# ZDD Boolean Synthesis*

Yi Lin ✉[1], Lucas M. Tabajara[1]**, and Moshe Y. Vardi[1]

Rice University, Houston TX 77005, USA
vardi@cs.rice.edu, l.martinelli.tabajara@gmail.com, yl182@rice.edu

**Abstract.** Motivated by applications in boolean-circuit design, boolean synthesis is the process of synthesizing a boolean function with multiple outputs, given a relation between its inputs and outputs. Previous work has attempted to solve boolean functional synthesis by converting a specification formula into a Binary Decision Diagram (BDD) and quantifying existentially the output variables. We make use of the fact that the specification is usually given in the form of a Conjunctive Normal Form (CNF) formula, and we can perform resolution on a symbolic representation of a CNF formula in the form of a Zero-suppressed Binary Decision Diagram (ZDD). We adapt the realizability test to the context of CNF and ZDD, and show that the *Cross* operation defined in earlier work can be used for witness construction. Experiments show that our approach is complementary to BDD-based Boolean synthesis.

**Keywords:** Boolean synthesis · Binary decision diagram · Zero-suppressed binary decision diagram · Quantifier elimination · Resolution.

## 1 Introduction

Boolean functions are widely used in electronic circuits, and thus in many aspects of computing, to describe operations over binary values. Often the most natural way to express such an operation is as a declarative relation between inputs and outputs. Implementing these operations in practice, however, requires a functional, rather than declarative, representation. The process of constructing a function that generates outputs directly from inputs, based on a given declarative relation between them, is called *boolean synthesis*. For example, boolean synthesis can be applied in constructing a full logical circuit from a relational specification [9,15] or an unknown intermediate component in an existing logical circuit [12]. Boolean synthesis is also useful for computing certificates for quantified boolean formulas (QBF), and advances in QBF solving and boolean synthesis are motivated by each other [3,20].

Formally, we are given a specification $f(\vec{x}, \vec{y})$, from $\mathbb{B}^m \times \mathbb{B}^n$ to $\mathbb{B}$, relating two sets of boolean variables. The specification holds true if and only if $\vec{y}$ is a

* Work supported in part by NSF grants CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

** Currently at Runtime Verification, Inc.


D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 64–83, 2022.
https://doi.org/10.1007/978-3-030-99524-9_4

correct output for the inputs $\vec{x}$. We solve the synthesis problem following the convention of splitting it into two sub-problems [9]:

1. *Realizability*: constructing the realizable set $R \subseteq \mathbb{B}^m$ of input assignments $\vec{x}$ for which there exists an output assignment $\vec{y}$ such that $f(\vec{x}, \vec{y}) = 1$.
2. *Witness construction*: constructing a witness function $g : \mathbb{B}^m \to \mathbb{B}^n$ that computes an output $\vec{y} = g(\vec{x})$ from an input $\vec{x} \in R$ such that $f(\vec{x}, \vec{y}) = 1$.

Given a propositional formula $f$ as the relational specification, we aim to synthesize a boolean function $g$ that is correct by construction, meaning that as long as the input is realizable the output will satisfy the specification.

Prior work solved the boolean functional synthesis by converting the specification formula into a Binary Decision Diagram (BDD), defined in Section 2, and quantifying the output variables existentially [9]. BDDs constitute a formalism for representing Boolean functions, supported by mature tools such as CUDD [22]. The size of a BDD representing a formula can, however, be exponential in the number of variables. Oftentimes, it is even not possible to construct the BDD before starting to solve the problem [9]. Noticing how this blow-up in BDD size has restricted the potential of existing BDD-based synthesis algorithms, we seek to develop an algorithm that reduces the impact of this exponential blowup. Hence we look for an alternative data structure that might be more promising in representing boolean formulas compactly.

We identify here Zero-Suppressed Binary Decision Diagram (ZDD) [16], defined in Section 2, as such an alternative approach. ZDDs have been shown to sometimes outperform BDDs in the context of QBF solving [19]. Unlike BDDs, which represent a boolean formula *semantically* via the set of satisfying assignments, ZDDs are designed to encode sets of sets [14], allowing them to represent *syntactically* a formula in Conjunctive Normal Form (CNF) as a set of clauses, which are themselves sets of literals. This means that it may require an exponential-size BDD to represent a CNF formula, which can be alternatively compactly encoded as a polynomial-size ZDD representation.

It can be expected, however, that this more compact representation comes at a cost. Since ZDDs do not represent the solution sets directly like BDDs do, solving realizability and synthesis over this representation might require additional effort. With this in mind. we perform here a full investigation comparing ZDDs and BDDs for boolean synthesis. We focus on the following research questions:

1. How do the sizes of the ZDD and BDD representations compare, and how does this affect the time of compiling the formulas into the diagram representation? Are ZDDs always more compact?
2. In realizability, how do ZDDs vs. BDDs perform, in time and space?
3. How do ZDDs perform, compared to BDDs, in witness construction?
4. How does the end-to-end synthesis performance of ZDDs compare to BDDs?
5. For scalable families of formulas, how does the time and space performance scale as the formula grows, comparing ZDDs to BDDs?

Our synthesis problem can often be expressed as boolean synthesis for CNF specifications, as the boolean specification in synthesis problems is often given

in CNF form, and even non-CNF specifications can be easily converted to CNF. Once specification formulas are given in CNF, it is possible to perform realizability by using the *resolution* operation, which is equivalent to existentially quantifying the output variables directly from the CNF formula. Each resolution step increases the number of clauses quadratically. But when a ZDD is used to represent the CNF formula, even when the number of clauses increases quadratically, the size of the ZDD tends to increase to a lesser extent.

The crux of our contribution is a boolean-synthesis algorithm that performs resolution on a symbolic, ZDD-based representation of CNF formulas. To solve the first sub-problem of realizability, we compute the set $R \subseteq \mathbb{B}^m$ of all realizable inputs, and then check the full and partial realizability of the input domain. The realizable set is generated by applying resolution to the ZDD representation of the CNF formula, based on operations defined in previous work [4,5,19].

The second sub-problem requires construction of a witness function $g : \mathbb{B}^m \to \mathbb{B}^n$ for the output variables $\vec{y} \in \mathbb{B}^n$. We adapt the formulas defined in previous work [9] to the context of CNF, eliminating one output variable $y_i \in \mathbb{B}$ at a time, and make use of the fact that resolution is equivalent to existential quantification. In this way we can extract a witness $g_i : \mathbb{B}^m \to \mathbb{B}$ for variable $y_i$ without abandoning the ZDD representation.

After substituting the witness of an output variable back in the formula, we need to compute the next witness. This leads to our next challenge, which is how to guarantee that the formula remains in CNF after performing this substitution. The overall form of the entire formula after substitution is dependent on the form of the substituted witness function $g_i$: clauses where $y_i$ is positive can be converted back to CNF if $g_i$ is also in CNF, but clauses where $y_i$ is negative require $g_i$ to be in *Disjunctive* Normal Form (DNF). Thus, what we need are two equivalent witness functions, one in CNF and the other in DNF.

Our solution is to use the *Cross* ZDD operation, first defined by Knuth [14]. We show that if the *Cross* operation, defined on "families of sets" [14], is applied to a ZDD representation of a CNF formula, then the result can be interpreted as the ZDD for an equivalent DNF. In this way, with the *Cross* operation, we can use the CNF version of a witness for positive occurrences of a variable, and use the equivalent DNF version for negative occurrences, while both preserving the equivalence and ensuring that the resulting formula remains in CNF.

