Towards Parallel Boolean Functional Synthesis

S. Akshay[1(✉)], Supratik Chakraborty[1(✉)],
Ajith K. John[2], and Shetal Shah[1(✉)]

[1] IIT Bombay, Mumbai, India
{akshayss,supratik,shetals}@cse.iitb.ac.in
[2] HBNI, BARC, Mumbai, India

Abstract. Given a relational specification $\varphi(X, Y)$, where X and Y are sequences of input and output variables, we wish to synthesize each output as a function of the inputs such that the specification holds. This is called the Boolean functional synthesis problem and has applications in several areas. In this paper, we present the first parallel approach for solving this problem, using compositional and CEGAR-style reasoning as key building blocks. We show by means of extensive experiments that our approach outperforms existing tools on a large class of benchmarks.

1 Introduction

Given a relational specification of input-output behaviour, synthesizing outputs as functions of inputs is a key step in several applications, viz. program repair [14], program synthesis [28], adaptive control [25] etc. The synthesis problem is, in general, uncomputable. However, there are practically useful restrictions that render the problem solvable, e.g., if all inputs and outputs are Boolean, the problem is computable in principle. Nevertheless, functional synthesis may still require formidable computational effort, especially if there are a large number of variables and the overall specification is complex. This motivates us to investigate techniques for Boolean functional synthesis that work well in practice.

Formally, let X be a sequence of m input Boolean variables, and Y be a sequence of n output Boolean variables. A relational specification is a Boolean formula $\varphi(X, Y)$ that expresses a desired input-output relation. The goal in Boolean functional synthesis is to synthesize a function $F : \{0,1\}^m \to \{0,1\}^n$ that satisfies the specification. Thus, for every value of X, if there exists some value of Y such that $\varphi(X, Y) = 1$, we must also have $\varphi(X, F(X)) = 1$. For values of X that do not admit any value of Y such that $\varphi(X, Y) = 1$, the value of $F(X)$ is inconsequential. Such a function F is also referred to as a *Skolem function* for Y in $\varphi(X, Y)$ [15,22].

An interesting example of Boolean functional synthesis is the problem of integer factorization. Suppose Y_1 and Y_2 are n-bit unsigned integers, X is a $2n$-bit unsigned integer and $\times_{[n]}$ denotes n-bit unsigned multiplication. The relational specification $\varphi_{\mathsf{fact}}(X, Y_1, Y_2) \equiv ((X = Y_1 \times_{[n]} Y_2) \wedge (Y_1 \neq 1) \wedge (Y_2 \neq 1))$ specifies that Y_1 and Y_2 are non-trivial factors of X. This specification can be easily encoded as a Boolean relation. The corresponding synthesis problem requires

A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 337–353, 2017.
DOI: 10.1007/978-3-662-54577-5_19

us to synthesize the factors Y_1 and Y_2 as functions of X, whenever X is non-prime. Note that this problem is known to be hard, and the strength of several cryptographic systems rely on this hardness.

Existing approaches to Boolean functional synthesis vary widely in their emphasis, ranging from purely theoretical treatments (viz. [3,6,7,10,20,23]) to those motivated by practical tool development (viz. [4,11,12,15,17,18,21,22,27–29]). A common aspect of these approaches is their focus on sequential algorithms for synthesis. In this paper, we present, to the best of our knowledge, the first parallel algorithm for Boolean functional synthesis. A key ingredient of our approach is a technique for solving the synthesis problem for a specification φ by composing solutions of synthesis problems corresponding to sub-formulas in φ. Since Boolean functions are often represented using DAG-like structures (such as circuits, AIGs [16], ROBDDs [1,8]), we assume w.l.o.g. that φ is given as a DAG. The DAG structure provides a natural decomposition of the original problem into sub-problems with a partial order of dependencies between them. We exploit this to design a parallel synthesis algorithm that has been implemented on a message passing cluster. Our initial experiments show that our algorithm significantly outperforms state-of-the-art techniques on several benchmarks.

Related Work: The earliest solutions to Boolean functional synthesis date back to Boole [6] and Lowenheim [20], who considered the problem in the context of Boolean unification. Subsequently, there have been several investigations into theoretical aspects of this problem (see e.g., [3,7,10,23]). More recently, there have been attempts to design practically efficient synthesis algorithms that scale to much larger problem sizes. In [22], a technique to synthesize Y from a proof of validity of $\forall X \exists Y \varphi(X, Y)$ was proposed. While this works well in several cases, not all specifications admit the validity of $\forall X \exists Y \varphi(X, Y)$. For example, $\forall X \exists Y \varphi_{\mathsf{fact}}(X, Y)$ is not valid in the factorization example. In [12,29], a synthesis approach based on functional composition was proposed. Unfortunately, this does not scale beyond small problem instances [11,15]. To address this drawback, a CEGAR based technique for synthesis from *factored* specifications was proposed in [15]. While this scales well if each factor in the specification depends on a small subset of variables, its performance degrades significantly if we have a few "large" factors, each involving many variables, or if there is significant sharing of variables across factors. In [21], Macii et al. implemented Boole's and Lowenheim's algorithms using ROBDDs and compared their performance on small to medium-sized benchmarks. Other algorithms for synthesis based on ROBDDs have been investigated in [4,17]. A recent work [11] adapts the functional composition approach to work with ROBDDs, and shows that this scales well for a class of benchmarks with pre-determined variable orders. However, finding a good variable order for an arbitrary relational specification is hard, and our experiments show that without prior knowledge of benchmark classes and corresponding good variable orders, the performance of [11] can degrade significantly. Techniques using *templates* [28] or *sketches* [27] have been found to be effective for synthesis when we have partial information about the set of candidate solutions. A framework for functional synthesis, focused on unbounded

domains such as integer arithmetic, was proposed in [18]. This relies heavily on tailor-made smart heuristics that exploit specific form/structure of the relational specification.

2 Preliminaries

Let $X = (x_1, \ldots x_m)$ be the sequence of input variables, and $Y = (y_1, \ldots y_n)$ be the sequence of output variables in the specification $\varphi(X, Y)$. Abusing notation, we use X (resp. Y) to denote the set of elements in the sequence X (resp. Y), when there is no confusion. We use 1 and 0 to denote the Boolean constants true and false, respectively. A *literal* is either a variable or its complement. An assignment of values to variables *satisfies* a formula if it makes the formula true.

We assume that the specification $\varphi(X, Y)$ is represented as a rooted DAG, with internal nodes labeled by Boolean operators and leaves labeled by input/output literals and Boolean constants. If the operator labeling an internal node N has arity k, we assume that N has k ordered children. Figure 1 shows an example DAG, where the internal nodes are labeled by AND and OR operators of different arities. Each node N in such a DAG represents a Boolean formula $\Phi(N)$, which is inductively defined as follows. If N is a leaf, $\Phi(N)$ is the label of N. If N is an internal node labeled by op with arity k, and if the ordered children of N are $c_1, \ldots c_k$, then $\Phi(N)$ is $\mathsf{op}(\Phi(c_1), \ldots \Phi(c_k))$. A DAG with root R is said to represent the formula $\Phi(R)$. Note that popular DAG representations of Boolean formulas, such as AIGs, ROBDDs and Boolean circuits, are special cases of this representation.

