The Open University of Israel
Department of Mathematics and Computer Science

On Dependent Variables in Linear Temporal Logic and Reactive Synthesis

Thesis submitted as partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By
Eliyahu Basa

Prepared under the supervision of **Dr. Dror Fried**.
Full collaboration with **Prof. S. Akshay** and **Prof. Supratik Chakraborty**.

April 2024

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 research, we investigate the notion of dependent variables in the context of reactive synthesis. Inspired by successful pre-processing steps in Boolean functional synthesis, we define dependent variables as output variables that are uniquely assigned, given an assignment, to all other variables and the history so far. We describe three equivalent approaches for dependent variables: LTL formula, NBA, and the ω-language. Using this, we show that dependent variables are 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 when reactive synthesis exploits dependent variables, it can solve some problems beyond the reach of several existing techniques. Further, among benchmarks with dependent variables, if the number of non-dependent variables is low (≤ 3 by our experiments), our method outperforms all state-of-the-art tools for synthesis. This thesis contributes in terms of theoretical aspects and practical aspects. From theoretical aspects, we show new definitions, novel approaches, and algorithms. In practice, we developed a tool that implements the suggested algorithms and definitions, in some terms, this tool outperforms state-of-the-art tools. The research was published at TACAS 2024 conference.

Acknowledgements

I am grateful to Dr. Dror Fried from the Open University of Israel for his supervision, insight, and guidance. My gratitude also goes to Professor Supratik Chakraborty and Professor S. Akshay from IIT Bombay for their essential collaboration and insights in this research.

Contents

List of Figures

List of Tables

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. Ever since Church introduced the problem [14] in 1962, 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 [18]. 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, 21, 22, 31].

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 abound in reactive synthesis benchmarks, and can be effectively exploited to obtain synthesis techniques with orthogonal strengths vis-a-vis existing state-of-the-art techniques.

Our work is inspired by dependency in propositional logic. There, we are given a propositional formula F over a set of variables V. A set $X \subseteq V$ of variables is called dependent on another set of variables Y in F if there are no two satisfying assignments for F that are the same for the Y variables but different for the X variables. In other words, the assignment for Y uniquely determines the assignment for X in any satisfying assignment. 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 [38]. Identifying such uniquely defined variables efficiently can be very helpful for various problems, such as checking satisfiability, 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. This explains the interest in identifying (and "removing") such variables in the spec before solving the given problem. As such, in several problems in propositional logic, state-of-the-art tools try to identify dependent variables in a pre-processing step and process them separately, such as model counting [42], certified DQBF solving [33] or in Boolean functional synthesis [5, 21, 22, 31]. The remarkable practical success of Boolean functional synthesis tools such as Manthan [22] 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.

As a first step, we define dependency in an LTL formula: 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 $Y \subseteq I \cup (O \setminus X)$ 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. We say that X is *dependent in* φ if X is dependent on $Y = I \cup (O \setminus X)$. We focus on finding dependency in the output variables as our evaluation showed that there are no dependent input variables in current reactive synthesis benchmarks. An immediate consequence of this choice is that our definition of dependency is a history-related notion. That is, two distinct words in $L(\varphi)$ can have at any point the same assignment for Y, but different assignments for X dependent on Y as long as the histories of these words are different. 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 first 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 3 equivalent definitions of dependency over LTL formula φ, (1) over the ω-language of φ (2) Over the NBA of φ (3) Emptiness of customized LTL formula. Next, we describe an abstract framework for finding a maximal set of dependent variables in an LTL formula, that is inspired by similar techniques used in the context of Boolean functional synthesis (e.g. [5]). We then ex-

plore two concrete implementations of the abstract framework for dependency finding: a formula-based technique and an automata-based technique. In the formula-based approach, we construct a reactive LTL formula Ψ which is empty if and only if the variables X are dependent on the variables Y in the given formula φ. Our experiments convincingly show that the automata-based approach to detecting dependency scales much better than the formula-based one. As a result, we focus on the automata-based approach for the remainder of the paper. 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 such as Spot [8]). 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 was developed in C++ using APIs from the widely used library Spot for representing and manipulating non-deterministic Büchi automata. We avoid the complexity of explicitly constructing power sets of states while constructing the transducers for dependent variables. Instead, we use symbolic methods to directly construct circuits that implement the required transducers. which we combine with the AIGER circuit that describes T_Y. This constitutes our third technical contribution, and as our experiments ratify, constructing the transducers for dependent variables is among the fastest steps in the synthesis process using dependent variables.

We performed a comparative analysis of our tool with winning entries of the SYNTCOMP [24] competition to evaluate how knowledge of dependent variables helps reactive synthesis, we compare DepSynt with state-of-the-art synthesis tools: Ltlsynt (Spot's reactive synthesis tool) with 4 different configurations and Strix, the winning tool in SYNTCOMP23′ [24].

Our tool outperforms state-of-the-art and specifically highly optimized synthesis tools on benchmarks that have at least one dependent variable and at most 3 non-dependent variables. Moreover, the evaluation showed that DepSynt did manage to solve several benchmarks not solvable by any of the other tools and utilized dependency to some extent. To conclude, our research shows that dependent variables are prevalent in reactive synthesis benchmarks and that better utilization of these can help in solving reactive synthesis problems.

1.1 Thesis Flow

This thesis is composed of 8 chapters:

1. Chapter 1 is an introduction and overview of the thesis.
2. Chapter 2 is the background and preliminaries of the research field, we describe concepts such as ω-regular language, LTL, and reactive synthesis.
3. Chapter 3 reviews previous works related to this research.
4. Chapter 4 defines the dependency concepts over the linear temporal field and introduces three equivalent definitions.
5. Chapter 5 examines how dependency can be used in the reactive synthesis field where the specification is given in linear temporal logic formula.
6. Chapter 6 shows the dependency concept over a symbolic representation of the reactive synthesis problem. This step is required since we implement the symbolic form of the dependency algorithms.
7. Chapter 7 displays experimental results of the algorithms and definitions that were discussed in previous chapters.
8. Chapter 8 is the conclusion of the research, a high-level overview of the research, and additional future directions.

1.2 Thesis Contribution

This thesis has the following contributions.

- Introduction of the dependency concept in the field of reactive synthesis, inspired by the concept in the Boolean functional synthesis.
- Theoretical contribution - Three novel equivalent definitions of dependency, an algorithm to find a maximal set of dependent variables and a synthesis framework.
- Practical contribution - We published a tool called DepSynt, written in C++ using Spot [8] that implements the concepts and algorithms suggested in this research.
- Experimental results - We compare DepSynt with state-of-the-art tools and showed improvement in the synthesis process.
- TACAS 2024 - The research was published in TACAS 2024 conference [2]. The artifact DepSynt, got the *available*, *functional*, and *reusable* badges.

2 Background

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) was introduced in [29]. LTL is constructed with a finite set of propositional variables AP, using Boolean operators such as $\vee, \wedge$, and $\neg$, and temporal operators such as next X, until U, etc. The set AP induces an alphabet $\Sigma_{AP} = 2^{AP}$ of all possible assignments ($true/false$) to the variables of AP. The language of an LTL formula φ, denoted $L(\varphi)$ is the set of all infinite words on the alphabet Σ_{AP}. The semantics of the operators and satisfiability relation are defined as usual [23]. We denote the number of variables AP by $|AP|$, and the size of the formula φ, i.e., the number of its subformulas by $|\varphi|$. We sometimes abuse notation and identify the singleton set of a variable $\{z\}$ with the variable z.

We start by defining the syntax of LTL.

Definition 1 (Linear Temporal Logic Syntax) *Linear Temporal Logic (LTL) is built over a finite set of propositional variables AP, logical operators $\vee, \wedge, \neg$ and temporal operators X, U, G, R, W, F. The set of LTL formulas over AP is defined as:*

- *$\top, \bot$ are LTL formulas.*
- *if $p \in AP$ then p is a LTL formula.*
- *if α and β are LTL formulas then $\alpha \vee \beta, \alpha \wedge \beta, \neg\alpha, X\alpha, F\alpha, G\alpha, \alpha R\beta, \alpha U\beta, \alpha W\beta$ are LTL formulas.*

The temporal operators stand for:

- *G - Globally (Always)*
- *X - Next (Next time α holds)*

- *F - Future (Sometime in the future)*
- *U - Until (α need to holds least until β holds, β must to hold sometime in future)*
- *W - Weak Until (α need to holds least until β holds, β is not necessary hold in future)*
- *R - Release (β need to holds until and including the time α holds, if α never holds then β must hold all the time)*

We introduce additional notations over LTL formulas to define the semantics and usability in theorems and proofs.

ω-language and ω-word. Given a finite alphabet Σ, an infinite *word* $w \in \Sigma^\omega$ is a sequence $a_0 a_1 a_2 \cdots$ where for every i, the i^{th} letter of w is denoted by $a_i \in \Sigma$. w is also called an *ω-word*. The *prefix* $a_0 \cdots a_i$ of size $i+1$ of an ω-word w is denoted by $w[0,i]$. Note that $w[i,i] = a_i$, also denoted as $w[i]$. We denote $w[0,-1]$ to be the empty word. We denote by $w^i = a_i a_{i+1}...$ the *i-suffix* of w. 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. When the alphabet Σ is combined from two distinct alphabets, $\Sigma = \Sigma_X \times \Sigma_Y$ for some sets of variables X, Y, we notate for a letter $a = (a^1, a^2) \in \Sigma$, denote by $a.X$ the projection of a on Σ_X, that is, the letter $a^1 \in \Sigma_X$. Similarly, $a.Y$ denotes the projection of a on Σ_Y, that is the letter $a^2 \in \Sigma_Y$. For letters $a^1 \in \Sigma_X$ and $a^2 \in \Sigma_Y$ we denote by $a^1 \oplus a^2$ the letter (a^1, a^2) in Σ. For words, we define $w.X$ to be ω-word obtained from a word $w \in \Sigma$ on Σ_X i.e. $w.X = a_1.X a_2.X a_3.X \cdots$. For words $w^1 \in \Sigma_X$ and $w^2 \in \Sigma_Y$ we denote by $w^1 \oplus w^2$ the word w in which for every i, $w_i = w^1_i \oplus w^2_i$.

Now we can define LTL semantically, when an ω-word satisfies an LTL formula.

Definition 2 (Semantic of LTL) *The satisfaction relation ($\models$) between a LTL formula over AP and ω-word over $\Sigma = 2^{AP}$ is inductively defined:*

$w \models \top$
$w \not\models \bot$
$w \models a$ *if and only if* $a \in w[0]$
$w \models \neg a$ *if and only if* $a \notin w[0]$
$w \models \varphi \wedge \psi$ *if and only if* $w \models \varphi$ *and* $w \models \psi$
$w \models \varphi \vee \psi$ *if and only if* $w \models \varphi$ *or* $w \models \psi$
$w \models X\varphi$ *if and only if* $w^1 \models \varphi$
$w \models F\varphi$ *if and only if* $\exists i \in \mathbb{N}$: $w^i \models \varphi$
$w \models G\varphi$ *if and only if* $\forall i \in \mathbb{N}$: $w^i \models \varphi$
$w \models \varphi U \psi$ *if and only if* $\exists k : w^k \models \psi$ *and* $\forall j < k : w^j \models \varphi$
$w \models \varphi W \psi$ *if and only if* $\forall k : w^k \models \varphi$ *or* $\exists k : w^k \models \psi$ *and* $\forall j < k : w^j \models \varphi$
$w \models \varphi R \psi$ *if and only if* $\forall k : w^k \models \psi$ *or* $(\exists k : w^k \models \varphi$ *and* $\forall j \leq k : w^j \models \psi)$

LTL formulas induce an ω-language.

Definition 3 (ω-language of LTL formula) *The ω-language of the LTL formula φ is denoted as $\mathbb{L}(\varphi)$ and defined to be the set of all the ω-words satisfying φ.*

$$\mathbb{L}(\varphi) = \{w \in \Sigma^\omega \quad | \quad w \models \varphi\}$$

For the definitions and approaches that will be introduced in this paper, we need to compare multiple Boolean variables, therefore we define the equal $\equiv$ operator.

Definition 4 (Equal Operator) *Given the sets of Boolean variables A, B where $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$. We define the binary operators $=, \neq$ as the following:*

$$(A = B) \equiv (a_1 \leftrightarrow b_1) \wedge ... \wedge (a_n \leftrightarrow b_n)$$
$$(A \neq B) \equiv \neg(A = B)$$

A common practical tool to work with LTL formulas is Nondeterministic Büchi Automata.

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. 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 then

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* of the run ρ is the sequence of labels seen along ρ, i.e., $w = \sigma_{i_0}\sigma_{i_1}\cdots$ where for every $j > 0$, $q_{i_{j+1}} \in \delta(q_{i_j}, \sigma_{i_j})$. As A is nondeterministic a word can have many runs, although every run has a single word. 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. Without loss of generality, we assume that all the edges and all the states exist at a single run at least. Finally, every LTL formula φ can be transformed in exponential time in the length of φ, to an NBA A_φ for which $L(\varphi) = L(A_\varphi)$ [23, 40]. 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., its number of states and transitions.

In this research we use the Rabin-Scott Theorem [32] to construct a strategy for dependent variables, originally, this method converts a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA).

Rabin-Scott Theorem [32] Rabin–Scott powerset construction is a method to convert a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) in a term that both have the same language. Formally, given the NDA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ we want to construct the DFA $\mathcal{D} = (Q', \Sigma, \delta', q_0', F')$ such that $\mathbb{L}(\mathcal{D}) = \mathbb{L}(\mathcal{A})$. The method defines the states of $\mathcal{D}$ to be powerset of $\mathcal{A}$, i.e., $Q' = 2^Q$ and the init state is $q_0' = \{q_0\}$. The transition in the DFA from state $S' \subseteq Q$ for the input $\sigma \in \Sigma$ is all the states in Q that are reachable from any state in S with the input $\sigma \in I$ in $\mathcal{A}$, formally, it's defined as: $\delta'(S', \sigma) = \cup\{\delta(q, \sigma) | q \in S'\}$. In DFA the state $S' \subseteq Q$ is accepting, i.e., $S' \in F'$ if one the states in S' is accepting in $\mathcal{A}$, i.e., $F' = \{S' | S' \subseteq Q, S' \cap F \neq \emptyset\}$. Figure 2.1 is an example of an NFA that accepts all strings ending with 1 and example 2.2 is a DFA that was constructed via the Rabin-Scott method from example 2.1.