Our experimental evaluation confirms the advantages of ZDDs in compilation, thanks to their linear size and direct correspondence to the CNF formula structure. As expected, this more compact representation can come with a trade-off of increasing the difficulty of constructing witnesses. Therefore, in synthesis performance, neither ZDDs nor BDDs dominate across the board, each performing better in different families of formulas. We therefore advocate for the ZDD-based approach as an addition to the portfolio of boolean synthesis tools, serving as a complement to BDD-based approaches [11].

As shown in related works on boolean synthesis, there exist alternative tools including CegarSkolem [13], BFSS [1] and Manthan [10]. Our focus of comparison here is, however, on improvements to decision-diagrams based tools

for boolean synthesis, rather than tools based, for example, on QBF solvers. Decision-diagram based approaches enjoy some unique advantage. For example, decision diagrams facilitate partitioned-form representation [23]. Also, decision diagrams can be used as intermediate-step representation in temporal synthesis [24]. These unique advantages justify, we believe, our focus here on decision-diagram based approaches. We return to this point in our discussion of future work.

## 2 Preliminaries

*Boolean Formulas and Functions.* Boolean formulas and boolean functions are built upon the boolean set $\mathbb{B} = \{0, 1\}$. We identify a boolean formula $f(\vec{x})$ over $m$ propositional variables $\vec{x} = (x_1, \ldots, x_m)$ with the boolean function $f : \mathbb{B}^m \to \mathbb{B}$ such that $f(\vec{a}) = 1$ for an assignment $\vec{a} = (a_1, \ldots, a_m) \in \mathbb{B}^m$ if and only if $\vec{a}$ is a satisfying assignment to $\vec{x}$ in the formula. Two boolean formulas $f$ and $f'$ are logically equivalent if they represent the same boolean function (and therefore have the same set of satisfying assignments). Substitution of a boolean expression $d(\vec{x})$ in place of a variable $x_i$ in a boolean formula $f(\vec{x})$ is denoted by $f[x_i \mapsto d]$ and defined by $f[x_i \mapsto d](\vec{x}) = f(x_1, \ldots, x_{i-1}, d(\vec{x}), x_{i+1}, \ldots, x_m)$.

*Conjunctive and Disjunctive Normal Forms.* A *literal* is either a variable or the negation of a variable. A *clause* is a disjunction of literals, and a *cube* is a conjunction of literals. A boolean formula in the form of a conjunction of clauses is said to be in *Conjunctive Normal Form* (CNF), and a boolean formula in the form of a disjunction of cubes is said to be in *Disjunctive Normal Form* (DNF).

**Definition 1 (Boolean Synthesis Problem).** *Given a boolean formula $f(\vec{x}, \vec{y})$ in CNF with $m + n$ boolean variables, partitioned into $m$* input variables *$\vec{x} = (x_1, \ldots, x_m)$ and $n$* output variables *$\vec{y} = (y_1, \ldots, y_n)$, construct:*

1. *The set $R \subseteq \mathbb{B}^m$, called the* realizability set*, of all assignments $\vec{a} \in \mathbb{B}^m$ to $\vec{x}$ for which there exists an assignment $\vec{b} \in \mathbb{B}^n$ to $\vec{y}$ such that $f(\vec{a}, \vec{b}) = 1$.*
2. *A function $g : \mathbb{B}^m \to \mathbb{B}^n$ such that $f(\vec{a}, g(\vec{a})) = 1$ for all $\vec{a} \in R$. This is called a* witness function*. In practice, arbitrary formulas can be converted to equi-realizable CNF formulas with a linear blowup using Tseytin encoding, quantifying existentially over Tseytin variables. The witnesses for the equi-realizable formula can then be used for the original formula.*

*Binary Decision Diagrams.* A (Reduced Ordered) Binary Decision Diagram (BDD) [2] is a directed acyclic graph that represents a boolean function. Internal nodes of the BDD represent boolean variables, and paths on the BDD correspond to assignments, leading either to a terminal node 1 if satisfying or 0 if unsatisfying. We assume that all BDDs are *ordered*, meaning that variables are ordered in the same way along every path, and *reduced*, meaning that superfluous nodes are removed and identical subgraphs are merged. Given these two conditions, BDDs are a *canonical* representation, meaning that two BDDs

with the same variable order that represent the same function will be identical [2]. The variable order used can have a major impact on the BDD's size, and two BDDs representing the same function but with different orders can have an exponential difference in size.

*Zero-Suppressed Decision Diagrams.* A Zero-Suppressed Binary Decision Diagram (ZDD), is a data structure first defined in [16]. ZDDs are similar to BDDs but use a different reduction rule: while BDDs remove nodes where both edges point to the same child, ZDDs remove nodes where the 1-edge (edge assigning the variable to 1) points directly to the 0-terminal. Specifically, the 0-ZDD encodes formulas that are always valid, and the 1-ZDD encodes contradiction.

*Semantics on Families of Sets.* ZDDs can be used to implicitly represent families of subsets of a set $S$, where the variables in the ZDD correspond to elements of $S$ that can be either present or absent in a subset [14]. For a ZDD $Z$, we denote by $[\![Z]\!]$ the family of subsets represented by $Z$. We define $[\![0]\!] = \varnothing$ and $[\![1]\!] = \{\varnothing\}$ for the terminals 0 and 1, respectively. Using $Z(x, Z_0, Z_1)$ to denote a ZDD with variable $x$ as the root, ZDD $Z_0$ as the 0-child and ZDD $Z_1$ as the 1-child, we define $[\![Z(x, Z_0, Z_1)]\!] = [\![Z_0]\!] \cup \{\{x\} \cup \alpha \mid \alpha \in [\![Z_1]\!]\}$. Note that using this interpretation every subset in the family corresponds to a path to the terminal 1 on the ZDD. Since CNF formulas can be viewed as sets of clauses, where a clause can be viewed as a set of literals, we can use ZDDs to represent CNF formulas syntactically. When representing a formula in CNF by a ZDD, for each atomic proposition $p$ we treat its positive and negative literals $p$ and $(\neg p)$ as two distinct variables $x_p$ and $x_{\neg p}$. Then every path leading to the 1-terminal corresponds to a clause in the CNF formula, where $x_l$ connects to its 1-edge in the path if and only if the literal $l$ is in the corresponding clause.

*ZDD Operations.* We use standard ZDD operations such as *Subset0*, *Subset1*, *Change*, *Union*, *Intersect*, and *Difference*, defined previously in [17] and implemented in the CUDD package [22]. In terms of families of sets, *Subset0*$(Z, x)$ returns the family of all sets $\alpha$ such that $\alpha \in [\![Z]\!]$ and $x \notin \alpha$, and *Subset1*$(Z, x)$ returns the family of all sets $\alpha \setminus \{x\}$ such that $\alpha \in [\![Z]\!]$ and $x \in \alpha$. *Change*$(Z, x)$ returns the family $\{\alpha \cup \{x\} \mid \alpha \in [\![Z]\!] \text{ and } x \notin \alpha\} \cup \{\alpha \setminus \{x\} \mid \alpha \in [\![Z]\!] \text{ and } x \in \alpha\}$. The operation *Resolution(x, Z)* returns the ZDD representing the result of applying resolution to variable $x$ in the CNF represented by $Z$. It is implemented following [4], using the operations *SubsumptionFreeUnion*, which takes the union of two families of sets while removing subsumed sets, and *ClauseDistribution*, which returns the family of sets resulting from applying distribution over two given sets of clauses. The witness-construction phase also requires the *Cross* operation defined in [14] to convert between CNF and DNF representations. See Section 4 for details.