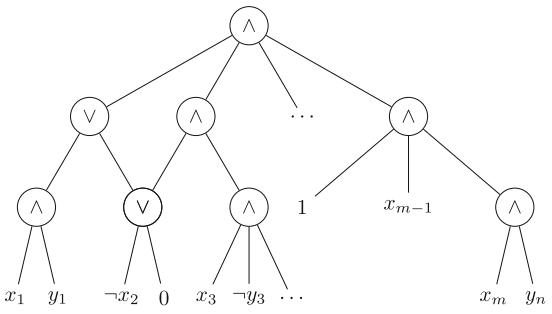


Fig. 1. DAG representing $\varphi(X, Y)$

A k-ary Boolean function f is a mapping from $\{0,1\}^k$ to $\{0,1\}$, and can be viewed as the semantics of a Boolean formula with k variables. We use the terms "Boolean function" and "Boolean formula" interchangeably, using formulas mostly to refer to specifications. Given a Boolean formula φ and a Boolean function f, we use $\varphi[y \mapsto f]$ to denote the formula obtained by substituting every occurrence of the variable y in φ with f. The set of variables appearing in φ is called the *support* of φ. If f and g are Boolean functions, we say that f *abstracts* g and g *refines* f, if $g \rightarrow f$, where $\rightarrow$ denotes logical implication.

Given the specification $\varphi(X, Y)$, our goal is to synthesize the outputs $y_1, \ldots y_n$ as functions of X. Unlike some earlier work [5,13,22], we *do not assume the validity of* $\forall X \exists Y\ \varphi(X, Y)$. Thus, we allow the possibility that for some values of X, there may be no value of Y that satisfies $\varphi(X, Y)$. This allows us to accommodate some important classes of synthesis problems, viz. integer factorization. If $y_1 = f_1(X), \ldots y_n = f_n(X)$ is a solution to the synthesis problem,

we say that $(f_1(X), \ldots f_n(X))$ *realizes* Y in $\varphi(X, Y)$. For notational clarity, we simply use $(f_1, \ldots f_n)$ instead of $(f_1(X), \ldots f_n(X))$ when X is clear from the context.

In general, an instance of the synthesis problem may not have a unique solution. The following proposition, stated in various forms in the literature, characterizes the space of all solutions, when we have one output variable y.

Proposition 1. *A function $f(X)$ realizes y in $\varphi(X, y)$ iff the following holds:* $\varphi[y \mapsto 1] \wedge \neg\varphi[y \mapsto 0] \;\rightarrow\; f(X)$ *and* $f(X) \;\rightarrow\; \varphi[y \mapsto 1] \vee \neg\varphi[y \mapsto 0]$.

As a corollary, both $\varphi[y \mapsto 1]$ and $\neg\varphi[y \mapsto 0]$ realize y in $\varphi(X, y)$. Proposition 1 can be easily extended when we have multiple output variables in Y. Let $\sqsubseteq$ be a total ordering of the variables in Y, and assume without loss of generality that $y_1 \sqsubseteq y_2 \sqsubseteq \cdots y_n$. Let $\overrightarrow{F}$ denote the vector of Boolean functions $(f_1(X), \ldots f_n(X))$. For $i \in \{1, \ldots n\}$, define $\varphi^{(i)}$ to be $\exists y_1 \ldots \exists y_{i-1}\, \varphi$, and $\varphi^{(i)}_{\overrightarrow{F}}$ to be $(\cdots(\varphi^{(i)}[y_{i+1} \mapsto f_{i+1}])\cdots)[y_n \mapsto f_n]$, with the obvious modifications for $i = 1$ (no existential quantification) and $i = n$ (no substitution). The following proposition, once again implicit in the literature, characterizes the space of all solutions $\overrightarrow{F}$ that realize Y in $\varphi(X, Y)$.

Proposition 2. *The function vector $\overrightarrow{F} = (f_1(X), \ldots f_n(X))$ realizes $Y = (y_1, \ldots y_n)$ in $\varphi(X, Y)$ iff the following holds for every $i \in \{1, \ldots n\}$:* $\varphi^{(i)}_{\overrightarrow{F}}[y_i \mapsto 1] \wedge \neg\varphi^{(i)}_{\overrightarrow{F}}[y_i \mapsto 0] \rightarrow f_i(X)$, *and* $f_i(X) \rightarrow \varphi^{(i)}_{\overrightarrow{F}}[y_i \mapsto 1] \vee \neg\varphi^{(i)}_{\overrightarrow{F}}[y_i \mapsto 0]$.

Propositions 1 and 2 are effectively used in [11,12,15,29] to sequentially synthesize $y_1, \ldots y_n$ as functions of X. Specifically, output y_1 is first synthesized as a function $g_1(X, y_2, \ldots y_n)$. This is done by treating y_1 as the sole output and $X \cup \{y_2, \ldots y_n\}$ as the inputs in $\varphi(X, Y)$. By substituting g_1 for y_1 in φ, we obtain $\varphi^{(2)} \equiv \exists y_1 \varphi(X, Y)$. Output y_2 can then be synthesized as a function $g_2(X, y_3, \ldots y_n)$ by treating y_2 as the sole output and $X \cup \{y_3, \ldots y_n\}$ as the inputs in $\varphi^{(2)}$. Substituting g_2 for y_2 in $\varphi^{(2)}$ gives $\varphi^{(3)} \equiv \exists y_1 \exists y_2\, \varphi(X, Y)$. This process is then repeated until we obtain y_n as a function $g_n(X)$. The desired functions $f_1(X), \ldots f_n(X)$ realizing $y_1, \ldots y_n$ can now be obtained by letting $f_n(X)$ be $g_n(X)$, and $f_i(X)$ be $(\cdots(g_i[y_{i+1} \mapsto f_{i+1}(X)])\cdots)[y_n \mapsto f_n(X)]$, for all i from $n-1$ down to 1. Thus, given $\varphi(X, Y)$, it suffices to obtain $(g_1, \ldots g_n)$, where g_i has support $X \cup \{y_{i+1}, \ldots y_n\}$, in order to solve the synthesis problem. We therefore say that $(g_1, \ldots g_n)$ *effectively realizes* Y in $\varphi(X, Y)$, and focus on obtaining $(g_1, \ldots g_n)$.

Proposition 1 implies that for every $i \in \{1, \ldots n\}$, the function $g_i \equiv \varphi^{(i)}[y_i \mapsto 1]$ realizes y_i in $\varphi^{(i)}$. With this choice for g_i, it is easy to see that $\exists y_i\, \varphi^{(i)}$ (or $\varphi^{(i+1)}$) can be obtained as $\varphi^{(i)}[y_i \mapsto g_i] = \varphi^{(i)}[y_i \mapsto \varphi^{(i)}[y_i \mapsto 1]]$. While synthesis using quantifier elimination by such *self-substitution* [11] has been shown to scale for certain classes of specifications with pre-determined optimized variable orders, our experience shows that this incurs significant overheads for general specifications with unknown "good" variable orders. An alternative technique for synthesis from *factored* specification was proposed by John et al. [15], in which

initial abstractions of $g_1, \ldots g_n$ are first computed quickly, and then a CEGAR-style [9] loop is used to refine these abstractions to correct Skolem functions. We use John et al.'s refinement technique as a black-box module in our work; more on this is discussed in Sect. 3.1.