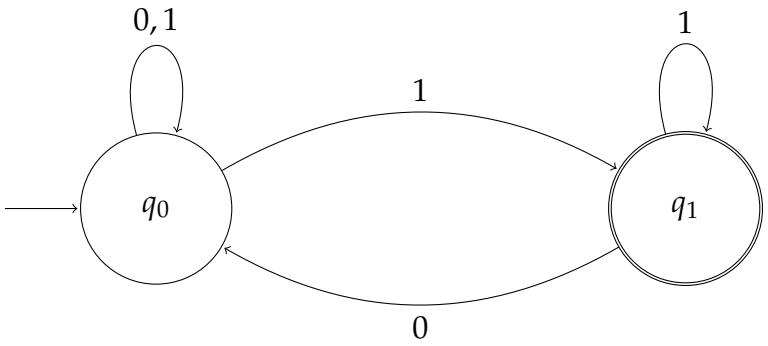


Figure 2.1: Example of non-deterministic finite automaton that accepts all the strings that end with 1.

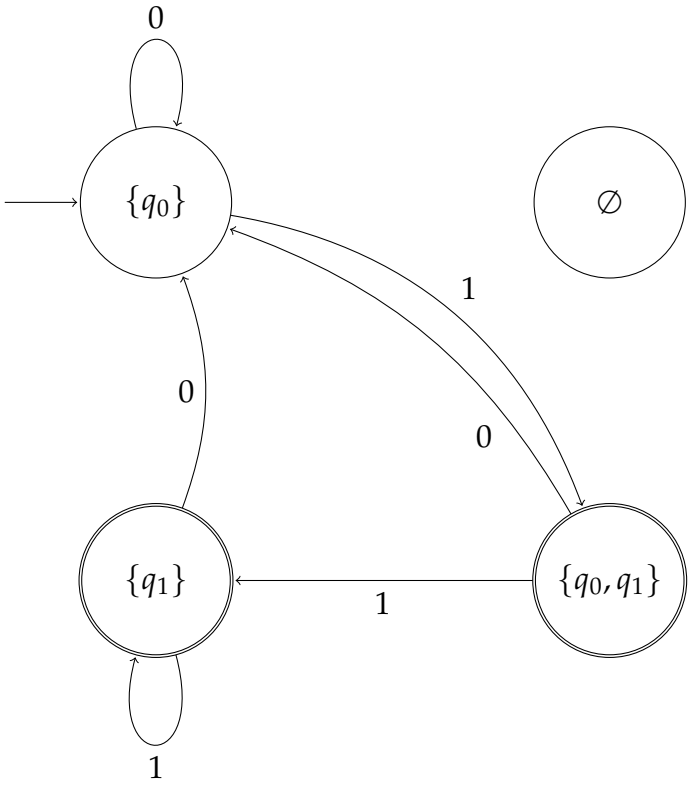


Figure 2.2: Deterministic finite automaton constructed with Rabin-Scott method on the NFA from example 2.2

2.2 Synthesis

Reactive Synthesis [30] 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*, $f : \Sigma_I^* \rightarrow \Sigma_O$ such that every word $w \in (\Sigma_I \times \Sigma_O)^\omega$ obtained by using this strategy 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 often omit the prefix reactive while referring to them. The synthesized strategy $f : \Sigma_I^* \rightarrow \Sigma_O$ that is given as output is typically described (explicitly or symbolically) in the form of 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, $\delta : S \times \Sigma_I \rightarrow S$ is a deterministic transition function, and $\lambda : S \times \Sigma_I \rightarrow \Sigma_O$ is the output function. A standard procedure in solving reactive synthesis is to transform the given LTL formula φ to an NBA A_φ for which $L(A_\varphi) = L(\varphi)$. Then transform A_φ to a Deterministic Parity Automata (DPA) which turns into a parity game, whose solution is described as a transducer T_{A_φ}. Many methods exist to solve the parity game [37] such as Zielonka's recursive algorithm [28].

Boolean function synthesis and dependency [5] In Boolean functional synthesis, the notion of dependent variables proved to be extremely powerful. Intuitively, a set of Boolean variables X is dependent on a set of variables Y in a Boolean formula F if for every satisfying assignment to F, the value of X is uniquely determined by the assignments for Y. Dependency in the Boolean formula has various applications, such as Boolean Functional synthesis and model counting. Specifically, in Boolean Functional Synthesis, there are tools [5] that find dependent variables and extract them as a pre-processing step, to have them later be uniquely assigned. This thesis proposes a possible elevation of the variable dependency concept from propositional logic and Boolean functional synthesis to temporal logic and reactive synthesis. To formally define dependencies within Boolean formulas, the projection operator must be defined.

Definition 5 (Projection variables over assignment) *Let V be a set of Boolean variables, X is a subset of V and $\sigma \in \Sigma_V$ is an assignment over V, the projection operator of σ on the variables X which is denoted as $\sigma \downarrow X$ is the assignment of all variables X in σ. For example, let $V = \{p, q, s, t\}$, $X = \{p, q\}$ and*

$\sigma = (p = \top, q = \bot, s = \top, t = \bot)$ *then* $\sigma \downarrow X = (p = \top, q = \bot)$.

We next define dependency in Boolean formulas.

Definition 6 (Variable dependency in Boolean Formula) *Given Boolean formula* $F(\hat{X}, \hat{Y}, c)$ *where* $\hat{X}, \hat{Y}$ *are sets of Boolean variables and* c *is a Boolean variable. The variable* c *is dependent on set* X *if for every assignments* σ, σ' *such that* $F(\sigma) = F(\sigma') = \top$ *then it holds that if* $\sigma \downarrow \hat{X} = \sigma \downarrow \hat{X}'$ *then* $\sigma \downarrow c = \sigma \downarrow c$.

As discussed on reactive synthesis, the synthesis process produces an transducer. A common way to represent transducer is with Mealy machine.

Mealy machine [25]. In our synthesis process, we describe our strategies as Mealy machines, which we also refer to as *transducers*. A Mealy machine, first introduced in [25], is a finite-state machine whose output values are determined both by its current state and the current inputs. In the field of reactive synthesis, the goal is to generate a mealy machine that satisfies the specification.

Formally, we describe a Meally machine (transducer) as a 6-tuple $T = (\Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda)$, where Σ_I is the input alphabet, Σ_O is the output alphabet, Q is the set of states of T, with $q_0 =$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is the state transition function, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is the output function. An example of a Mealy machine can be found on Figure 2.3.

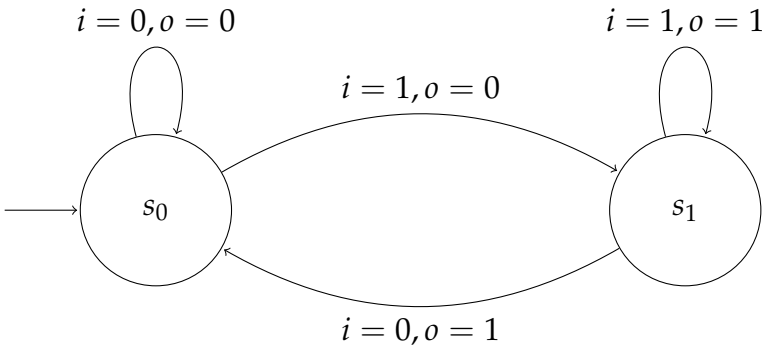


Figure 2.3: Example of a Mealy machine for the specification $\neg o \wedge G(o \leftrightarrow Xi)$, where $\Sigma_I = \{i\}, \Sigma_O = \{o\}, Q = \{s_0, s_1\}$, the state transition is the edges between the states combined with the input assignment, the output function is the assignments of the output variable on the edges and the init state is s_0 .

2.3 Symbolic Representations

In our work we symbolically describe Mealy machines in a format called AIGER that relies on a structure called AIG. We describe these below.

Definition 7 (And-Inverter Graphs) *And-Inverter Graphs (AIGs) are directed acyclic graphs representing the structural implementation of a circuit's logical functionality, initially introduced in Turing's work [39]. Figure 2.4 is an example of an And-Inverter Graph. An And-Inverter graph (AIG) over the variables* $X = \{x_1, ..., x_n\}$ *is directed acyclic graph* $G = (V, E)$ *where* $V = V_X \cup V_g$ *with the following properties:*

- *Each node* $v \in V_X$ *is labeled by* $x_i \in X$ *and has no outgoing edges, i.e.,* v .
- *Each non-leaf node* $v \in V_g$ *represents a Boolean conjunction (AND) of the functions represented by the two incoming edges.*
- *An edge* $e \in E$ *can optionally contain markers indicating logical negation.*

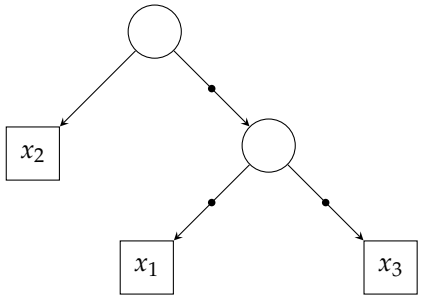


Figure 2.4: AIG example of the formula $x_2 \wedge \neg(\neg x_1 \wedge \neg x_3) \equiv x_2 \wedge (x_1 \vee x_3)$

Using the definition of AIG, we can now define the AIGER format with which we will make use in the symbolic representation of our synthesis.

Definition 8 (AIGER) *AIGER [7] is a circuit logic that is represented with the tuple* $(I, O, L, \lambda, \delta)$ *where* I *are the boolean input variables,* O *are the boolean output variables and* L *are the latches. Latches are flip-flops that are used to store information within the circuit logic. Flip-flops are digital logic circuits that store binary information and can change their value. For every output variable* $o \in O$*, there is an AIG* $\lambda^o : I \cup L \rightarrow \{0, 1\}$ *that uses the input and latches to determine the value of* o*, i.e.,* λ *is a vector of Boolean functions* $\lambda = (\lambda^{o_1}, ..., \lambda^{o_{|O|}})$*. For*

every latch $l \in L$ there is an AIG $\delta^l : I \cup L \to \{0,1\}$ that uses the input and latches to determine the next value of the latch l, i.e., δ is a vector of Boolean functions $\delta = (\delta^{l_1}, ..., \delta^{l_{|L|}})$.

Representing Mealy machine as AIGER Mealy machine can be represented as an AIGER. Assume we have the Mealy machine $M = (\Sigma_I, \Sigma_O, Q, q_0, \delta_M, \lambda_M)$ and we would like to get the corresponding AIGER $A = (I, O, L, \lambda_A, \delta_A)$ that induces the same strategy. We show a method that is used in this paper, although additional methods exist as well. The input and output would be the same in the Mealy machine and AIGER, for every state Q in the Mealy machine, we create a corresponding latch, for example, if we have the states $Q = \{s_1, s_2, s_3\}$ we would have the latches $L = \{l_1, l_2, l_3\}$ where the corresponding latch of s_i is l_i. For the state-transition function $\delta_M : Q \times \Sigma_I \to Q$ and for every $q \in Q$ we can create a Boolean function $\delta_{M,q} : Q \times \Sigma_I \to \mathbb{B}$ that takes $Q \times \Sigma_I$ and returns true if and only if q suppose to be the next state in the Mealy machine, i.e., $\delta_{M,q}(q', i) = \top$ if and only if $q = \delta_M(q', i)$. The function $\delta_{M,q}$ can be represented as an AIG [39]. Therefore, we define the AIGER transition vector as $\delta_A = (\delta_{M,q_1}, ..., \delta_{M,|Q|})$. In the same way, we define the AIGER output vector λ_A, we define for every $o \in O$ the function $\lambda_{M,o} : Q \times \Sigma_I \to \mathbb{B}$ such that $\lambda_{M,o}$ is true if and only if the variable o is assigned true in the Mealy machine, i.e., $o \in \lambda_M(q, i)$ if and only if $\lambda_{M,o}(q, i) = \top$. We define the AIGER output vector as $\lambda_A = (\lambda_{M,o_1}, ..., \lambda_{M,o_{|O|}})$.

In figure 2.5 we can see the corresponding AIGER $A = (I, O, L, \lambda_A, \delta_A)$ of the Mealy machine in figure 2.3, in both the AIGER and the Mealy machine we have the input $I = \{i\}$ and output $O = \{o\}$. In the Mealy machine, we have the states $Q = \{s0, s_1\}$ therefore the AIGER would have 2 latches $L = \{s_0, s_1\}$ corresponding to the states in the Mealy machine. 3The Mealy machine transition function induces the functions $\delta_{s_0} : \{s_0, s_1\} \times \{i\} \to \mathbb{B}$ which returns true if and only if s_0 is the next state and the same for δ_{s_1}. We have a single output function $\lambda_o : \{s_0, s_1\} \times \{i\} \to \mathbb{B}$ that returns true if and only if in the Mealy machine, o is assigned to true.

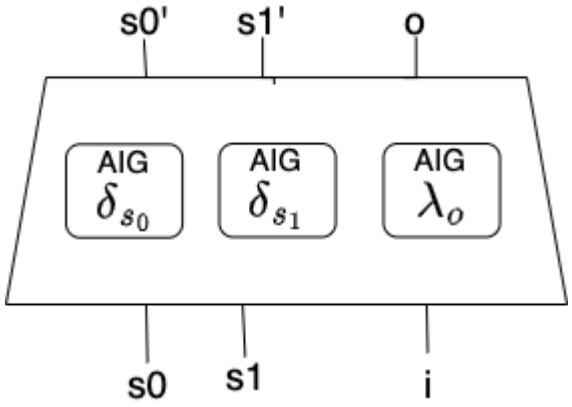


Figure 2.5: Illustration of the AIGER corresponds to the Mealy machine in figure 2.3.

In many practical ω-regular frameworks, such as Spot [8], the Binary Decision Diagram is a common way to represent symbolically edges between states, it's discussed widely in section 6. Here we introduce the Binary Decision Diagram.