## 3 Realizability Using ZDDs

We describe in this work a ZDD-based algorithm to solve the Boolean-Synthesis Problem described in Definition 1. This means that the specification $f(\vec{x}, \vec{y})$, the realizability set $R$ and the witness function $g$ are all represented by CNF formulas encoded as ZDDs, as defined in Section 2. In this section we describe how to compute the realizability set $R$ and analyze it to answer whether the specification is partially or fully realizable. In Section 4 we describe how to compute the witness function $g$.

### 3.1 Realizable Set R

In order to construct the set $R$ of realizable assignments to the input variables $\vec{x}$, as described in Definition 1, we need to quantify existentially the output variables $\vec{y}$, analogously to the BDD-based approach of [9].

Let $f_0, \ldots, f_n$ be CNF formulas such that

$$
\begin{aligned}
&f_n \equiv f \\
&f_{n-1} \equiv (\exists y_n) f \\
&\ldots \\
&f_i \equiv (\exists y_{i+1}) \ldots (\exists y_{n-1})(\exists y_n) f \\
&\ldots \\
&f_1 \equiv (\exists y_2) \ldots (\exists y_{n-1})(\exists y_n) f \\
&f_0 \equiv (\exists y_1) \ldots (\exists y_{n-1})(\exists y_n) f
\end{aligned}
$$

As in [9], the last formula $f_0 = (\exists y_1) \ldots (\exists y_{n-1})(\exists y_n) f$ implicitly represents the realizable set $R$, describing the set of satisfying assignments of $f_0$.

To compute $f_0, \ldots, f_n$ as CNF formulas, we apply the *resolution* operation, which is equivalent to existential quantification [7]. We first state a normal-form lemma.

**Lemma 1.** [4] *Let $f$ be a CNF formula. Let $f_p^+$ denote the conjunction of all clauses $\alpha$ such that $(p \vee \alpha)$ is a clause in $f$. Let $f_p^-$ denote the conjunction of all clauses $\beta$ such that $((\neg p) \vee \beta)$ is a clause in $f$. Let $f_p'$ denote the conjunction of clauses $\gamma$ in $f$ where neither $p$ nor $(\neg p)$ is a literal in $\gamma$. Then $f$ is logically equivalent to $(p \vee f_p^+) \wedge (\neg p \vee f_p^-) \wedge f_p'$ for a boolean variable $p$.*

*Proof.* The claim follows from [4]. □

Next we show how to use resolution to existentially quantify a variable from a formula in the normal form of Lemma 1.

**Lemma 2.** *Let $y$ be a boolean variable, then the boolean formula $(\exists y)((y \vee f_y^+) \wedge (\neg y \vee f_y^-) \wedge f_y')$ is logically equivalent to $((f_y^+ \vee f_y^-) \wedge f_y')$.*

*Proof.*

$$
\begin{aligned}
&(\exists y)((y \vee f_y^+) \wedge (\neg y \vee f_y^-) \wedge f_y') \\
&\equiv (\exists y)(((y \wedge \neg y) \vee (y \wedge f_y^-) \vee (f_y^+ \wedge \neg y) \vee (f_y^+ \wedge f_y^-)) \wedge f_y') \\
&\equiv ((\exists y)(y \wedge f_y^-) \vee (\exists y)(f_y^+ \wedge \neg y) \vee (\exists y)(f_y^+ \wedge f_y^-)) \wedge f_y' && (f_y' \text{ excludes } y) \\
&\equiv (((\exists y)(y \wedge f_y^-) \vee (\exists y)(f_y^+ \wedge \neg y) \vee (f_y^+ \wedge f_y^-)) \wedge f_y') && (f_y^+, f_y^- \text{ excludes } y) \\
&\equiv ((f_y^- \vee f_y^+ \vee (f_y^+ \wedge f_y^-)) \wedge f_y') && (f_y^+, f_y^- \text{ excludes } y) \\
&\equiv ((f_y^- \vee f_y^+) \wedge f_y')
\end{aligned}
$$

□

We call the formula $((f_y^+ \vee f_y^-) \wedge f_y')$ the *resolution of the variable $y$ in $f$*. Note that this formula (specifically the subformula $(f_y^+ \vee f_y^-)$) is not in CNF, but can be easily rewritten in CNF by distributing the clauses in $f_y^+$ over the clauses in $f_y^-$. The equivalence of resolution and existential quantification then follows from Lemmas 1 and 2 above:

**Corollary 1.** *For a formula $f$ and a boolean variable $y$, the formula $(\exists y)f$ is logically equivalent to $((f_y^+ \vee f_y^-) \wedge f_y')$.*

*Proof.* The claim follows from Lemmas 1 and 2. □

We represent $f_y^+, f_y^-, f_y'$ by ZDDs by applying the *Subset0* and *Subset1* operations described in Section 2: $Z_y^+ = Subset1(Z, y)$, $Z_y^- = Subset1(Z, \neg y)$, and $Z_y' = Subset0(Subset0(Z, y), \neg y)$. We then use the *ClauseDistribution* operation to distribute the clauses of $Z_y^+$ over $Z_y^-$, and the *SubsumptionFreeUnion* operation to combine all clauses into a single ZDD. This implements the operation $Resolution(y_i, Z)$ mentioned in Section 2. In practice, we follow the *Cut-Elimination Algorithm* of [4], which also eliminates tautologies by removing clauses where the same variable appears both positively and negatively. Therefore we can assume that the ZDD representations of $f_0, \ldots, f_n$ do not include subsumed and tautological clauses, which may also lead to smaller ZDDs.

The advantage of applying resolution symbolically over a ZDD representation, rather than directly over the CNF formula is that every resolution step increases the number of clauses in the formula quadratically. Thus, the number of clauses after multiple resolution steps can easily grow exponentially. ZDDs, compared to representing clauses explicitly, are well-equipped for representing compactly large sets of clauses, often being able to represent an exponential set of clauses in polynomial space [16]. The ZDD representation also makes it easy to remove subsumed and tautological clauses, further reducing size.

### 3.2 Full and Partial Realizability

When the realizable set $R$ is represented by a BDD, as in [9], it is easy to check whether $R = \varnothing$ or $R = \mathbb{B}^m$, as this corresponds to the BDD being equal to 0 or 1, respectively. This is less straightforward for a ZDD representation, which

expresses $R$ indirectly by the set of clauses in its CNF representation $f_0$, rather than the set of assignments itself. We say that a CNF specification $f(\vec{x}, \vec{y})$ is *fully realizable* if and only if all $\vec{a} \in \mathbb{B}^m$ have some $\vec{b} \in \mathbb{B}^n$ so that $f(\vec{a}, \vec{b})$ holds. This corresponds to $R = \mathbb{B}^m$. Similarly, we say that $f$ is *partially realizable* if and only if there is some $\vec{a}$ for which there exists some $\vec{b}$ so that $f(\vec{a}, \vec{b})$ holds. This corresponds to $R \neq \varnothing$. After computing a ZDD representation of $R$, we wish to check full and partial realizability over this representation.

