BDD-Based Boolean Functional Synthesis

Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi

Department of Computer Science, Rice University

Abstract. *Boolean functional synthesis* is the process of automatically obtaining a constructive formalization from a declarative relation that is given as a Boolean formula. Recently, a framework was proposed for Boolean functional synthesis that is based on Craig Interpolation and in which Boolean functions are represented as And-Inverter Graphs (AIGs). In this work we adapt this framework to the setting of Binary Decision Diagrams (BDDs), a standard data structure for representation of Boolean functions. Our motivation in studying BDDs is their common usage in *temporal synthesis*, a fundamental technique for constructing control software/hardware from temporal specifications, in which Boolean synthesis is a basic step. Rather than using Craig Interpolation, our method relies on a technique called *Self-Substitution*, which can be easily implemented by using existing BDD operations. We also show that this yields a novel way to perform quantifier elimination for BDDs. In addition, we look at certain BDD structures called *input-first*, and propose a technique called *TrimSubstitute*, tailored specifically for such structures. Experiments on scalable benchmarks show that both Self-Substitution and TrimSubstitute scale well for benchmarks with good variable orders and significantly outperform current Boolean-synthesis techniques.

1 Introduction

Boolean functions appear in all levels of computing, and can fairly be considered as one of the most fundamental building block of modern digital computers. Often, the most intuitive way of defining a Boolean function is not *constructively*, describing how the outputs can be computed from the inputs, but rather *declaratively*, as a *relation* between input and output values that must be satisfied [4]. Nevertheless, in order to implement a function in a practical format, such as in a circuit or program, a declarative definition is not enough, and a constructive description of how to compute the output from the input is necessary. The process of going from a declarative formalization to a constructive one is called *functional synthesis* [14]. This transformation is a challenging algorithmic problem, which we focus on in this paper.

In this work, we follow a framework proposed in [16, 17] for algorithmically synthesizing a correct-by-construction constructive representation of a desired Boolean function from a relational specification. Such relation is given as a propositional formula that relates input and output variables. Our construction ensures that when the input is *realizable*, that is, there is a corresponding

output for that specific input, the function that we synthesize produces this output. More formally, given a specification in the form of a characteristic function $f : \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$, where $f(\boldsymbol{x}, \boldsymbol{y}) = 1$ iff $\boldsymbol{y}$ is a correct output for the input $\boldsymbol{x}$, we synthesize functions $r_f : \mathbb{B}^m \to \mathbb{B}$ and $g : \mathbb{B}^m \to \mathbb{B}^n$ with the guarantee that $r_f(\boldsymbol{x}) = 1$ precisely when there exists some $\boldsymbol{y}$ for which $f(\boldsymbol{x}, \boldsymbol{y}) = 1$ and $f(\boldsymbol{x}, g(\boldsymbol{x})) = 1$ for every input vector $\boldsymbol{x}$ for which $r_f(\boldsymbol{x}) = 1$. As such, our framework consists of two phases. The first phase is the *realizability* phase, and requires the computation of the Boolean realizability function r_f. The second phase is the *function-construction* phase, in which we construct the function g.

The proposed framework in [17] is based on representing Boolean functions by means of And-Inverter Graphs (AIGs) [20]. In this paper we adapt this framework to the setting of Reduced Ordered Binary Decision Diagrams (BDDs) [6], a data structure designed for the efficient representation and manipulation of Boolean functions. BDDs provide easy-to-manipulate canonical (and minimal) representations of Boolean functions in which Boolean operations can be implemented efficiently. BDDs have found numerous applications in a variety of settings, including model checking [7], equivalence checking [24], and others. Our main motivation for using BDDs is that Boolean functional synthesis is also a basic step in *temporal synthesis*, a fundamental technique for constructing control software/hardware from temporal specifications [26], which is most often implemented by using BDDs, cf. [3]. Thus, our approach can be easily incorporated to temporal-synthesis tools. We discuss the differences between the AIG-based and BDD-based approaches below.

At the heart of our approach there is a technique we call *Self-Substitution*, a simplification of the Craig Interpolation-based approach that appears in [17]. A single-step Self-Substitution enables us to extract a function g *syntactically* from the function f for a case in which there is a single output variable. When there are multiple output variables, we iterate the single-step for each of the output variables. In this way we can use Self-Substitution both for quantifier elimination, in the realizability phase, and for constructing a function g, one output variable at a time. We use the software tool CUDD [28] for our implementation, and show that Self-Substitution can be efficiently implemented through basic BDD operations by using the CUDD API. Thus, Self-Substitution provides a novel way to perform quantifier elimination for BDDs, where the standard technique has been *Shannon Expansion* [6].

We begin the synthesis process by converting the relational specification f into a BDD. To obtain r_f we quantify out the output variables existentially one by one, which can be done by either Shannon Expansion or Self-Substitution. Eliminating the output variables one by one yields a *realizability sequence* $f_n, \ldots, f_0$ of BDDs, where f_n is the specification f, and f_0 is the desired realizability function r_f. In the function-construction phase again we use Self-Substitution, leveraging the realizability sequence $f_1, \ldots, f_n$ to construct a function, represented as a BDD, for each output variable. At the end, we obtain an n-rooted BDD for the implementation function g, where each root represents a single output variable. Motivated by [22], we study Self-Substitution on

a specific BDD order called *input-first*. In input-first BDDs, all input variables precede output variables. We develop a novel method, *TrimSubstitute*, which tailors Self-Substitution for input-first BDDs.

Our experimental evaluation relies on *scalable* problem instances rather than a random collection of problem instances, so we can evaluate the scalability of our techniques. Our evaluations demonstrate that the proposed framework scales well when the problem admits a good variable order, which is a well-established property of BDDs [6]. Our comparisons also showed that our method outperforms the previous AIG-based approach and other state-of-the-art tools by orders of magnitude. We also compare Self-Substitution as a quantifier-elimination technique against the standard Shannon-Expansion technique, and show that in many cases Self-Substitution scales better than Shannon Expansion. In addition, we show that TrimSubstitute outperforms Self-Substitution on input-first BDDs. The tool we built to implement our framework, RSynth, is available on-line[1].

The contributions of this paper are as follows: We offer a BDD-based approach for Boolean Synthesis that is simple and requires only basic BDD procedures. We show that our method outperforms other synthesis tools on scalable benchmarks. Our method also suggests a novel way for BDD based quantifier elimination. In addition we also offer a technique for input-first BDD in which we tailor our method specifically to this BDD order and show that we outperform all others tools.

Related Work Functional Boolean synthesis has been the goal of a number of different works over the years, focusing on different applications in both hardware and software design. In some literature, our definition of functional synthesis is also called *uniformization* [13]. Note however, that our definition is different than that of *logic synthesis*, which is used in tools such as ABC, in which a given circuit structure is transformed to meet certain criteria [5].

Several approaches have been proposed for functional Boolean synthesis, but one trend that can be observed among them is that BDDs seem to be a popular choice of data structure to use for the underlying representation of Boolean functions, despite a common concern regarding their scalability. Kuncak et al. developed a general framework for functional synthesis, focusing mainly on unbounded domains, such as integer arithmetic and sets with size constraints [22]. For Boolean logic, they suggest to start with a BDD following what we call an input-first order. Our work on input-first BDDs can be viewed as an elaboration of their work. Tronci considered synthesis of Boolean functions in a work that is focused on synthesizing optimal controllers [29]. He mentioned a basic form of Self-Substitution for function construction to extract an implementation function from a relational specification, but did not develop the idea and did not exploit Self-Substitution as a quantifier-elimination technique. Kukula and Shiple [21] were the first to address explicitly the issue of converting relational to functional specifications. They present a direct mapping of a BDD for a relation to a circuit, where each node of the BDD is converted into a hardware module

[1] see `http://www.cs.rice.edu/~lm30/RSynth/`

composed of several logic gates. Their approach is quite complex, and was not accompanied by an empirical evaluation. Bañeres, Cortadella, and Kishinevsky also addressed the problem for converting relational to functional specifications [1]. Their approach is based on a recursive search in which the cost function is a parameter. In this sense, their work is focused on optimizing the output size, rather than scalability, as in our work. Another search-based synthesis tool is Sketch [27], where the user specifies the behavior of a desired function, which the tool finds by searching through the space of possible implementations. A very recent work of [19] adapts [17] to synthesis for relations specified as large conjunctions of small formulas. Their work also makes use of Self-Substitution for function construction, but does not use BDDs. Certain QBF solvers [2, 15] include the capability of producing witnesses from the proof of validity of a formula. For valid QBF formulas these witnesses can be computed fairly efficiently, but cannot be applied when the formula is not valid.