Binary Decision Diagram Binary Decision Diagram (BDD) is a data structure that is used to represent Boolean functions efficiently. BDD is a rooted, directed, and acyclic graph that includes two terminal nodes to represent the constants 0/False and 1/True. Each node (except the terminal node) has a corresponding Boolean variable and two child nodes: high child and low child. The high child represents the case where the variable in the node is assigned to 1, respectively, and the low child represents the case where the variable is assigned to 0. ROBDD is a BDD where the low and high successors of every node are distinct (redundancy) and no two distinct nodes are testing the same variable with the same successors (uniqueness). Figure 2.6 is a graph illustration of an ROBDD. A lot of information and problems can be solved given a ROBDD in a polynomial time over the ROBDD, such as Model counting, SAT, etc.

2.4 Experiments background

As part of the research, we introduce a reactive synthesis tool and compare it with 2 state-of-the-art tools:

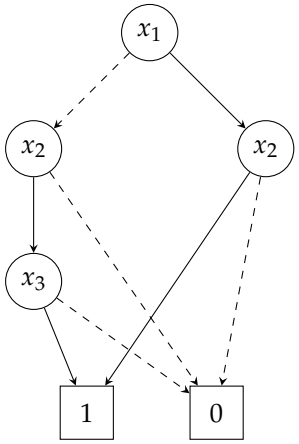


Figure 2.6: ROBBD example of the function $x_2 \wedge (x_1 \vee x_3)$ with the variable order: x_1, x_2, x_3.

1. **Spot [8]** is a state-of-the-art platform for LTL, ω-automata manipulation and model checking. Many algorithms are implemented in the Spot library, such as: Converting LTL to NBA, Converting NBA to deterministic parity automaton, solving parity games, representing Mealy machines, and working with AIG. The implementation of the tools that were created for this thesis is mainly based on Spot. Spot has a tool called `ltlsynt` for reactive synthesis problems.

2. **Strix [26]** is a tool for reactive LTL synthesis combining a direct translation of LTL formulas into deterministic parity automata (DPA) and an efficient, multi-threaded explicit state solver for parity games. In brief, Strix (1) decomposes the given formula into simpler formulas, (2) translates these on-the-fly into DPAs based on the queries of the parity game solver, (3) converts the DPAs into a parity game, and at the same time already solves the intermediate games using strategy iteration, and (4) finally translates the winning strategy, if it exists, into a Mealy machine or an AIGER circuit with optional minimization.

We use benchmarks from The Reactive Synthesis Competition (SYNTCOMP) [24] to search for LTL-dependent variables and compare the reactive synthesis tool we introduce in this research. SYNTCOMP is a competition for reactive synthesis tools to search for LTL dependency and compare our reactive synthesis tool. The SYNTCOMP has a variety types of benchmarks for different reactive synthesis problems, (1) Parity game solving. (2) LTL

synthesis problem. (3) LTLf synthesis problem. This thesis focuses on the LTL synthesis problem. In the years, 2018-2023 (the year of writing this thesis) the tool Strix [26] wins the competition and `ltlsynt` gets to the second place.

3 Previous Work

Reactive Synthesis has been an extremely active research area for the last several decades (see e.g. [10, 14, 18, 19, 30]). Not only is the theoretical investigation of the problem rich but there are also several tools that are available to solve synthesis problems in practice. These include solutions like `ltlsynt` [27] based on Spot [8], Strix [26] and BoSY [17]. 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 the converted NBA to a deterministic parity automaton (DPA) (see [9] for an overview of the challenges of reactive synthesis).

Reactive synthesis specifications are usually represented in Linear Temporal Logic (LTL) formulas. LTL was introduced in ”The temporal logic of programs” by Amir Pnueli [29] and allows to specify temporal properties on the logic, the time in LTL is represented as an infinity linear sequence of discrete moments. The LTL extends the propositional logic by adding temporal operators that express properties over the infinity-length sequence of the time: ”until”, ”globally” ”eventually”, ”and always”, as defined in definition 1.

The approach for synthesizing the reactive system in this paper begins by converting the LTL formula into a non-deterministic Büchi automaton (NBA). The non-deterministic Büchi automaton was introduced in ”On a Decision Method in Restricted Second Order Arithmetic” by Büchi [12] to serve as a model for defining the behavior of systems with infinite-length execution. To convert the LTL formula to NBA we use the algorithm that we call ”Vardi-Wolper” which was introduced in the paper [41].

To synthesize the non-deterministic Büchi automaton (NBA) we convert it to deterministic parity automata (DPA) [43], in which the acceptance conditions are expressed as parity conditions. The DPA provides a deterministic framework for the synthesis process, where the system’s behav-

ior is fully specified by the state transitions and parity conditions. Solving the parity game involves finding a winning strategy of a two-player game, where one player represents the environment and the other represents the system. In this game, the goal is to ensure that for every action of the environment which symbolizes an assignment to input variables of the specification, the follow-action of the system, which symbolizes an assignment to output variables of the specification leads to the specification being satisfied. The strategy of the system for the environment's actions can be represented as a Mealy machine. If no such Mealy machine exists, we say the specification is unrealizable. The algorithm to construct DPA from NBA is called Safraless [34].

Given a deterministic parity automaton, which represents a game of the required specification to synthesize, we can use "Zielonka's Recursive Algorithm" which is an EXP-time recursive divide-and-conquer approach to solve parity games, the algorithm is presented in [43].

Finally, our work may be viewed as lifting the idea of uniquely defined variables in Boolean functional synthesis to the context of reactive synthesis. Dependency in Boolean functional synthesis is discussed widely in [4]. The dependency concept in Boolean functional synthesis significantly impacts the synthesis process's performance. Finding dependent variables reduces the problem's complexity since every output dependent variable can be synthesized with good performance as described in [4].

4 Dependent variables in Linear Temporal Logic

In this section, we define dependent variables for (reactive) LTL formulas and suggest a framework for finding a maximal set of dependent variables. Then we provide two characterizations of dependent variables: using LTL formula, and using NBA, which allow us to have efficient procedures for identifying such variables. Our research starts with the formula approach, this approach indicated that there are many reactive synthesis benchmarks in SYNTCOMP [24] with dependency but could not complete on many benchmarks and was not efficient enough in terms of time and memory. Those issues led us to search for another approach, the automaton approach, for identifying dependent variables that could be scaled efficiently. As discussed in chapter 7, we show that the automaton approach in practice scales better and outperforms the formula approach. Yet, the formula approach is a naive and intuitive approach, this approach can be researched further.

4.1 Defining dependent variables

Our notion of dependent variables for LTL formulas, specifically suited to reactive synthesis, asks that the dependency be maintained at every step of the word satisfying the formula. While there are several notions of dependency that can be considered, we describe the one that we use throughout the paper. As mentioned earlier, our definition of dependency is restricted to output variables, since having dependent input variables would imply that not all input values are possible for these variables, which makes the formula unrealizable. While this can be used for a quick way to detect unrealizability, empirically, we did not find a single benchmark among those we encountered with dependent input variables. Nevertheless, all the def-

initions in this section are suitable for finding dependent variables that are also input variables just as well.

Definition 9 (Variable Dependency in LTL ω-language) *Let φ be a reactive LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \setminus I$. Let X, Y be 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 $i \geq 1$ 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 still have different values for X for the same values for Y, and X can still be defined dependent on Y. In that sense, our definition is much more permissive than defining, e.g., X to be dependent on Y if the value of Y determines the value of X for all words in $L(\varphi)$ and in all time steps.

As an example, consider an LTL formula φ with an input variable y, an output variable x and a language $L = \{w^1, w^2, w^3\}$ 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 thus the dependency of x is not violated, although $w^1_2.y = w^3_2.y$ and $w^1_2.x \neq w^3_2.x$. Observe that, if X is dependent on Y in φ for some Y, then it is also dependent in φ. We next show how to find a maximal set of dependent variables.

4.2 Finding dependent variables

Definition 10 (Maximal dependent set) *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 that strictly contains X is not dependent in φ.*

Finding a maximal dependent set is desirable since we are interested in exploiting as many dependent variables as possible. Note, however, that as in the propositional logic case, finding an independent set of variables with *maximum* size, is an intractable task [36]. Therefore, propositional logic deploys various heuristics for finding dependent variables of a large size. In this work, we extend this to the temporal setting.

As such, Algorithm 1 called **FindDependent** for finding a maximal dependent set works as follows. **FindDependent** gets as input an LTL formula

φ over the set of variables $V = I \cup O$, and initialize the set X that includes the dependent variables found so far to be empty (line 3). Then at every step (line 4) we choose a variable $z \in O$ that was not tested before and check if z is dependent of the remaining variables, excluding those that were already found dependent (line 5). This is done through a procedure called isDependent() which we discuss next. If so then x is added to X (line 6). At the end of the process, we have that X is returned as an output of **FindDependent**.

The heart of the framework is naturally in the procedure **isDependent** that takes a variable z and a set Y and returns true if and only if $\{z\}$ is dependent on Y in φ according to definition 9. The implementation of this procedure varies, and in the next sections, we discuss two possible implementations at length: formula-based and automata-based. A second point to notice is the order in which the output variables are chosen. Currently, we use the most standard order of the variables which is an order of appearance. Note however that the order may play a significant role, since a different order may produce a maximal independent set of a different size. We leave this problem of choosing a more efficient order heuristic for future work.

Algorithm 1: Find a maximal set of dependent variables

```
1 Input: LTL Formula φ with variables V = I ∪ O
2 Output: A set X ⊆ O which is maximal dependent in φ
3 X ← ∅;
4 for z ∈ O do
5     if isDependent(z, V\(X ∪ {z})) then
6         X ← X ∪ {z};
7 return X;
```

In order to prove that Algorithm 1 finds a maximal set of dependent variables, i.e. Theorem 1, we use the following claim:

Claim 1 *Let X' be a set of output variables and $X \subseteq X'$. If X' depends on $V \setminus X'$ φ, then so is X.*

Proof: Set $w, w' \in L(\varphi)$ with the same prefix $w[0, i-1] = w'[0, i-1]$ for some $i \geq 1$. Assume $w_i.(V \backslash X) = w'_i.(V \backslash X)$. Then since $V \backslash X' \subseteq V \backslash X$ we have $w_i.V \backslash X' = w'_i.V \backslash X'$. Therefore $w'_i.X' = w_i.X'$, which means that $w'_i.X = w_i.X$ as well. ■

Theorem 1 *Given an LTL formula φ over the set of variables V, **FindDependent** returns a maximal dependent set in φ.*

Proof: Assume that *isDependent* is well defined. That is, for a variable z and a set Y *isDependent* returns *true* if and only if $\{z\}$ is dependent on Y in φ. Denote by X^i the set X in the i'th step of the loop obtained in line 6. We show by induction that X^i is dependent on $V\backslash X^i$ in φ. For step 0, we have that the empty set is naturally dependent on V. Assume by induction that X^{i-1} is dependent on $V\backslash X^{i-1}$. Then at step i, if no element was added to X^{i-1} then $X^i = X^{i-1}$ and we are done. Otherwise, we have $X^i = X^{i-1} \cup \{z\}$. Then let $j \geq 0$ be such that for two words w, w' in $L(\varphi)$, $w[0, j-1] = w'[0, j-1]$ and $w_j.V\backslash X^i = w'_j.V\backslash X^i$. Then since z was just added, it follows that $w_j.\{z\} = w'_j.\{z\}$. But then since $X^i = X^{i-1} \cup \{z\}$ it follows that $w_j.V\backslash X^{i-1} = w'_j.V\backslash X^{i-1}$ and by induction $w_j.X^{i-1} = w'_j.X^{i-1}$. hence in total $w_j.X^i = w'_j.X^i$. Note that it also follows that at the end of the algorithm, X is dependent on $V\backslash X$ in φ and hence X is dependent in φ.

To see that X is maximal, assume for contradiction that it is not and let $X' \subseteq O$ be a dependent set that strictly contains X. Let $z \in X'\backslash X$. Let i be the step in the algorithm in which z was considered and ruled-out (otherwise z would have been a member of X). Then there exist two words w, w' in $L(\varphi)$ and $j \geq 0$, for which $w[0, j-1] = w'[0, j-1]$, $w_j.V\backslash X^i = w'_j.V\backslash X^i$ and $w_j.\{z\} \neq w'_j.\{z\}$. On the other hand since $X^i \subseteq X \subset X'$ we have that $V\backslash X' \subseteq V\backslash X^i$. So that means that also $w_j.V\backslash X' = w'_j.V\backslash X'$. Then, since $w[0, j-1] = w'[0, j-1]$ it follows that $w_j.X' = w_j.X'$ and specifically $w_j.\{z\} = w'_j.\{z\}$, a contradiction.

■

4.3 Dependent variables by formula

Towards a practical approach for finding dependent variables, we provide a formula characterization for dependency.

Definition 11 (Variable formula dependency) *Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V\backslash I$. Let X, Y be sets of variables where $X \subseteq O$. Let V' be a copy of the variables V. Then we say that X is formula dependent on Y in φ if $L(\Psi) = \emptyset$, where*

$$\Psi \equiv \varphi(V) \wedge \varphi(V') \wedge ((V = V')U(Y = Y' \wedge X \neq X'))$$

We say that X is *formula dependent* in φ if $\mathcal{L}(\Psi) = \emptyset$ and $Y = V \backslash X$ (i.e. $A = \emptyset$).

The following theorem shows that formula dependency captures our notion of dependency.

Theorem 2 *Let φ be an LTL formula over variables V. Then X is dependent on Y in φ if and only if X is formula dependent on Y in φ.*

Proof: Let X be formula dependent in Y. That means that the following formula is false.

$$\Psi \equiv \varphi(V) \wedge \varphi(V') \wedge ((V = V')U(Y = Y' \wedge X \neq X'))$$

Which means that for every two words, $w, w' \in L(\varphi)$, we have that

$$(w, w') \models \neg(((V = V')U(Y = Y' \wedge X \neq X')))$$

Then suppose that for some $i \geq 0$ we have $w[0, i-1] = w'[0, i-1]$ and $w_i.Y = w'_i.Y$. Then it must be that $w_i.X = w'_i.X$ as otherwise w, w' satisfy Ψ.