**Theorem 1.** *The CNF specification $f(\vec{x}, \vec{y})$ is fully realizable if and only if the ZDD for $f_0$ is equivalent to the 0-ZDD.*

*Proof.* The specification $f(\vec{x}, \vec{y})$ is fully realizable if and only if the CNF formula $f_0$ representing $R$ is a tautology, which means that every clause of $R$ has both $p$ and $\neg p$ for some variable $p$, i.e., every clause is a tautology. Tautologies, however, are automatically removed by the *Resolution* operation, as explained in Section 3.1. Thus, full realizability occurs if and only if the set of clauses is empty, represented by the ZDD 0. □

Note that the realizability $R$ is represented by the CNF formula $f_0 \equiv (\exists y_1) \ldots (\exists y_n) f$, which does not contain any free occurrences of $\vec{y}$ variables. We then perform resolution on the $\vec{x}$ variables in the same way as we did for the $\vec{y}$ variables. Then the original formula is partially realizable if and only if $(\exists x_1)(\exists x_2) \ldots (\exists x_m) f_0$ is true, meaning that resolution does not derive a contradiction. If a contradiction is derived, the resulting ZDD is the terminal 1, representing the empty clause. Otherwise it is the terminal 0.

**Theorem 2.** *The CNF specification $f(\vec{x}, \vec{y})$ is partially realizable if and only if the ZDD representing $(\exists x_1)(\exists x_2) \ldots (\exists x_m) f_0$ is equivalent to the 0-ZDD.*

*Proof.* Since all variables are existentially quantified, the ZDD must be either the terminal 0 (representing the empty CNF, equivalent to *true*) or the terminal 1 (representing a CNF with an empty clause, equivalent to *false*). In the first case, the formula $(\exists x_1) \ldots (\exists x_m)(\exists y_1) \ldots (\exists y_n) f$ is true, meaning that there is an assignment that satisfies $f(\vec{x}, \vec{y})$, which by definition makes $f$ partially realizable. In the second case, the formula is false, meaning that there is no such assignment, and therefore $f$ is not partially realizable. □

## 4 Synthesis Using ZDDs

As described in [9], once we have computed the formulas $f_1, \ldots, f_n$ with the output variables existentially quantified, we can construct the witness $g_i$ for variable $y_i$ from the formula $f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}]$, after having computed the witnesses $g_1, \ldots, g_{i-1}$ for the preceding variables. In [9], two witness functions were presented for variable $y_i$: the default-1 witness $f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}][y_i \mapsto 1]$ and the default-0 witness $(\neg f_i)[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}][y_i \mapsto 0]$. In this work, however, we additionally want to ensure that we maintain the CNF form of the specification after substituting $g_1, \ldots, g_{i-1}$ into $f_i$, to enable ZDD representation.

In this section we show how to construct and substitute witnesses so that the result remains in CNF.

For ZDD-based algorithms, the iterated substitution approach requires more sophistication for the construction of the witnesses, compared to the iterated-substitution approach for BDDs. We solve this problem in Section 4.2. As in [9], the resulting witnesses guarantee that $f(\vec{a}, g_1(\vec{a}), \ldots, g_n(\vec{a})) = 1$ for all realizable input assignments $\vec{a} \in R$.

### 4.1 Witnesses for Single-Dimension Output Variable

As in [9], we start by defining witnesses for the case when there is a single output variable:

**Lemma 3.** *Let $f$ be a CNF formula over boolean variables $x_1, \ldots, x_m, y$. Then the formulas $f_y^-$ and $\neg f_y^+$ are witnesses for the variable $y$.*

*Proof.* The realizability set, as defined in Section 3.1, is $R = \{\vec{a} \in \mathbb{B}^m \mid (\exists y) f[\vec{x} \mapsto \vec{a}] \equiv 1\}$. Thus, by Corollary 1, for all $\vec{a} \in R$

$$((f_y^+ \vee f_y^-) \wedge f_y')[\vec{x} \mapsto \vec{a}] \equiv 1. \tag{1}$$

Hence $f_y'[\vec{x} \mapsto \vec{a}] \equiv 1$ and either $f_y^+[\vec{x} \mapsto \vec{a}] \equiv 1$ or $f_y^-[\vec{x} \mapsto \vec{a}] \equiv 1$.

Now we want to show $f(\vec{a}, g(\vec{a})) = 1$, i.e., $f[y \mapsto g(\vec{x})][\vec{x} \mapsto \vec{a}] = 1$, for both $g(\vec{x}) = f_y^-$ and $g(\vec{x}) = \neg f_y^+$.

For $g(\vec{x}) = f_y^-$, since $f \equiv f_y' \wedge (y \vee f_y^+) \wedge ((\neg y) \vee f_y^-)$, we are left to show $f[y \mapsto f_y^-][\vec{x} \mapsto \vec{a}] \equiv (f_y' \wedge (f_y^- \vee f_y^+) \wedge ((\neg f_y^-) \vee f_y^-))[\vec{x} \mapsto \vec{a}] \equiv 1$. By (1) we are only left to show $((\neg f_y^-) \vee f_y^-)[\vec{x} \mapsto \vec{a}] \equiv 1$, which follows from the left-hand side being a tautology.

Similarly, for $g(\vec{x}) = \neg f_y^+$, we need to show $f(\vec{a}, g(\vec{a})) = f[y \mapsto (\neg f_y^+)][\vec{x} \mapsto \vec{a}] \equiv 1$. This is equivalent to showing that $(f_y' \wedge ((\neg f_y^+) \vee f_y^+) \wedge (f_y^+ \vee f_y^-))[\vec{x} \mapsto \vec{a}] \equiv 1$. By (1) we are only left to show $((\neg f_y^+) \vee f_y^+)[\vec{x} \mapsto \vec{a}] \equiv 1$, a tautology.

□

Note that the witness $f_y^-$ is in CNF, while the witness $\neg f_y^+$, being the negation of a CNF formula, can be more easily represented in DNF. Note also that these witnesses do not correspond exactly to the default-1 and default-0 witnesses of [9], which would more specifically be equivalent to $f_y^- \wedge f_y'$ and $\neg(f_y^+ \wedge f_y')$, respectively. We choose the alternative witnesses because they contain fewer clauses, and thus are more likely to produce a more efficient ZDD representation.

### 4.2 Preserve CNF by Equivalent Witnesses

We now explain how to construct witnesses of multiple output variables. Let $f_n, \ldots, f_0$ be as defined in Section 3.1. We can then compute a witness for each $y_i$ iteratively, as in [9]. Using the $f_y^-$ witness from Lemma 3, for example, this means $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}])_{y_i}^-$, where $g_i$ is the witness for variable $y_i$.

The substitution $f_i[y \mapsto g]$, however, is not necessarily in CNF. But Lemma 3 requires that the formula is in CNF in order to extract the next witness. This means that we need to find a way to perform the substitution in a way that the result remains in CNF.

Recall that, since our *Resolution* operation removes tautological clauses, each variable can only occur in positive or negative form in a clause, but not both. If the witness $g$ is in CNF, e.g., $g = f_y^-$, we can substitute this witness in a clause $(y \vee l_1 \vee l_2 \vee \ldots)$ where $y$ occurs in positive form. The result is a disjunction of the literals $l_1, l_2, \ldots$ and the CNF $g = (cl_1 \wedge cl_2 \wedge \ldots)$. By distribution, we can write this as an equivalent CNF $((cl_1 \vee l_1 \vee l_2 \vee \ldots) \wedge (cl_2 \vee l_1 \vee l_2 \vee \ldots) \wedge \ldots)$. Likewise, if the witness $g$ for $y$ is in DNF, e.g., $g = (\neg f_y^+)$, then, after the substitution, every clause $(\neg y \vee l_1 \vee l_2 \vee \ldots)$ where $y$ appears in negative form can be converted to the CNF $(\neg(\neg(cl_1 \wedge cl_2 \wedge \ldots)) \vee l_1 \vee l_2 \vee \ldots) \equiv ((cl_1 \wedge cl_2 \wedge \ldots) \vee l_1 \vee l_2 \vee \ldots) \equiv ((cl_1 \vee l_1 \vee l_2 \vee \ldots) \wedge (cl_2 \vee l_1 \vee l_2 \vee \ldots) \wedge \ldots)$.