Boolean synthesis lies at the heart of temporal synthesis, as temporal synthesis for the temporal formula $\Box f$ ("globally f"), where f is a Boolean formula, essentially requires functional Boolean synthesis for the formula f. There are several tools for temporal synthesis [10, 18, 25], yet the focus of such tools is on dealing with temporal formulas, while dealing with the underlying Boolean formulas is ignored or delegated to Boolean-synthesis tools.

Our framework is based on that of Jiang [16, 17], also concerned with extracting functions from Boolean relations. That work uses *and-inverters graphs* (AIGs) as the basic data structure, and uses Craig Interpolation for quantifier elimination and witness extraction. As seen below, our experiments have shown that this interpolation-based approach does not scale well and is very unpredictable. This has also been noted in [19].

Since our approach can also be used for quantifier elimination, we compare it with the standard quantifier elimination technique of Shannon Expansion. Other quantifier elimination techniques such as Goldberg and Manolios [12], require the formulas to be in CNF form, rather than the BDD representation we use. In addition, these techniques eliminate variables in blocks, while to compute each witness variable separately, our synthesis requires the variables to be eliminated individually.

2 Preliminaries

Boolean Functions We denote by $\mathbb{B} = \{0, 1\}$ the set of Boolean values. For simplicity, we often conflate an m-ary Boolean function $f : \mathbb{B}^m \to \mathbb{B}$ with its representation by means of a Boolean formula f with m propositional variables. Then $f(\boldsymbol{\sigma}) = 1$ if and only if $\boldsymbol{\sigma} \in \mathbb{B}^m$ is a satisfying assignment for f. Two formulas $f(\boldsymbol{x})$ and $f'(\boldsymbol{x})$ are *logically equivalent*, denoted $f \equiv f'$, if $f(\boldsymbol{\sigma}) = f'(\boldsymbol{\sigma})$ for every assignment $\boldsymbol{\sigma}$ for $\boldsymbol{x}$. Given formulas $f(x_1, \ldots, x_m)$ and $f'(y_1, \ldots, y_n)$, we use $f[x_i \mapsto f']$ to denote the formula $f(x_1, \ldots, x_{i-1}, f'(y_1, \ldots, y_n), x_{i+1}, \ldots, x_m)$, representing the functional composition of f in variable x_i with f'. A *Quantified Boolean Formula*, or QBF, is a Boolean formula in which some variables can be

universally or existentially quantified. In this work we assume that all the QBF are in *prenex normal form* in which all the quantifiers are grouped together before the quantifier-free part of the formula. Every QBF can be converted into a logically equivalent quantifier-free formula through a process called *quantifier elimination*. This is usually performed using the technique of *Shannon Expansion* [6], where $\forall x f \equiv f[x \mapsto 0] \wedge f[x \mapsto 1]$, and $\exists x f \equiv f[x \mapsto 0] \vee f[x \mapsto 1]$. Given a QBF in prenex normal form, we can obtain its equivalent quantifier-free formula by eliminating the quantifiers from the inside out.

Binary Decision Diagrams A *[Reduced Ordered] Binary Decision Diagram*, or BDD, is a data structure that represents a Boolean function as a directed acyclic graph [6]. BDDs can be seen as a reduced representation of a binary decision tree of a Boolean function. We require that variables are ordered the same way along every path of the BDD ("ordered") and that the BDD is minimized to eliminate duplication ("reduced"). For a given variable order, the reduced BDD is *canonical*. The variable order used can have a major impact on its size, and two BDDs representing the same function but with different orders can have an exponential difference in size. Consequently, finding a good variable order is essential for BDD-based Boolean reasoning. Since BDDs represent Boolean functions, they can be manipulated using standard Boolean operations. We overload the notation of the operators $\neg$, $\wedge$, $\vee$ and functional composition (e.g. $B[x_i \mapsto B']$) with equivalent semantics to their counterparts for Boolean formulas.

3 Theoretical Framework

3.1 Realizability and Synthesis

The problem of synthesis of Boolean functions is formally defined as follows.

Problem 1. Given a relation between two vectors of Boolean variables represented by the characteristic function $f : \mathbb{B}^m \times \mathbb{B}^n \to \mathbb{B}$, obtain a function $r_f : \mathbb{B}^m \to \mathbb{B}$ such that $r_f(\boldsymbol{x}) = 1$ exactly for the inputs $\boldsymbol{x} \in \mathbb{B}^m$ for which $\exists \boldsymbol{y} f(\boldsymbol{x}, \boldsymbol{y})$, and a function $g : \mathbb{B}^m \to \mathbb{B}^n$ such that $f(\boldsymbol{x}, g(\boldsymbol{x})) = 1$ if and only if $r_f(\boldsymbol{x}) = 1$.

In the context of this problem, f is called the *specification*, g is called the *implementation* or *witness function*, and r_f is called a *realizability function*. The specification is interpreted as describing a desired relationship between inputs and outputs of a function, and the implementation describes how to obtain an output from an input such that this relationship is maintained. The realizability function indicates for which inputs the specification can be satisfied. in the expression $f(\boldsymbol{x}, \boldsymbol{y})$, $\boldsymbol{x} = (x_1, \ldots, x_m)$ are called the *input variables*, and $\boldsymbol{y} = (y_1, \ldots, y_n)$ the *output variables*. The function that gives the i-th bit of $g(\boldsymbol{x})$ is called a *witness-bit function*, and is denoted by $g_i(\boldsymbol{x})$. A Boolean function $f(\boldsymbol{x}, \boldsymbol{y})$ is said to be *realizable for an input* $\boldsymbol{\sigma}$ if $\exists \boldsymbol{y} f(\boldsymbol{\sigma}, \boldsymbol{y}) \equiv 1$. We say that f is *realizable* if $\forall \boldsymbol{x} \exists \boldsymbol{y} f(\boldsymbol{x}, \boldsymbol{y}) \equiv 1$. For every assignment $\boldsymbol{\sigma}$ for $\boldsymbol{x}$ such that f is

realizable for $\boldsymbol{\sigma}$ we will have that $r_f(\boldsymbol{\sigma}) = 1$. In this case, $g(\boldsymbol{\sigma})$ is called a *witness* for $\boldsymbol{\sigma}$. In case $r_f(\boldsymbol{\sigma}) = 0$ we are not concerned about the output of g since f is unrealizable for $\boldsymbol{\sigma}$.

Following [17], the structure of our solution takes two steps, 1) Realizability, where we obtain r_f by constructing a sequence of formulas with progressively fewer output variables, and 2) Function construction, in which we synthesize a witness-bit function from every formula in the sequence obtained in the realizability step.

To perform both steps, we suggest a novel method called *Self-Substitution*. In Section 2 we observed that Boolean quantifier elimination is usually performed via Shannon Expansion. More recently, it was proposed to use Craig Interpolation for quantifier elimination (see [16]). We now introduce Self-Substitution as an alternative quantifier-elimination technique.

