On Dependent Variables in Reactive Synthesis

S. Akshay[1(✉)], Eliyahu Basa[2], Supratik Chakraborty[1], and Dror Fried[2]

[1] IIT Bombay, Mumbai, India
akshayss@cse.iitb.ac.in
[2] The Open University of Israel, Ra'anana, Israel

Abstract. Given a Linear Temporal Logic (LTL) formula over input and output variables, reactive synthesis requires us to design a deterministic Mealy machine that gives the values of outputs at every time step for every sequence of inputs, such that the LTL formula is satisfied. In this paper, we investigate the notion of dependent variables in the context of reactive synthesis. Inspired by successful pre-processing techniques in Boolean functional synthesis, we define dependent variables in reactive synthesis as output variables that are uniquely assigned, given an assignment to all other variables and the history so far. We describe an automata-based approach for finding a set of dependent variables. Using this, we show that dependent variables are surprisingly common in reactive synthesis benchmarks. Next, we develop a novel synthesis framework that exploits dependent variables to construct an overall synthesis solution. By implementing this framework using the widely used library Spot, we show that reactive synthesis that exploits dependent variables can solve some problems beyond the reach of existing techniques. Furthermore, we observe that among benchmarks with dependent variables, if the count of non-dependent variables is low (≤ 3 in our experiments), our method outperforms state-of-the-art tools for synthesis.

Keywords: Reactive synthesis · Functionally dependent variables· BDDs

1 Introduction

Reactive synthesis concerns the design of deterministic transducers (often Mealy or Moore machines) that generate a sequence of outputs in response to a sequence of inputs such that a given temporal logic specification is satisfied. Church introduced the problem [12] in 1962, and there has been a rich and storied history of work in this area over the past six decades. Recently, it was shown that a form of pre-processing, viz. decomposing a Linear Temporal Logic (LTL) specification, can lead to significant performance gains in downstream synthesis steps [15]. The general idea of pre-processing a specification to simplify synthesis has also been used very effectively in the context of Boolean functional synthesis [4,5,17,18,25]. Motivated by the success of one such pre-processing step, viz. identification of uniquely defined outputs, in Boolean functional synthesis, we introduce the notion of dependent outputs in the context of reactive synthesis in this paper. We develop its theory and show by means of extensive experiments that dependent outputs are common in reactive synthesis benchmarks, and can be effectively

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14570, pp. 123–143, 2024.
https://doi.org/10.1007/978-3-031-57246-3_8

exploited to obtain synthesis techniques with orthogonal strengths vis-a-vis existing state-of-the-art techniques.

In the context of propositional specifications, it is not uncommon for a specification to uniquely define an output variable in terms of the input variables and other output variables. A common example of this arises when auxiliary variables, called Tseitin variables, are introduced to efficiently convert a specification not in conjunctive normal form (CNF) to one that is in CNF [28]. Being able to identify such uniquely defined variables efficiently can be very helpful, whether it be for checking satisfiability, for model counting or synthesis. This is because these variables do not alter the basic structure or cardinality of the solution space of a specification regardless of whether they are projected out or not. Hence, one can often simplify the reasoning about the specification by ignoring (or projecting out) these variables. In fact, the remarkable practical success of Boolean functional synthesis tools such as Manthan [18] and BFSS [4,5] can be partly attributed to efficient techniques for identifying a large number of uniquely defined variables. We draw inspiration from these works and embark on an investigation into the role of uniquely defined variables, or *dependent variables*, in the context of reactive synthesis. To the best of our knowledge, this is the first attempt at directly using dependent variables for reactive synthesis.

We start by first defining the notion of dependent variables in LTL specifications for reactive synthesis. Specifically, given an LTL formula φ over a set of input variables I and output variables O, a set of variables $X \subseteq O$ is said to be *dependent* on a set of variables $Y \subseteq I \cup (O \backslash X)$ in φ, if at every step of every infinite sequence of inputs and outputs satisfying φ, the finite history of the sequence together with the current assignment for Y uniquely defines the current assignment for X. The above notion of dependency generalizes the notion of uniquely defined variables in Boolean functional synthesis, where the value of a uniquely defined output at any time is completely determined by the values of inputs and (possibly other) outputs at that time. We show that our generalization of dependency in the context of reactive synthesis is useful enough to yield a synthesis procedure with improved performance vis-a-vis competition-winning tools, for a non-trivial number of reactive synthesis benchmarks.

We present a novel automata-based technique for identifying a subset-maximal set of dependent variables in an LTL specification φ. Specifically, we convert φ to a language-equivalent non-deterministic Büchi automaton (NBA) A_φ, and then deploy practically efficient techniques to identify a subset-maximal set of outputs X that are dependent on $Y = I \cup (O \setminus X)$. We implemented our method to determine the prevalence of dependent variables in existing reactive synthesis benchmarks. Our finding shows that out of 1141 benchmarks taken from the SYNTCOMP [21] competition, 300 had at least one dependent output variable and 26 had all output variables dependent.

Once a subset-maximal set, say X, of dependent variables is identified, we proceed with the synthesis process as follows. Referring to the NBA A_φ alluded to above, we first transform it to an NBA A'_φ that accepts the language L' obtained from $L(\varphi)$ after removing (or projecting out) the X variables. Our

experiments show that A'_φ is more compactly representable compared to A_φ, when using BDD-based representations of transitions (as is done in state-of-the-art tools like Spot [7]). Viewing A'_φ as a new (automata-based) specification with output variables $O \setminus X$, we now synthesize a transducer T_Y from A' using standard reactive synthesis techniques. This gives us a strategy $f^Y : \Sigma_I^* \to \Sigma_{O \setminus X}$ for each non-dependent variable in $O \setminus X$. Next, we use a novel technique based on Boolean functional synthesis to directly construct a circuit that implements a transducer T_X that gives a strategy $f_X : \Sigma_Y^* \to \Sigma_X$ for the dependent variables. Significantly, this circuit can be constructed in time polynomial in the size of the (BDD-based) representation of A_φ. The transducers T_Y and T_X are finally merged to yield an overall transducer T that describes a strategy $f : \Sigma_I^* \to \Sigma_O$ solving the synthesis problem for φ.

We implemented our approach in a tool called DepSynt. Our tool is developed in C++ using APIs from the widely used library Spot for representing and manipulating non-deterministic Büchi automata. We performed a comparative analysis of our tool with winning entries of the SYNTCOMP [21] competition to evaluate how knowledge of dependent variables helps reactive synthesis. Our experimental results show that identifying and utilizing dependent variables results in improved synthesis performance when the count of non-dependent variables is low. Specifically, our tool outperforms state-of-the-art and highly optimized synthesis tools on benchmarks that have at least one dependent variable and at most 3 non-dependent variables. This leads us to hypothesize that exploiting dependent variables benefits synthesis when the count of non-dependent variables is below a threshold. Given the preliminary and un-optimized nature of our implementation, we believe there is significant scope for improvement.

Related work. Reactive synthesis has been an extremely active research area for the last several decades (see e.g. [9, 12, 15, 16, 24]). Not only is the theoretical investigation of the problem rich, there are also several tools that are available to solve synthesis problems in practice. These include solutions like `ltlsynt` [23] based on Spot [7], Strix [22] and BoSY [14]. Our tool relies heavily on Spot and its APIs, which we use liberally to manipulate non-deterministic Büchi automata. Our synthesis approach is based on the standard conversion of LTL formula to NBA, and then from NBA to deterministic parity automata (DPA) (see [8] for an overview of the challenges of reactive synthesis).