Next, Assume that X is dependent on Y, and let w, w' be two words in $L(\varphi)$. Then for every $i > 0$, if $w[0, i-1] = w'[0, i-1]$ and $w_i.Y = w'_i.Y$, then $w_i.X = w'_i.X$. Therefore we have that:

$$(w, w') \models \neg(\varphi(V) \wedge \varphi(V') \wedge ((V = V')U(Y = Y' \wedge X \neq X')))$$

Then, since $w \models \varphi$, and $w' \models \varphi$, we have that Ψ is empty:

$$\Psi \equiv \varphi(V) \wedge \varphi(V') \wedge ((V = V')U(Y = Y' \wedge X \neq X'))$$

■

We use the formula Ψ in Algorithm 1 by implementing **isDependent** as the procedure **isFormulaDependent(**$z, V \backslash (X \cup \{z\})$**)**: Given a variable z with the set X and the whole set of variables V, define variables $V_\Psi = V$, in which $X_\Psi = \{z\}$ is a subset of V_Ψ of the candidate dependent variable, and variables $Y_\Psi = Y'_\Psi = V \backslash (X \cup \{z\})$ which are also a subset of V_Ψ. Let V'_Ψ be a copy of V_Ψ. Then return *True* if and only if $\Psi(V_\Psi, V'_\Psi)$ is empty. Note that verifying whether Ψ is empty can be done by standard means, e.g. [23, 40].

We call the resulting algorithm that is obtained from Algorithm 1 implemented using **isFormulaDependent** as **FindFormulaDependency**. We then have the following.

Corollary 1 *Algorithm* ***FindFormulaDependency*** *returns a maximal set dependent in φ.*

Implementing variables dependency based formula Given an LTL formula $\varphi(V)$, we would like to know if the variables $X \subset V$ are dependent on $Y \subset V$. The construction of the LTL formula in Definition 11 requires to duplicate every variable in the formula to achieve the corresponding formula with new variables $\varphi(V')$ and construct the formula $((V = V')U(Y = Y' \wedge X \neq X'))$ as required in the definition. We get the formula Ψ, as in Definition 11 by applying conjunction of all the 3 formulas: $\varphi(V) \wedge \varphi(V') \wedge ((V = V')U(Y = Y' \wedge X \neq X'))$. To check the emptiness of Ψ, we construct the NBA N_Ψ from Ψ and check if the language of N_Ψ is empty by using the standard emptiness algorithm for Buchi automata [15]. From the proof of Theorem 2, we get that X is dependent on Y in φ if and only if Ψ is false (hence N_Ψ is empty).

In section 7 we display the experimental results of the described approach.

4.4 Dependent variables by automaton

Using the *FindFormulaDependency* algorithm to verify if a variable is dependent on a set of variables, requires repeated calls to emptiness checking of LTL formula and does not scale in practice (see chapter 7). Instead, we suggest another approach to verify if a variable is dependent on a set of variables based on the nondeterministic Büchi automaton of the original LTL formula. Our framework uses the notion of compatible pairs of states of the automaton defined as follows.

Definition 12 *Let $A = (\Sigma, Q, \delta, q_0, F)$ be an NBA . The pair $(s, s') \in Q \times Q$ is* compatible *in A if there is any run from q_0 to s and from q_0 to s' with 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 13 *Let φ be an LTL formula over V with input variables $I \subseteq V$ and output variables $O = V \backslash I$. Let X, Y be 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 $Y = V \backslash X$.

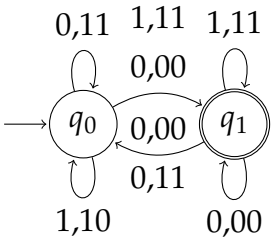


Figure 4.1: An Example NBA

As an example, consider NBA A_1 from adjoining Figure 4.1, constructed from some LTL formula with input $I = \{i\}$ and outputs $O = \{o_1, o_2\}$. Here $\Sigma_I = \{0,1\}, \Sigma_O = \{0,1\}^2$ and edges are labeled by values of $(i, o_1 o_2)$. It is immediate that, $(q_0, q_0), (q_1, q_1)$ are compatible pairs but so are $(q_0, q_1), (q_1, q_0)$ since they can both be reached from the initial state on reading the length 2 word $(0,00)(0,00)$. 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 we can see that 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 shows us the relation between automata dependency and dependency in LTL as defined earlier.

Theorem 3 *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.*

Proof: Assume that X is dependent on Y in φ and let s, s' be two compatible states in A with a joint word prefix u for size $i \geq 0$. Assume for contradiction that there are assignments σ, σ' for V, where $\sigma.Y = \sigma'.Y$, $\sigma.X \neq \sigma'.X$. Note that both $\delta(s,\sigma)$ and $\delta(s,\sigma')$ exist in A_φ. Then this means

that $\delta(s, \sigma)$ and $\delta(s', \sigma')$ are each a part of an accepting run r and r' respectively, on words w, w' respectively for which $u = w[0, i-1] = w'[0, i-1]$. But then $w_i.Y = w_i.Y$ and $w_i.X \neq w_i.X$, contradicting X being dependent on Y in φ.

Next, assume that X is automata dependent on Y in A_φ. Let w, w' be two words in $L(\varphi)$, and thus in $L(A_\varphi)$. Let $i \geq 0$ be such that $u = w[0, i-1] = w'[0, i-1]$. Let r, r' be accepting runs of w, w' respectively and let s_1, s_2 be the states respectively in the run r, r' that has the same history u. Note that this makes s_1, s_2 compatible (specifically if u is the empty word then $s_1 = s_2$ are the initial state s_0). Next assume that $e_1 = (s_1, s_1')$ with a label σ_1 is on r , and $e_2 = (s_2, s_2')$ with a label σ_2 is on r'. Further, assume $\sigma_1.Y = \sigma_2.Y$. Note that it means that $w_i.Y = w_i'.Y$. Then since X is automata dependent in Y, we have that $\sigma_1.X = \sigma_2.X$ as well, hence $w_i.X = w_i'.X$. ■

Finding Compatible States We find all compatible states in an automaton in Algorithm 2 as follows. We maintain a list of in-process compatible pairs C to which we start by adding the initial pair (q_0, q_0), which is of course compatible. At each step, until C becomes empty, for each pair $(s_i, q_j) \in C$, we add it to the compatible pair set P, and remove it from C (in lines 6-7). Then (in lines 9-11), we check (in line 10) if outgoing transitions from (s_i, s_j) lead to a new pair (s_i', s_j') not already in P or C, which can be reached on reading the same letter σ and if so, we add this pair to the in-process set C. All pairs that we put in P, C are indeed compatible, nothing is removed from P. When the algorithm terminates, C is empty, which means all possible ways (from the initial state pair) to reach a possible compatible pair have been explored, thus showing correctness.

To prove that algorithm 2 finds all compatible states we use lemma 4.4.1.

Lemma 4.4.1 *Let (s, s') be compatible states. Then either $(s, s') = (q_0, q_0)$ or there are states s_1, s_1' that have the edges (s_1, s), (s_1', s') such that (s_1, s_1') are compatible.*

Proof: Assume (s, s') are compatible such that either s or s' are not q_0. Then they have a shared word w. Let r, r' be the runs that lead from q_0 to s and s' respectively with the label w, and let a be the last letter in w. Then $w = w'a$ for a prefix w' of w. Then there are states (s_1, s_1') for which r reaches the s_1 with the word w' and $\delta(s_1, a) = s$, and r' reaches the s_1' with the word w' and $\delta(s_1', a) = s'$. Therefore (s_1, s_1') are compatible. ■

Therefore, we claim that algorithm 2 find all compatible states in an NBA.

Algorithm 2: 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 $(s'_i, s'_j) \in out(s_i) \times out(s_j)$ **do**
        if $(s'_i, s'_j) \notin P \cup C$ *and* $\exists \sigma \in 2^\Sigma$ *such that* $s'_i \in \delta(s_i, \sigma) \wedge s'_j \in \delta(s_j, \sigma)$ **then**
            $C \leftarrow C \cup \{(s'_i, s'_j)\}$;
return P;

Claim 2 *Algorithm 2 finds all the compatible states in the automaton.*

Proof: First note that only compatible states enter P, and no pair leaves P once it enters P. To see that P is the set of all compatible states, assume for contradiction that it is not. Then there is a compatible pair (s, s') not in p with a shared word w from q_0 to s and from q_0 to s', which is minimal in length among all shared words of compatible pairs not in P. If $|w| = 0$ then $s = s' = q_0$ so $(s, s') \in P$, a contradiction. Then otherwise by Lemma 4.4.1 there are some s_1, s'_1 for which $s_1 \in E(s)$ and $s'_1 \in E(s')$ for which (s_1, s'_1) are compatible and therefore in P, since they have history of length $w' < w$ where $w = w'a$ for some letter a. But then since Algorithm 2 returns all pairs in P unmarked, we have that at some step j of the algorithm, (s_1, s'_1) was selected, and therefore (s, s') would have been identified and added to P as well, a contradiction. ■

Finally, we show how to implement **isDependent** from Algorithm 1 by implementing the following procedure **isAutomataDependent**, described in Algorithm 3. **isAutomataDependent** works by trying to find a witness to $\{z\}$ being *not* dependent on Y. If no such witness exists then it means that $\{z\}$ is dependent on Y. Given a variable z, a set $Y = V \setminus \{z\}$ and a list P of all compatible pairs in A which is returned from Algorithm 2, the algorithm **isAutomataDependent** checks for every pair $(s, s') \in P$ (line 9) if there exists an assignment σ, σ' for which both $\delta(s, \sigma)$ and $\delta(s', \sigma')$ exist,

$\sigma.Y = \sigma'.Y$ and $\sigma.\{z\} \neq \sigma'.\{z\}$ (lines 10-11).

Algorithm 3: Check Dependency Based Automaton

```
1  Input: NBA A_φ = (Σ, Q, δ, q_0, F) from φ.
2         Candidate dependent variable {z}
3         Candidate dependency set Y
4         All compatible states P made by the function
    FindAllCompatibleStates(A_φ).
5  Output: Is {z} dependent on Y by definition 13
6  Function AreStateColliding(p, q):
7      return ∃σ_p, σ_q ∈ 2^Σ such that
        δ(p, σ_p) ≠ ∅ ∧ δ(q, σ_q) ≠ ∅ ∧ σ_p.Y = σ_q.Y ∧ σ_p.{z} ≠ σ_q.{z};
8  begin
9      for (s_1, s_2) ∈ P do
10         if AreStateColliding(s_1, s_2) then
11             return False;
12     return True;
```

Lemma 4.4.2 *Algorithm 3 returns* True *if and only if* $\{z\}$ *is automata-dependent on* $V \backslash \{z\}$ *in* A_φ.

Proof: Since P is finite, The algorithm of course terminates. The algorithm 3 returns *false* if and only if there are two compatible states s, s' and assignment σ_Y for Y, for which there are distinct assignments z, z' which both $\delta(s, zy)$ and $\delta(s, z'y)$ exist. Then by definition 13 we see that $\{z\}$ is not dependent on Y. ■

Definition 14 (Algorithm FindAutomataDependency) *Algorithm 1 finds a maximal set of dependent variables given a function that checks if a variable* $\{x\}$ *is dependent on the set of variables. The algorithm* ***FindAutomataDependency*** *is algorithm 1 with the function* ***isAutomatonDependent*** *as defined in algorithm 3 as a function to check if a variable is dependent.*

Corollary 2 *The algorithm* ***FindAutomataDependency*** *(Definition 14), returns a maximal dependent set in* φ.

Thus using the above algorithm to perform a dependency check we can compute maximal sets of dependent variables (as explained earlier), which

we will use next to improve synthesis. Note that all the above algorithms run in time polynomial (in fact, quadratic) in size of the NBA.

Corollary 3 *Given NBA A_φ, computing compatible pairs, checking dependency, and building maximal dependent sets can be done in time polynomial in the size of A_φ.*

In the experimental results, which are discussed widely in section 7.1, the automaton approach scaled and outperforms the formula approach. Therefore, in this research, we focus mainly on the automaton approach.

5 Dependency in Reactive Synthesis

In this section, we explain how dependencies can be beneficially exploited in a reactive synthesis pipeline. Our synthesis algorithm focuses on the NBA A_φ of the LTL specification, it finds dependents on the NBA and project from it the dependent variables that lead to the NBA A'_φ which is smaller in terms of total output variables (in case there are dependent variables). The reduced NBA A'_φ is synthesized using existing algorithms [9] and the construction of the dependent variables strategy is described in this chapter. This section is a theoretical explanation of exploiting dependency in reactive synthesis and uses notions and definitions from the literature. In practice, tools in the field of reactive synthesis, such as Spot [8], use symbolic manipulation in the implementation, such as ROBDD to represent edges between states in the NBA and AIGER to represent the transducers. In the next section, we will discuss the implementation using symbolic representations and practical tools.

5.1 High Level Overview

Our approach can be described at a high level as shown in Figure 5.1. This flowchart has the following 6 steps. Every box is a step that is described in this section. Every step's output edge in the Figure is passed as an input to the next step, together with the relevant variables. This is illustrated by arrows between the steps. For example, Step 2 passes the automaton A_φ with variable X to Step 5.

The framework follows the following steps:

1. Given an LTL formula φ over a set of variables V with input variables $I \subseteq V$ and output variables $O = V \setminus I$, we first construct a

language-equivalent NBA $A_\varphi = (\Sigma_I \cup \Sigma_O, S, s_0, \delta, F)$ by standard means, e.g [40].

2. Then, as described in Section 4, 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 $\Sigma_I \times \Sigma_{O \backslash X}$.