Lemma 1. *(Self-Substitution for Quantifier Elimination) Let $\varphi = Qyf(\boldsymbol{x}, y)$ be a QBF formula, where Q is either a universal or existential quantifier and f is quantifier-free. Let q be 0 if Q is universal and 1 if Q is existential. Then, $Qyf(\boldsymbol{x}, y)$ is logically equivalent to $f(\boldsymbol{x}, f(\boldsymbol{x}, q))$, and is also logically equivalent to $f(\boldsymbol{x}, \neg f(\boldsymbol{x}, \neg q))$.*

Proof. If Q is an existential quantifier, we prove that for every assignment $\boldsymbol{\sigma}$ for $\boldsymbol{x}$, $\exists y f(\boldsymbol{\sigma}, y) = 0$ iff $f(\boldsymbol{\sigma}, f(\boldsymbol{\sigma}, 1)) = 0$: If $\exists y f(\boldsymbol{\sigma}, y) = 0$, then $f(\boldsymbol{\sigma}, y) = 0$ for all possible assignments of y. Since this includes $f(\boldsymbol{\sigma}, 1)$, then $f(\boldsymbol{\sigma}, f(\boldsymbol{\sigma}, 1)) = 0$. On the other hand, if $f(\boldsymbol{\sigma}, f(\boldsymbol{\sigma}, 1)) = 0$, then it cannot be the case that $f(\boldsymbol{\sigma}, 1) = 1$ (otherwise $f(\boldsymbol{\sigma}, f(\boldsymbol{\sigma}, 1)) = f(\boldsymbol{\sigma}, 1) = 1$). Therefore, $f(\boldsymbol{\sigma}, 1) = 0$, and so $f(\boldsymbol{\sigma}, 0) = f(\boldsymbol{\sigma}, f(\boldsymbol{\sigma}, 1)) = 0$. Since both $f(\boldsymbol{\sigma}, 1) = 0$ and $f(\boldsymbol{\sigma}, 0) = 0$, then $\exists y f(\boldsymbol{\sigma}, y) = 0$. The claim that for every assignment $\boldsymbol{\sigma}$, $\exists y f(\boldsymbol{\sigma}, y) = 0$ iff $f(\boldsymbol{\sigma}, \neg f(\boldsymbol{\sigma}, 0)) = 0$ is proved analogously. The proof when Q is a universal quantifier is derived by using the identity $\forall y f(\boldsymbol{x}, y) \equiv \neg \exists y \neg f(\boldsymbol{x}, y)$. □

Following Lemma 1, quantifier elimination can be performed by replacing quantified formulas by their quantifier-free equivalents. Table 1 compares the formulas produced by quantifier elimination using Shannon Expansion and Self-Substitution.

The Self-Substitution method looks surprising at first glance. In the Shannon-Expansion method it is easy to see that the size of the quantifier-free formula becomes exponential compared to its quantified version, as it is a disjunction of all possible assignments. In Self-Substitution such a blow-up also takes place, but the encapsulation of all assignments is more subtle. The depth of the nested functions for a formula with n quantified variables is $n + 1$. Therefore all the possible assignments for the quantified variables can be obtained recursively. For example let $q_i = 1$ if the quantifier Q_i is existential, and $q_i = 0$ if Q_i is universal. Then a possible expansion for two quantified variables is $Q_1 y_1 Q_2 y_2 f(\boldsymbol{x}, y_1, y_2) = Q_1 y_1 f(\boldsymbol{x}, y_1, f(\boldsymbol{x}, y_1, q_2)) = f(\boldsymbol{x}, f(\boldsymbol{x}, q_1, f(\boldsymbol{x}, q_1, q_2)), f(\boldsymbol{x}, f(\boldsymbol{x}, q_1, f(\boldsymbol{x}, q_1, q_2)), q_2))$.

The following lemma which appeared in many forms in various places, e.g. [1, 17, 29], is derived from Lemma 1 and shows how Self-Substitution can be used for synthesis purposes.

Table 1: Equivalent formulas using each method of quantifier elimination

	$\forall y f(\boldsymbol{x}, y)$	$\exists y f(\boldsymbol{x}, y)$
Shannon Expansion	$f(\boldsymbol{x}, 0) \wedge f(\boldsymbol{x}, 1)$	$f(\boldsymbol{x}, 0) \vee f(\boldsymbol{x}, 1)$
Self-Substitution 1	$f(\boldsymbol{x}, f(\boldsymbol{x}, 0))$	$f(\boldsymbol{x}, f(\boldsymbol{x}, 1))$
Self-Substitution 2	$f(x, \neg f(\boldsymbol{x}, 1))$	$f(\boldsymbol{x}, \neg f(\boldsymbol{x}, 0))$

Lemma 2. *(Synthesis by Self-Substitution) Let $f(\boldsymbol{x}, y)$ be a Boolean formula with free variables $\boldsymbol{x}$ and y. Then $f(\boldsymbol{x}, 1)$ and $\neg f(\boldsymbol{x}, 0)$ are witness functions to $f(\boldsymbol{x}, y)$.*

Proof. By Lemma 1, $\exists y f(\boldsymbol{x}, y)$ is logically equivalent to $f(\boldsymbol{x}, f(\boldsymbol{x}, 1))$ and to $f(\boldsymbol{x}, \neg f(\boldsymbol{x}, 0))$. Since $r_f(\boldsymbol{x}) = 1$ exactly for those $\boldsymbol{x}$ for which $\exists y f(\boldsymbol{x}, y)$ holds, both $f(\boldsymbol{x}, f(\boldsymbol{x}, 1))$ and $f(\boldsymbol{x}, \neg f(\boldsymbol{x}, 0))$ return 1 if and only if $r_f(\boldsymbol{x}) = 1$. Thus, $f(\boldsymbol{x}, 1)$ and $\neg f(\boldsymbol{x}, 0)$ are witness functions to $f(\boldsymbol{x}, y)$. □

The witness $f(\boldsymbol{x}, 1)$ is called the *default*-1 witness, while the witness $\neg f(\boldsymbol{x}, 0)$ is called the *default*-0 witness. The observation in [17] is that when f is realizable for all $\boldsymbol{x}$, the conjunction of the two formulas $\neg f(\boldsymbol{x}, 0)$ and $\neg f(\boldsymbol{x}, 1)$ is unsatisfiable. From a resolution proof of this unsatisfiability, one can extract a Craig interpolant, which may be smaller than either $f(\boldsymbol{x}, 1)$ or $\neg f(\boldsymbol{x}, 0)$. Our experimental evaluation for our benchmarks does not support this expectation, where we show the advantage of using the witness function $f(\boldsymbol{x}, 1)$ for synthesis and $f(\boldsymbol{x}, f(\boldsymbol{x}, 1))$ for existential-quantifier elimination.

3.2 Realizability and Function-Construction Using BDDs

Similarly to [17], we separate the synthesis approach into two phases. We call the first the *realizability* phase, and the second the *function construction* phase. We assume that the input is in the form of a BDD B_f that describes the function $f(\boldsymbol{x}, \boldsymbol{y})$. When f is obvious from the reference, we denote B_f by B.

Realizability Our definition of r_f requires that r_f returns 1 exactly for those assignments of $\boldsymbol{x}$ for which $\exists \boldsymbol{y} f(\boldsymbol{x}, \boldsymbol{y})$. This means r_f can be obtained by applying quantifier elimination on the output variables. Recall that f has n output variables $y_1, \ldots, y_n$. Typically, the order of variables makes a major difference in constructing a BDD. However, in this section we assume no specific order.