Our work may be viewed as lifting the idea of uniquely defined variables used in Boolean functional synthesis to the context of reactive synthesis. Viewed from this perspective, our work is not the first to lift ideas from Boolean functional synthesis to the reactive context. Following an approach for Boolean functional synthesis that decomposes a specification into separate formulas on input variables and on output variables [11], the work in [6] constructed a reactive synthesis tool for specific benchmarks that admit a separation of the specification into formulas for only environment variables and formulas for only system variables. The current work serves as an additional example in support of the hypothesis that intuition from Boolean functional synthesis can be helpful and effective in the reactive synthesis context.

The remainder of the paper is structured as follows. We introduce definitions and notations in Section 2. In Section 3 we define dependent variables for LTL formulas, and describe an algorithm to find them. In Section 4 we describe our automata-based synthesis framework and discuss its implementation details in Section 5. We describe our evaluation in Section 6 and conclude in Section 7. Missing proofs and additional experiments can be found in the full-version [2].

2 Preliminaries

Given a finite alphabet Σ, an infinite *word* w is a sequence $w_0 w_1 w_2 \cdots$ where for every i, the i^{th} letter of w, denoted w_i, is in Σ. The *prefix* $w_0 \cdots w_i$ (of size $i+1$) of w is denoted by $w[0,i]$. Note that $w[0,0] = w_0$. We use $w[0,-1]$ to denote the empty word. The set of all infinite words over Σ is denoted by Σ^ω. We call $L \subseteq \Sigma^\omega$ a *language* over infinite words in ω. For our work, the alphabet Σ is often the product of two distinct alphabets Σ_X and Σ_Y, i.e. $\Sigma = \Sigma_X \times \Sigma_Y$. In such cases, for every $a = (a_1, a_2) \in \Sigma$, we abuse notation and use $a.X$ to denote the projection of a on Σ_X, i.e. the letter $a_1 \in \Sigma_X$. Similarly, $a.Y$ denotes the projection of a on Σ_Y, i.e. the letter $a_2 \in \Sigma_Y$. For an infinite word $w \in \Sigma^\omega$, we use $w.X$ to denote the infinite word in Σ_X^ω obtained by projecting each letter in w on Σ_X i.e. $w.X = w_0.X w_1.X \ldots$.

Linear Temporal Logic. A Linear Temporal Logic (LTL) formula is constructed with a finite set of propositional variables V, using Boolean operators such as $\vee, \wedge$, and $\neg$, and temporal operators such as next (X), until (U), etc. The set V induces an alphabet $\Sigma_V = 2^V$ of all possible assignments ($true/false$) to the variables of V. The semantics of the operators and satisfiability relation are defined as usual [20]. The language of an LTL formula φ, denoted $L(\varphi)$ is the set of all words in Σ_V^ω that satisfy φ. For an LTL formula φ over V, we use $|V|$ to denote the number of variables in V, and $|\varphi|$ to denote the size of the formula, i.e., count of its subformulas. For clarity of exposition, we sometimes abuse notation and identify the singleton variable set $\{z\}$ with z. We also use Σ for Σ_V, when V is clear from the context.

Nondeterministic Büchi Automata. A Nondeterministic Büchi Automaton (NBA) is a tuple $A = (\Sigma, Q, \delta, q_0, F)$ where Σ is the alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a non-deterministic transition function, q_0 is the initial state and $F \subseteq Q$ is a set of accepting states. Automaton A can be seen as a directed labeled graph with vertices Q and an edge (q, q') exists with a label a if $q' \in \delta(q, a)$. We denote the set of incoming edges to q by $in(q)$ and the set of outgoing edges from q by $out(q)$. A *path* in A is a (possibly infinite) sequence of states $\rho = (q_{i_0}, q_{i_1}, \cdots)$ in which for every $j > 0$, $(q_{i_j}, q_{i_{j+1}})$ is an edge in A. A *run* is a path that starts in q_0, and is *accepting* if it visits a state in F infinitely often. A *word* $w = \sigma_{i_0}\sigma_{i_1} \cdots$ induces a run $\rho = (q_{i_0}, q_{i_1}, \cdots)$ of A if $q_{i_0} = q_0$ and for every $j \geq 0$, $q_{i_{j+1}} \in \delta(q_{i_j}, \sigma_{i_j})$. Since A is nondeterministic, a word can have many runs. A word is *accepting* if it has an accepting run in A. The language

$L(A)$ is the set of all accepting words in A. Wlog, we assume that all states and edges that are not a part of any accepting run (i.e. do not reach a cycle with an accepting state) are removed. This can be done by a simple pre-processing pass on the NBA. Finally, every LTL formula φ can be transformed in time exponential time in the size of φ to an NBA A_φ for which $L(\varphi) = L(A_\varphi)$ [20,29]. When φ is clear from the context we omit the subscript and refer to A_φ as A. We denote by $|A|$ the size of an automaton, i.e., number of its states and transitions.

Reactive Synthesis. A *reactive LTL formula* is an LTL formula φ over a set of input variables I and output variables O, with $I \cap O = \emptyset$. In *reactive synthesis* we are given a reactive LTL formula φ, and the challenge is to synthesize a function, called *strategy*, $f : \Sigma_I^* \to \Sigma_O$ such that every word $w \in (\Sigma_I \times \Sigma_O)^\omega$ obtained by using this strategy at every time step is in $L(\varphi)$. If such a strategy exists we say that φ is *realizable*. Otherwise, we say that φ is *unrealizable*. In what follows, we always consider only reactive LTL formulas and hence omit the "reactive" prefix while referring to them. The synthesized strategy $f : \Sigma_I^* \to \Sigma_O$ is typically described (explicitly or symbolically) as a *transducer* $T = (\Sigma_I, \Sigma_O, S, s_0, \delta, \lambda)$ in which Σ_I and Σ_O are input and output alphabet respectively, S is a set of states with an initial state s_0, $\delta : S \times \Sigma_I \to S$ is a deterministic transition function, and $\lambda : S \times \Sigma_I \to \Sigma_O$ is the output function. A standard procedure in solving reactive synthesis is to transform a given LTL formula φ to an NBA A_φ for which $L(A_\varphi) = L(\varphi)$. Subsequently, A_φ is transformed to a Deterministic Parity Automata (DPA) that turns to a parity game, whose solution is described as a transducer T_{A_φ}. As the following theorem shows, this approach incurs a double exponential blowup in the worst-case.

Theorem 1. *1. Reactive synthesis can be solved in $O(2^{n \cdot 2^n})$, where n is the size of the LTL formula.*
2. Given an NBA A with n states, computing transducer T_A takes $\Omega(2^{n \log n})$.

3 Dependent variables in reactive LTL

We begin by defining dependent variables for (reactive) LTL formulas and propose an algorithm for finding a maximal set of dependent variables. While there are several notions of dependency that can be considered, we discuss one that we have found to be useful in reactive synthesis. Specifically, we require that the value of a dependent output variable be completely determined by the values of inputs and other output variables and their finite history at every step of the interaction between the reactive system and its environment. We consider dependencies restricted to output variables, since having dependent input variables would preclude some input sequences, rendering the specification unrealizable.