Definition 1. *Given a specification $\varphi(X, Y)$, we define $\Delta_{y_i}(\varphi)$ to be the formula $(\neg\exists y_1 \ldots y_{i-1}\, \varphi)\,[y_i \mapsto 0]$, and $\Gamma_{y_i}(\varphi)$ to be the formula $(\neg\exists y_1 \ldots y_{i-1}\, \varphi)\,[y_i \mapsto 1]$, for all $i \in \{1, \ldots n\}$[1]. We also define $\overrightarrow{\Delta}(\varphi)$ and $\overrightarrow{\Gamma}(\varphi)$ to be the vectors $(\Delta_{y_1}(\varphi), \ldots \Delta_{y_n}(\varphi))$ and $(\Gamma_{y_1}(\varphi), \ldots \Gamma_{y_n}(\varphi))$ respectively.*

If N is a node in the DAG representation of the specification, we abuse notation and use $\Delta_{y_i}(N)$ to denote $\Delta_{y_i}(\Phi(N))$, and similarly for $\Gamma_{y_i}(N)$, $\overrightarrow{\Delta}(N)$ and $\overrightarrow{\Gamma}(N)$. Furthermore, if both Y and N are clear from the context, we use Δ_i, Γ_i, $\overrightarrow{\Delta}$ and $\overrightarrow{\Gamma}$ instead of $\Delta_{y_i}(N)$, $\Gamma_{y_i}(N)$, $\overrightarrow{\Delta}(N)$ and $\overrightarrow{\Gamma}(N)$, respectively. It is easy to see that the supports of both Γ_i and Δ_i are (subsets of) $X \cup \{y_{i+1}, \ldots y_n\}$. Furthermore, it follows from Definition 1 that whenever Γ_i (resp. Δ_i) evaluates to 1, if the output y_i has the value 1 (resp. 0), then φ must evaluate to 0. Conversely, if Γ_i (resp. Δ_i) evaluates to 0, it doesn't hurt (as far as satisfiability of $\varphi(X, Y)$ is concerned) to assign the value 1 (resp. 0) to output y_i. This suggests that both $\neg\Gamma_i$ and Δ_i suffice to serve as the function $g_i(X, y_{i+1}, \ldots y_n)$ when synthesizing functions for multiple output variables. The following proposition, adapted from [15], follows immediately, where we have abused notation and used $\neg\overrightarrow{\Gamma}$ to denote $(\neg\Gamma_1, \ldots \neg\Gamma_n)$.

Proposition 3. *Given a specification $\varphi(X, Y)$, both $\overrightarrow{\Delta}$ and $\neg\overrightarrow{\Gamma}$ effectively realize Y in $\varphi(X, Y)$.*

Proposition 3 shows that it suffices to compute $\overrightarrow{\Delta}$ (or $\overrightarrow{\Gamma}$) from $\varphi(X, Y)$ in order to solve the synthesis problem. In the remainder of the paper, we show how to achieve this compositionally and in parallel by first computing refinements of Δ_i (resp. Γ_i) for all $i \in \{1, \ldots n\}$, and then using John et al.'s CEGAR-based technique [15] to abstract them to the desired Δ_i (resp. Γ_i). Throughout the paper, we use δ_i and γ_i to denote refinements of Δ_i and Γ_i respectively.

3 Exploiting Compositionality

Given a specification $\varphi(X, Y)$, one way to synthesize $y_1, \ldots y_n$ is to decompose $\varphi(X, Y)$ into sub-specifications, solve the synthesis problems for the sub-specifications in parallel, and compose the solutions to the sub-problems to obtain the overall solution. A DAG representation of $\varphi(X, Y)$ provides a natural recursive decomposition of the specification into sub-specifications. Hence, the key technical question relates to compositionality: how do we compose solutions to synthesis problems for sub-specifications to obtain a solution to the synthesis problem for the overall specification? This problem is not easy, and no state-of-the-art tool for Boolean functional synthesis uses such compositional reasoning.

[1] In [15], equivalent formulas were called $Cb0_{y_i}(\varphi)$ and $Cb1_{y_i}(\varphi)$ respectively.

Our compositional solution to the synthesis problem is best explained in three steps. First, for a simple, yet representationally complete, class of DAGs representing $\varphi(X, Y)$, we present a lemma that allows us to do compositional synthesis at each node of such a DAG. Next, we show how to use this lemma to design a parallel synthesis algorithm. Finally, we extend our lemma, and hence the scope of our algorithm, to significantly more general classes of DAGs.

3.1 Compositional Synthesis in AND-OR DAGs

For simplicity of exposition, we first consider DAGs with internal nodes labeled by only AND and OR operators (of arbitrary arity). Figure 1 shows an example of such a DAG. Note that this class of DAGs is representationally complete for Boolean specifications, since every specification can be expressed in negation normal form (NNF). In the previous section, we saw that computing $\Delta_i(\varphi)$ or $\Gamma_i(\varphi)$ for all i in $\{1, \ldots n\}$ suffices for purposes of synthesis. The following lemma shows the relation between Δ_i and Γ_i at an internal node N in the DAG and the corresponding formulas at the node's children, say $c_1, \ldots c_k$.

Lemma 1 (Composition Lemma). *Let* $\Phi(N) = \mathsf{op}(\Phi(c_1), \ldots, \Phi(c_k))$, *where* $\mathsf{op} = \vee$ *or* $\mathsf{op} = \wedge$. *Then, for each* $1 \leq i \leq n$:

$$\left(\bigwedge_{j=1}^{k} \Delta_i(c_j)\right) \leftrightarrow \Delta_i(N) \quad \textit{and} \quad \left(\bigwedge_{j=1}^{k} \Gamma_i(c_j)\right) \leftrightarrow \Gamma_i(N) \textit{ if } \mathsf{op} = \vee \tag{1}$$

$$\left(\bigvee_{j=1}^{k} \Delta_i(c_j)\right) \rightarrow \Delta_i(N) \quad \textit{and} \quad \left(\bigvee_{j=1}^{k} \Gamma_i(c_j)\right) \rightarrow \Gamma_i(N) \textit{ if } \mathsf{op} = \wedge \tag{2}$$

The proof of this lemma can be found in [2]. Thus, if N is an OR-node, we obtain $\Delta_i(N)$ and $\Gamma_i(N)$ directly by conjoining Δ_i and Γ_i at its children. However, if N is an AND-node, disjoining the Δ_i and Γ_i at its children only gives refinements of $\Delta_i(N)$ and $\Gamma_i(N)$ (see Eq. (2)). Let us call these refinements $\delta_i(N)$ and $\gamma_i(N)$ respectively. To obtain $\Delta_i(N)$ and $\Gamma_i(N)$ exactly at AND-nodes, we must use the CEGAR technique developed in [15] to iteratively abstract $\delta_i(N)$ and $\gamma_i(N)$ obtained above. More on this is discussed below.

A CEGAR step involves constructing, for each i from 1 to n, a Boolean *error formula* Err_{δ_i} (resp. Err_{γ_i}) such that the error formula is unsatisfiable iff $\delta_i(N) \leftrightarrow \Delta_i(N)$ (resp. $\gamma_i(N) \leftrightarrow \Gamma_i(N)$). A SAT solver is then used to check the satisfiability of the error formula. If the formula is unsatisfiable, we are done; otherwise the satisfying assignment can be used to further abstract the respective refinement. This check-and-abstract step is then repeated in a loop until the error formulas become unsatisfiable. Following the approach outlined in [15], it can be shown that if we use $\mathsf{Err}_{\delta_i} \equiv \neg\delta_i \wedge \bigwedge_{j=1}^{i} (y_j \leftrightarrow \delta_j) \wedge \neg\varphi$ and $\mathsf{Err}_{\gamma_i} \equiv \neg\gamma_i \wedge \bigwedge_{j=1}^{i} (y_j \leftrightarrow \neg\gamma_j) \wedge \neg\varphi$, and perform CEGAR in order from $i = 1$ to $i = n$, it suffices to gives us Δ_i and Γ_i. For details of the CEGAR implementation, the reader is referred to [15]. The above discussion leads to

a straightforward algorithm COMPUTE (shown as Algorithm 1) that computes $\overrightarrow{\Delta}(N)$ and $\overrightarrow{\Gamma}(N)$ for a node N, using $\overrightarrow{\Delta}(c_j)$ and $\overrightarrow{\Gamma}(c_j)$ for its children c_j. Here, we have assumed access to a black-box function PERFORM_CEGAR that implements the CEGAR step.