The basic idea is as follows: from the input BDD B, we construct a sequence $\boldsymbol{B} = \{B_n, B_{n-1}, \ldots B_1, B_0\}$ of BDDs, where $B_n = B$, such that B_{i-1} is logically equivalent to $\exists y_i B_i$. Therefore, the BDD B_{i-1} is constructed from B_i by eliminating the existentially quantified variable y_i. The elimination process guarantees that B_0 represents the realizability function r_f.

The elimination of y_i from B_i can be done via either Shannon Expansion, or via Self-Substitution. For Shannon Expansion, we define $B_{i-1} = B_i[y_i \mapsto$

$0] \vee B_i[y_i \mapsto 1]$. To use the Self-Substitution method, we define either $B_{i-1} = B_i[y_i \mapsto B_i[y_i \mapsto 1]]$ to construct the default-1 witness for y_i or $B_{i-1} = B_i[y_i \mapsto \neg B_i[y_i \mapsto 0]]$ to construct the default-0 witness for y_i.

Function Construction We next use the BDD sequence obtained in the realizability process to construct a sequence of BDDs $\boldsymbol{W} = \{W_n, W_{n-1}, \ldots, W_1\}$, each emitting an output bit.

By using Lemma 2, we perform the function-construction step as follows. Let $\boldsymbol{B} = \{B_n, B_{n-1}, \ldots, B_1, B_0\}$ be the BDD sequence obtained in the realizability step, and note that the output variables in the BDD B_i are $y_1, \ldots, y_i$. We first construct W_1 from B_1 by setting $W_1 = B_1[y_1 \mapsto 1]$ for a default-1 witness for y_1 or $W_1 = \neg B_1[y_1 \mapsto 0]$ for a default-0 witness for y_1. The structure of BDDs allows us to define both B_1 and $\neg B_1$ without extra effort. Next, we inductively define either $W_i = B_i[y_1 \mapsto W_1, \ldots, y_{i-1} \mapsto W_{i-1}, y_i \mapsto 1]$ for a default 1 witness for y_i, or $W_i = \neg B_i[y_1 \mapsto W_1, \ldots, y_{i-1} \mapsto W_{i-1}, y_i \mapsto 0]$ for a default 0 witness for y_i. Thus, every W_i has only the input variables $\boldsymbol{x}$, and represents the witness-bit function $g_i(\boldsymbol{x})$. Thus, the proof for the following theorem follows from Lemma 2.

Theorem 1. *For every assignment $\boldsymbol{\sigma}$ for $\boldsymbol{y}$, the sequence $(g_1(\boldsymbol{\sigma}), \ldots g_n(\boldsymbol{\sigma}))$ is a witness to $\boldsymbol{\sigma}$. Thus $\boldsymbol{W}$ describes a witness function for B.*

In practice, we chose, for simplicity, to use only the default-1 witnesses. In principle, one could always choose the best among the default-0 and default-1 witnesses. Since, however, we have n output variables, and the assignment of one of them affects the others, finding the optimal combination of bit-witness functions requires optimizing over an exponentially large space, which is an expensive undertaking. Finding such combinations of functions is a matter of future work.

3.3 Synthesis of Input-First BDD

An *input-first* BDD is a BDD in which all the input (universal) variables precede all the output (existential) variables. Synthesis using input-first BDDs was suggested in [22], but an explicit way to do it was not provided. This specific order of variables of input-first BDDs has led us to develop a method called *Trim-Substitute* for synthesis of input-first BDDs, in which we tailor Self-Substitution specifically for the input-first order. Given the input BDD, the running time of TrimSubstitute is at most quadratic in its size. In Section 4 we show that Trim-Substitute indeed outperforms Self-Substitution on input-first BDDs. In this section we give an outline of our method. Full proof with an example appears in the appendix. For simplicity TrimSubstitute produces default-1 witnesses. With minor modification the TrimSubstitute method can produce any desired combination of bit-witness functions.

An *output node* (resp. *input node*) in a BDD B is a node labeled with an output (resp. input) variable. Recall that every non-terminal node in B has exactly two children called *high-child* and *low-child*. Let B be an input-first BDD. We define $Fringe(B)$ to be the collection of all output nodes and terminal nodes

in B that have an input node as an immediate parent. Note that $Fringe(B)$ can be found by performing standard graph-search operations (e.g. Depth-First-Search) on B. Also note that B is realizable exactly for those assignments for which the corresponding node in $Fringe(B)$ is not the terminal node 0.

Given an input-first BDD B, we assume without loss of generality that the order of the output variables in B is $y_1, \ldots y_n$. We construct a sequence of witness BDDs $\boldsymbol{W} = (W_1, \ldots, W_n)$, in which every W_i contains only input variables, and is the witness-bit function $g_i(\boldsymbol{x})$. To obtain $\boldsymbol{W}$, we construct a sequence of BDDs $\boldsymbol{B'} = (B'_1, \ldots, B'_n)$ in which every B'_i is an input-first BDD that contains all input variables, plus only output variables from $y_i, \ldots, y_n$. We obtain W_i from B'_i by an operation called "trim", and obtain B'_{i+1} from B'_i and W'_i by an operation called "substitute", hence our method's name TrimSubstitute. We next describe how $\boldsymbol{B'}$ and $\boldsymbol{W}$ are obtained.

We assume by induction on $i \leq n$ that B'_i is an input-first BDD that is realizable for exactly the same inputs as B, and that contains input variables plus only output variables from $y_i, \ldots, y_n$. Setting $B'_1 = B$, we already satisfy these assumptions for the base case. We first construct W_i by "trimming" B'_i, which means replacing each node v in $Fringe(B'_i)$ with either the terminal node 0 or 1. Intuitively we construct W_i to produce an output bit for y_i in the "default-1" sense, i.e., W_i always produces 1 unless 1 is not a possible output bit for y_i. Formally, this is done as follows.

Note that if a $v \in Fringe(B'_i)$ is the terminal node 0, then the assignment to y_i is irrelevant since the path to v corresponds to an unrealizable input, and so it can be left as 0. If v is a variable node, it cannot be that both children of v are the terminal node 0, as otherwise v itself would be reduced to 0. Therefore, if v is labeled by y_i, and the high-child of v is not the terminal node 0, replace v with the terminal node 1. Otherwise, if v is labeled by y_i and the high-child of v is 0 (then the low-child of v is not 0), replace v with the terminal node 0. For all other cases (v is labeled y_j, where $j > i$, or v is the terminal node 1), replace v with the terminal node 1. Note that W_i has only input variables.

Finally, we use B'_i and W_i to construct B'_{i+1}. To do that, we define $B'_{i+1} = B'_i[y_i \mapsto W_i]$. That is, B'_{i+1} is constructed from B'_i by "substituting" y_i with W_i. By construction we have that B'_{i+1} is an input-first BDD that is realizable for the same inputs as B and that contains input variables plus only output variables from $y_{i+1}, \ldots, y_n$. Therefore, the induction assumption is maintained. An example of the construction can be found in the appendix.

Theorem 2. *The BDD sequence $\boldsymbol{W} = (W_1, \ldots, W_n)$ describes a witness function for B.*

In the last induction step we obtain an additional BDD B'_{n+1} which is realizable for the same inputs as B, but contains only input variables. As such, B'_{n+1} encodes the realizability function r_f.

4 Experimental Evaluation

We compare our approach with two current state-of-the-art methods: the Craig Interpolation-based approach [17] and Sketch [27]. In addition, we compare between Shannon Expansion and Self-Substitution as quantifier-elimination methods to be used for the realizability phase. Finally, we see how the TrimSubstitute method, specialized to input-first BDDs, compares with the generic Self-Substitution method when using this type of BDD.