Definition 1 (Variable Dependency in LTL). *Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \setminus I$. Let X, Y be disjoint sets of variables where $X \subseteq O$. We say that X is dependent on Y in φ if for every pair of words $w, w' \in L(\varphi)$ and $i \geq 0$ if $w[0, i-1] = w'[0, i-1]$ and*

$w_i.Y = w'_i.Y$, then we have $w_i.X = w'_i.X$. Further, we say that X is dependent in φ if X is dependent on $V \setminus X$ in φ, i.e., it is dependent on all the remaining variables.

Note that two words in $L(\varphi)$ with different prefixes can have different values for X for the same values for Y, if X is dependent on Y. Also, observe that if X is dependent on Y in φ for some Y, then it is also dependent in φ.

As an example, consider an LTL formula φ with input variable y and output variable x. The corresponding input and output alphabets are $\Sigma_X = \{x, \neg x\}$ and $\Sigma_Y = \{y, \neg y\}$ respectively. Suppose $L(\varphi) = \{w^1, w^2, w^3\}$ where $w^1 = (y, x)^\omega$, $w^2 = (\neg y, x)^\omega$ and $w^3 = (y, x)(\neg y, x)(y, \neg x)^\omega$. Then x is dependent on y in φ. Specifically, note that $w^1[0, 1] \neq w^3[0, 1]$, and hence the dependency of x is not violated although $w^1_2.y = w^3_2.y$ and $w^1_2.x \neq w^3_2.x$.

3.1 Maximally dependent sets of variables Given an LTL formula $\varphi(I, O)$, we say that a set $X \subseteq O$ is a *maximal dependent set* in φ if X is dependent in φ and every set of outputs that strictly contains X is not dependent in φ. As in the propositional case [27], finding maximum or minimum dependent sets is intractable, hence we focus on subset-maximality. Given a variable z and set Y, checking whether z is dependent on Y, can easily be used to finding a maximal dependent set. Indeed, we would just need to start from the empty set and iterate over output variables, checking for each if it is dependent on the remaining variables. We give the pseudocode for this in [2]. Note that when all output variables are not dependent, the order in which output variables are chosen may play a significant role in the size of the maximal set obtained. We currently use a naive ordering (first appearance), and leave the problem of better heuristics for getting larger maximal independent sets to future work.

3.2 Finding dependent variables via automata As explained above, the heart of the dependency check is to verify whether a given output variable is dependent on a set of other variables. We now develop an approach for doing so based on the nondeterministic Büchi automaton A_φ that represents the same language as the LTL formula φ. Our framework uses the notion of compatible pairs of states of the automaton:

Definition 2. *Let $A = (\Sigma, Q, \delta, q_0, F)$ be an NBA with states s, s' in Q. Then the pair (s, s') is* compatible *in A if there are runs from q_0 to s and from q_0 to s' on the same word $w \in \Sigma^*$.*

Recall that in our definition, only states and edges that are part of an accepting run exist in A. Then we have the following definition.

Definition 3. *Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \setminus I$. Let X, Y be disjoint sets of variables where $X \subseteq O$. Let A_φ be an NBA that describes φ. We say that X is* automata dependent *on Y in A_φ, if for every pair of compatible states s, s' and assignments σ, σ' for V, where $\sigma.Y = \sigma'.Y$ and $\sigma.X \neq \sigma'.X$, $\delta(s, \sigma)$ and $\delta(s, \sigma')$ cannot both exist in A_φ. We say that X is automata dependent in A_φ if X is automata dependent on Y in A_φ and $Y = V \setminus X$.*

As an example, consider NBA A_1 in Figure 1, constructed from some LTL formula with input $I = \{i\}$ and outputs $O = \{o_1, o_2\}$. For notational simplicity, we use $\Sigma_I = \{0, 1\}$, $\Sigma_O = \{0, 1\}^2$, and edges are labeled by values of $(i, o_1 o_2)$. It is easy to see that $(q_0, q_0), (q_1, q_1)$ are compatible pairs, but so are $(q_0, q_1), (q_1, q_0)$ since both q_0 and q_1 be reached from the initial state on reading the word $(0, 00)(0, 00)$ of length 2. Now consider output o_1. It is not dependent on $\{i\}$, i.e., only the input, since from q_0 with $i = 0$, we can go to different states with different values of o_1. But o_1 is indeed dependent on $\{i, o_2\}$. To see this consider every pair of compatible states – in this case all pairs. Then if we fix the values of i and o_2, there is a unique value of o_1 that permits state transitions to happen from the compatible pair. For example, regardless of which state we are in, if $i = 0, o_2 = 0$, o_1 must be 0 for a state transition to happen. On the other hand, o_2 is not dependent on either $\{i\}$ or $\{i, o_1\}$ (as can be seen from (q_0, q_1) with $i = 1, o_1 = 1$). The following theorem relates automata-based dependency and dependency in LTL (for proof, see [2]), allowing us to focus only on the former.

Theorem 2. *Let φ be an LTL formula with set of variables $V = I \cup O$, where $X \subseteq O$ and $Y \subseteq I \cup (O \setminus X)$. Let A_φ be an NBA with $L(\varphi) = L(A_\varphi)$. Then X is dependent on Y in φ if and only if X is automata dependent on Y in A_φ.*

Finding Compatible States. We find all compatible states in an automaton in Algorithm 1 as follows. We maintain a list of in-process compatible pairs C that is initialized with (q_0, q_0) – an undoubtedly compatible pair. At each step, until C becomes empty, we pick a pair $(s_i, s_j) \in C$, add it to the compatible pair set P, and remove it from C (in line 4). Then (in lines 5-8), we check (in line 6) if outgoing transitions from (s_i, s_j) lead to a new pair (s'_i, s'_j) not already in P or C, that can be reached on reading the same letter σ. If so, we add this pair to the in-process set C. All pairs that we add to P, C are indeed compatible, and nothing is removed from P. When the algorithm terminates, C is empty, which means all possible ways (from initial state pair) to reach a compatible pair have been explored, thus showing correctness.

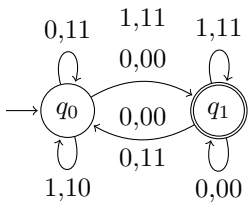

Fig. 1. An Example NBA A_1

Finally, we show how to check dependency using automata, by implementing procedure **isAutomataDependent**, shown in Algorithm 2. This procedure takes an NBA A_φ, a candidate dependent output z and a candidate dependency set $Y \subseteq V \setminus \{z\}$ as inputs, and tries to find a witness to z *not* being dependent on Y. If no such witness exists, then z is declared as being dependent on Y. Procedure **isAutomataDependent** first uses Algorithm 1 to construct a list P of all compatible pairs in A (line 4). Then for every pair $(s, s') \in P$, the algorithm checks using procedure AreStatesColliding (lines 1-2) whether there exists an assignment σ, σ' for which both $\delta(s, \sigma)$ and $\delta(s', \sigma')$ exist, $\sigma.Y = \sigma'.Y$ and $\sigma.\{z\} \neq \sigma'.\{z\}$. If so, z is not dependent on Y (line 7) and the algorithm returns *false*. Otherwise, afterchecking all the pairs, the algorithm returns *true*.