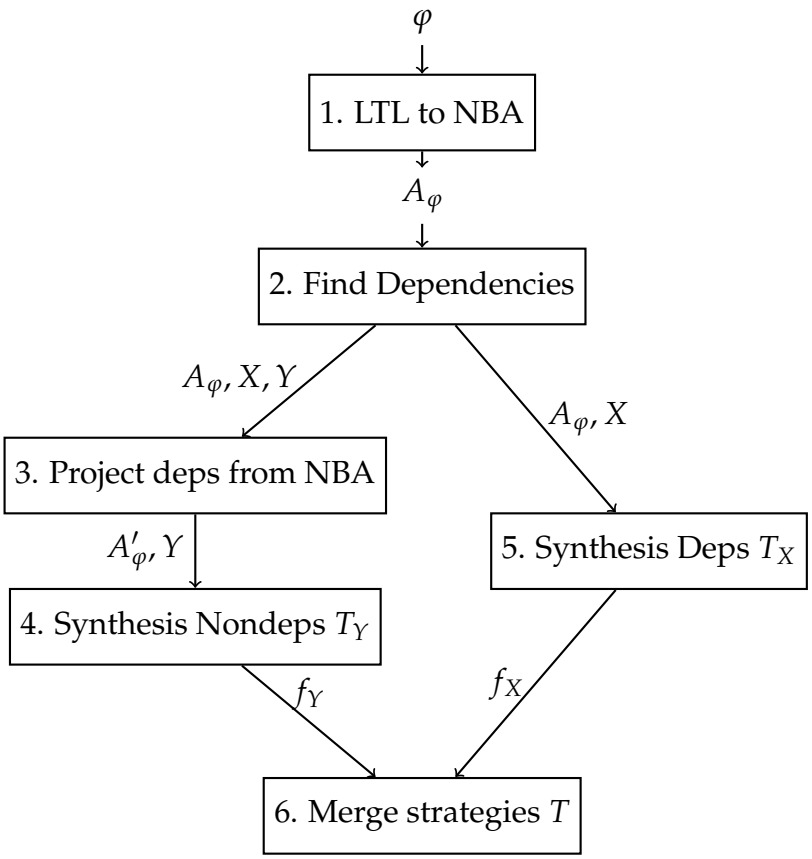
3. Next, we construct an NBA A'_φ from A_φ by projecting out (or eliminating) all X variables from labels of transitions. Thus, A'_φ has the same sets of states and transitions as A_φ. We simply remove valuations of variables in X from the label of every state transition in A_φ to obtain A'_φ. Note that after this step, $L(A'_\varphi) = \{w \mid \exists u \in L(A_\varphi)$ such that $w = u.Y\} \subseteq \Sigma_Y^\omega$.

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., [9]) to obtain a transducer T_Y that describes a strategy $f_Y : \Sigma_I^* \to \Sigma_{O \backslash X}$ for $L(A'_\varphi)$. If no such strategy exists, then φ is unrealizable.

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.

Figure 5.1: Synthesis using dependencies



In the above synthesis flow, we use standard techniques from the literature for Steps 1 and 4, as explained above. Step 2 was already described in detail in Section 4. Step 3 is easy when we have an explicit representation of the automata. As we will discuss in the next section, it has interesting consequences when we use symbolic representations of automata. Step 6, as explained above, is straightforward. Hence, in the remainder of this section, we focus on Step 5, which is also a key contribution of this paper.

5.2 Synthesis Dependent Variables

In this section, we explain the construction of the strategy of the dependent variables, i.e., the transducer T_X as described in the high-level framework. An example of this process is shown at subsection 5.2. Let $A_\varphi = (\Sigma_I \times \Sigma_O, Q, \delta, q_0, F)$ be the NBA of the specification obtained in step 1

of the pipeline shown above. The transducer T_X we wish to construct is a deterministic Mealy machine described by the 6-tuple $T_X = (\Sigma_Y, \Sigma_X \cup \{\bot\}, Q^X, q_0^X, \delta^X, \lambda^X)$, where the dependency variables $\Sigma_Y = \Sigma_I \times \Sigma_{(O \setminus X)}$ 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. We define the states of the transducer as the powerset of Q, i.e., $Q^X = 2^Q$, and the initial state to be $q_0^X = \{q_0\}$. The domain and codomain of the transition function and the output functions are $\delta^X : Q^X \times \Sigma_I \times \Sigma_{(O \setminus X)} \to Q^X$ and $\lambda^X : Q^X \times \Sigma_I \times \Sigma_{(O \setminus X)} \to \Sigma_X$ correspondingly. The state transition function δ^X is defined by the Rabin-Scott subset construction [32], which is described in section 2.1 applied to the automaton A_φ [32]. Formally, for every $U \subseteq Q$, $\sigma_I \in \Sigma_I$ and $\sigma \in \Sigma_{(O \setminus X)}$, we define: $\delta^X\big(U, (\sigma_I, \sigma)\big) = \{q' \mid q' \in Q, \exists q \in U \text{ and } \exists \sigma' \in \Sigma_X \text{ such that } q' \in \delta\big(q, (\sigma_I, \sigma, \sigma')\big)\}$.

Before defining the output function λ^X, we state important properties of T^X that follow from the definition of δ^X above.

Lemma 5.2.1 *Let U be a state reachable from q_0^X in T_X and let $\sigma_Y \in \Sigma_Y$, then there is a single $\sigma_X \in \Sigma_X$ such that for every $q \in U$ it has an outgoing edge in A_φ, i.e., $\delta(q, (\sigma_Y, \sigma_X)) \neq \emptyset$ and for every $\sigma'_X \neq \sigma_X$ it holds that $\delta(q, (\sigma_Y, \sigma_X)) = \emptyset$.*

Proof: Since X is automata dependent in A_φ, it follows from Definition 13 that for every $(q, q') \in U \times U$, if $\delta\big(q, (\sigma_I, \sigma_{O \setminus X}, \sigma_X)\big) \neq \emptyset$ and $\delta\big(q', (\sigma_I, \sigma_{O \setminus X}, \sigma'_X)\big) \neq \emptyset$ for $\sigma_X, \sigma'_X \in \Sigma_X$, then $\sigma_X = \sigma'_X$. ■

Lemma 5.2.2 *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.*

Proof: We prove by induction on the number of steps, say k, needed to reach a state U from q_0^X in T_X. The base case follows from the fact that q_0^X is a singleton set, and every singleton set trivially satisfies the desired property. Therefore, the claim holds for $k = 0$.

Suppose the claim holds for all states U reachable from q_0^X in k or fewer steps, for $k \geq 0$. Hence, (q, q') is compatible in A_φ for all $q, q' \in U$. We wish to prove that if $U' = \delta^X\big(U, (\sigma_I, \sigma)\big)$ for any $(\sigma_I, \sigma) \in \Sigma_I \times \Sigma_{O \setminus X}$, then (s, s') is also compatible in A_φ for every $s, s' \in U'$.

From the definition of δ^X and lemma 5.2.1, it now follows that if $U' = \delta^X\big(U, (\sigma_I, \sigma_{O \setminus X})\big)$ and if $U' \neq \emptyset$, then there exists a unique $\sigma_X \in \Sigma_X$ such

that $U' = \{q' \mid \exists q \in U \text{ such that } q' \in \delta\big(q, (\sigma_I, \sigma_{O\setminus X}, \sigma_X)\big)\}$. This shows that for every $s, s' \in U'$, the pair (s, s') is compatible in A_φ. ■

Given these properties, the output function λ_X needs to be defined as the unique $\sigma_X \in \Sigma_X$ which exists for every $U \subseteq Q, \sigma_Y \in \Sigma_Y$ as shown in lemmas 5.2.1, 5.2.2, i.e., $\lambda^X\big(U, (\sigma_I, \sigma_{O\setminus X})\big) = \sigma_X$. The correctness of this strategy is shown in the next section. In terms of intuition, every possible state on the transducer T_X is a set of compatible states 5.2.2, and every input of this transducer is an assignment of dependency variables. Compatibility ensures that these states agree on a unique value of the dependent variables for every possible dependency assignment, which is the input of the transducer.

Example of dependent strategy construction

As an illustration of the above construction, consider the NBA A_φ in Fig. 4.1. We saw in Section 4.4 that o_1 is dependent on $\{i, o_2\}$ in A_φ, hence $X = \{o_1\}$ and $Y = \{i, o2\}$. The transducer T_X in this case has the states: $\{q_0\}, \{q_1\}$ and $\{q_0, q_1\}$, with $\{q_0\}$ being the initial state. The input of the transducer is $\Sigma_X = \{i, o_2\}^*$ and $\Sigma_X = \{o_1\}^*$ is its sole output, i.e., the variables $\Sigma_I, \Sigma_{O\setminus X}$ leads to Σ_X. In this NBA example, from state q_1 there are two outgoing edges with the label $(i = 0, o_1 = 0, o_2 = 0)$ therefore, $\delta^X(\{q_1\}, 00) = \{q_0, q_1\}$ and $\lambda^X(\{q_1\}, 00) = 0$. In state $\{q_0\}$ for input $(i = 0, o_1 = 0, o_2 = 0)$ it leads to the state $\{q_1\}$ and in state $\{q_1\}$ for input $(i = 0, o_1 = 0, o_2 = 0)$ it leads to the state $\{q_0, q_1\}$, where necessarily both label $o_2 = 0$, therefore $\lambda^X(\{q_0, q_1\}, 00) = 0$ and $\delta^X(\{q_0, q_1\}, 00) = \{q_0, q_1\}$. This transducer is also visualized as a diagram in figure 5.2.

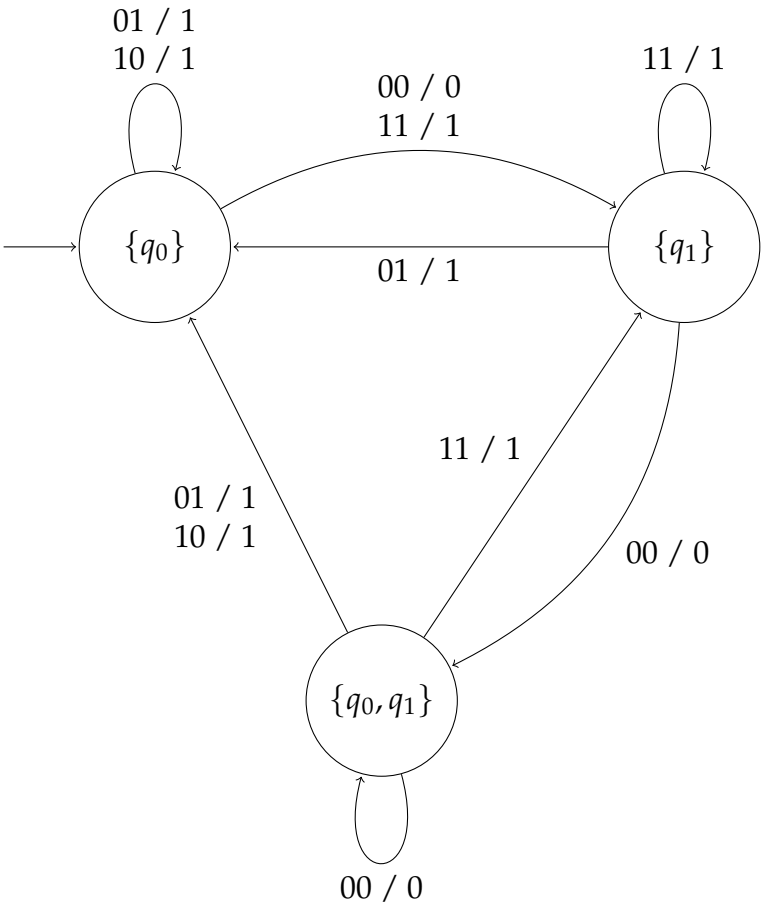


Figure 5.2: Example of a transducer of dependent variables displayed as an automaton.

5.3 Synthesis Correctness

We explain in detail our synthesis approach. We first give some definitions and notations that are necessary to describe our approach, then describe a more general framework from which the correctness of the automata implementation easily follows.

Given a language L over alphabet $\Sigma_I \cup \Sigma_O$, where Σ_I and Σ_O are distinct finite alphabets, a *two players game* of L is a tuple $G_L = (\Sigma_I, \Sigma_O, L)$ played between two players P_0 and P_1 as follows. Starting with P_0 at step 0, at every step i, the player P_0 chooses a word from Σ_I followed by P_1 that chooses a word from Σ_O. Thus every step i becomes a word over $\Sigma_I \cup$

Σ_O. The game goes forever at which an infinite word $w \in (\Sigma_I \cup \Sigma_O)^\omega$ is formed. P_1 wins the game if $w \in L$.

Given such a game, the challenge is to devise a strategy $f : \Sigma_I^* \to \Sigma_O$ for P_1, of which word to pick such that every word $w \in (\Sigma_I \cup \Sigma_O)^\omega$ obtained by using this strategy is in L. Such a strategy is also called a *winning strategy* or a *solution* for L. By default, L is compactly described as an LTL formula φ (as in reactive synthesis). Note however that L can also be compactly given as an NBA A, and then we call the game played an *NBA game*, denoted for simplification: $G_A = (\Sigma_I, \Sigma_O, A)$. The set of infinite words obtained by the strategy f is denoted by $L(f)$. That is $w \in L(f)$ if and only if for every prefix $w[0,i] \cap I$ we have that $w_i \cap O = f(w[0,i] \cap I)$. Then if f is a solution to φ then $L(f) \subseteq L(\varphi)$. $L(f)$ is also called the *language* of the strategy f.

Next, we are given a reactive LTL formula φ over input variables I and output variables O. We do as follows.

1. Find in φ a maximal set of output variables X that are dependent on φ. Set $Y = I \cup (O \backslash X)$. Define a language $L' = \{w.Y \mid w \in L(\varphi)\}$, and set a function $g^Y : L(\varphi) \to L'$ as follows $g^Y(w) = w.Y$. Similarly set a language $g^X : L(\varphi) \to X^\omega$ as follows $g^X(w) = w.X$. Note that for every word $w \in \Sigma_V^\omega$ we have $w = g^Y(w) \oplus g^X(w)$. The notions follow the definitions in subsection 2.1.

2. We synthesize a solution $f_Y : \Sigma_I^* \to (O \backslash X)$ to solve the game $G_{L'}$. If no such solution exists then it means that φ is unrealizable.

3. We construct a function $f_X : (\Sigma_Y^* \to \Sigma_X)$ with the following property: for every word $w' \in L'$ there exists a unique word $w \in W$ in which $g^Y(w) = w'$ and $g^X(w) = f_X(w')$.

4. From f_Y and f_X we construct the following strategy $f : \Sigma_I^* \to \Sigma_O$. $f(i) = f_Y(i) \oplus f_x(i \oplus f_Y(i))$. We call f the *combination* of f_Y and f_X. We then have that f is a solution to φ if and only if f is realizable.