Algorithm 1. COMPUTE(NODE N)

Input: A DAG Node N labelled either AND or OR
Precondition: Children of N, if any, have their $\overrightarrow{\Delta}$ and $\overrightarrow{\Gamma}$ computed.
Output: $\overrightarrow{\Delta}(N), \overrightarrow{\Gamma}(N)$

1 **if** *N is a leaf* `// Φ(N) is a literal/constant; use Definition 1`
2 **then**
3 for all $y_i \in Y$, $\Delta_i(N) = \neg\exists y_1 \dots y_{i-1}(\Phi(N))[y_i \mapsto 0]$;
4 for all $y_i \in Y$, $\Gamma_i(N) = \neg\exists y_1 \dots y_{i-1}(\Phi(N))[y_i \mapsto 1]$;
5 **else**
  `// N is an internal node; let its children be` $c_1, \dots c_k$
6 **if** *N is an OR-node* **then**
7 **for** ***each*** $y_i \in Y$ **do**
8 $\Delta_i(N) := \Delta_i(c_1) \wedge \Delta_i(c_2) \dots \wedge \Delta_i(c_k)$;
9 $\Gamma_i(N) := \Gamma_i(c_1) \wedge \Gamma_i(c_2) \dots \wedge \Gamma_i(c_k)$;
10 **if** *N is an AND-node* **then**
11 **for** ***each*** $y_i \in Y$ **do**
12 $\delta_i(N) := \Delta_i(c_1) \vee \Delta_i(c_2) \dots \vee \Delta_i(c_k)$; `/* δ_i(N) → Δ_i(N) */`
13 $\gamma_i(N) := \Gamma_i(c_1) \vee \Gamma_i(c_2) \dots \vee \Gamma_i(c_k)$; `/* γ_i(N) → Γ_i(N) */`
14 $\left(\overrightarrow{\Delta}(N), \overrightarrow{\Gamma}(N)\right) =$ PERFORM_CEGAR$(N, (\delta_i(N), \gamma_i(N))_{y_i \in Y})$;
15 **return** $\left(\overrightarrow{\Delta}(N), \overrightarrow{\Gamma}(N)\right)$;

3.2 A Parallel Synthesis Algorithm

The DAG representation of $\varphi(X, Y)$ gives a natural, recursive decomposition of the specification, and also defines a partial order of dependencies between the corresponding synthesis sub-problems. Algorithm COMPUTE can be invoked in parallel on nodes in the DAG that are not ordered w.r.t. this partial order, as long as COMPUTE has already been invoked on their children. This suggests a simple parallel approach to Boolean functional synthesis. Algorithm PARSYN, shown below, implements this approach, and is motivated by a message-passing architecture. We consider a standard manager-worker configuration, where one out of available m cores acts as the manager, and the remaining $m-1$ cores act as workers. All communication between the manager and workers is assumed to happen through explicit **send** and **receive** primitives.

The manager uses a queue Q of ready-to-process nodes. Initially, Q is initialized with the leaf nodes in the DAG, and we maintain the invariant that all

Algorithm 2. ParSyn

Input: AND-OR DAG with root Rt representing $\varphi(X, Y)$ in NNF form
Output: $(g_1, \ldots g_n)$ that effectively realize Y in $\varphi(X, Y)$

```
   /* Algorithm for Manager                                                    */
 1 Queue Q ;
   /* Invariant: Q has nodes that can be processed in parallel, i.e.,
      leaves or nodes whose children have their ⃗Δ, ⃗Γ computed.                */
 2 Insert all leaves of DAG into Q;
 3 while all DAG nodes not processed do
 4     while a worker W is idle and Q is not empty do
 5         Node N := Q.front();
 6         send node N for processing to W;
 7         if N has children c_1, ... c_k then send ⃗Δ(c_j), ⃗Γ(c_j) for 1 ≤ j ≤ k to W;
 8     wait until some worker W' processing node N' becomes free;
 9     receive (⃗Δ, ⃗Γ) from W', and store as (⃗Δ(N'), ⃗Γ(N'));
10     Mark node N' as processed;
11     for each parent node N'' of N' do
12         if all children of N'' are processed then insert N'' into Q

   /* All DAG nodes are processed; return ¬⃗Γ or ⃗Δ from root Rt               */
13 return (¬Γ_1(Rt), ... ¬Γ_n(Rt))      // or alternatively (Δ_1(Rt), ... Δ_n(Rt))
```

```
   /* Algorithm for Worker W                                                   */
14 receive node N to process, and ⃗Δ(c_j), ⃗Γ(c_j) for every child c_j of N, if any;
15 (⃗Δ, ⃗Γ) := Compute(N) ;
16 send (⃗Δ, ⃗Γ) to Manager ;
```

nodes in Q can be processed in parallel. If there is an idle worker W and if Q is not empty, the manager assigns the node N at the front of Q to worker W for processing. If N is an internal DAG node, the manager also sends $\overrightarrow{\Delta}(c_j)$ and $\overrightarrow{\Gamma}(c_j)$ for every child c_j of N to W. If there are no idle workers or if Q is empty, the manager waits for a worker, say W', to finish processing its assigned node, say N'. When this happens, the manager stores the result sent by W' as $\overrightarrow{\Delta}(N')$ and $\overrightarrow{\Gamma}(N')$. It then inserts one or more parents N'' of N' in the queue Q, if all children of N'' have been processed. The above steps are repeatedly executed at the manager until all DAG nodes have been processed. The job of a worker W is relatively simple: on being assigned a node N, and on receiving $\overrightarrow{\Delta}(c_j)$ and $\overrightarrow{\Gamma}(c_j)$ for all children c_j of N, it simply executes Algorithm Compute on N and returns $\left(\overrightarrow{\Delta}(N), \overrightarrow{\Gamma}(N)\right)$.

Note that Algorithm ParSyn is guaranteed to progress as long as all workers complete processing the nodes assigned to them in finite time. The partial order

of dependencies between nodes ensures that when all workers are idle, either all nodes have already been processed, or at least one unprocessed node has $\overrightarrow{\Delta}$ and $\overrightarrow{\Gamma}$ computed for all its children, if any.

3.3 Extending the Composition Lemma and Algorithms

So far, we have considered DAGs in which all internal nodes were either AND- or OR-nodes. We now extend our results to more general DAGs. We do this by generalizing the Composition Lemma to arbitrary Boolean operators. Specifically, given the refinements $\delta_i(c_j)$ and $\gamma_i(c_j)$ at all children c_j of a node N, we show how to compose these to obtain $\delta_i(N)$ and $\gamma_i(N)$, when N is labeled by an arbitrary Boolean operator. Note that the CEGAR technique discussed in Sect. 3.1 can be used to abstract the refinements δ_i and γ_i to Δ_i and Γ_i respectively, at any node of interest. Therefore, with our generalized Composition Lemma, we can use compositional synthesis for specifications represented by general DAGs, even without computing Δ_i and Γ_i exactly at all DAG nodes. This gives an extremely powerful approach for parallel, compositional synthesis.