Algorithm 1 Find All Compatible States in NBA

Input NBA $A_\varphi = (\Sigma, Q, \delta, q_0, F)$ of φ.
Output Set $P \subseteq Q \times Q$ of all compatible state pairs in A_φ

$P \leftarrow \emptyset$; $C \leftarrow \{(q_0, q_0)\}$
while $C \neq \emptyset$ **do**
  Let $(s_i, s_j) \in C$
  $P \leftarrow P \cup \{(s_i, s_j)\}$; $C \leftarrow C \setminus \{(s_i, s_j)\}$
  for $(s_i', s_j') \in out(s_1) \times out(s_2)$ **do**
    if $(s_i', s_j') \notin P \cup C$ and $\exists \sigma \in 2^\Sigma$ s.t. $s_i' \in \delta(s_i, \sigma) \wedge s_j' \in \delta(s_j, \sigma)$ **then**
      $C \leftarrow C \cup \{(s_i', s_j')\}$
    end if
  end for
end while
return P

Algorithm 2 Check Dependency Based Automaton

Input NBA $A_\varphi = (\Sigma, Q, \delta, q_0, F)$ from φ, Candidate dependent variable z, Candidate dependency set Y.
Output Is z dependent on Y by Definition 3

procedure AreStateColliding(p, q)
  return $\exists \sigma_p, \sigma_q \in 2^\Sigma$ s.t. $\delta(p, \sigma_p) \neq \emptyset \wedge \delta(q, \sigma_q) \neq \emptyset \wedge \sigma_p.Y = \sigma_q.Y \wedge \sigma_p.\{z\} \neq \sigma_q.\{z\}$
end procedure
$P \leftarrow FindAllCompatibleStates(A_\varphi)$
for $(s_1, s_2) \in P$ **do**
  if $AreStateColliding(s_1, s_2)$ **then**
    return False
  end if
end for
return True

Lemma 1. *Algorithm 2 returns* True *if and only if z is automata-dependent on Y in A_φ.*

Using the above algorithm to perform dependency check, it is easy to compute a maximal set of dependent variables (as explained earlier). Note that all the above algorithms run in time polynomial (in fact, quadratic) in size of the NBA.

Corollary 1. *Given NBA A_φ, a maximal dependent set of outputs can be computed in time polynomial in the size of A_φ.*

Note that if all output variables are dependent, then regardless of the order in which the outputs are considered, for every finite history of inputs, there is a unique value for each output that makes the specification true. Therefore, there is a unique winning strategy for the specification, assuming it is realizable.

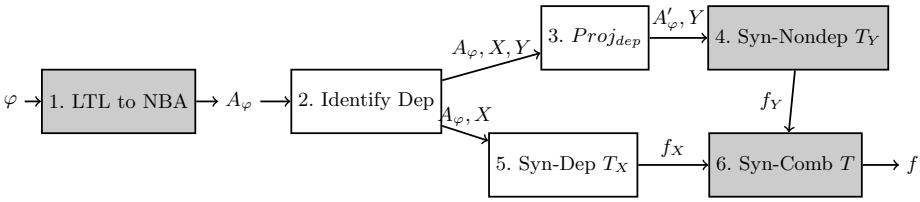

Fig. 2. Synthesis using dependencies. Note that Steps 2., 3., 5, are novel, while Steps 1., 4., 6. (shaded in gray) use pre-existing techniques.

4 Exploiting Dependency in Reactive Synthesis

In this section, we explain how dependencies can be beneficially exploited in a reactive synthesis pipeline. Our approach can be described at a high level as shown in Figure 2. This flow-chart has the following 6 steps:

1. Given an LTL formula φ over a set of variables V with input variables $I \subseteq V$ and output variables $O = V \backslash I$, we first construct a language-equivalent NBA $A_\varphi = (\Sigma_I \cup \Sigma_O, S, s_0, \delta, F)$ by standard means, e.g [29].
2. Then, as described in Section 3, we find in A_φ a maximal set of output variables X that are dependent in φ. For notational convenience, in the remainder of the discussion, we use Y for $I \cup (O \backslash X)$ and Σ_Y for $\Sigma_I \times \Sigma_{O \backslash X}$.
3. Next, we construct an NBA A'_φ from A_φ by projecting out (or eliminating) all X variables from labels of transitions. Thus, A'_φ has the same sets of states and transitions as A_φ. We simply remove valuations of variables in X from the label of every state transition in A_φ to obtain A'_φ. Note that after this step, $L(A'_\varphi) = \{w \mid \exists u \in L(A_\varphi) \text{ s.t. } w = u.Y\} \subseteq \Sigma_Y^\omega$.
4. Treating A'_φ as a (automata-based) specification with inputs I and outputs $O \setminus X$, we next use existing reactive synthesis techniques (e.g., [8]) to obtain a transducer T_Y that describes a strategy $f_Y : \Sigma_I^* \to \Sigma_{O \backslash X}$ for $L(A'_\varphi)$.
5. We also construct a transducer T_X that describes a function $f_X : (\Sigma_Y^* \to \Sigma_X)$ with the following property: for every word $w' \in L(A'_\varphi)$ there exists a unique word $w \in L(\varphi)$ such that $w.Y = w'$ and for all i, $w_i.X = f_X(w'[0, i])$.
6. Finally, we compose T_X and T_Y to construct a transducer T that defines the final strategy $f : \Sigma_I^* \to \Sigma_O$. Recall that transducer T_Y has I as inputs and $O \setminus X$ as outputs, while transducer T_X has I and $O \setminus X$ as inputs and X as outputs. Composing T_X and T_Y is done by simply connecting the outputs $O \setminus X$ of T_Y to the corresponding inputs of T_X.

In the above flow, we use standard techniques from the literature for Steps 1 and 4, as explained above. Hence we do not dwell on these steps in detail. Step 2 was detailed in Section 3. Step 3 is easy when we have an explicit representation of the automata, but it has interesting consequences when using symbolic representations of automata. Step 6 is also easy to implement. Hence, in the remainder of this section, we focus on Step 5, a key contribution of this paper. In the next section, we will discuss how steps 2, 3 and 5 are implemented using symbolic representations (viz. ROBDDs).

Constructing transducer T_X Let $A = (\Sigma_I \times \Sigma_O, Q, \delta, q_0, F)$ be the NBA A_φ obtained in step 1 of the pipeline shown above. Since each letter in Σ_O can be thought of as a pair (σ, σ'), where $\sigma \in \Sigma_{O\setminus X}$ and $\sigma' \in \Sigma_X$, the transition function δ can be viewed as a map from $Q \times (\Sigma_I \times \Sigma_{O\setminus X} \times \Sigma_X)$ to 2^Q. The transducer T_X we wish to construct is a deterministic Mealy machine described by the 6-tuple $(\Sigma_Y, \Sigma_X \cup \{\bot\}, Q^X, q_0^X, \delta^X, \lambda^X)$, where $\Sigma_Y = \Sigma_I \times \Sigma_{(O\setminus X)}$ is the input alphabet, Σ_X is the output alphabet with $\bot \notin \Sigma_X$ being a special symbol that is output when no symbol of Σ_X suffices, $Q^X = 2^Q$, that is the powerset of Q is the set of states of T_X, $q_0^X = \{q_0\}$ is the initial state, $\delta^X : Q^X \times \Sigma_I \times \Sigma_{(O\setminus X)} \to Q^X$ is the state transition function, and $\lambda^X : Q^X \times \Sigma_I \times \Sigma_{(O\setminus X)} \to \Sigma_X$ is the output function. The state transition function δ^X is defined by the Rabin-Scott subset construction applied to the automaton A_φ [19]. Formally, for every $U \subseteq Q$, $\sigma_I \in \Sigma_I$ and $\sigma \in \Sigma_{(O\setminus X)}$, we define $\delta^X\big(U, (\sigma_I, \sigma)\big) = \{q' \mid q' \in Q, \exists q \in U$ and $\exists \sigma' \in \Sigma_X$ s.t. $q' \in \delta\big(q, (\sigma_I, \sigma, \sigma')\big)\}$. Before defining the output function λ^X, we state an important property of T^X that follows from the definition of δ^X above.