We now have the following

Lemma 5.3.1 *g^Y is well defined and is a bijection.*

Proof: We see that g^Y is a bijection. To see that g^Y is onto, let $w' \in L'$. Then by the definition of L', there is $w \in L$ such that $w' = w.Y$. Then

$g^y(w) = w'$. To see that g^Y is an injection, assume that w, h are words in L, and $w \neq h$. We see that $w.Y \neq h.Y$. Let i be the first index in which $w_i \neq h_i$. If $w_i.Y = h_i.Y$, then since $w[0, i-1] = h[0, i-1]$, we have that $w_i.X = h_i.X$ since X is dependent on Y, a contradiction since then $w_i = h_i$. Therefore $w_i.Y \neq h_i.Y$, which means that $g^Y(w) = w.Y \neq h.Y = g^Y(h)$. ■

Lemma 5.3.2 *f^X is well defined.*

Proof: Let $w' \in L'$, then since g^Y is a bijection, $w = {g^Y}^{-1}(w')$ is a unique word for which $g^Y(w) = w'$. Then f_X is defined to be such that $f_X(w') = g^X({g^Y}^{-1}(w'))$. ■

We are now ready for the main lemma.

Lemma 5.3.3 *φ is realizable if and only if $G_{L'}$ is solved.*

Proof: Suppose that φ is realizable with a solution $f : \Sigma_I^* \to \Sigma_O$. We construct a strategy $f' : \Sigma_I^* \to \Sigma_{(O \setminus X)}$ as follows. For every $i > 0$ and a word $\sigma \in \Sigma_I^i$, we have $f'(\sigma) = f(\sigma).(O \setminus X)$. We see that f' solves L'. Let w' be a word over $(\Sigma_I \times \Sigma_{(O \setminus X)})^\omega$ in $L(f')$. Then there is a word $w \in L(f)$ (and therefore in $L(\varphi)$ since f is a solution) for which $w = w' \oplus g^X(w)$. Then $g^Y(w) = w'.Y$. Then from Lemma 5.3.1 we have that since $w \in L$ then $w' \in L'$.

Next, assume that L' has a solution $f_Y : \Sigma_I^* \to \Sigma_{(O \setminus X)}$. We show that the strategy $f : \Sigma_I^* \to \Sigma_O$, where for every word $w \in \Sigma_I^i$ we have that $f(w) = f_Y(w) \oplus f_x(w \oplus f_Y(w))$, solves $L(\varphi)$. Let $w \in L(f)$ and denote $w^I = w.I$. Then we have that $w.Y = w^I \oplus f_Y(w^I)$ is in $L(f_Y)$, and therefore is in L'. Hence there is a word $w' \in L(\varphi)$ for which $g^Y(w') = w.Y$ and $w'.X = g^X(w') = f_X(w')$. But then $w'.X = w.X$, which means that $w = w'$, hence $w \in L$. ■

Corollary 4 *If φ is realizable then f is well defined and solves φ.*

Proof: Follows immediately from Lemma 5.3.3. ■

Finally, following the construction described in Section 5.2 we make the following proof.

Theorem 4 *If φ is realizable, the transducer T obtained by composing T_X and T_Y as in step 6 of Fig. 5.1 solves the synthesis problem for φ.*

Proof: For the NBA A' constructed from A, we have that $L(A') = L'$. Then by definition, the synthesized transducer T_Y describes a strategy f_Y that solves L'. We also have that the construction of T_X basically describes f_X and its properties. Finally, see that the merge T of T_Y and T_X describes the strategy f that combines f_Y and f_X. ■

Corollary 5 *If φ is realizable then our synthesis framework returns True with a solution f to φ. Otherwise, our framework returns False.*

Proof: From Theorem 4 we have that if φ is realizable then our synthesis framework returns *True* with the solution f. From Lemma 5.3.3 we have that if φ is not realizable then there is no solution to $G_{L'}$ hence our framework returns *False*. ■

Therefore from Corollary 5, if φ is realizable then f solves φ and therefore T solves φ as well, and if φ is unrealizable then there is no solution and the framework return $false$.

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.

6 Symbolic Implementation

In this section, we describe symbolic implementations of each of the non-shaded blocks in the synthesis flow depicted in Fig. 5.1. There are two symbolic implementations in our algorithm that are slightly different than those discussed in sections 4, 5: 1) ROBDD as NBA edges. 2) AIGER, as defined in definition 8, to represent the synthesis strategy. This chapter is composed of 3 sections. Section 6.1 describes how the algorithm handles the cases where the edges between states are ROBDDs, section 6.2 describes how to construct AIGER which is a symbolic strategy for the dependent variables, and section 6.3 describes how to merge two AIGERs which are symbolic strategies.

6.1 Identifying and projecting dependency

ROBDD as NBA edges We use the same representation as used in Spot [8] – 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) [11], as is done in Spot. Each labeled edge represents a set of state transitions from s to s', with one transition for each satisfying assignment of $B_{(s,s')}$. For example, consider $I = \{i_1, i_2\}$ and $O = \{o_1, o_2\}$ and suppose an edge from s to s' is labeled by the Boolean function $B_{(s,s')} \equiv (i_1 \vee o_1) \wedge i_2 \wedge \neg o_2$. There are three satisfying assignments of $B_{(s,s')}$, i.e. $i_1 i_2 o_1 o_2 = 1100, 0110$ and 1110. Therefore, the edge from s to s' labeled $B_{(s,s')}$ represents three state transitions from s to s', one corresponding to each of the satisfying assignments of $B_{(s,s')}$.

Implementing Find Dependencies with NBA We described algorithms for finding dependent variables from the NBA where states of the NBA A_φ are explicitly represented as nodes of a graph. Algorithm 2 search for all compatible states and algorithm 3 verifies dependency based on the automaton and its pair states. Since in the implementation of the framework, the transition function is not a set of labels, i.e., $\delta(s, s') \subset \Sigma$ but a ROBDD, i.e. $\delta(s, s') = B_{s,s'}$ an appropriate changes needs to be done. In algorithm 2 we determinate that the pair-states $(s'_i, s'_j) \in out(s_i) \times out(s_j)$ are compatible if $\exists \sigma \in 2^\Sigma$ such that $s'_i \in \delta(s_i, \sigma) \wedge s'_j \in \delta(s_j, \sigma)$, i.e., there is an assignment σ such that it leads s_i to s'_i and s_j to s'_j. In the ROBDD case, we replace this condition with $B_{(s_i,s'_i)} \wedge B_{(s_j,s'_j)} \neq False$, it has exactly the same meaning as described in Claim 3.

Claim 3 *Given the NBA $A = (\Sigma, Q, \delta, q_0, F)$ where the transition is function $\delta : Q \times \Sigma \to 2^Q$ and the equivalent NBA $A' = (\Sigma, Q, \delta', q_0, F)$ where the transition function's output is a BDD, i.e., $\delta(s, s') = B_{s,s'}$. Then for every pair states (s, s'), it holds that $\exists \sigma \in 2^\Sigma$ such that $s'_i \in \delta(s_i, \sigma) \wedge s'_j \in \delta(s_j, \sigma)$ if and only if $B_{(s_i,s'_i)} \wedge B_{(s_j,s'_j)} \neq False$.*

Proof: By the definition, it holds that $s'_i \in \delta(s_i, \sigma)$ if and only if $\sigma \models B_{s_i,s'_i}$. Therefore, $\exists \sigma \in 2^\Sigma$ such that $s'_i \in \delta(s_i, \sigma) \wedge s'_j \in \delta(s_j, \sigma)$ if and only if $\exists \sigma \in 2^\Sigma$ such that $\sigma \models B_{(s_i,s'_i)}$ and $\sigma \models B_{(s_j,s'_j)}$, i.e., $\sigma \models B_{(s_i,s'_i)} \wedge B_{(s_j,s'_j)}$. In other words, it means that $B_{(s_i,s'_i)} \wedge B_{(s_j,s'_j)}$ is satisfiable, i.e., $B_{(s_i,s'_i)} \wedge B_{(s_j,s'_j)} \neq False$. ■

Algorithm 3 decides that a variable z is dependent if there is no collision between any compatible pair-state, where a collision in (p, q) is defined as $\exists \sigma_p, \sigma_q \in 2^\Sigma$ such that $\delta(p, \sigma_p) \neq \emptyset \wedge \delta(q, \sigma_q) \neq \emptyset \wedge \sigma_p.Y = \sigma_q.Y \wedge \sigma_p.\{z\} \neq \sigma_q.\{z\}$, i.e., there are different assignments σ_p, σ_q that are the same on the non-dependent variables Y but different on the candidate dependent variable z. In the ROBDD case, we define collision in (p, q) if formula 6.1 is satisfiable:

$$\bigvee_{(s,s') \in out(p) \times out(q)} B_{(p,s)}(I, O) \wedge B_{(q,s')}(I', O') \wedge \bigwedge_{y \in Y} (y \leftrightarrow y') \wedge (z \leftrightarrow \neg z') \tag{6.1}$$

In the above formula, I' (resp. O') denotes a set of fresh, primed copies of variables in I (resp. O).

Lemma 6.1.1 *Assume that formula 6.1 is satisfiable. Then there are assignments σ_p, σ_q which are the same on non-dependent variables Y but different on the candidate dependent variable z, and therefore the states collide.*

Proof: Assume that the formula 6.1 is satisfiable, it means that exists $(s, s') \in out(p) \times out(q)$ that satisfies $B_{(p,s)}(I, O) \wedge B_{(q,s')}(I', O') \wedge \bigwedge_{y \in Y}(y \leftrightarrow y') \wedge (z \leftrightarrow \neg z')$. It means that exists $\sigma_p, \sigma_q \in \Sigma^{I \cup O}$ such that $\sigma_p \models B_{(p,s)}(I, O)$, $\sigma_q \models B_{(q,s')}(I', O')$ and $\sigma_p.Y = \sigma_q.Y, \sigma_p.\{z\} \neq \sigma_q.\{z\}$. ■

By lemma 6.1.1 we infer that algorithm 3 finds dependent variables where the edges in the NBA are symbolically presented as ROBDDs.

Projecting dependent variables The synthesis process transforms A_φ to A'_φ by removing dependent variables from A_φ as described in section 5. Let X be the dependent variables, 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.

Theorem 5 *There exists an NBA A_φ with a single dependent output and an edge e such that the ROBDD that labels e is exponentially larger (in number of inputs and outputs) than the labeling of e in A'_φ.*

Proof: Let $I = \{i_1, \ldots i_n\}$ and $O = \{o_1, \ldots o_{n+1}\}$ be sets of input and output variables. Consider the LTL formula $\mathsf{G}\big(o_{n+1} \leftrightarrow f(i_1, \ldots i_n, o_1, \ldots o_n)\big)$, where f is a Boolean function that gives the value of the n^{th} least significant bit (or middle bit) in the product obtained by multiplying the unsigned integers represented by the $i_1, \ldots i_n$ and $o_1, \ldots o_n$. The automaton A_φ has a single state, which is also an accepting state, with a self-loop, denoted e, labeled by an ROBDD representing $o_{n+1} \leftrightarrow f(i_1, \ldots i_n, o_1, \ldots o_n)$. It is known that this ROBDD has a size in $\Omega(2^n)$ regardless of the variable ordering. However, once we detect that o_{n+1} is dependent on $\{i_1, \ldots i_n\} \cup \{o_1, \ldots o_n\}$, we can project away o_{n+1}, and the ROBDD after projection is simply a single node representing True. This is because for every combination of values of $\{i_1, \ldots i_n\} \cup \{o_1, \ldots o_n\}$, there is a value of o_{n+1} that matches $f(i_1, \ldots i_n, o_1, \ldots o_n)$. ■

6.2 Implementing T_X

We now describe how to construct an AIGER $(I, O, L, \lambda, \delta)$ (definition 8) as a symbolic implementation of the transducer T_X which is described in section 5. The I is the input variables of the specification and the non-dependent variables and the O is the output which are the dependent variables. 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 an AIGER implementing symbolically 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 $I \cup (O \setminus X)$.

Mealy Machine States and AIGER Latches Let $S = \{s_0, \ldots s_{k-1}\}$ be the set of states of A_φ. To implement the desired AIGER, we define the set of latches L to have k latches (state-holding flip-flops). Every subset of the automaton A_φ states $U \subseteq S$ is represented in the AIGER as a state of k flip-flops, i.e. by a k-dimensional Boolean vector. Specifically, the i^{th} component in this vector is set to 1 if and only if $s_i \in L$. For example, if $S = \{s_0, s_1, s_2\}$ and $L = \{s_0, s_2\}$, 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} latch.

Describing the transition function The next-state function $\delta = (\delta^1, \cdots \delta^k)$ of the AIGER is implemented as follows. Every δ^i is implemented by a circuit, 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 $\bigvee_{s_j \in \mathsf{in}(s_i)} \left(p_j \wedge \exists X\, B_{(s_j, s_i)}\right)$. Theorem 6 shows that δ is a symbolic implementation of the transition function δ^x in T_X. It is known from the knowledge compilation literature (see e.g. [1, 4, 16]) that every ROBDD can be compiled in linear time to a Boolean circuit in an And-inverter graph (AIG), and that every AIG circuit admits linear time projection of variables, yielding a resultant AIG 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 symbolically the next-state transition logic of our Mealy machine, in time (and space) linear in $|X|$ and $|A_\varphi|$.

Output function Next, we turn to constructing the output functions λ in T_X. The functions $\lambda = (\lambda^{x_1}, ..., \lambda^{x_{|X|}})$ takes as an inputs $\{p_0, \dots p_{k-1}\} \cup I \cup (O \setminus X)$ and outputs X. Since X is automata dependent on $I \cup (O \setminus X)$ in A_φ, we have the following claim:

Claim 4 *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 $\sigma_X \in \Sigma_X$ such that $(\sigma_I, \sigma, \sigma_X) \models B_{(s,s')}$.*

Proof: Let (σ_I, σ) be such that $(\sigma_I, \sigma) \models \exists X\, B_{(s,s')}$, by the definition of the $\exists$ operator there is $\sigma_X \in \Sigma_X$ such that $(\sigma_I, \sigma, \sigma_X) \models B_{(s,s')}$. Assume in contradiction that σ_X is not unique, i.e., exists σ'_X that $(\sigma_I, \sigma, \sigma'_X) \models B_{(s,s')}$. By definition 13 of automaton dependency, for every pair compatible states it holds that $(\sigma_I, \sigma, \sigma'_X)$ and $(\sigma_I, \sigma, \sigma_X)$ cannot both satisfies $B_{(s,s')}$ since (s, s) is compatible. ■

Considering only the transition (s, s') referred to in claim 4, we first discuss how to synthesize a vector of Boolean functions, say $F^{(s,s')} = \langle F_1^{(s,s')}, \dots F_{|X|}^{(s,s')} \rangle$, 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 $F^{(s,s')}$ such that $\forall (I \cup (O \setminus X)) \left(\exists X B_{(s,s')} \rightarrow B_{(s,s')}[X \mapsto F^{(s,s')}]\right)$ 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, 13]). In fact, we know from [1, 35] that if $B_{(s,s')}$ is represented as a ROBDD, then a Boolean circuit for $F_{(s,s')}$ can be constructed in $\mathcal{O}\left(|X|^2.|B_{(s,s')}|\right)$ time, where $|B_{(s,s')}|$ denotes the size of the ROBDD for $B_{(s,s')}$. Then, 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 $\bigvee_{\textit{trans. } (s,s') \textit{ in } A} \left(p_s \wedge (B_{(s,s')}[X \mapsto F^{(s,s')}]) \wedge F_i^{(s,s')}\right)$.