Let $\Phi(N) = \mathsf{op}(\Phi(c_1), \ldots \Phi(c_r))$, where op is an r-ary Boolean operator. For convenience of notation, we use $\neg N$ to denote $\neg\Phi(N)$, and similarly for other nodes, in the subsequent discussion. Suppose we are given $\delta_i(c_j)$, $\gamma_i(c_j)$, $\delta_i(\neg c_j)$ and $\gamma_i(\neg c_j)$, for $1 \leq j \leq r$ and for $1 \leq i \leq n$. We wish to compose these appropriately to compute $\delta_i(N)$, $\gamma_i(N)$, $\delta_i(\neg N)$ and $\gamma_i(\neg N)$ for $1 \leq i \leq n$. Once we have these refinements, we can adapt Algorithm 1 to work for node N, labeled by an arbitrary Boolean operator op.

To understand how composition works for op, consider the formula $\mathsf{op}(z_1, \ldots z_r)$, where $z_1, \ldots z_r$ are fresh Boolean variables. Clearly, $\Phi(N)$ can be viewed as $(\cdots(\mathsf{op}(z_1, \ldots z_r)[z_1 \mapsto \Phi(c_1)])\cdots)[z_r \mapsto \Phi(c_r)]$. For simplicity of notation, we write op instead of $\mathsf{op}(z_1, \ldots, z_r)$ in the following discussion. W.l.o.g., let $z_1 \prec z_2 \prec \cdots \prec z_r$ be a total ordering of the variables $\{z_1, \ldots z_r\}$. Given $\prec$, suppose we compute the formulas $\delta_{z_l}(\mathsf{op})$, $\gamma_{z_l}(\mathsf{op})$, $\delta_{z_l}(\neg\mathsf{op})$ and $\gamma_{z_l}(\neg\mathsf{op})$ in negation normal form (NNF), for all $l \in \{1, \ldots r\}$. Note that these formulas have support $\{z_{l+1}, \ldots z_r\}$, and do not have variables in $X \cup Y$ in their support. We wish to ask if we can compose these formulas with $\delta_i(c_j)$, $\gamma_i(c_j)$, $\delta_i(\neg c_j)$ and $\gamma_i(\neg c_j)$ for $1 \leq j \leq r$ to compute $\delta_i(N)$, $\gamma_i(N)$, $\delta_i(\neg N)$ and $\gamma_i(\neg N)$, for all $i \in \{1, \ldots n\}$. It turns out that we can do this.

Recall that in NNF, negations appear (if at all) only on literals. Let $\Upsilon_{l,\mathsf{op}}$ be the formula obtained by replacing every literal $\neg z_s$ in the NNF of $\gamma_{z_l}(\mathsf{op})$ with a fresh variable $\overline{z_s}$. Similarly, let $\Omega_{l,\mathsf{op}}$ be obtained by replacing every literal $\neg z_s$ in the NNF of $\delta_{z_l}(\mathsf{op})$ with the fresh variable $\overline{z_s}$. The definitions of $\Upsilon_{l,\neg\mathsf{op}}$ and $\Omega_{l,\neg\mathsf{op}}$ are similar. Replacing $\neg z_s$ by a fresh variable $\overline{z_s}$ allows us to treat the literals z_s and $\neg z_s$ independently in the NNF of $\gamma_{z_l}(\mathsf{op})$ and $\delta_{z_l}(\mathsf{op})$. The ability to treat these independently turns out to be important when formulating the generalized Composition Lemma. Let $(\Upsilon_{l,\mathsf{op}}\,[z_s \mapsto \delta_i(\neg c_s)]\,[\overline{z_s} \mapsto \delta_i(c_s)])^{r}_{s=l+1}$ denote the formula obtained by substituting $\delta_i(\neg c_s)$ for z_s and $\delta_i(c_s)$ for $\overline{z_s}$, for every $s \in \{l+1, \ldots r\}$, in $\Upsilon_{l,\mathsf{op}}$. The interpretation of $(\Omega_{l,\mathsf{op}}\,[z_s \mapsto \delta_i(\neg c_s)]\,[\overline{z_s} \mapsto \delta_i(c_s)])^{r}_{s=l+1}$ is analogous. Our generalized Composition Lemma can now be stated as follows.

Lemma 2 (Generalized Composition Lemma). *Let* $\Phi(N) = \mathsf{op}(\Phi(c_1), \ldots \Phi(c_r))$*, where* op *is an* r*-ary Boolean operator. For each* $1 \leq i \leq n$ *and* $1 \leq l \leq r$*:*

$$1.\ \delta_i(c_l) \wedge (\Omega_{l,\mathsf{op}}\,[z_s \mapsto \delta_i(\neg c_s)]\,[\overline{z_s} \mapsto \delta_i(c_s)])_{s=l+1}^{r} \ \rightarrow\ \Delta_i(N)$$
$$2.\ \delta_i(\neg c_l) \wedge (\Upsilon_{l,\mathsf{op}}\,[z_s \mapsto \delta_i(\neg c_s)]\,[\overline{z_s} \mapsto \delta_i(c_s)])_{s=l+1}^{r} \ \rightarrow\ \Delta_i(N)$$
$$3.\ \gamma_i(c_l) \wedge (\Omega_{l,\mathsf{op}}\,[z_s \mapsto \gamma_i(\neg c_s)]\,[\overline{z_s} \mapsto \gamma_i(c_s)])_{s=l+1}^{r} \ \rightarrow\ \Gamma_i(N)$$
$$4.\ \gamma_i(\neg c_l) \wedge (\Upsilon_{l,\mathsf{op}}\,[z_s \mapsto \gamma_i(\neg c_s)]\,[\overline{z_s} \mapsto \gamma_i(c_s)])_{s=l+1}^{r} \ \rightarrow\ \Gamma_i(N)$$

If we replace op *by* $\neg\mathsf{op}$ *above, we get refinements of* $\Delta_i(\neg N)$ *and* $\Gamma_i(\neg N)$*.*

The reader is referred to [2] for a proof of Lemma 2. We simply illustrate the idea behind the lemma with an example here. Suppose $\Phi(N) = \Phi(c_1) \wedge \neg\Phi(c_2) \wedge (\neg\Phi(c_3) \vee \Phi(c_4))$, where each $\Phi(c_j)$ is a Boolean function with support $X \cup \{y_1, \ldots y_n\}$. We wish to compute a refinement of $\Delta_i(N)$, using refinements of $\Delta_i(c_j)$ and $\Delta_i(\neg c_j)$ for $j \in \{1, \ldots 4\}$. Representing N as $\mathsf{op}(c_1, c_2, c_3, c_4)$, let $z_1, \ldots z_4$ be fresh Boolean variables, not in $X \cup \{y_1, \ldots y_n\}$; then $\mathsf{op}(z_1, z_2, z_3, z_4) = z_1 \wedge \neg z_2 \wedge (\neg z_3 \vee z_4)$. For ease of exposition, assume the ordering $z_1 \prec z_2 \prec z_3 \prec z_4$. By definition, $\Delta_{z_2}(\mathsf{op}) = (\neg\exists z_1\,(z_1 \wedge \neg z_2 \wedge (\neg z_3 \vee z_4)))\,[z_2 \mapsto 0] = z_3 \wedge \neg z_4$, and suppose $\delta_{z_2}(\mathsf{op}) = \Delta_{z_2}(\mathsf{op})$. Replacing $\neg z_4$ by $\overline{z_4}$, we then get $\Omega_{2,\mathsf{op}} = z_3 \wedge \overline{z_4}$.