Rather than using a random collection of problem instances for our experiments, we selected a collection of *scalable* benchmarks, presented in Table 2, that operate over vectors of Boolean variables. Each entry in the table represents a class of benchmarks parameterized by the length n of the vectors. This allows us to produce benchmarks of different size to measure how our techniques scale. For our experiments we vary n in powers of 2 between 8 and 1024, totaling 42 benchmark instances. The first five benchmark classes represent linear-arithmetic functions in which the vectors encode the binary representation of integers in n bits, while the sixth represents the sorting of a bit array of size n. The first column in Table 2 describes the function we synthesize, where $\boldsymbol{x}$ and $\boldsymbol{x'}$ are vectors of input variables and $\boldsymbol{y}$ is a vector of output variables. The relational specification of these functions are shown in the second column. These specifications are translated to propositional-logic formulas (see appendix for details) and given as input to the algorithm, which then constructs a BDD for the relational specification and synthesizes the implementation function. All benchmarks are realizable for every input[2], therefore the realizability function is just the constant 1.

Table 2: Benchmark classes used for synthesis. See the appendix for translation into propositional-logic formulas.

	Function to synthesize	Specification
Subtraction	$\boldsymbol{y} = \boldsymbol{x'} - \boldsymbol{x}$	$\boldsymbol{y} + \boldsymbol{x} = \boldsymbol{x'}$
Maximum	$\boldsymbol{y} = max(\boldsymbol{x}, \boldsymbol{x'})$	$(\boldsymbol{y} \geq \boldsymbol{x}) \wedge (\boldsymbol{y} \geq \boldsymbol{x'}) \wedge ((\boldsymbol{y} = \boldsymbol{x}) \vee (\boldsymbol{y} = \boldsymbol{x'}))$
Minimum	$\boldsymbol{y} = min(\boldsymbol{x}, \boldsymbol{x'})$	$(\boldsymbol{y} \leq \boldsymbol{x}) \wedge (\boldsymbol{y} \leq \boldsymbol{x'}) \wedge ((\boldsymbol{y} = \boldsymbol{x}) \vee (\boldsymbol{y} = \boldsymbol{x'}))$
Floor of average	$\boldsymbol{y} = \lfloor \frac{\boldsymbol{x} + \boldsymbol{x'}}{2} \rfloor$	$(2\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{x'}) \vee (2\boldsymbol{y} + 1 = \boldsymbol{x} + \boldsymbol{x'})$
Ceiling of average	$\boldsymbol{y} = \lceil \frac{\boldsymbol{x} + \boldsymbol{x'}}{2} \rceil$	$(2\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{x'}) \vee (2\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{x'} + 1)$
Sorting	$\boldsymbol{y} = sort(\boldsymbol{x})$	$sorted(\boldsymbol{y}) \wedge (\Sigma_{i=1}^{n} x_i = \Sigma_{j=1}^{n} y_j)$

For purposes of evaluation we have constructed a tool, called RSynth, implemented in C++11 using the CUDD BDD library [28]. Self-Substitution was implemented using the built-in method `Compose` for BDD composition. That way, for a BDD B representing a function $f(\boldsymbol{x}, y)$, the BDD for $f(\boldsymbol{x}, f(\boldsymbol{x}, 1))$

[2] The *Subtraction* class of benchmarks is defined for subtraction modulo 2^n, or equivalently subtraction in two's complement, which is realizable for all inputs.

is computed as `B.Compose(i, B.Compose(i, bddOne))`, where `bddOne` is the BDD for the constant 1 and `i` is the index of variable y. All the experiments in this paper were carried out on a computer cluster consisting of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8 GB of RAM per core. Since the algorithm has not been parallelized, the cluster was used solely to run different experiments simultaneously. The execution of each benchmark for a given n had a maximum time limit of 8 hours.

Scalability comparison with previous approaches We compared the performance of RSynth with the Craig Interpolation approach from [17] that synthesizes functions in the format of AIGs, and the Sketch synthesis tool [27] that uses syntax-guided search-based synthesis. The original tool for Craig Interpolation from [17] was not available, therefore we used an implementation of the same method, which is called MonoSkolem, from [19].

Since BDD sizes can blow up if a poor variable order is chosen, causing initial BDD construction time to dominate the overall running time, we selected a variable order that can be expected to produce efficient BDDs for our benchmarks. For that, we chose an order called *fully interleaved*, in which the variables are ordered according to their index, alternating input and output variables.

We show the results of the comparison for the *Subtraction*, *Maximum* and *Ceiling of Average* benchmark classes in Fig. 1. Similar results were obtained for the other arithmetic benchmarks. Recall that n is the number of variables in each vector $\boldsymbol{x}$, $\boldsymbol{x'}$ and $\boldsymbol{y}$, therefore the total number of variables in each case is $3n$, with $2n$ input variables and n output variables.

Sketch is omitted from Fig. 1 because it was unable to synthesize the benchmarks for any n greater than 3, in all cases either timing out or running out of memory. For the two remaining approaches, it is noticeable that RSynth outperformed MonoSkolem by orders of magnitude, and scaled significantly better.

Although these results seem to lean considerably in favor of our approach, note that the benchmark classes used so far are deterministic (relations that have a unique implementation), while Craig Interpolation is reported to produce better results for non-deterministic relations by exploiting the flexibility in the choice of witness. To address these factors, we added to the same setting an additional collection of linear arithmetic operations, represented in Table 3, this time of *non-deterministic* benchmarks.

Table 3: Non-deterministic benchmark classes

	Input	Output	Specification
Decomposition	$\boldsymbol{x}$	$\boldsymbol{y}$, $\boldsymbol{y'}$	$\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{y'}$
Equalization	$\boldsymbol{x}$, $\boldsymbol{x'}$	$\boldsymbol{y}$, $\boldsymbol{y'}$	$\boldsymbol{x} + \boldsymbol{y} = \boldsymbol{x'} + \boldsymbol{y'}$
Intermediate value	$\boldsymbol{x}$, $\boldsymbol{x'}$	$\boldsymbol{y}$	$(\boldsymbol{x} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{x'}) \vee (\boldsymbol{x'} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{x})$

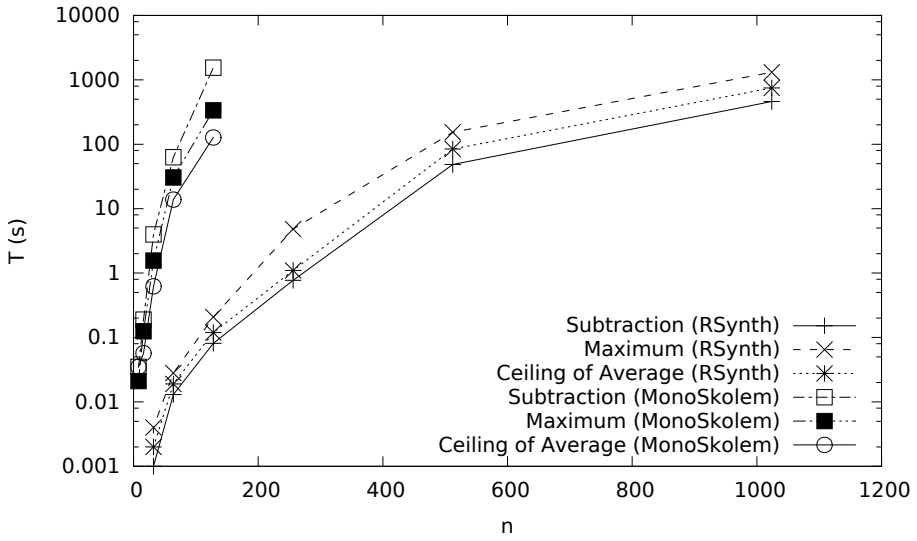


Fig. 1: Comparison of running time of RSynth against MonoSkolem

Contrary to expectations, as Fig. 2 shows, our method gives better performance for the non-deterministic benchmark classes as well. From this we can conclude that despite the flexibility that Craig Interpolation provides, it does not necessarily exploit the don't-cares of the input specification efficiently. These results are supported by the ones obtained in [19], which reported that the quality of the results obtained when using Craig Interpolation depended strongly on the interpolation procedure of finding good interpolants, something which is not guaranteed to happen. Comparison of the size of the implementation between RSynth and MonoSkolem also showed that the functions constructed by Craig Interpolation are much larger.