Lemma 2. *If X is automata dependent in A_φ, then every state U reachable from q_0^X in T_X satisfies the property: $\forall q, q' \in U$, (q, q') is compatible in A_φ.*

The lemma is easily proved by induction on the number of steps needed to reach U from q_0^X. Details of the proof may be found in [2]. We are now ready to define the output function λ^X of T_X. Let U be a state reachable from q_0^X in T_X and let $U' = \delta^X\big(U, (\sigma_I, \sigma)\big)$, where $(\sigma_I, \sigma) \in \Sigma_Y$. If $U' \neq \emptyset$, we can infer that (see Proof of Lemma 2 in [2]) that there is a unique $\sigma_X \in \Sigma_X$ s.t. $U' = \{q' \mid \exists q \in U$ s.t. $q' \in \delta\big(q, (\sigma_I, \sigma, \sigma_X)\big)\}$. We define $\lambda^X\big(U, (\sigma_I, \sigma)\big) = \sigma_X$ in this case. If, on the other hand, $U' = \emptyset$, we define $\lambda^X\big(U, (\sigma_I, \sigma)\big) = \bot$.

Theorem 3. *If φ is realizable, the transducer T obtained by composing T_X and T_Y as in step 6 of Fig. 2 solves the synthesis problem for φ.*

An interesting corollary of the above result is that for realizable specifications with all output variables dependent, we can solve the synthesis problem in time $O(2^k)$ instead of $\Omega(2^{k \log k})$, where $k = |A_\varphi|$. This is because the subset construction on A_φ suffices to obtain T_X, while A_φ must be converted to a deterministic parity automaton to solve the synthesis problem in general.

5 Symbolic Implementation

In this section, we describe symbolic implementations of each of the non-shaded steps in the synthesis flow depicted in Fig. 2. Before we delve into the details, a note on the representation of NBAs is relevant. We use the same representation as used in Spot [7] – a state-of-the-art platform for representing and manipulating LTL formulas and ω-automata. Specifically, the transition structure of an NBA A is represented as a directed graph, with nodes representing states of A, and directed edges representing state transitions. Furthermore, every edge from state s to state s' is labeled by a Boolean function $B_{(s,s')}$ over $I \cup O$. The Boolean

function can itself be represented in several forms. We assume it is represented as a Reduced Ordered Binary Decision Diagram (ROBDD) [10], as is done in Spot. Each such labeled edge represents a set of state transitions from s to s', with one transition for each satisfying assignment of $B_{(s,s')}$.

Implementing Algorithms 1 and 2 (Step 2) : Since states of the NBA A_φ are explicitly represented as nodes of a graph, it is straightforward to implement Algorithms 1 and 2. The check in line 6 of Algorithm 1 is implemented by checking the satisfiability of $B_{(s_i,s'_i)}(I,O) \wedge B_{(s_j,s'_j)}(I,O)$ using ROBDD operations. Similarly, the check in line 2 of Algorithm 2 is implemented by checking the satisfiability of $\bigvee_{(s,s')\in out(p)\times out(q)} B_{(p,s)}(I,O) \wedge B_{(q,s')}(I',O') \wedge \bigwedge_{y\in Y}(y \leftrightarrow y') \wedge (z \leftrightarrow \neg z')$ using ROBDD operations. In the above formula, I' (resp. O') denotes a set of fresh, primed copies of variables in I (resp. O).

Implementing transformation of A_φ to A'_φ (Step 3): To obtain A'_φ, we simply replace the ROBDD for $B_{(s,s')}$ on every edge (s,s') of the NBA A_φ by an ROBDD for $\exists X\, B_{(s,s')}$. While the worst-case complexity of computing $\exists X\, B_{(s,s')}$ using ROBDDs is exponential in $|X|$, this doesn't lead to inefficiencies in practice because $|X|$ is typically small. Indeed, our experiments reveal that the total size of ROBDDs in the representation of A'_φ is invariably smaller, sometimes significantly, compared to the total size of ROBDDs in the representation of A_φ. Indeed, this reduction can be significant in some cases, as the following proposition shows (see proof in [2]).

Proposition 1. *There exists an NBA A_φ with a single dependent output such that the ROBDD labeling its edge is exponentially (in number of inputs and outputs) larger than that labeling the edge of A'_φ.*

Implementing transducer T_X (Step 5): We now describe how to construct a Mealy machine corresponding to the transducer T_X. As explained in the previous section, the transition structure of the Mealy machine is obtained by applying the subset construction to A_φ. While this requires $O(2^{|A_\varphi|})$ time if states and transitions are explicitly represented, we show below that a sequential circuit implementing the Mealy machine can be constructed directly from A_φ in time polynomial in $|X|$ and $|A_\varphi|$. This reduction in construction complexity crucially relies on the fact that all variables in X are dependent on $I \cup (O \setminus X)$.

Let $S = \{s_0, \ldots s_{k-1}\}$ be the set of states of A_φ, and let $\mathsf{in}(s_i)$ denote the set of states that have an outgoing transition to s_i in A_φ. To implement the desired Mealy machine, we construct a sequential circuit with k state-holding flip-flops. Every state U ($\subseteq S$) of the Mealy machine is represented by the state of these k flip-flops, i.e. by a k-dimensional Boolean vector. Specifically, the i^{th} component is set to 1 iff $s_i \in U$. For example, if $S = \{s_0, s_1, s_2\}$ and $U = \{s_0, s_2\}$, then U is represented by the vector $\langle 1,0,1\rangle$. Let n_i and p_i denote the next-state input and present-state output of the i^{th} flip-flop. The next-state function δ^X from $p'_i s$ to $n'_i s$ of the Mealy machine is implemented by a circuit, say Δ^X, with inputs $\{p_0, \ldots p_{k-1}\} \cup I \cup (O \setminus X)$ and outputs $\{n_0, \ldots n_{k-1}\}$. For $i \in \{0, \ldots k-$

1}, output n_i of this circuit implements the Boolean function $\bigvee_{s_j \in \mathsf{in}(s_i)} \big(p_j \wedge \exists X\, B_{(s_j,s_i)}\big)$. To see why this works, suppose $\langle p_0, \ldots p_{k-1}\rangle$ represents the current state $U \subseteq S$ of the Mealy machine. Then the above function sets n_i to true iff there is a state $s_j \in U$ (i.e. $p_j = 1$) s.t. there is a transition from s_j to s_i on some values of outputs X and for the given values of $I \cup (O \setminus X)$ (i.e. $\exists X\, B_{(s_j,s_i)} = 1$). This is exactly the condition for s_i to be present in the state $U' \subseteq S$ reached from U for the given values of $I \cup (O \setminus X)$ in the Mealy machine obtained by subset construction.