Note that $\delta^X\big(U, (\sigma_I, \sigma)\big) = \emptyset$ if and only if 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.

To conclude our construction, we have the following theorem.

Theorem 6 *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* $j \in \{1, \ldots k-1\}$*, and (b) the output is interpreted as* $\perp$ *whenever* $\bigvee_{i=0}^{k-1} n_i$ *evaluates to* 0*.*

Proof: To see why δ is a symbolic implementation of the transition function δ^x in T_X, suppose that $\langle p_0, \ldots p_{k-1} \rangle$ represents the current state $U \subseteq S$ of the AIGER. Then the above function sets n_i to true if and only if there is a state $s_j \in U$ (i.e. $p_j = 1$) such that 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 requirement for s_i to be member in the Mealy machine state $U' \subseteq S$, that is reached from U by subset construction for the given values of $I \cup (O \setminus X)$. To see why λ^x is a symbolic implementation of the of the output function in T_X, we know from the proof of Lemma 5.2.2 that there is a unique $\sigma_X \in \Sigma_X$ such that from every pair states (s, s') and for every $\sigma_y \in \Sigma_{I \cup O \setminus X}$ there is a transition from s to s', therefore, when for a single pair-state (s, s') there is a transition where x is True for a given $\Sigma_{I \cup O \setminus X}$, it's the correct assignment to x, this is as discussed in section 5.3. If there is no (s, s') where x is True for a given $\Sigma_{I \cup O \setminus X}$, there x must be assigned to false. This is exactly the meaning of the formula of δ^x:

$$\delta^x = \bigvee_{\textit{trans. } (s,s') \textit{ in } A} \left(p_s \wedge (B_{(s,s')}[X \mapsto F^{(s,s')}]) \wedge F_i^{(s,s')}\right)$$

Explanation of the formula: is there any transition (s, s') (the big $\vee$ operator), where the corresponding latch of state s is on, which is defined to be p_s, and there is a transition to the next state given the current input and non-dependent variables, which is defined as $F_i^{(s,s')}$ and the value of x is supposed to be true in this transition which is $(B_{(s,s')}[X \mapsto F^{(s,s')}]$. ■

6.3 Merge strategies

In chapter 5 we showed how to construct a strategy $T_Y : I^* \to (O \setminus X)$ for the non-dependent variables, the state-of-the-art tools construct AIGER as the strategy. In this chapter, section 6.2 we showed how to construct AIGER $T_X : (I \cup (O \setminus X))^* \to X$ for dependent variables. We next show how to merge the strategies T_X, T_Y in order to get a strategy $T : I^* \to O$. We use Figures 6.1 6.2 6.3, to better illustrate our construction.

We are also given the AIG T_X, as shown in figure 6.2, which has as input the specification input variables and the non-dependent variables $I \cup Y =$

$\{i_1, ..., i_n, y_1, ..., y_{|Y|}\}$, as an output the dependent variables $X = \{x_1, ..., x_{|X|}\}$ and the latches $L = \{0,1\}^j$.

Then from T_Y and T_X, we construct the AIG T, shown in figure 6.3, which describes the required strategy for the specification, and is defined as follows: The input is $I = \{i_1, ..., i_n\}$, output $O = X \cup Y$, latches $L + S = \{0,1\}^{k+j}$. The input I is wired to the input of T_Y and T_X. The first k latches are wired to the latches of T_Y and the remind j latches are wired to the latches of T_X. The first k next-latches slots are connected to the next-latches slot of T_Y and the remind j latches are wired to the next-latches of T_X. The output Y of T_Y is wired to the output of T and as input to T_X. The output X of T_X is wired to the output of Y.

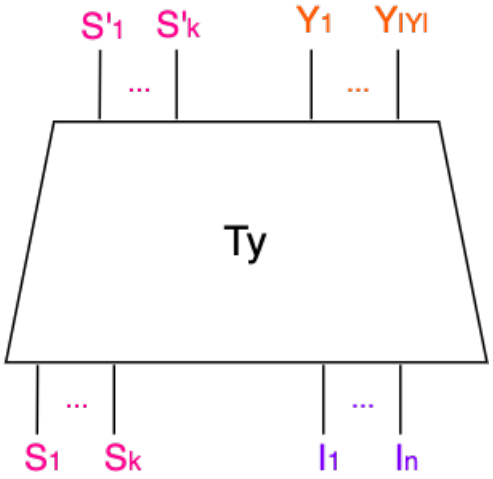


Figure 6.1: AIG T_Y, strategy of non-dependent variables

We are given the AIG T_Y, as shown in figure 6.1, which has as input the input of specification $I = \{i_1, ..., i_n\}$, and as an output the non-dependent variables $Y = \{y_1, ..., y_{|Y|}\}$ and the latches $S = \{0,1\}^k$.

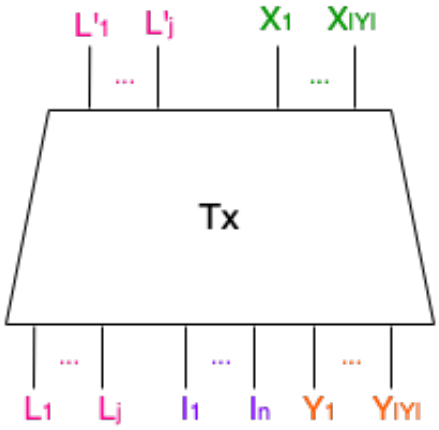


Figure 6.2: AIG T_X, strategy of dependent variables

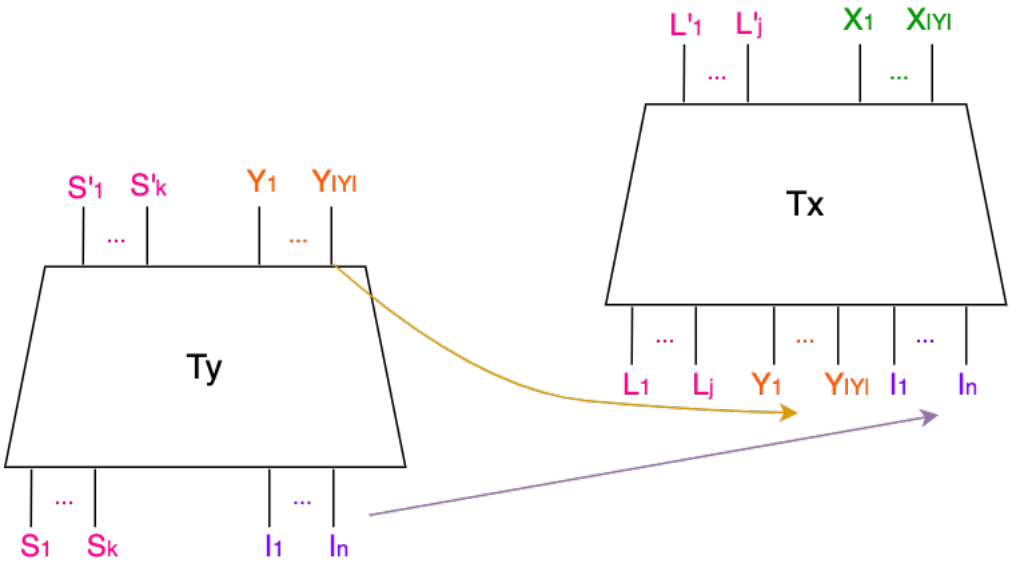


Figure 6.3: AIG T, strategy of the specification

7 Experiments and Evaluation

We implemented the synthesis pipeline depicted in Figure 5.1 in a tool called DepSynt[1] , using the symbolic approach of chapter 6 supporting both the automaton approach and the formula approach as description in chapter 4. For steps 1 and 4 of the pipeline, i.e., construction of A_φ and synthesis of T_Y, we used the tool Spot [8], a widely used library for representing and manipulating NBAs, while the other steps are using our custom C++ code. We then experimented with 1,141 LTL specifications over 31 benchmark families from the SYNTCOMP competition [24]. 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 focused on answering the following research questions:
RQ1: Between automaton-based and formula-based identification of dependent output variables, which performs better in practice?
RQ2: How prevalent are dependent output variables in existing reactive synthesis benchmarks?
RQ3: Is there any evidence to suggest that reactive synthesis is benefited by identification and separate processing of dependent output variables?

7.1 Comparing automata approach and formula approach

To answer **RQ1**, we implemented a tool called FindDeps that finds a maximal set of dependent in an LTL formula, as described in algorithm 1 sup-

[1]The source code and experiment results can be accessed at: https://github.com/eliyaoo32/DepSynt

porting both the formula approach and the automaton approach, the automaton approach is implemented symbolically as described in chapter 6. Our tool is implemented via Spot [8] that includes building NBA from the LTL formula, checking the emptiness of the LTL formula (for formula approach, using NBA emptiness checking [15]) and NBA traversal (for automaton approach). We applied FindDeps on all 1,141 SYNTCOMP [24] benchmarks with a timeout of 60 minutes on each benchmark. To compare the approaches we measure the metrics:

1. How many benchmarks each approach could complete?
2. How many unique benchmarks could the approach complete that the other one could not?

Approach	**Uniquely solved**	**Total Completed**
Formula	0	297
Automaton	581	878

Table 7.1: Total and unique completed benchmarks of finding dependency tool by approach.

In the formula approach, 703 benchmarks had out-of-memory errors and 141 had a timeout. On the other hand, in the automaton approach, 222 benchmarks had an out-of-memory error and 41 had a timeout. In the automaton approach, 33 benchmarks successfully constructed the NBA of the LTL specification but could not find the dependent variable within an hour. As indicated in the experiments results and table 7.1 shows, all the benchmarks that could be solved in the formula approach could be solved in the automaton approach but not vice versa.

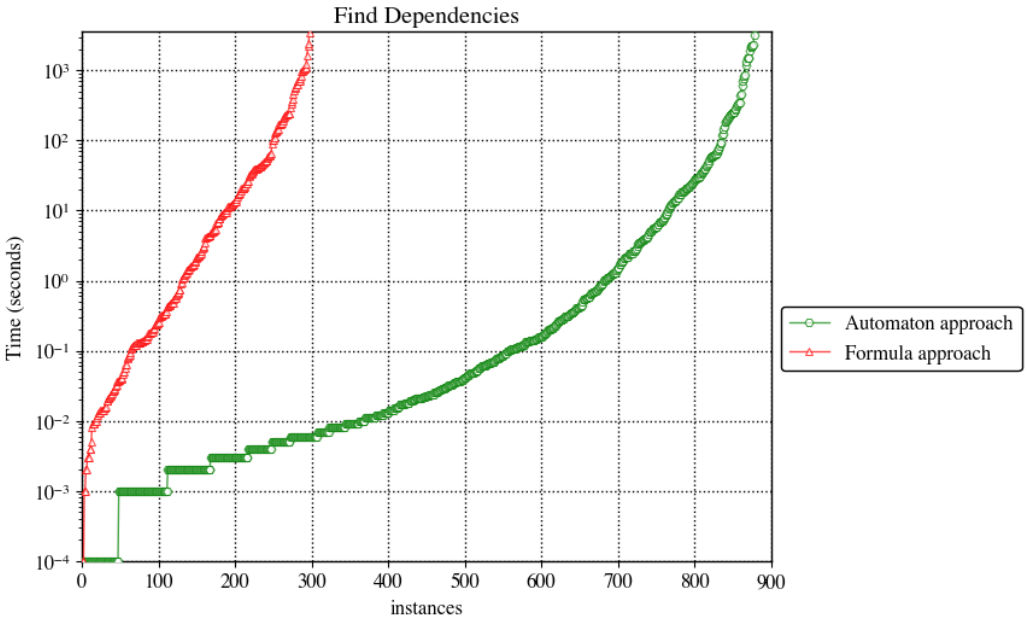


Figure 7.1: Cactus plot of finding a maximal set of dependent variables per approach.

As figure 7.1 shows the automaton approach outperforms the formula approach in terms of times and total amount of solved benchmarks. Based on all the experiment results, we can see that the automaton approach outperforms the formula approach in every aspect.

7.2 Dependency Prevalence

To answer **RQ2**, we used the experiment results from the previous section. We were able to identify 300 benchmarks out of 1,141 SYNTCOMP benchmarks, that had at least 1 dependent output variable (as per Definition 13). We found that all the benchmarks with at least 1 dependent variable belong to one of 5 benchmark families, as seen in Table 7.2. To measure the prevalence of dependency we evaluate the following:

1. Quantity of dependent variables.
2. The *dependency ratio*, we define it as the ratio between total dependent variables to total output variables, i.e. $\frac{\text{Total dependent vars}}{\text{Total output vars}}$.

Table 7.2 summarizes the dependency prevalence for 5 benchmark families, indicating the number 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.

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 7.2: Summary for dependency prevalence over 5 benchmark families.

Out of those depicted, Mux (for multiplexer) and shift (for shift-operator operator) were two benchmark families where the 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.

Figure 7.2 is a graph that indicates the cumulative count of benchmarks for each unique value of total dependent variables, where $F(x)$ on the y-axis represents how many benchmarks have at most x (on the x-axis) dependent variables.