These results allow us to conclude that with a good variable order to the function being synthesized, our method scales well and outperforms previous approaches. For linear arithmetic operations, we can identify fully-interleaved to be such an order.

Shannon Expansion vs. Self-Substitution As mentioned in Section 3.2, the first step of the synthesis, realizability, requires quantifier elimination, which can be performed by either Shannon Expansion or Self-Substitution. We compared these two techniques by measuring the running time of the realizability phase using each of them. Our experiments show that the realizability step is responsible for only a small fraction of the running time of the synthesis. For the arithmetic benchmarks with fully-interleaved order, this step is performed in under 1s in all cases, even for $n = 1024$. In order to better observe the difference between

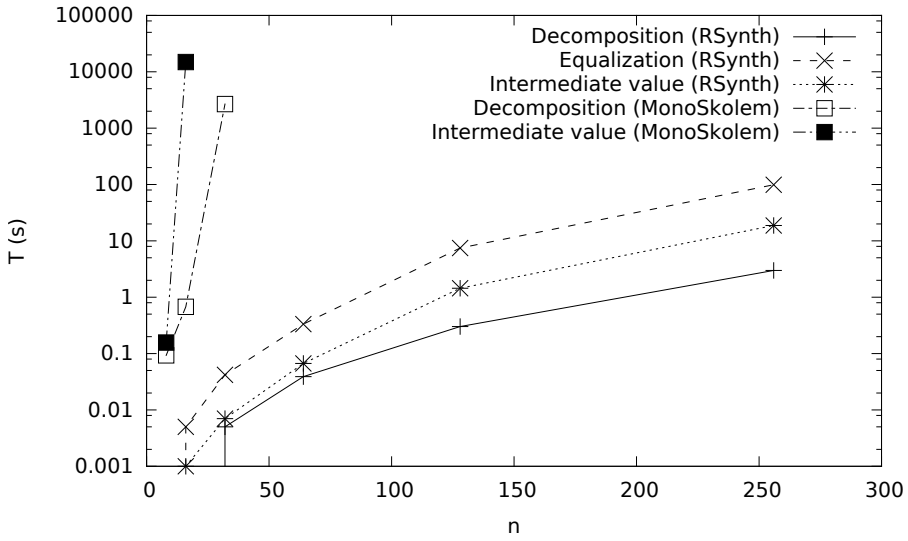


Fig. 2: Comparison of running time using non-deterministic benchmarks. The results for *Equalization* using MonoSkolem are not shown due to the synthesis timing out for $n > 8$.

the two quantifier-elimination techniques, we measured them using the *Sorting* benchmark class, for which the BDD representation is not as efficient.

As can be seen in Fig. 3, as n grows Self-Substitution tends to perform better than Shannon Expansion, taking approximately 40% less time to perform the realizability step for $n = 256$ (the same behavior was observed on the arithmetical benchmarks, using different variable orders). Thus, our experiments show an advantage in using Self-Substitution for quantifier elimination in the realizability step. Note that both Self-Substitution and Shannon Expansion are semantically equivalent, and thus produce identical BDDs. Therefore, the difference in performance between the two methods originates solely from the application of the CUDD operation itself over the constructed BDD. Shannon Expansion is currently the standard way of performing quantifier elimination on BDDs, but our experiments indicate that Self-Substitution interacts more efficiently with this type of data structure and should be considered as an alternative for practical applications.

Synthesis for input-first BDDs Following a suggestion in [22] for synthesis of propositional logic, we presented in Section 3.3 the TrimSubstitute method for BDDs that follow an input-first order. We compared the performance of TrimSubstitute with Self-Substitution (using Self-Substitution for both realizability and function construction) on input-first BDDs.

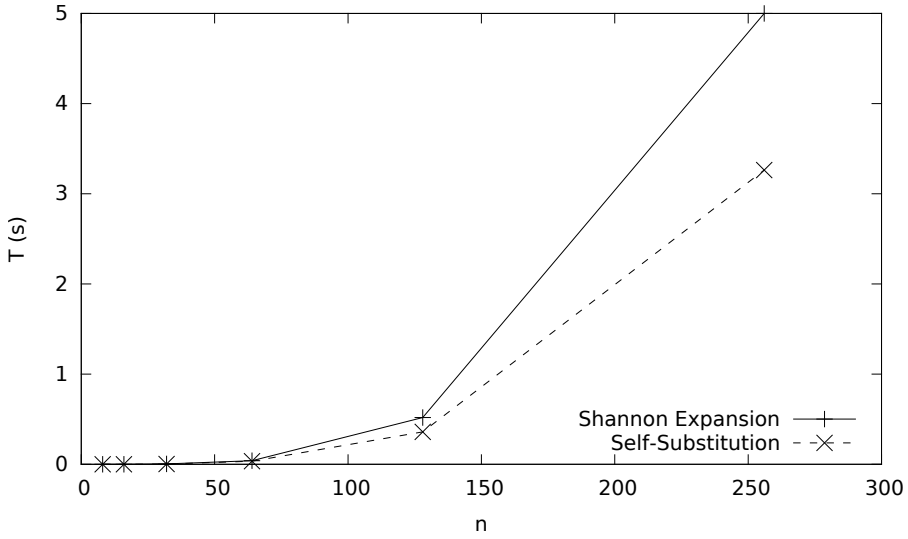


Fig. 3: Comparison of Shannon Expansion and Self-Substitution for realizability, for the *Sorting* benchmark class

We first observed that construction time of the input-first BDD for the arithmetic benchmark classes scales poorly and was very large even for a relatively small n. The reason is that in the input-first order, the BDD is forced to keep track of all relevant information about the input before looking at the output variables. Thus, the constructed BDD must have a path for every possible output of the function being synthesized. Since in the arithmetic benchmarks, the number of such paths is 2^n, it does not pay off to use an input-first order for these benchmarks, regardless of the efficiency of the synthesis algorithm used.

On the other hand, for other classes of specifications the amount of information that must be memorized about the input can be polynomial or even linear in size. An example for that is the *Sorting* benchmark class, in which it is only necessary to keep track of the number of 1s in the input; thus, only n paths are required in the constructed BDD. In this case, although the construction time of the initial BDD still dominates the running time (experiments showed construction to take around 1200s for $n = 256$), the size of the constructed BDD scales much better and makes synthesis feasible for a larger number of bits. The development of techniques to lessen the impact of construction time is a matter of future work.

Fig. 4 shows a comparison of running time between the Self-Substitution and TrimSubstitute methods for *Sorting*. We can see that TrimSubstitute greatly improves over Self-Substitution, performing around 50 times faster for $n = 256$. These results imply that when the specification can be efficiently represented as

an *input-first* BDD, TrimSubstitute can be used to obtain a significant improvement in synthesis time.