Recalling the definition of $\delta_{z_2}(\cdot)$, if we set $z_3 = 1$, $z_4 = 0$ and $z_2 = 0$, then op must evaluate to 0 regardless of the value of z_1. By substituting $\delta_i(\neg c_3)$ for z_3 and $\delta_i(c_4)$ for $\overline{z_4}$ in $\Omega_{2,\mathsf{op}}$, we get the formula $\delta_i(\neg c_3) \wedge \delta_i(c_4)$. Denote this formula by χ and note that its support is $X \cup \{y_{i+1}, \ldots y_n\}$. Note also from the definition of $\delta_i(\cdot)$ that if χ evaluates to 1 for some assignment of values to $X \cup \{y_{i+1}, \ldots y_n\}$ and if $y_i = 0$, then $\neg\Phi(c_3)$ evaluates to 0 and $\Phi(c_4)$ evaluates to 0, regardless of the values of $y_1, \ldots y_{i-1}$. This means that $z_3 = 1$ and $z_4 = 0$, and hence $\delta_{z_2}(\mathsf{op}) = 1$. If z_2 (or $\Phi(c_2)$) can also be made to evaluate to 0 for the same assignment of values to $X \cup \{y_i, y_{i+1}, \ldots y_n\}$, then $N = \mathsf{op}(c_1, \ldots c_r)$ must evaluate to 0, regardless of the values of $\{y_1, \ldots y_{i-1}\}$. Since $y_i = 0$, values assigned to $X \cup \{y_{i+1}, \ldots y_n\}$ must therefore be a satisfying assignment of $\Delta_i(N)$. One way of having $\Phi(c_2)$ evaluate to 0 is to ensure that $\Delta_i(c_2)$ evaluates to 1 for the same assignment of values to $X \cup \{y_{i+1}, \ldots y_n\}$ that satisfies χ. Therefore, we require the assignment of values to $X \cup \{y_{i+1}, \ldots y_n\}$ to satisfy $\chi \wedge \Delta_i(c_2)$, or even $\chi \wedge \delta_i(c_2)$. Since $\chi = \delta_i(\neg c_3) \wedge \delta_i(c_4)$, we get $\delta_i(c_2) \wedge \delta_i(\neg c_3) \wedge \delta_i(c_4)$ as a refinement of $\Delta_i(N)$.

Applying the Generalized Composition Lemma: Lemma 2 suggests a way of compositionally obtaining $\delta_i(N)$, $\gamma_i(N)$, $\delta_i(\neg N)$ and $\gamma_i(\neg N)$ for an arbitrary Boolean operator op. Specifically, the disjunction of the left-hand sides of implications (1) and (2) in Lemma 2, disjoined over all $l \in \{1, \ldots r\}$ and over all total orders ($\prec$) of $\{z_1, \ldots z_r\}$, gives a refinement of $\Delta_i(N)$. A similar disjunction of the left-hand sides of implications (3) and (4) in Lemma 2 gives a refinement of $\Gamma_i(N)$. The cases of $\Delta_i(\neg N)$ and $\Gamma_i(\neg N)$ are similar. This suggests that for each operator op that appears as label of an internal DAG node, we can pre-compute a template

of how to compose δ_i and γ_i at the children of the node to obtain δ_i and γ_i at the node itself. In fact, pre-computing this template for $\mathsf{op} = \vee$ and $\mathsf{op} = \wedge$ by disjoining as suggested above, gives us exactly the left-to-right implications, i.e., refinements of $\Delta_i(N)$ and $\Gamma_i(N)$, as given by Lemma 1. We present templates for some other common Boolean operators like *if-then-else* in [2].

Once we have pre-computed templates for composing δ_i and γ_i at children of a node N to get $\delta_i(N)$ and $\gamma_i(N)$, we can use these pre-computed templates in Algorithm 1, just as we did for AND-nodes. This allows us to apply compositional synthesis on general DAG representations of Boolean relational specifications.

Optimizations Using Partial Computations: Given δ_i and γ_i at children of a node N, we have shown above how to compute $\delta_i(N)$ and $\gamma_i(N)$. To compute $\Delta_i(N)$ and $\Gamma_i(N)$ exactly, we can use the CEGAR technique outlined in Sect. 3.1. While this is necessary at the root of the DAG, we need not compute $\Delta_i(N)$ and $\Gamma_i(N)$ exactly at each intermediate node. In fact, the generalized Composition Lemma allows us to proceed with $\delta_i(N)$ and $\gamma_i(N)$. This suggests some optimizations: (i) Instead of using the error formulas introduced in Sect. 3.1, that allow us to obtain $\Delta_i(N)$ and $\Gamma_i(N)$ exactly, we can use the error formula used in [15]. The error formula of [15] allows us to obtain some Skolem function for y_i (not necessarily $\Delta_i(N)$ or $\neg\Gamma_i(N)$) using the sub-specification $\Phi(N)$ corresponding to node N. We have found CEGAR based on this error formula to be more efficient in practice, while yielding refinements of $\Delta_i(N)$ and $\Gamma_i(N)$. In fact, we use this error formula in our implementation. (ii) We can introduce a *timeout* parameter, such that $\overrightarrow{\Delta}(N)$, $\overrightarrow{\Gamma}(N)$ are computed exactly at each internal node until timeout happens. Subsequently, for the nodes still under process, we can simply combine δ_i and γ_i at their children using our pre-computed composition templates, and not invoke CEGAR at all. The only exception to this is at the root node of the DAG where CEGAR must be invoked.

4 Experimental Results

Experimental Methodology. We have implemented Algorithm 2 with the error formula from [15] used for CEGAR in Algorithm 1 (in function PERFORM_CEGAR), as described at the end of Sect. 3.3. We call this implementation ParSyn in this section, and compare it with the following algorithms/tools: (*i*) CSk: This is based on the sequential algorithm for conjunctive formulas, presented in [15]. For non-conjunctive formulas, the algorithm in [15], and hence CSk, reduces to [12,29]. (*ii*) RSynth: The *RSynth* tool as described in [11]. (*iii*) Bloqqer: As prescribed in [22], we first generate special QRAT proofs using the preprocessing tool bloqqer, and then generate Boolean function vectors from the proofs using the qrat-trim tool.

Our implementation of ParSyn, available online at [26], makes extensive use of the ABC [19] library to represent and manipulate Boolean functions as AIGs. We also use the default SAT solver provided by ABC, which is a variant of MiniSAT. We present our evaluation on three different kinds of *benchmarks*.

1. *Disjunctive Decomposition Benchmarks*: Similar to [15], these benchmarks were generated by considering some of the larger sequential circuits in the HWMCC10 benchmark suite, and formulating the problem of disjunctively decomposing each circuit into components as a problem of synthesizing a vector of Boolean functions. Each generated benchmark is of the form $\exists Y \varphi(X, Y)$ where $\exists X(\exists Y \varphi(X, Y))$ is true. However, unlike [15], where each benchmark (if not already a conjunction of factors) had to be converted into factored form using Tseitin encoding (which introduced additional variables), we have used these benchmarks without Tseitin encoding.
2. *Arithmetic Benchmarks*: These benchmarks were taken from the work described in [11]. Specifically, the benchmarks considered are *floor*, *ceiling*, *decomposition*, *equalization* and *intermediate* (see [11] for details).
3. *Factorization Benchmarks*: We considered the integer factorization problem for different bit-widths, as discussed in Sect. 1.