The problem, therefore, is that if we want the result to be in CNF, CNF witnesses work well for positive occurrences, while DNF witnesses work well for negative occurrences. Thus, as long as we can find an efficient conversion between CNF formulas and their equivalent DNF formulas, we can ensure that the substitution formula $f_i[y \mapsto g]$ can be written as a CNF. For this purpose, we introduce the *Cross* operator from [14].

**Definition 2 (*Cross* operation).**
*Let $S$ be a family of sets of literals. Then*

$$Cross(S) = Minimal\{t \mid \forall s_i \in S : t \cap s_i \neq \varnothing\},$$

*where*

$$Minimal(S) = \{t \in S \mid \forall s \in S : s \subseteq t \rightarrow s = t\}.$$

Hence, $Cross(S)$ is a family of sets of literals, such that every set $t$ of literals in $Cross(S)$ has at least a common literal with every set of literals in $S$. Moreover, every set $t$ in $Cross(S)$ is irredundant [14], meaning they are the smallest possible sets satisfying this property.

Specifically, if $S$ represents a given CNF $f$, where every set $s_i \in S$ represents a clause and the elements of $s_i$ are the literals in that clause, then $Cross(S)$ represents the set of smallest possible sets $t$ such that $t$ has at least one common literal with every disjunctive clause of $f$. Equivalently speaking, $Cross(S)$ collects all $t$ such that every disjunctive clause is satisfied, i.e., it is a collection of all irredundant sets of literals corresponding to irredundant assignments to variables. This further means $Cross(S)$ is a collection of prime implicants of $f$ [6,14], whose disjunction has been proved to be a DNF equivalent to the CNF $f$. Therefore, whenever a CNF is given, we can construct a set $S$ of sets, where every set in $S$ collects literals in a disjunctive clause of the CNF. Then $Cross(S)$ returns a set of sets representing an equivalent DNF. Conversely, when interpreted as a DNF, $Cross(S)$ is equivalent to $S$ interpreted as a CNF.

By the analysis above, we can extend Definition 2 of the *Cross* operation to CNF formulas:

**Definition 3 (CNF Cross operation).** *Let $f$ be a CNF formula $cl_1 \wedge \ldots \wedge cl_k$, where every $cl_i = \bigvee_{\ell \in L_i} \ell$ is a clause formed by the disjunction of a set of literals $L_i$. Let $S = \{L_1, \ldots, L_k\}$ be the representation of $f$ as a family of sets. Then,*

$$Cross(f) = \bigvee_{L'_i \in Cross(S)} \bigwedge_{\ell \in L'_i} \ell$$

Note that $Cross(f)$ is a DNF formula. We can similarly define in an analogous way the *Cross* of a DNF formula as a CNF formula. We can verify that $Cross(f)$ and $f$ are equivalent:

**Lemma 4.** *For a CNF formula $f$, $Cross(f) \equiv f$.*

*Proof.* By analysis above, the set $Cross(S)$ includes elements $L'_i s$ which are irredundant smallest sets that each has common literal with every set of literals in $S$. Therefore, every conjunction $\bigwedge_{\ell \in L'_i} \ell$, or cube, has common literal with every disjunctive clauses in CNF $f$, and thus every cube has the same boolean values under the same set of truth assignments as a prime implicant [6,21] of CNF $f$. Then it follows that the DNF $Cross(f)$, as a disjunction of these conjunctions, is logically equivalent to the disjunction of all prime implicants of the CNF $f$, as proved by previous works [21]. □

Note that the same result also holds for DNF formulas, following from the fact that $Cross(f) \equiv f$ if and only if $\neg Cross(f) \equiv \neg f$.

Now we aim to show how to construct witnesses one by one, why this construction is correct, and why this construction is viable. First, if we fix the witness $g_j = (f_i)^-_{y_j}$, and substitute positive and negative occurrences with $g_j$ and $Cross(g_j)$ in the CNF formula $f_i$, then the equivalence and CNF form of $f_i[y_j \mapsto g_j]$ can both be preserved. We use the following lemma:

**Lemma 5.** *Let $f$ and $g$ be given as CNF formulas. Then $f[y \mapsto g]$ is equivalent to $(g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$.*

*Proof.* By Lemmas 1 and 4, $f[y \mapsto g] \equiv ((y \vee f^+_y) \wedge (\neg y \vee f^-_y) \wedge f'_y)[y \mapsto g] = (g \vee f^+_y) \wedge (\neg g \vee f^-_y) \wedge f'_y \equiv (g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$.

□

Since $g = f^-_y$ is a CNF formula, $Cross(g)$ is a DNF formula, and $\neg Cross(g)$ is a CNF. By distribution of $f^+_y$ over clauses in $g$, and distribution of $f^-_y$ over clauses in $\neg Cross(g)$, the resulting expression $(g \vee f^+_y) \wedge (\neg Cross(g) \vee f^-_y) \wedge f'_y$ can be converted to CNF form.

Alternatively, we can pick the witness $g = \neg f^+_y$, and instead substitute $Cross(g)$ on positive occurrences and $g$ on negative occurrences of $y$. Similarly, the formula $(Cross(g) \vee f^+_y) \wedge (\neg g \vee f^-_y) \wedge f'_y$ can also be converted to an equivalent CNF. Therefore, the equivalence and CNF form is preserved for $f_i[y_j \mapsto g_j]$, leading to the following corollary.

**Corollary 2.** *Every step in $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1] \ldots [y_{i-1} \mapsto g_{i-1}])^-_{y_i}$ can be performed so it returns a CNF formula.*

*Proof.* Corollary 2 follows from Lemma 3, Definition 3, and Lemma 5 □

Finally, we have the witnesses constructed in this process:

**Theorem 3.** *Let* $g_i(\vec{x}) = (f_i[y_1 \mapsto g_1] \dots [y_{i-1} \mapsto g_{i-1}])^-_{y_i}$ *for* $0 \leq i \leq n$. *Then,* $g_i$ *is a witness for* $y_i$ *in* $f$, *for every* $y_i$. *The same applies if* $g_i(\vec{x}) = \neg(f_i[y_1 \mapsto g_1] \dots [y_{i-1} \mapsto g_{i-1}])^+_{y_i}$

*Proof.* Theorem 3 follows from Lemma 5 and Corollary 2. □

### 4.3 Algorithm for Constructing Witnesses

In the last subsection we described how to uses Knuth's Cross operation to facilitate CBF/DNF conversion, enabling the use of iterated substitution. We describe our novel algorithm for synthesis using ZDDs.

We start by presenting the ZDD implementation of *Cross* function from Definition 2, following [14]:

**if** *ZDD Z is the 1-terminal* **then**
  **return** *0-terminal*;
**else if** *ZDD Z is the 0-terminal* **then**
  **return** *1-terminal*;
**else**
  // $Z_l$ denotes the ZDD rooted at 0-child of root of $Z$
  // $Z_h$ denotes the ZDD rooted at 1-child of root of $Z$
  $Z_r = Union(Z_l, Z_h)$;
  $Z_{ll} = Cross(Z_r)$;
  $Z_r = Cross(Z_l)$;
  $Z_{hh} = Difference(Z_r, Z_{ll})$;
  // $Var(Z)$ denotes the variable at the root node of $Z$
  $Z' = NewZDD(Var(Z), Z_{ll}, Z_{hh})$;
  **return** *Z'*;
**end**

We now explain how to perform the substitution following Lemma 5, where we want to construct a ZDD of $f[y \mapsto g]$, where $f$ and $g$ are CNF formulas and $y$ is a variable. Denote the ZDD representation of $f$ as $Z_f$ and that of $g$ as $Z_g$. Then we compute the ZDD $Cross(Z_g)$ using the algorithm above. Recall that this ZDD represents a DNF formula that is equivalent to $g$.