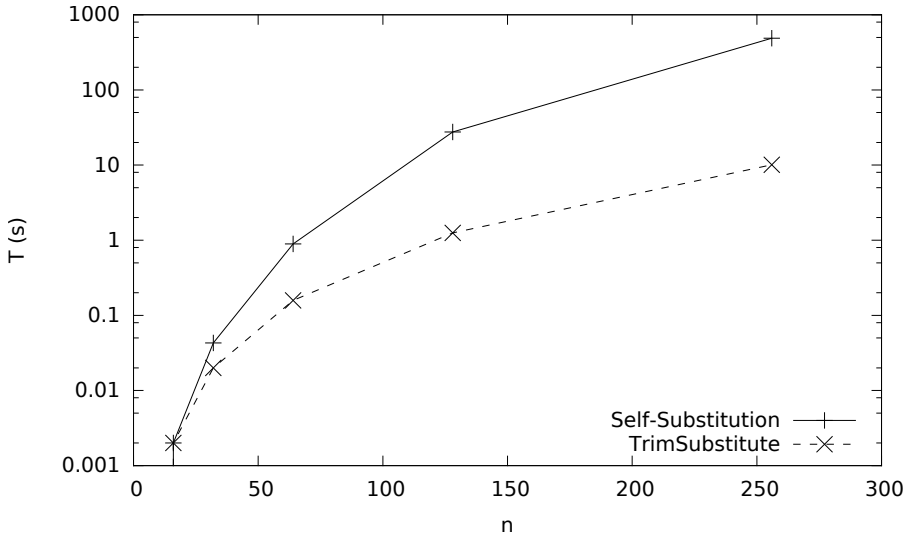


Fig. 4: Comparison of methods for synthesis using *input-first* BDDs for the *Sorting* benchmark class

5 Concluding Remarks

In this work we introduced BDD-based methods for synthesizing Boolean functions from relational specifications. We suggested a method called Self-Substitution for both quantifier elimination and function construction. We also suggested a method called TrimSubstitute, which outperforms Self-Substitution on input-first BDDs. We demonstrated that our methods scale well for benchmarks for which we have good BDD variable order, and outperform prior techniques.

A key challenge venue is to lessen the impact of the BDD size in the synthesis process. Factored representation of BDDs and early-quantification techniques, used in both symbolic model checking [8] and satisfiability testing [23], may be also helpful for synthesis. Another research direction is to find a good combination of bit-witness functions for specific benchmarks. There may also be BDD variants that can bring benefits in this area. For example, Free Binary Decision Diagrams (FBDDs) [11] relax the variable-order requirement in BDDs by allowing separate paths to use different orders. This might allow for more efficient representation of specifications in cases where an efficient global order is difficult to find. The Self-Substitution method as a technique for quantifier

elimination calls for further research, both in applied settings, for example, in symbolic model checking [8], and in theoretical settings, for example, in the study of Post classes and algebraic clones [9]. Finally, we plan to explore the extension of our techniques to the setting of temporal synthesis.

Acknowledgement This work is supported in part by NSF grants CCF-1319459 and IIS-1527668, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", by BSF grant 9800096, and by the Brazilian agencies CAPES and CNPq through the Ciência Sem Fronteiras program.

References

1. D. Bañeres, J. Cortadella, and M. Kishinevsky. A Recursive Paradigm to Solve Boolean Relations. *IEEE Trans. Computers*, 58(4):512–527, 2009.
2. Marco Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, pages 369–376, 2005.
3. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic Hardware Synthesis from Specifications: a Case Study. In *Proc. Conference on Design, Automation and Test in Europe*, pages 1188–1193. ACM, 2007.
4. R.K. Brayton and F. Somenzi. Minimization of Boolean Relations. In *IEEE Int'l Symp. on Circuits and Systems*, pages 738–743. IEEE, 1989.
5. Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 24–40, 2010.
6. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.
7. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^20 States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.
8. J.R. Burch, E.M. Clarke, and D.E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proc. 28th ACM/IEEE Design Automation Conference*, pages 403–407. ACM, 1991.
9. M. Couceiro, S. Foldes, and E. Lehtonen. Composition of Post Classes and Normal Forms of Boolean Functions. *Discrete Mathematics*, 306(24):3223–3243, 2006.
10. Rüdiger Ehlers. Symbolic Bounded Synthesis. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 365–379, 2010.
11. J. Gergov and C. Meinel. Boolean Manipulation with Free BDDs: An Application in Combinational Logic Verification. In *IFIP Congress (1)*, pages 309–314, 1994.
12. Eugene Goldberg and Panagiotis Manolios. Quantifier elimination via clause redundancy. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 85–92, 2013.
13. Yuri Gurevich and Saharon Shelah. Rabin's Uniformization Problem. *J. Symb. Log.*, 48(4):1105–1119, 1983.
14. G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

15. Marijn Heule, Martina Seidl, and Armin Biere. Efficient extraction of Skolem functions from QRAT proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 107–114, 2014.

16. J.R. Jiang. Quantifier Elimination via Functional Composition. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2009.

17. J.R. Jiang, H. Lin, and W. Hung. Interpolating Functions From Large Boolean Relations. In *2009 International Conference on Computer-Aided Design (ICCAD'09), November 2-5, 2009, San Jose, CA, USA*, pages 779–784. IEEE, 2009.

18. Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A Tool for Property Synthesis. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 258–262, 2007.

19. Ajith K. John, Shetal Shah, Supratik Chakraborty, Ashutosh Trivedi, and S. Akshay. Skolem Functions for Factored Formulas. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 73–80, 2015.

20. A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-Based Boolean Reasoning. In *Proc. Design Automation Conference*, pages 232–237. IEEE, 2001.

21. J.H. Kukula and T.R. Shiple. Building Circuits from Relations. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 113–123. Springer, 2000.

22. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete Functional Synthesis. In B. G. Zorn and A. Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 316–329. ACM, 2010.

23. G. Pan and M.Y. Vardi. Symbolic Techniques in Satisfiability Solving. *J. Autom. Reasoning*, 35(1-3):25–50, 2005.

24. V. Paruthi and A. Kuehlmann. Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation. In *Proc. Int'l Conf. on Computer Design*, pages 459–464. IEEE, 2000.

25. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) Designs. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 364–380, 2006.

26. A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. 16th ACM Symp. on Principles of Programming Languages*, pages 179–190, 1989.

27. A. Solar-Lezama, R.M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by Sketching for Bit-Streaming Programs. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, pages 281–294. ACM, 2005.

28. F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.5. 0.* University of Colorado at Boulder, 2012.

29. E. Tronci. Automatic Synthesis of Controllers from Formal Specifications. In *ICFEM*, pages 134–143, 1998.

A APPENDIX

A.1 Proof of TrimSubstitute

We prove Theorem 2. Let B be an input-first BDD, and let $\boldsymbol{B'} = (B'_1, \ldots, B'_n)$ and $\boldsymbol{W} = (W_1, \ldots, W_n)$ as defined in Section 3.3. Fig. 5 depicts our construction. Given a BDD D, and a node v in D, the subgraph D_v of D is obtained by restricting D to all the nodes that can be reached from v. Assume $z_1, \ldots z_k$ are the variables of D. Then by following a partial assignment $\boldsymbol{\nu}$ to the variables of $z_1, \ldots z_i$ for some i, we follow a unique path in D that ends up in a node v. Then the subgraph D_v is called the subgraph *reached* by following $\boldsymbol{\nu}$ in D.

Theorem 2. *The BDD sequence $\boldsymbol{W} = (W_1, \ldots, W_n)$ describes a witness function for B.*

Proof. Let $g_i : \mathbb{B}^m \to \mathbb{B}$ be the function that describes W_i. The following facts are easily proved by induction on i.

1. Following the construction of W_i, for every realizable assignment $\boldsymbol{\sigma}$ to the input variables, the path followed by $(\boldsymbol{\sigma}, g_i(\boldsymbol{\sigma}))$ in B'_i does not end in the terminal node 0.
2. Following fact (1), and the construction of B'_{i+1}, we have that for every realizable assignment $\boldsymbol{\sigma}$ to the input variables, the subgraph reached by following $\boldsymbol{\sigma}$ in B'_{i+1} is identical to the subgraph reached by following $(\boldsymbol{\sigma}, g_i(\boldsymbol{\sigma}))$ in B'_i. Therefore B'_{i+1} is realizable for $\boldsymbol{\sigma}$ as well.

