductive finding a reachable state-space over-approximation. In *IWLS*, June 2006, pp. 172-179.

[6] M. L. Case, A. Mishchenko, and R. K. Brayton. Cut-based inductive invariant computation. In *IWLS*, June 2008, pp. 172-179.

[7] M. L. Case, V. N. Kravets, A. Mishchenko, and R. K. Brayton. Merging nodes under sequential observability. In *Proc. DAC*, June 2008, pp. 540-545.

[8] J. Cortadella. Timing-driven logic bi-decomposition. *IEEE Trams. on CAD,* 22(6):675-685, 2003.

[9] K. T. Cheng and L. A. Entrena. Sequential logic optimization by redundancy addition and removal. In *Proc ICCAD*, Nov. 1993, pp. 310-315.

[10] J. Cho, G. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state decomposition for approximate FSM traversal based on circuit analysis. *IEEE TCAD,* 15(12):1451-1464, 1996.

[11] N. Een and N. Sorensson. An extensible SAT-solver. In *Proc. SAT*, 2003, pp. 502-518.

[12] C. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE TCAD*, July 2000, pp. 814-819.

[13] V. N. Kravets et al. Automated synthesis of limited-switch dynamic logic (LSDL) circuits. *Prior Art Database (ip.com)*, March 2008.

[14] B. Lin, H. Touati, and R. Newton. Don't care minimization of multi-level sequential networks. In *Proc ICCAD*, Nov. 1990, pp. 414-417.

[15] R.-R. Lee, J.-H. Jiang, and W.-L. Hung. Bi-decomposing large Boolean functions via interpolation and satisfiability solving. In *Proc. DAC*, June 2008, pp. 636-641.

[16] A. Mishchenko, B. Steinbach and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proc. DAC*, June 2001, pp. 103-108.

[17] A. Mishchenko, M. L. Case, R.K. Brayton, and S. Jang. Scalable and scalable-verifiable sequential synthesis. In *Proc. ICCAD*, Nov. 2008, pp.234-241.

[18] J. P. Roth and R. Karp. Minimization over boolean graphs. *IBM J. Res. and Develop.*, 6(2):227–238, April 1962.

[19] T. Sasao and J. Butler. On bi-decomposition of logic functions. In *IWLS*, June 1997.

[20] H. Savoj and R. K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *Proc. DAC*, June 1990, pp. 297-301.

[21] T. Stanion and C. Sechen. Quasi-algebraic decomposition of switching functions. In *Proc. 16th Conference on Advance Research in VLSI*, September 1998, pp. 358-367.

[22] B. Steinbach and A. Wereszczynski. Synthesis of multi-level circuits using EXOR-gates. In *Proc. IFIP WG 10.5 - Workshop on Application of the Reed-Muller Expansion in Circuit Design*, Japan, 1995, pp. 161-168.

[23] C. Yang, M. Cieselski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proc. DAC*, June 2000, pp. 92-97.