To construct a ZDD for the formula in Lemma 5, we need a ZDD for $\neg Cross(g)$. But note that the ZDD for the CNF $\neg Cross(g)$ is equal to the ZDD for the DNF $Cross(g)$ except replacing every positive literal $p$ with the its negative literal $\neg p$ and vice-versa. Therefore, we want to swap $p$ and $\neg p$ in $Cross(Z_g)$.

We retrieve the clauses with neither $p$ nor $\neg p$ by

$$Z_1 = Subset0(Subset0(Cross(Z_g), p), \neg p).$$

Then we swap $p$ with $\neg p$ in every clause where $p$ appears positively:

$$Z_2 = Change(Subset1(Cross(Z_g), p), \neg p).$$

And we swap $\neg p$ with $p$ in every clause where $p$ appears negatively:

$$Z_3 = Change(Subset1(Cross(Z_g), \neg p), p).$$

Finally, taking the union of $Z_1, Z_2$ and $Z_3$ gives us the ZDD $\neg Cross(Z_g)$ encoding the CNF for the negation of $Cross(Z_g)$.

Let $Z_y^+$, $Z_y^-$ and $Z_y'$ be the ZDDs for $f_y^+$, $f_y^-$ and $f_y'$, respectively, constructed as described in Section 3.1. We compute the ZDDs for $(g \vee f_y^+)$ and $(\neg Cross(g) \vee f_y^-)$ by $ClauseDistribution(Z_g, Z_y^+)$ and $ClauseDistribution(\neg Cross(Z_g), Z_y^-)$, respectively. We then take the *Union* of these two ZDDs and $Z_y'$ to get the ZDD for $(g \vee f_y^+) \wedge (\neg Cross(g) \vee f_y^-) \wedge f_y'$, which is exactly the ZDD for $f[y \mapsto g]$ by Lemma 5.

## 5 Experimental Evaluations

### 5.1 Experimental Methodology and and Setting

We perform a comparison between our ZDD-based synthesizer, ZSynth, and the tool RSynth described by [9], using challenging $\Pi_2^P$ benchmarks from the QBFEVAL 2016 data set [18], the latest QBFEVAL set that includes a 2QBF (forall-exists) track, which is the format our benchmarks require. Each benchmark ran for 24 hours on Rice University's NOTS cluster with 64G RAM size. We focus our comparison on the Fixpoint Detection, MutexP, and QShifter benchmark families, omitting those families that are either too easy or too hard to solve for both tools, namely, the Tree, Ranking Functions, Reduction Finding, and Sorting Networks families [18]. For those families, either both tools solved all instances or none. Of these omitted benchmark families, Tree is very simple and is solved very quickly by both tools, while the others could be synthesized by neither tool. therefore we choose to focus on the three families that provide an interesting comparison. Fixpoint Detection, MutexP and QShifter have, respectively, 146, 7, and 6 instances.

For each tool we evaluate both total time and peak memory consumption for compilation, realizability, and synthesis, as well as the DD size for the original formula in each symbolic representation. We use the maximum cardinality search (MCS) heuristic [23] to determine the ordering of variables in both ZDDs and BDDs.Due to restrictions on available time and space resources, some benchmarks show out-of-time and out-of-memory failures. We measure the performance of both tools on the benchmarks that are solved. The experimental evaluations conclude that the ZDD-based approach is complementary to the BDD-based approach.

### 5.2 Compilation Time and Size of Diagram Representing Original Formula

We first compare the performance of CNF compilation into ZDDs and BDDs, following the first research question proposed in Section 1. The log-scale bar plot in

Fig. 1 presents compilation time for the benchmarks from the selection families, per Section 5.1. The size of the bars representing each formula is proportional to the compilation time.

The compilation into a ZDD takes polynomial (at most quadratic) time, because paths in the ZDD correspond to clauses, and therefore the size of the ZDD is always linear in the size of the formula. In contrast, the compilation into a BDD can be exponential, because paths in a BDD correspond to assignments, and therefore the number of paths can be exponential. The advantage of ZDDs as a compact representation is consistent with our conjecture. Across all benchmark families in QBFEVAL'16, compilation into ZDDs takes less time and space than BDDs in most cases.

It is worth noting that we construct here the ZDD representation of the CNF formulas by adding one clause at a time using the *Union* operator. Compilation could be further optimized by using a divide-and-conquer approach, where we split the set of clauses in half, construct ZDDs for each half recursively, and then take their union.

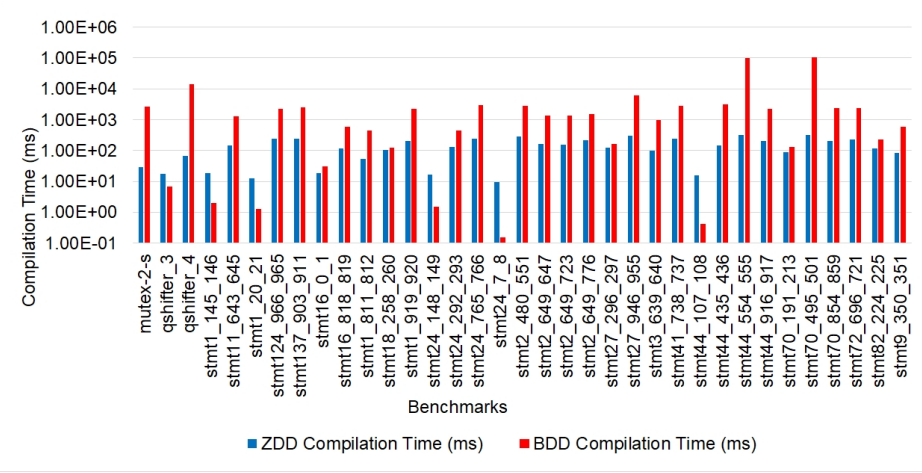


Fig. 1: Compilation time of the CNF: red = BDD, blue = ZDD

### 5.3 Realizability Time

The plot in Fig. 2 summarizes for each family the time spent on constructing the realizability set and checking partial and full realizability. The dashed lines in red illustrate RSynth results, while the solid lines in dark blue with the same shapes show ZSynth results. As each solvers have the families where it has an advantage in, we note how many instances of each family each solver is able to solve within a given time. We include data for all benchmarks that completed the