For each arithmetic and factorization benchmark, we first specified the problem instance as an SMT formula and then used *Boolector* [24] to generate the Boolean version of the benchmark. For each arithmetic benchmark, three variants were generated by varying the bit-width of the arguments of arithmetic operators; specifically, we considered bit-widths of 32, 128 and 512. Similarly, for the factorization benchmark, we generated four variants, using 8, 10, 12 and 16 for the bit-width of the product. Further, as Bloqqer requires the input to be in `qdimacs` format and RSynth in `cnf` format, we converted each benchmark into `qdimacs` and `cnf` formats using Tseitin encoding [30]. All benchmarks and the procedure by which we generated them are detailed in [26].

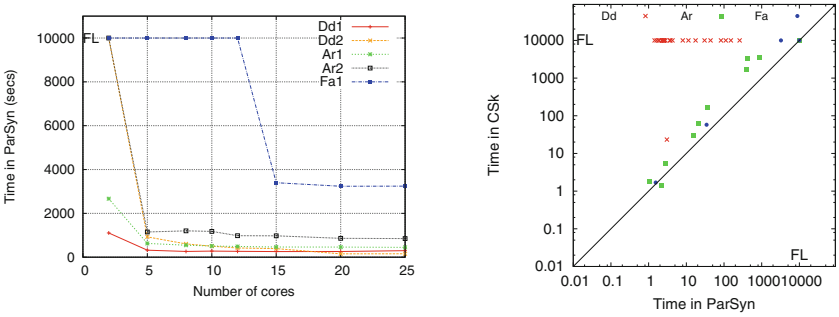
Variable Ordering: We used the same ordering of variables for all algorithms. For each benchmark, the variables are ordered such that the variable which occurs in the transitive fan-in of the least number of nodes in the AIG representation of the specification, appears at the top. For RSynth this translated to an interleaving of most of the input and output variables.

Machine Details: All experiments were performed on a message-passing cluster, where each node had 20 cores and 64 GB main memory, each core being a 2.20 GHz Intel Xeon processor. The operating system was Cent OS 6.5. For CSk, Bloqqer, and RSynth, a single core on the cluster was used. For all comparisons, ParSyn was executed on 4 nodes using 5 cores each, so that we had both intra-node and inter-node communication. The maximum time given for execution was 3600 s, i.e., 1 h. We also restricted the total amount of main memory (across all cores) to be 16 GB. The metric used to compare the different algorithms was the time taken to synthesize Boolean functions.

Results. Our benchmark suite consisted of 27 disjunctive decomposition benchmarks, 15 arithmetic benchmarks and 4 factorization benchmarks. These benchmarks are fairly comprehensive in size i.e., the number of AIG nodes ($|SZ|$)

in the benchmark, and the number of variables ($|Y|$) for which Boolean functions are to be synthesized. Amongst disjunctive decomposition benchmarks, $|SZ|$ varied from 1390 to 58752 and $|Y|$ varied from 21 to 205. Amongst the arithmetic benchmarks, $|SZ|$ varied from 442 to 11253 and $|Y|$ varied from 31 to 1024. The factorization benchmarks are the smallest and the most complex of the benchmarks, with $|SZ|$ varying from 122 to 502 and $|Y|$ varying from 8 to 16.

We now present the performance of the various algorithms. On 4 of the 46 benchmarks, none of the tools succeeded. Of these, 3 belonged to the *intermediate* problem type in the arithmetic benchmarks, and the fourth one was the 16 bit factorization benchmark.



(a) ParSyn on different cores

(b) ParSyn vs CSk

Fig. 2. Legend: `Ar`: arithmetic, `Fa`: factorization, `Dd`: disjunctive decomposition. `FL`: benchmarks for which the corresponding algorithm was unsuccessful.

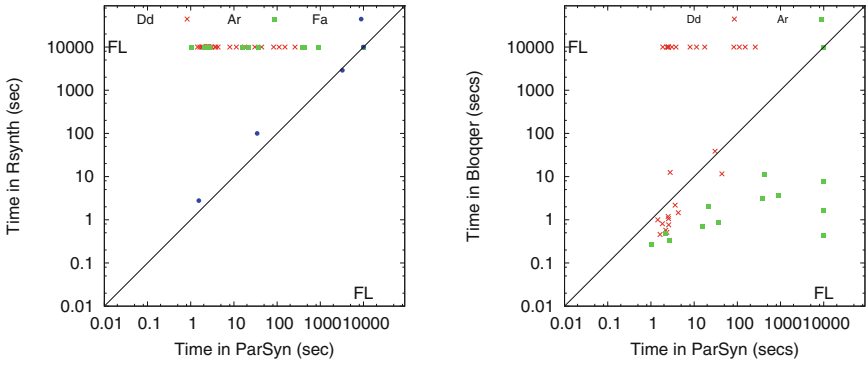
Effect of the Number of Cores. For this experiment, we chose 5 of the larger benchmarks. Of these, two benchmarks belonged to the disjunctive decomposition category, two belonged to the arithmetic benchmark category and one was the 12 bit factorization benchmark. The number of cores was varied from 2 to 25. With 2 cores, ParSyn behaves like a sequential algorithm with one core acting as the manager and the other as the worker with all computation happening at the worker core. Hence, with 2 cores, we see the effect of compositionality without parallelism. For number of cores > 2, the number of worker cores increase, and the computation load is shared across the worker cores.

Figure 2a shows the results of our evaluation. The topmost points indicated by `FL` are instances for which ParSyn timed out. We can see that, for all 5 benchmarks, the time taken to synthesize Boolean function vectors when the number of cores is 2 is considerable; in fact, ParSyn times out on three of the benchmarks. When we increase the number of cores we observe that (a) by synthesizing in parallel, we can now solve benchmarks for which we had timed out earlier, and (b) speedups of about 4–5 can be obtained with 5–15 cores. From 15 cores to 25 cores, the performance of the algorithm, however, is largely invariant and any further increase in cores does not result in further speed up.

To understand this, we examined the benchmarks and found that their AIG representations have more nodes close to the leaves than to the root (similar to the DAG in Fig. 1). The time taken to process a leaf or a node close to a leaf is typically much less than that for a node near the root. Furthermore, the dependencies between the nodes close to the root are such that at most one or two nodes can be processed in parallel leaving most of the cores unutilized. When the number of cores is increased from 2 to 5–15, the leaves and the nodes close to the leaves get processed in parallel, reducing the overall time taken by the algorithm. However, the time taken to process the nodes close to the root remains more or less the same and starts to dominate the total time taken. At this point, even if the number of cores is further increased, it does not significantly reduce the total time taken. This behaviour limits the speed-ups of our algorithm. For the remaining experiments, the number of cores used for ParSyn was 20.

ParSyn *vs* **CSk***:* As can be seen from Fig. 2b, CSk ran successfully on only 12 of the 46 benchmarks, whereas ParSyn was successful on 39 benchmarks, timing out on 6 benchmarks and running out of memory on 1 benchmark. Of the benchmarks that CSk was successful on, 9 belonged to the arithmetic category, 2 to the factorization and 1 to the disjunctive decomposition category. On further examination, we found that factorization and arithmetic benchmarks (except the *intermediate* problems) were conjunctive formulae whereas disjunctive decomposition benchmarks were arbitrary Boolean formulas. Since CSk has been specially designed to handle conjunctive formulas, it is successful on some of these benchmarks. On the other hand, since disjunctive decomposition benchmarks are not conjunctive, CSk treats the entire formula as one factor, and the algorithm reduces to [12,29]. The performance hit is therefore not surprising; it has been shown in [15] and [11] that the algorithms of [12,29] do not scale to large benchmarks that are not conjunctions of small factors. In fact, among the disjunctive decomposition benchmarks, CSk was successful on only the smallest one.