It is known from the knowledge compilation literature (see e.g. [1,4,13]) that every ROBDD can be compiled in linear time to a Boolean circuit in Decomposable Negation Normal Form (DNNF), and that every DNNF circuit admits linear time projection of variables, yielding a resultant DNNF circuit. Hence, a Boolean circuit for $\exists X\, B_{(s_j,s_i)}$ can be constructed in time linear in the size of the ROBDD representation of $B_{(s_j,s_i)}$. This allows us to construct the circuit Δ^X, implementing the next-state transition logic of our Mealy machine, in time (and space) linear in $|X|$ and $|A_\varphi|$.

Next, we turn to constructing a circuit Λ^X that implements the output function λ^X of our Mealy machine. It is clear that Λ^X must have inputs $\{p_0, \ldots p_{k-1}\} \cup I \cup (O \setminus X)$ and outputs X. Since X is automata dependent on $I \cup (O \setminus X)$ in A_φ, the following proposition is easily seen to hold.

Proposition 2. *Let $B_{(s,s')}$ be a Boolean function with support $I \cup O$ that labels a transition (s, s') in A_φ. For every $(\sigma_I, \sigma) \in \Sigma_I \times \Sigma_{O\setminus X}$, if $(\sigma_I, \sigma) \models \exists X\, B_{(s,s')}$, then there is a unique $\sigma' \in \Sigma_X$ such that $(\sigma_I, \sigma, \sigma') \models B_{(s,s')}$.*

Considering only the transition (s, s') referred to in Proposition 2, we first discuss how to synthesize a vector of Boolean functions, say $F^{(s,s')} = \langle F_1^{(s,s')}, \ldots F_{|X|}^{(s,s')}\rangle$, where each component function has support $I \cup (O \setminus X)$, such that $F^{(s,s')}[I \mapsto \sigma_I][O \setminus X \mapsto \sigma] = \sigma'$. Generalizing beyond the specific assignment of $I \cup O$, our task effectively reduces to synthesizing an $|X|$-dimensional vector of Boolean functions $F^{(s,s')}$ s.t. $\forall I \cup (O \setminus X)\, \big(\exists X B_{(s,s')} \;\rightarrow\; B_{(s,s')}[X \mapsto F^{(s,s')}]\big)$ holds. Interestingly, this is an instance of *Boolean functional synthesis* – a problem that has been extensively studied in the recent past (see e.g. [1,3,4,6,11]). In fact, we know from [1,26] that if $B_{(s,s')}$ is represented as an ROBDD, then a Boolean circuit for $F_{(s,s')}$ can be constructed in $\mathcal{O}\big(|X|^2.|B_{(s,s')}|\big)$ time, where $|B_{(s,s')}|$ denotes the size of the ROBDD for $B_{(s,s')}$. For every $x_i \in X$, we use this technique to construct a Boolean circuit for $F_i^{(s,s')}$ for every edge (s, s') in A. The overall circuit Λ^X is constructed such that the output for $x_i \in X$ implements the function $\bigvee_{transition\ (s,s')\ in\ A} \big(p_s \wedge (B_{(s,s')}[X \mapsto F^{(s,s')}]) \wedge F_i^{(s,s')}\big)$.

Lemma 3. *Let $U \subseteq S$ be a non-empty set of pairwise compatible states of A. For $(\sigma_I, \sigma) \in \Sigma_I \times \Sigma_{O\setminus X}$, if $\delta^X\big(U, (\sigma_I, \sigma)\big) \neq \emptyset$, then the outputs X of Λ^X evaluate to $\lambda^X\big(U, (\sigma_I, \sigma)\big)$. In all other cases, every output of Λ^X evaluates to 0.*

Note that $\delta^X\big(U, (\sigma_I, \sigma)\big) = \emptyset$ iff all outputs n_i of the circuit Δ^X evaluate to 0. This case can be easily detected by checking if $\bigvee_{i=0}^{k-1} n_i$ evaluates to 0. We therefore have the following result.

Theorem 4. *The sequential circuit obtained with Δ^X as next-state function and Λ^X as output function is a correct implementation of transducer T_X, assuming (a) the initial state is $p_0 = 1$ and $p_j = 0$ for all $j \in \{1, \dots k-1\}$, and (b) the output is interpreted as $\bot$ whenever $\bigvee_{i=0}^{k-1} n_i$ evaluates to 0.*

6 Experiments and Evaluation

We implemented the synthesis pipeline depicted in Figure 2 in a tool called DepSynt (accessible at `https://github.com/eliyaoo32/DepSynt`), using symbolic approach of Section 5. For Steps 1., 4., of the pipeline, i.e., construction of A_φ and synthesis of T_Y, we used the tool Spot [7], a widely used library for representing and manipulating NBAs. We then experimented with all available reactive synthesis benchmarks from the SYNTCOMP [21] competition, a total of 1,141 LTL specifications over 31 benchmark families.

All our experiments were run on a computer cluster, with each problem instance run on an Intel Xeon Gold 6130 CPU clocking at 2.1 GHz with 2GB memory and running Rocky Linux 8.6. Our investigation was focussed on answering two main research questions:
RQ1: How prevalent are dependent outputs in reactive synthesis benchmarks?
RQ2: Under what conditions, if any, is reactive synthesis benefited by our approach, i.e., of identifying and separately processing dependent output variables?

Dependency Prevalence. To answer **RQ1**, we implemented the algorithm in Section 3 and executed it with a timeout of 1 hour. Within this time, we were able to find 300 benchmarks out of 1,141 SYNTCOMP benchmarks, that had at least 1 dependent output variable (as per Definition 3). Out of the 1,141 benchmarks, 260 had either timeout (41 total) or out-of-memory (219 total), out of which 227 failed because of the NBA construction (adapted from Spot), i.e, Step 1 in our pipeline, did not terminate. We found that all the benchmarks with at least 1 dependent variable in fact belong to one of 5 benchmark families, as seen in Table 1. In order to measure the prevalence of dependency we evaluated (1) the number of dependent variables and (2) the dependency ratio $= \frac{\text{Total dependent vars}}{\text{Total output vars}}$. Out of those depicted, Mux (for multiplexer) and shift (for shift-operator operator) were two benchmark families

Benchmark Family	Total	Completed	Found Dep	Avg Dep Ratio
ltl2dpa	24	24	24	.434
mux	12	12	4	1
shift	11	4	4	1
tsl-paper	118	117	115	.46
tsl-smart-home-jarvis	189	167	153	.33

Table 1. Summary for 5 benchmark families, indicating the no. of benchmarks, where the dependency-finding process was completed, the total count of benchmarks with dependent variables, and the average dependency ratio among those with dependencies.

where dependency ratio was 1. In total, among all those where our dependency checking algorithm terminated, we found 26 benchmarks with all the output variables dependent. Of these 4 benchmarks were from Shift, 4 benchmarks from mux, 14 benchmarks from tsl-paper, and 4 from tsl-smart-home-jarvis. Looking beyond total dependency, among the 300 benchmarks with at least 1 dependent variable, we found a diverse distribution of dependent variables as shown in Figure 3 (distribution wrt dependency ratio is in [2]).