As a result, we specifically have that for every realizable assignment $\boldsymbol{\sigma}$ to the input variables, the assignment $(\boldsymbol{\sigma}, g_1(\boldsymbol{\sigma}), \ldots, g_n(\boldsymbol{\sigma}))$ leads to the terminal node 1. This means that the BDD sequence $\boldsymbol{W} = (W_1, \ldots, W_n)$ describes a witness function for B. □

A.2 Encoding of Specifications in Propositional Logic

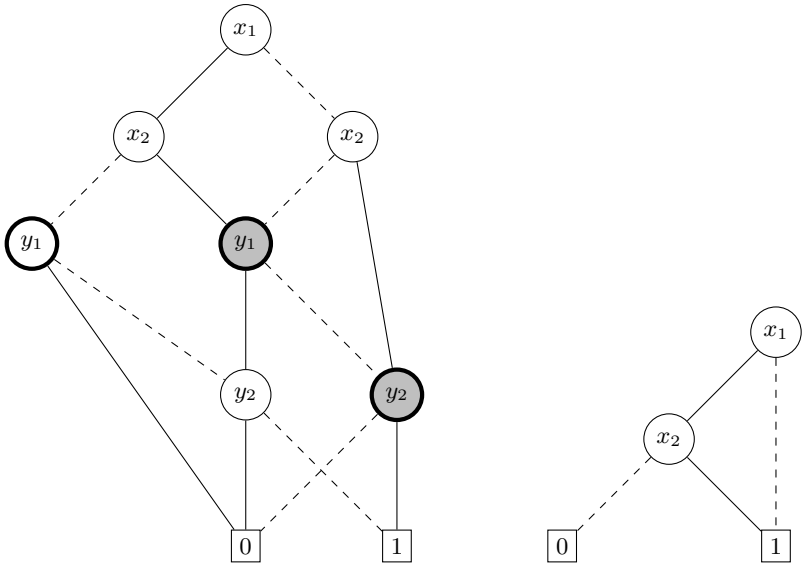
For completeness we show how to encode the specification given in Section 4, Table 2, into propositional logic formulas (later represented as a BDD). We assume that an integer is described by a vector of variables $\boldsymbol{z} = (z_n, z_{n-1}, \ldots, z_2, z_1)$, where z_n represents the most significant bit and z_1 the least significant bit. We now describe how specific operations used in the high-level specifications are encoded in propositional logic.

Relational Operations The formulas $(z = z')$, $(z \leq z')$ and $(z \geq z')$ are encoded respectively as $\varphi^{=}$, $\varphi^{\leq}$ and $\varphi^{\geq}$, as follows:

$$\varphi^{=} = \bigwedge_{i=1}^{n} (z_i \leftrightarrow z'_i) \tag{1}$$

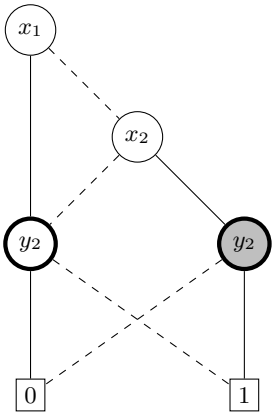
$$\varphi^{\leq} = \varphi_n, \text{ where } \varphi_i = (\neg z_i \wedge z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1}) \text{ and } \varphi_0 = 1 \tag{2}$$

$$\varphi^{\geq} = \varphi_n, \text{ where } \varphi_i = (z_i \wedge \neg z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1}) \text{ and } \varphi_0 = 1 \tag{3}$$

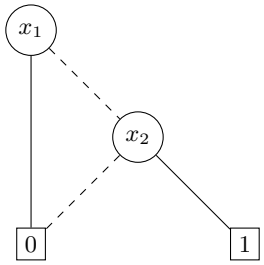


(a) $B'_1 = B$

(b) W_1



(c) B'_2



(d) W_2

Fig. 5: Example of the TrimSubstitute method for a BDD representing the formula $(((x_1 \to \neg y_1) \wedge (x_1 \oplus x_2) \wedge (x_1 \oplus y_2)) \vee ((x_1 \leftrightarrow x_2) \wedge (y_1 \oplus y_2)))$. Nodes with bold outlines are in $\mathit{Fringe}(B'_i)$, and are either white if they should be replaced by the leaf node 0 or gray if they should be replaced by the leaf node 1.

Addition Since addition is an operation that returns an integer rather than a Boolean it cannot be implemented as a single Boolean formula. Rather, it produces n formulas $\varphi_n^+, \ldots, \varphi_1^+$ representing a new integer, which can be later combined into a single formula through one of the relational operators above. The encoding for the + operator follows the usual representation of addition in binary: $\varphi_i^+ = z_i \oplus z'_i \oplus c_{i-1}$ where $c_i = (z_i \wedge z'_i) \vee (z_i \wedge c_{i-1}) \vee (z'_i \wedge c_{i-1})$.

In this encoding, c_i represents the carry-out from the addition in the i-th position. The carry-in for the first position, c_0, is normally 0, but can be set to 1 to add an extra term of 1 to the sum, which is useful in the formulas for average.

Recall that in the *Subtraction* benchmark class, + is interpreted as addition modulo n. On the other hand, in the high-level formulas for average we need the result of the addition with an extra bit added if necessary. This extra bit can be obtained by simply taking c_n. Therefore the comparisons in these formulas are actually performed over $(n+1)$-bit integers.

Sorting The specification for the *Sorting* benchmark class requires a more careful encoding. Recall that its high-level specification is given as $\mathit{sorted}(\boldsymbol{y}) \wedge (\Sigma_{i=1}^n x_i = \Sigma_{j=1}^n y_j)$ where $\boldsymbol{x}$ and $\boldsymbol{y}$ are interpreted as bit arrays. The first conjunct says that the output must be sorted, meaning that all 0 bits must precede all 1 bits. This is defined recursively for a range of consecutive positions $y_i, \ldots, y_j$ by saying that either all the variables are assigned to 1, or the first is 0 and the rest are sorted. The function $\mathit{sorted}(\boldsymbol{y})$ is defined as $\mathit{sorted}(y_i, \ldots, y_j) = 1$ if $i = j$ and $(\bigwedge_{k=i}^{j} y_k) \vee (\neg y_i \wedge \mathit{sorted}(y_{i+1}, \ldots, y_j))$ otherwise.

The second conjunct in the Sorting specification says that the output must have the same number of bits set to 1 as the input. In the high-level representation, this can be represented by $\Sigma_{i=1}^n x_i = \Sigma_{j=1}^n y_j$, but in practice it is not necessary to use summation in the encoding. Instead, the propositional logic formula for this property can be represented by a recurrence and constructed using dynamic programming:

$$
\begin{aligned}
\varphi_{0,0} &= 1 \\
\varphi_{i,0} &= \neg x_i \wedge \varphi_{i-1,0} \\
\varphi_{0,j} &= \neg y_j \wedge \varphi_{0,j-1} \\
\varphi_{i,j} &= ((x_i \leftrightarrow y_j) \wedge \varphi_{i-1,j-1}) \vee (x_i \wedge \neg y_j \wedge \varphi_{i,j-1}) \vee (\neg x_i \wedge y_j \wedge \varphi_{i-1,j}) \qquad (4)
\end{aligned}
$$

In this encoding, $\varphi_{i,j}$ means that $x_1, \ldots, x_i$ has the same number of 1s as $y_1, \ldots, y_j$. This is obtained by matching each bit that is set to 1 in the input with a bit that is set to 1 in the output, and skipping bits that are set to 0.

Note that some of these encodings can be optimized, for example the specification for *Sorting* can be reduced by testing at the same time if the input is sorted and it has the same number of 1s as the output. This can shorten the construction time, but since it is logically equivalent to the original formula, by the canonicity property of BDDs, the resulting BDD for the specification will be the same.