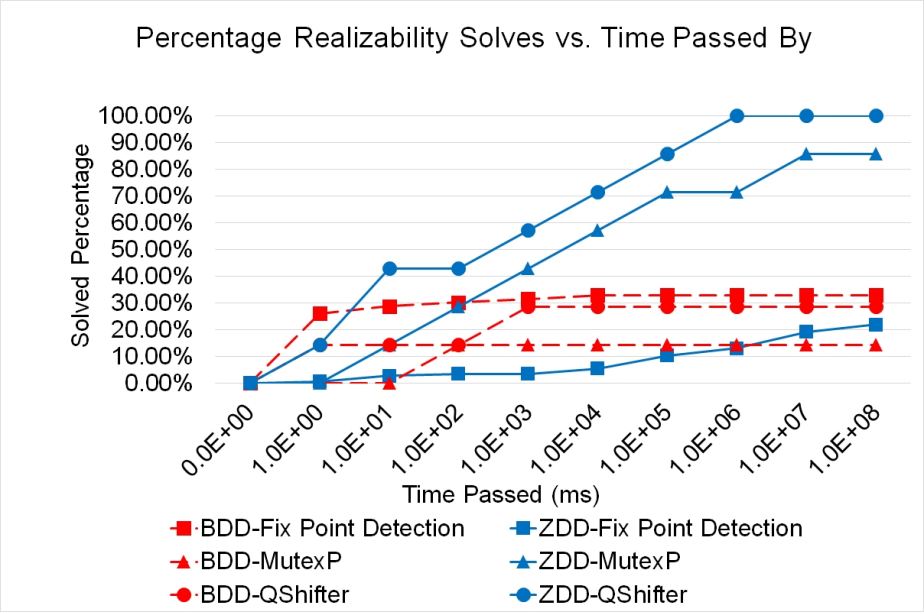


Fig. 2: Percentage solved for realizability within a given timeout. Dashed red = BDD, solid blue = ZDD.

realizability phase. The graph plots the percentage of benchmarks in each family that RSynth and ZSynth complete for a given timeout, with 100% meaning that all instances of that family were solved.

We see from Fig. 2 that RSynth solves more cases of the Fixpoint Detection family, and does so faster than ZSynth. Most of the cases it solves are completed in under 10ms. On the other hand, ZSynth has the advantage in the QShifter and MutexP families, for which it is able to solve more cases in a shorter time. Therefore, ZSynth and RSynth each performs better on different families of benchmarks. This allows us to answer the second research question proposed in Section 1 with the observation that neither approach dominates across the board, rather realizability performance is dependent on the benchmark family. As we see below in Section 5.4, these general results also extend to end-to-end synthesis.

### 5.4 End-to-End Time and Peak Memory

Our observations for end-to-end synthesis time–including compilation, realizability, and witness construction–are plotted in Fig. 3. Similarly to realizability time, the total end-to-end synthesis time shows strongly family-dependent results. Both ZSynth and RSynth display better relative performance on the same families as they did for realizability. In families where ZSynth solves more instances, including QShifter and MutexP, ZSynth also takes less time in most

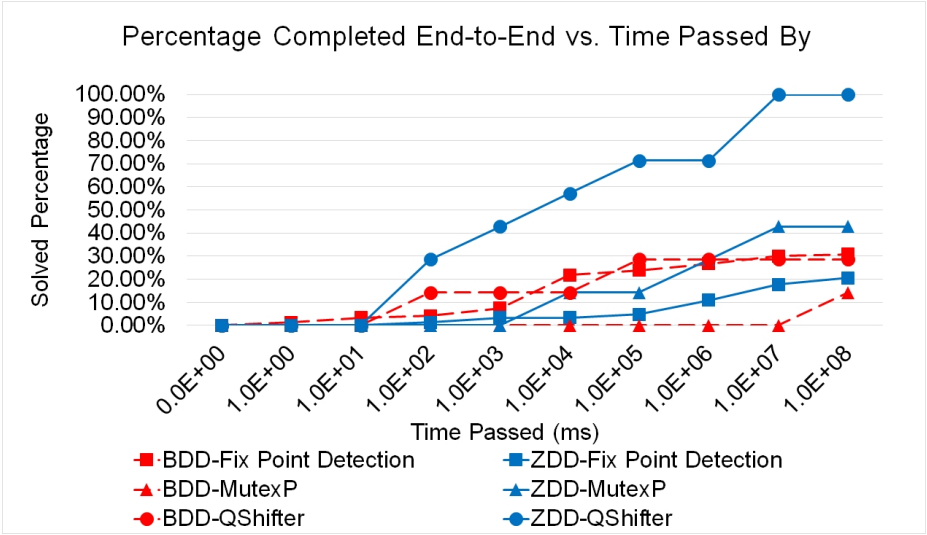


Fig. 3: Percentage solved end-to-end within a given timeout. Dashed red = BDD, solid blue = ZDD.

cases, and vice-versa for those families where RSynth solves more benchmarks end-to-end.

We observed in our experiments that memory and time were generally correlated, meaning that benchmarks that took more time also consumed more memory. This is expected when dealing with algorithms based on decision diagrams, since the biggest factor that impacts the performance of such algorithms is diagram size. In practice, memory comparison between RSynth and ZSynth in compilation, realizability and witness construction have similar patterns as the time comparison. Even if ZDDs have an advantage in representing the initial specification, the overall memory consumption for realizability and synthesis is, similarly to running time, largely dependent on the benchmark family.

### 5.5 Scalable Benchmarks Show ZDD has Slower Growing Demands of Time and Space

To analyze the scalability of ZDDs in relation to BDDs, as per the fifth research question in Section 1, we take a closer look at the running time and node counts of ZSynth and RSynth in the benchmarks of the QShifter family. All benchmarks in this family follow the same structure, just scaled based on a numerical parameter. For a parameter $n$, `qshifter_n` has $2^{2n+1}$ clauses, $2^n + n$ input variables and $2^n$ output variables, so we expect to see exponential trends in the measured values.

The results can be found in Fig. 4, which considers only QShifter because it can be scaled based on a parameter, and RSynth did not solve enough instances of MutexP to have an interesting scalability comparison. Since RSynth solves

only up to the smallest instances in the QShifter family, we use the maximal time limit, illustrated by the "X" in the plot not connected to any line, as a conservative underestimation for the running time of further instances. (Therefore, the compilation, realizability and end-to-end times for RSynth in `qshifter_5` must be higher than the "X" mark.) As QShifter benchmarks are regular in their constructions, we can observe the trend of the exponent.

The results for RSynth, both for time and number of nodes, always has a steeper slope in the parameter $n$. Since the graph is in log scale, straight lines represent an exponential increase, and the slope represents the coefficient of the exponent. Therefore, although both ZSynth and RSynth grow exponentially, in both time and space, ZSynth is more efficient by an exponential factor.

These results suggest that there are families for which we can expect ZDD synthesizers to require significantly fewer resources in time and space as the size of the formulas grows. The QShifter family is one example of a family where the ZDD algorithm performs better by an exponential factor.

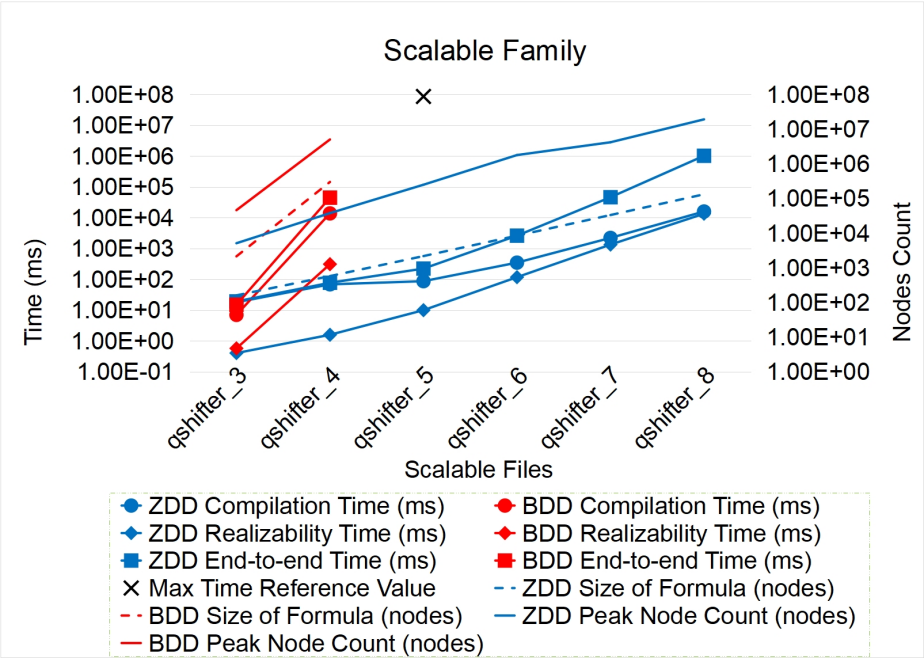


Fig. 4: Scalable family evaluations: dashed red = BDD, solid blue = ZDD.