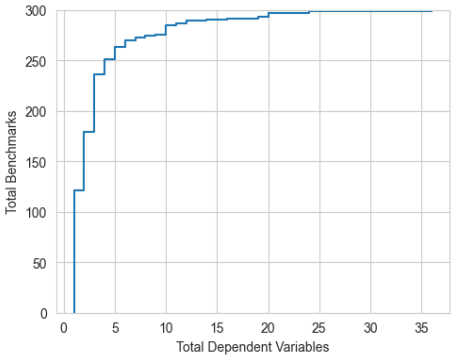

Fig. 3. Cumulative count of benchmarks for each unique value of Total Dependent Variables. $F(x)$ on y-axis represents how many benchmarks have at most x (on x-axis) dependent variables.

Utilizing Dependency for Reactive Synthesis: Comparison with other tools. Despite a large 1 hr time out, we noticed that most dependent variables were found within 10-12 seconds. Hence, in our tool DepSynt, we limited the time for dependency-check to an empirically determined 12 seconds, and declared unchecked variables after this time as non-dependent. Since synthesis of non-dependents T_Y (Step 5. of the pipeline) is implemented directly using Spot APIs, the difference between our approach and Spot is minimal when there are a large number of non-dependent variables. This motivated us to divide our experimental comparison, among the 300 benchmarks where at least one dependent variables was found, into benchmarks with at most 3 non-dependent variables (162 benchmarks) and more than 3 non-dependent variables (138 benchmarks). We compared DepSynt with two state-of-the-art synthesis tools, that won in different tracks of SYNTCOMP23' [21]: (i) Ltlsynt (based on Spot) [7] with different configurations ACD, SD, DS, LAR, and (ii) Strix [22] with the configuration of BFS for exploration and FPI as parity game solver (the overall winning configuration/tool in SYNTCOMP'23). All the tools had a total timeout of 3 hours per benchmark. As can be seen from Figure 4, indeed for the case of ≤ 3 non-dependent variables, DepSynt outperforms the highly optimized competition-winning tools. Even for > 3 case, as shown in Figure 5, the performance of DepSynt is comparable to other tools, only beaten eventually by Strix. DepSynt uniquely solved 2 specifications for which both Strix and Ltlsynt timed out after 3600s, the benchmarks are mux32, and mux64, and solved in 2ms, and 4ms respectively.

Analyzing time taken by different parts of the pipeline. In order to better understand where DepSynt spends its time, we plotted in Figure 6 the normalized time distribution of DepSynt. We can see that synthesizing a strategy for dependent variables is very fast (the yellow portion)- justifying its theoretical linear complexity bound, and so is the pink region depicting searching for dependency

(again, a poly-time algorithm), especially compared to the blue synthesizing a strategy for the non-dependent variables, and the green which is NBA build time. This also explains why having a high dependency ratio alone does not help our approach, since even with a high ratio, the number of non-dependent variables could be large, resulting in worse performance overall.

Analysis of the Projection step (Step 3.) of Pipeline. The rationale for projecting variables from the NBA is to reduce the number of output non-dependent variables in the synthesis of the NBA, which is the most expensive phase as Figure 6 shows. To see if this indeed contributes to our better performance, we asked if projecting the dependent variables reduces the BDDs' sizes, in terms of total nodes, (the BDD represents the transitions). Figure 7 shows that the BDDs' sizes are reduced significantly where the total of non-dependent variables is at most 3, in cases of total dependency, the BDD just vanishes and is replaced by the constant true/false. For the case of total non-dependent is 4 or more, the BDD size is reduced as well.

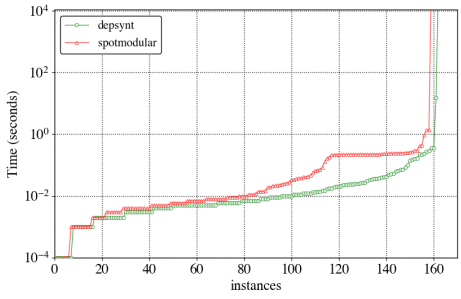

Fig. 8. Cactus plot comparing DepSynt and SpotModular on 162 benchmarks with at most 3 non-dependent variables.

An ablation experiment with Spot. As a final check, that dependency was causing the improvements seen, we conducted a control/ablation experiment where in DepSynt we gave zero-timeout to find dependency, classified all output variables as non-dependent, and called this SpotModular. As can be seen in Figure 8, for the case of benchmarks with at least 1 dependent and at most 3 non-dependent variables, this clearly shows the benefit of dependency-checking. In the full version [2], we show that for other cases we do not see this.

Summary. Overall, we answered both the research questions we started with. Indeed there are several benchmarks with dependent variables, and using our pipeline does give performance benefits when no. of non-dependent variables is low. Our recipe would be to first run our poly-time check to see if there are dependents and use our approach if there are not too many non-dependents; otherwise switch to any existing method. To summarize our comparisons: wrt Strix, we found 252 benchmarks that had dependent variables in which DepSynt took less time than Strix. Out of which, in 126 benchmarks DepSynt took at least 1 second less than Strix. Among these, for 10 benchmarks (shift16, LightsTotal_d65ed84e, LightsTotal_9cbf2546, LightsTotal_06e9cad4, Lights2_f3987563, Lights2_0f5381e9, FelixSpecFixed3.core_b209ff21, Lights2_b02056d6, Lights2_06e9cad4, LightsTotal_2c5b09da) the time taken by DepSynt was at least 10 seconds less than that taken by Strix. These are the examples that are easier to solve by DepSynt than by Strix. For shift16, the difference was more than 1056 seconds in favor of DepSynt. Interestingly, shift16 also has all output variables dependent.

When comparing with Ltlsynt, we found 193 benchmarks that had dependent variables in which DepSynt took less time than Ltlsynt. Among these, in 27 benchmarks DepSynt took at least 1 second less than Ltlsynt. Of these, there is one benchmark (ModifiedLedMatrix5X) for which the time taken by DepSynt was at least 10 seconds less than that taken by Ltlsynt. Specifically, DepSynt took 5 seconds and Ltlsynt took 55 seconds.

7 Conclusion

In this work, we have introduced the notion of dependent variables in the context of reactive synthesis. We showed that dependent variables are prevalent in reactive synthesis benchmarks and suggested a synthesis approach that may utilize these dependency for better synthesis. As part of future work, we wish to explore heuristics for choosing "good" maximal subsets of dependent variables. We also wish to explore integration of our method in other reactive synthesis tools such as Strix.

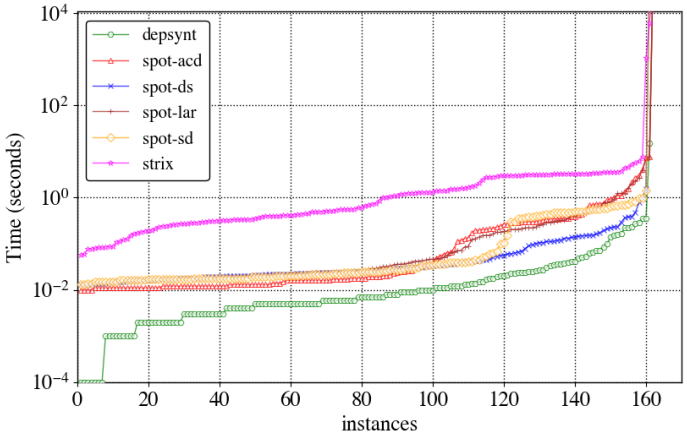

Fig. 4. Cactus plot comparing DepSynt, LtlSynt, and Strix on 162 benchmarks with at most 3 non-dependent variables.