ParSyn *vs RSynth:* As seen in Fig. 3a, RSynth was successful only on 3 of the 46 benchmarks; it timed out on 37 and ran out of memory on 6 benchmarks. The 3 benchmarks that RSynth was successful on were the smaller factorization benchmarks. Note that the arithmetic benchmarks used in [11] are semantically the same as the ones used in our experiments. In [11], custom variable orders were used to construct the ROBDDs, which resulted in compact ROBDDs. In our case, we use the variable ordering heuristic mentioned earlier, and include the considerable time taken to build BDDs from `cnf` representation. As mentioned in Sect. 1, if we know a better variable ordering, then the time taken can potentially reduce. However, we may not know the optimal variable order for an arbitrary specification in general. We also found the memory footprint of RSynth to be higher as indicated by the memory-outs. This is not surprising, as RSynth uses BDDs to represent Boolean formulas and it is well-known that BDDs can have large memory requirements.



(a) ParSyn vs RSynth (b) ParSyn vs Bloqqer

Fig. 3. Legend: `Ar`: arithmetic, `Fa`: factorization, `Dd`: disjunctive decomposition. `FL`: benchmarks for which the corresponding algorithm was unsuccessful.

ParSyn *vs Bloqqer:* Since Bloqqer cannot synthesize Boolean functions for formulas wherein $\forall X \exists Y \varphi(X, Y)$ is not *valid*, we restricted our comparison to only the disjunctive decomposition and arithmetic benchmarks, totalling 42 in number. From Fig. 3b, we can see that Bloqqer successfully synthesizes Boolean functions for 25 of the 42 benchmarks. For several benchmarks for which it is successful, it outperforms ParSyn. In line 14 of Algorithm 1, PERFORM_CEGAR makes extensive use of the SAT solver, and this is reflected in the time taken by ParSyn. However, for the remaining 17 benchmarks, Bloqqer gave a *Not Verified* message indicating that it could not synthesize Boolean functions for these benchmarks. In comparison, ParSyn was successful on most of these benchmarks.

Effect of Timeouts on ParSyn. Finally, we discuss the effect of the timeout optimization discussed in Sect. 3.3. Specifically, for 60 s (value set through a *timeout* parameter), starting from the leaves of the AIG representation of a specification, we synthesize exact Boolean functions for DAG nodes. After timeout, on the remaining intermediate nodes, we do not invoke the CEGAR step at all, except at the root node of the AIG.

This optimization enabled us to handle 3 more benchmarks, i.e., ParSyn with this optimization synthesized Boolean function vectors for all the *equalization* benchmarks (in <340 s). Interestingly, ParSyn without timeouts was unable to solve these problems. This can be explained by the fact that in these benchmarks many internal nodes required multiple iterations of the CEGAR loop to compute exact Boolean functions, which were, however, not needed to compute the solution at the root node.

5 Conclusion and Future Work

In this paper, we have presented the first parallel and compositional algorithm for complete Boolean functional synthesis from a relational specification. A key

feature of our approach is that it is agnostic to the semantic variabilities of the input, and hence applies to a wide variety of benchmarks. In addition to the disjunctive decomposition of graphs and the arithmetic operation benchmarks, we considered the combinatorially hard problem of factorization and attempted to generate a functional characterization for it. We found that our implementation outperforms existing tools in a variety of benchmarks.

There are many avenues to extend our work. First, the ideas for compositional synthesis that we develop in this paper could potentially lead to parallel implementations of other synthesis tools, such as that described in [11]. Next, the factorization problem can be generalized to synthesis of inverse functions for classically hard one-way functions, as long as the function can be described efficiently by a circuit/AIG. Finally, we would like to explore improved ways of parallelizing our algorithm, perhaps exploiting features of specific classes of problems.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **27**(6), 509–516 (1978). http://dx.doi.org/10.1109/TC.1978.1675141
2. Akshay, S., Chakraborty, S., John, A., Shah, S.: Towards Parallel Boolean Functional Synthesis. ArXiv e-prints (2017). CoRR abs/1703.01440
3. Baader, F.: On the complexity of Boolean unification. Technical report (1999)
4. Bañeres, D., Cortadella, J., Kishinevsky, M.: A recursive paradigm to solve Boolean relations. IEEE Trans. Comput. **58**(4), 512–527 (2009)
5. Benedetti, M.: sKizzo: a suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 369–376. Springer, Heidelberg (2005). doi:10.1007/11532231_27
6. Boole, G.: The Mathematical Analysis of Logic. Philosophical Library (1847). https://books.google.co.in/books?id=zv4YAQAAIAAJ
7. Boudet, A., Jouannaud, J.P., Schmidt-Schauss, M.: Unification in Boolean rings and Abelian groups. J. Symb. Comput. **8**(5), 449–477 (1989). http://dx.doi.org/10.1016/S0747-7171(89)80054–9
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986). http://dx.doi.org/10.1109/TC.1986.1676819
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
10. Deschamps, J.P.: Parametric solutions of Boolean equations. Discret. Math. **3**(4), 333–342 (1972). http://dx.doi.org/10.1016/0012-365X(72)90090–8
11. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based Boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Heidelberg (2016). doi:10.1007/978-3-319-41540-6_22
12. Jiang, J.-H.R.: Quantifier elimination via functional composition. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 383–397. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_30
13. Balabanov, V., Jiang, J.-H.R.: Resolution proofs and skolem functions in QBF evaluation and applications. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 149–164. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_12

14. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005). doi:10.1007/11513988_23
15. John, A., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: FMCAD, pp. 73–80 (2015)
16. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. CAD of Integr. Circuits Syst. **21**(12), 1377–1394 (2002). http://dblp.uni-trier.de/db/journals/tcad/tcad21.html#KuehlmannPKG02
17. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 113–123. Springer, Heidelberg (2000). doi:10.1007/10722167_12
18. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. SIGPLAN Not. **45**(6), 316–329 (2010)
19. Logic, B., Group, V.: ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/
20. Lowenheim, L.: Über die Auflösung von Gleichungen in Logischen Gebietkalkul. Math. Ann. **68**, 169–207 (1910)
21. Macii, E., Odasso, G., Poncino, M.: Comparing different Boolean unification algorithms. In: Proceedings of 32nd Asilomar Conference on Signals, Systems and Computers, pp. 17–29 (2006)
22. Marijn Heule, M.S., Biere, A.: Efficient extraction of Skolem functions from QRAT proofs. In: Proceedings of FMCAD (2014)
23. Martin, U., Nipkow, T.: Boolean unification - the story so far. J. Symb. Comput. **7**(3–4), 275–293 (1989). http://dx.doi.org/10.1016/S0747-7171(89)80013–6
24. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Satisfi. Boolean Model. Comput. **9**, 53–58 (2014 (published 2015))
25. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
26. Akshay, S., Chakraborty, S., John, A., Shah, S.: Website for TACAS 2017 Experiments (2016). https://drive.google.com/drive/folders/0BwmvCTZAETPvVExUQkx6WVEtWWs
27. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005, pp. 281–294 (2005)
28. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT **15**(5–6), 497–518 (2013)
29. Trivedi, A.: Techniques in symbolic model checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India (2003)
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Structures in Constructive Mathematics and Mathematical Logic, Part II. Seminars in Mathematics, pp. 115–125 (1968)