### 5.6 Overall Comparison

As explained in Section 5.1, we focus on evaluating the synthesizers on the Fixpoint Detection, QShifter, and MutexP families of benchmarks [18]. ZSynth

Table 1: Percentage of end-to-end completed instances in each family.

| Benchmark Family Name | RSynth (BDD) | ZSynth (ZDD) |
|---|---|---|
| Fixpoint Detection | 30.82% | 20.55% |
| MutexP | 14.29% | 42.86% |
| QShifter (scalable) | 28.57% | 100% |

shows clear time and space advantages on the MutexP and QShifter families, while RSynth performs better in the Fixpoint Detection family. In Table 1, we show how much of each family either tool was able to solve.

Next, we summarize the overall results of our experimental performance comparison. In families where ZDD completed more instances end-to-end, we can see that ZDD has better performance in all bases of comparison, including compilation, realizability, and end-to-end time, as well as diagram node count for the original formula and peak node count. Additionally, Section 5.5 shows that there exist families of scalable benchmarks for which the time and space demands of ZDDs grow more slowly than BDDs by an exponential factor, as illustrated by the smaller slope in Fig. 4.

Even in the Fixpoint Detection family, where BDDs solve more instances, ZDDs show advantages in compilation time, initial diagram size, and smaller scaling slopes in time and space. In realizability and overall synthesis performance, neither our ZDD-based algorithm nor the BDD-based algorithm dominates across the board, each performing better in those families where it can solve more instances.

## 6 Conclusion

We conclude that ZDD-based algorithms are competitive with those based on BDDs, and both have their place in a portfolio of solvers for boolean synthesis. Since both BDDs and ZDDs can be converted to circuits, we advocate that an industrial solver would benefit from both approaches. In CNF-specified boolean-synthesis problems, BDD and ZDD are orthogonal approaches, and circumstances exist where each one of the solvers shows leading performance. For this type of problems, our portfolio advocates a multi-engine approach that is inclusive of both approaches.

As most tools for QBF solving and synthesis solving handle the input formula monolithically, future research based on this work includes an exploration of partitioning of variables [8] and factored synthesis [23] in the context of ZDDs. Another direction is to explore the usage of ZDD-based techniques in the context of temporal synthesis, cf. [24].

## References

1. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about Boolean functional synthesis? In: Proc. 30th Int'l Conf. on Computer Aided Ver-

ification, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 251–269. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_14, https://doi.org/10.1007/978-3-319-96145-3_14
2. Bryant, R.: Graph-based algorithms for Boolean-function manipulation. IEEE Transactions on Computing **C-35**(8), 677–691 (1986)
3. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: Proc. 2018 IEEE Conf. on Formal Methods in Computer Aided Design. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603000
4. Chatalic, P., Simon, L.: Multi-resolution on compressed sets of clauses. In: Proc 12th IEEE Int'l Conf. on Tools with Artificial Intelligence. pp. 2–10. IEEE Computer Society (2000). https://doi.org/10.1109/TAI.2000.889839
5. Chatalic, P., Simon, L.: ZRES: The old Davis-Putman procedure meets ZBDD. In: Proc. 17th Int'l Conf. on Automated Deduction. Lecture Notes in Computer Science, vol. 1831, pp. 449–454. Springer (2000)
6. Crama, Y., Hammer, P.L.: Boolean functions: Theory, algorithms, and applications. Cambridge University Press (2011)
7. Dechter, R., van Beek, P.: Local and global relational consistency. Theor. Comput. Sci. **173**(1), 283–308 (1997)
8. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: Procount: Weighted projected model counting with graded project-join trees. In: Proc. 24th Int'l Conf. on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 12831, pp. 152–170. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_11
9. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based Boolean functional synthesis. In: Proc. 28th Int'l Conf. on Computer Aided Verification. Part II. Lecture Notes in Computer Science, vol. 9780, pp. 402–421. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_22
10. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for Boolean function synthesis. In: Proc. 32nd Int'l Conf. on on Computer Aided Verification, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 611–633. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_31, https://doi.org/10.1007/978-3-030-53291-8_31
11. Gomes, C.P., Selman, B.: Algorithm portfolio design: Theory vs. practice. arXiv preprint arXiv:1302.1541 (2013)
12. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.R., Bloem, R.: Synthesizing multiple Boolean functions using interpolation on a single proof. In: Proc. 2013 IEEE Conf. Formal Methods in Computer-Aided Design. pp. 77–84. IEEE (2013)
13. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: Proc. 2015 IEEE Conf. Formal Methods in Computer-Aided Design. pp. 73–80. IEEE (2015)
14. Knuth, D.E.: The Art of Computer Programming, Volume 4, Pre-Fascicle 1B: A Draft of Section 7.1.4: Binary Decision Diagrams. Addison-Wesley Professional, 12th edn. (2009)
15. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: Proc. 12th Int'l Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 1855, pp. 113–123. Springer (2000). https://doi.org/10.1007/10722167_12
16. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. 30th Design Automation Conference. pp. 272–277. ACM Press (1993). https://doi.org/10.1145/157485.164890

17. Mishchenko, A.: Introduction to zero-suppressed decision diagrams. Synthesis Lectures on Digital Circuits and Systems **45** (2001)
18. Narizzano, M., Pulina, L., Tacchella, A.: The QBFEVAL web portal. In: Proc. 10th European Conf. on Logics in Artificial Intelligence. Lecture Notes in Computer Science, vol. 4160, pp. 494–497. Springer (2006). https://doi.org/10.1007/11853886_45, https://doi.org/10.1007/11853886_45
19. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Proc. 10th Int'l Conf. Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 3258, pp. 453–467. Springer (2004). https://doi.org/10.1007/978-3-540-30201-8_34
20. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Proc. 19th Int'l Conf. on Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 9710, pp. 375–392. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_23
21. Sasao, T., Butler, J.T.: Applications of zero-suppressed decision diagrams. Synthesis Lectures on Digital Circuits and Systems **9**(2), 1–123 (2014)
22. Somenzi, F.: CUDD: CU Decision Diagram Package Release 3.0.0. University of Colorado at Boulder (2015)
23. Tabajara, L.M., Vardi, M.Y.: Factored Boolean functional synthesis. In: Proc. 2017 IEEE Conf. on Formal Methods in Computer Aided Design. pp. 124–131. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102250
24. Zhu, S., Tabajara, L.M., Li, J., Pu, G., Vardi, M.Y.: Symbolic LTLf synthesis. In: Proc. 26th Int'l Joint Conf. on Artificial Intelligence. pp. 1362–1369. ijcai.org (2017)