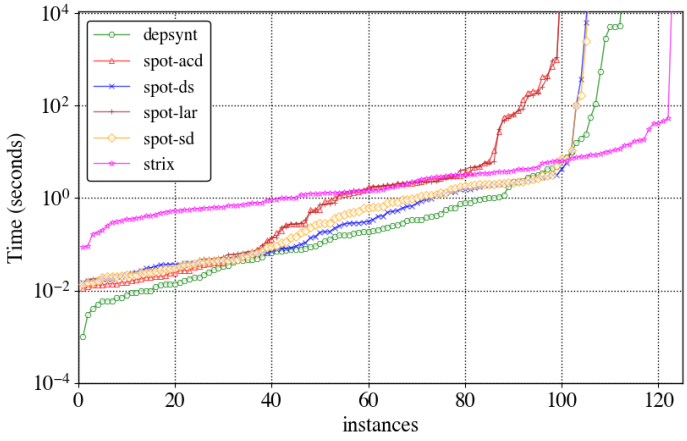

Fig. 5. Cactus plot comparing DepSynt, LtlSynt, and Strix on 138 benchmarks with more than 3 non-dependent variables.

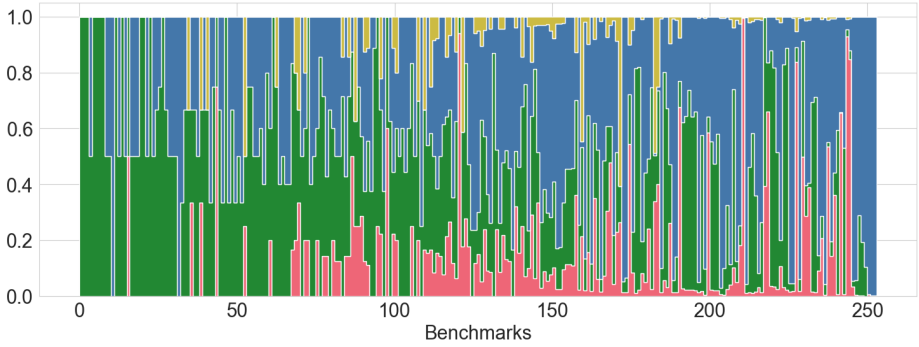

Fig. 6. Normalized time distribution of DepSynt sorted by total duration over benchmarks that could be solved successfully by DepSynt. Each color represents a different phase of DepSynt. Pink is searching for dependency, green is the NBA build, blue is synthesis of non-dependent variables and yellow is dependent variables synthesis.

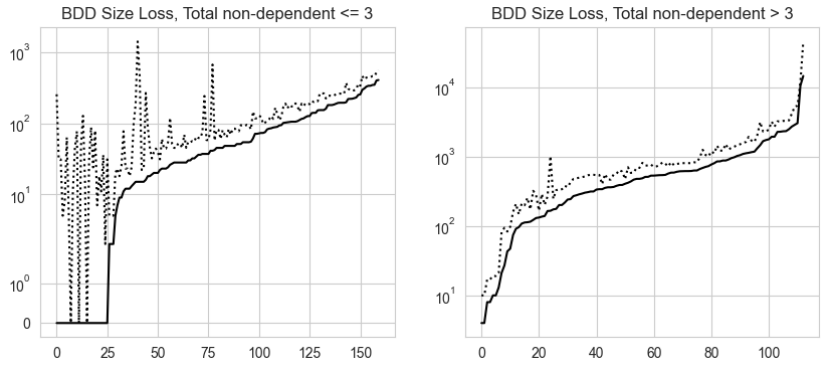

Fig. 7. This figure illustrates the total BDD sizes of the NBA edges before and after the projection of the dependent variables from the NBA edges, the left figure is over benchmarks with at most 3 non-dependent variables and the right figure is over benchmarks with 4 or more non-dependent variables. The solid line presents the projected BDD size and the dotted line presents the original BDD size. The y-axis is presented in symmetric log-scale. Benchmarks are sorted by the projected NBA's BDD total size.

References

1. Akshay, S., Arora, J., Chakraborty, S., Krishna, S.N., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 161–169. IEEE (2019)
2. Akshay, S., Basa, E., Chakraborty, S., Fried, D.: On dependent variables in reactive synthesis (full version). arXiv preprint arXiv:2401.11290 (2024)
3. Akshay, S., Chakraborty, S.: Synthesizing skolem functions: A view from theory and practice. In: Sarukkai, S., Chakraborty, M. (eds.) Handbook of Logical Thought in India, pp. 1–36. Springer (2022)
4. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about boolean functional synthesis? In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 251–269. Springer (2018)
5. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: Boolean functional synthesis: hardness and practical algorithms. Formal Methods Syst. Des. **57**(1), 53–86 (2021). https://doi.org/10.1007/s10703-020-00352-2, `https://doi.org/10.1007/s10703-020-00352-2`
6. Amram, G., Bansal, S., Fried, D., Tabajara, L.M., Vardi, M.Y., Weiss, G.: Adapting behaviors via reactive synthesis. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12759, pp. 870–893. Springer (2021)
7. Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV'20). Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (Jul 2020)
8. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 921–962. Springer (2018)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
10. Bryant, R.E.: Binary decision diagrams and beyond: Enabling technologies for formal verification. In: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD). pp. 236–243. IEEE (1995)
11. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. Formal Methods Syst. Des. **60**(2), 228–258 (2022)
12. Church, A.: Logic, arithmetic, and automata. In: International Congress of Mathematicians. p. 23–35 (1962)
13. Darwiche, A.: Decomposable negation normal form. J. ACM **48**(4), 608–647 (2001)
14. Faymonville, P., Finkbeiner, B., Tentrup, L.: Bosy: An experimentation framework for bounded synthesis. In: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. pp. 325–332. Springer (2017)
15. Finkbeiner, B., Geier, G., Passing, N.: Specification decomposition for reactive synthesis. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12673, pp. 113–130. Springer (2021)

16. Finkbeiner, B., Schewe, S.: Bounded synthesis. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 519–539 (2013)
17. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for boolean function synthesis. Computer Aided Verification **12225**, 611 – 633 (2020)
18. Golia, P., Slivovsky, F., Roy, S., Meel, K.S.: Engineering an efficient boolean functional synthesis engine. 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD) pp. 1–9 (2021)
19. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
20. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, USA (2004)
21. Jacobs, S., Perez, G.A., Abraham, R., Bruyere, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (syntcomp): 2018-2021 (2022)
22. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. pp. 578–586. Springer (2018)
23. Michaud, T., Colange, M.: Reactive synthesis from ltl specification with spot. In: Proceedings of the 7th Workshop on Synthesis, SYNT@ CAV (2018)
24. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 179–190 (1989)
25. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. pp. 375–392 (2016)
26. Shah, P., Bansal, A., Akshay, S., Chakraborty, S.: A normal form characterization for efficient boolean skolem function synthesis. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470741, `https://doi.org/10.1109/LICS52264.2021.9470741`
27. Soos, M., Meel, K.S.: Arjun: An efficient independent support computation technique and its applications to counting and sampling. In: ICCAD (Nov 2022)
28. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Automation of reasoning: 2: Classical papers on computational logic 1967–1970 pp. 466–483 (1983)
29. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation **115**(1), 1–37 (1994). https://doi.org/10.1006/inco.1994.1092 https://www.sciencedirect.com/science/article/pii/S0890540184710923