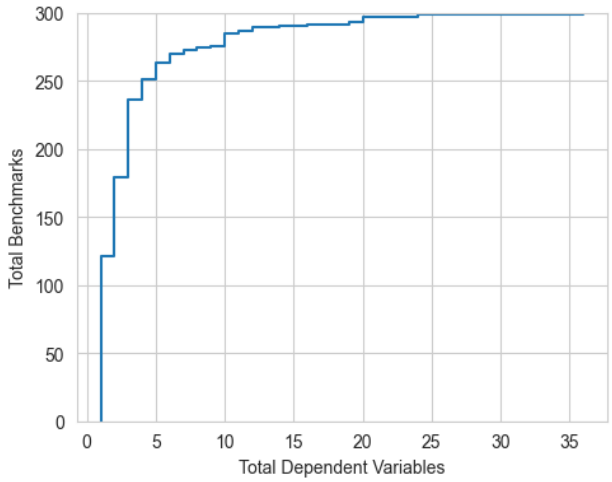


Figure 7.2: Cumulative count of benchmarks for each unique value of Total Dependent Variables.

Figure 7.3 is a plot that illustrates the cumulative count of benchmarks for each unique value of the Dependency Ratio, where the value of $F(x)$ on the y-axis represents how many benchmarks are at most x (on the x-axis) dependency ratio.

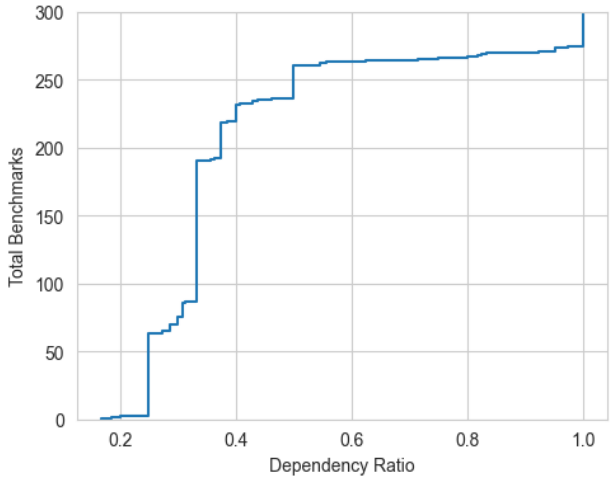


Figure 7.3: Plot illustrates the cumulative count of benchmarks for each unique value of the Dependency Ratio.

Looking beyond total dependency, among the 300 benchmarks with at least 1 dependent variable, we found a diverse distribution of dependent

variables and ratio as shown in Figures 7.2, 7.3. It shows that there are various dependent variables and dependency ratios over the benchmarks. We can see that the dependency ratios is not unified, we could not find a clear relation between the dependency ratio and performance in synthesis, but as will be discussed in section 7.3 we found a relation between synthesis performance and the total of non-dependent variables.

7.3 Dependency in Synthesis

Despite a large 1 hour timeout, 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 the 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 variable 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' [24]: (i) Ltlsynt (Spot [8]) (ii) Strix [26], where all the tools had a total timeout of 3 hours per benchmark. As part of the evaluation, we compare DepSynt with a control tool we call SpotModular, which is the same code-base and pipeline as DepSynt except we skip the phase of finding dependent variables, and all variables are classified as non-dependent automatically. Furthermore, this section analyzes different components of DepSynt, such as, how long it takes to synthesize the strategy of dependent variables and the impact of the projection of dependent variables on the BDD sizes.

Comparing with state-of-the-art tools

To answer **RQ3** and show that reactive synthesis is benefited by dependency, we compare DepSynt with two state-of-the-art tools:

(i) Ltlsynt (based on Spot) [8] with all its possible configurations: ACD, SD, DS, LAR.

(ii) Strix [26] with the configuration of BFS for exploration and FPI as parity game solver. Strix is the overall winning tool in SYNTCOMP'23.

All the tools had a total timeout of 3 hours per benchmark. As can be seen from Figure 7.4. As a first attempt we tried to correlate the performance of DepSynt relatively to other tools based on the dependency ratio, yet, we could not find such a correlation. We did find a correlation with the number of non-dependent variables. Indeed for the case of ≤ 3 non-dependent variables, DepSynt outperforms the highly optimized competition-winning tools. Even for the > 3 case, as shown in Figure 7.5, the performance of DepSynt is comparable to other tools, only beaten eventually by Strix.

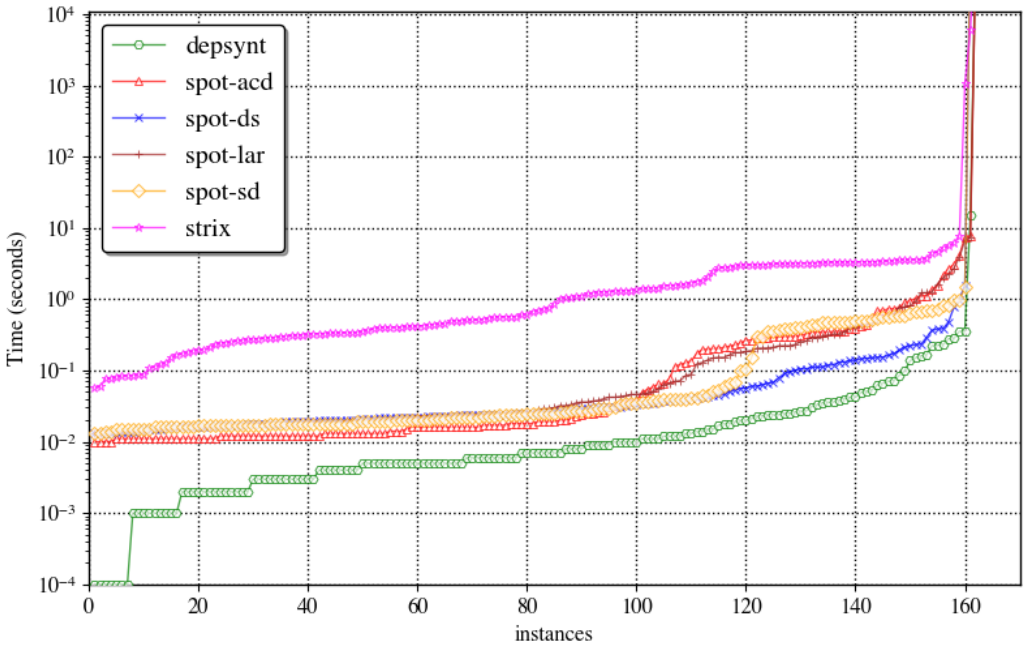


Figure 7.4: Cactus plot comparing DepSynt, LtlSynt, and Strix on 162 benchmarks with at most 3 non-dependent variables.

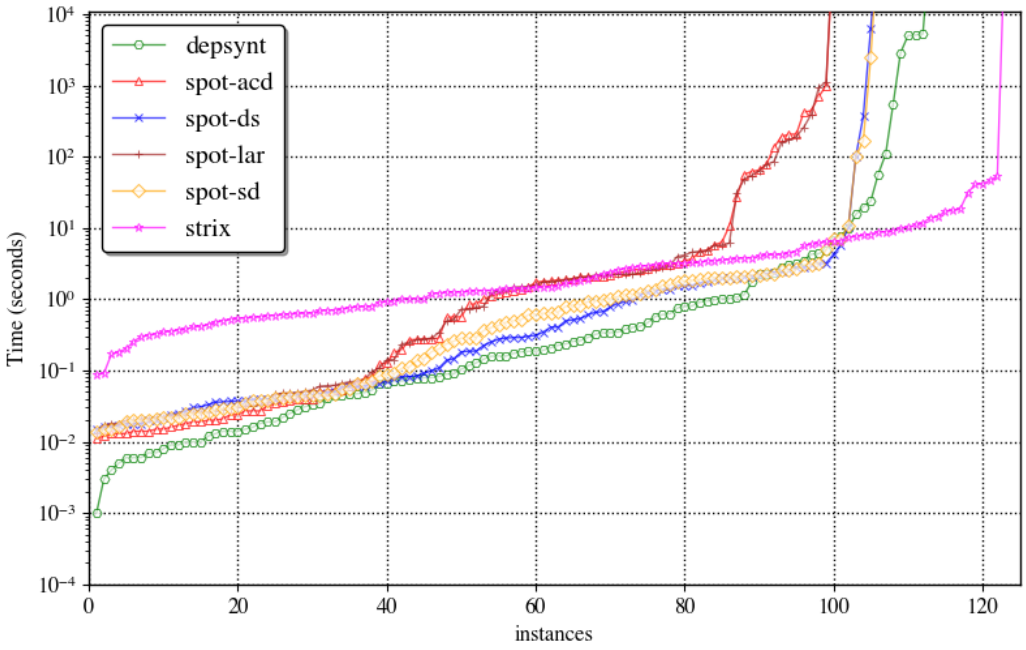


Figure 7.5: Cactus plot comparing DepSynt, LtlSynt, and Strix on 138 benchmarks with more than 3 non-dependent variables.

In addition, we observe that there are two specifications, mux32 and mux64, in the SYNTCOMP benchmarks for which both Strix and Ltlsynt timed out after 3600 seconds, but DepSynt solved these benchmarks (2 milliseconds for mux32 and 4 milliseconds for mux64 on the same computing platform). Note that all output variables are dependent on both mux32 and mux64. Table 7.3 summarizes the number of benchmarks each tool solved and the number of benchmarks it uniquely solved.

Comparing DepSynt against 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, there are 10 benchmarks for which 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. Comparing against 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.

Tool	Uniquely solved	Total Completed
DepSynt	2	273
Ltlsynt (SD)	0	264
Ltlsynt (DS)	0	264
Ltlsynt (ACD)	0	259
Ltlsynt (Lar)	0	259
Strix	5	282

Table 7.3: Summarize comparison with state-of-the-art tools over 300 benchmarks with dependency.

Analyzing time taken by different parts of the pipeline

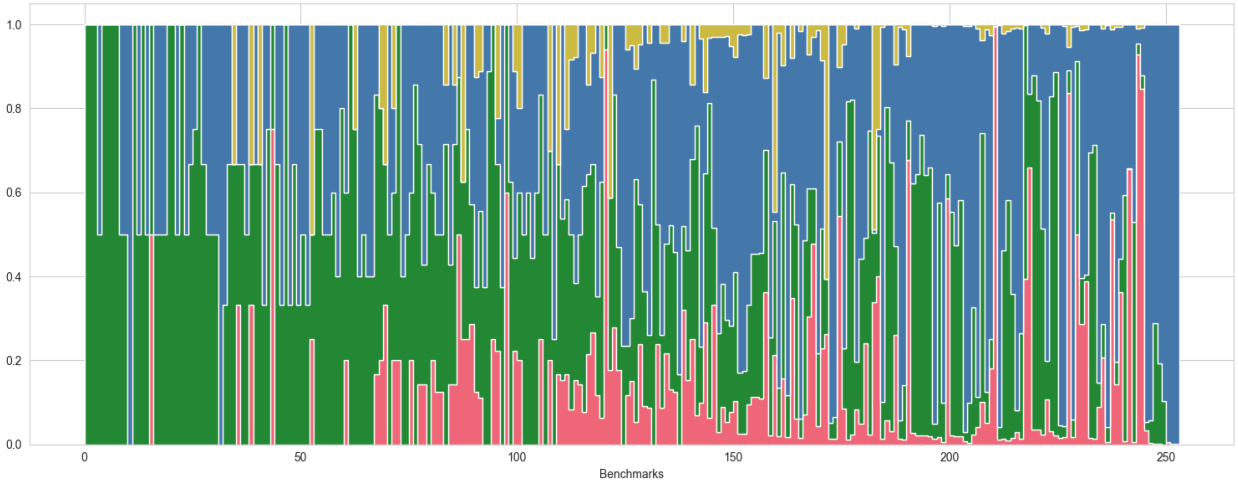


Figure 7.6: Normalized time distribution of DepSynt sorted by total duration.

In order to better understand where DepSynt spends its time, we plotted in Figure 7.6 The 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.

The pink is searching for dependency, the green is the NBA build, the blue is the non-dependent variables and the yellow is the dependent variables

synthesis. 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 region synthesizing a strategy for the non-dependent variables. 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

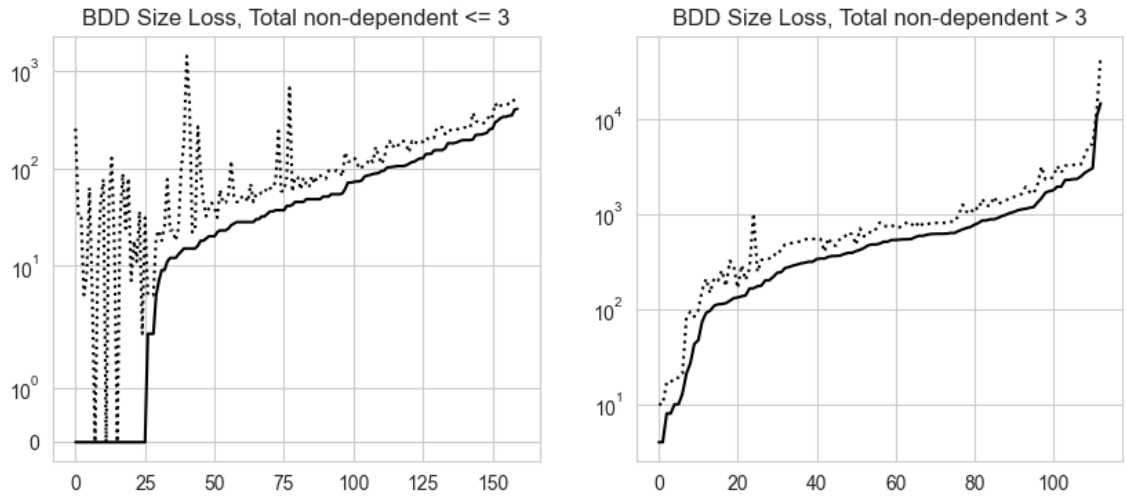


Figure 7.7: Total BDD sizes of the NBA edges before and after the projection of the dependent variables from the NBA edges.

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 7.6 shows. Figure 7.6 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 the benchmark with at most 3 non-dependent variables and the right figure is over the 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 a symmetric log scale. The benchmarks are sorted by the projected NBA's BDD total size. 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 7.7 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 3 or more, the BDD size is reduced as well.

An ablation experiment with SpotModular

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, and classifies all output variables that are classified as non-dependent, and called this SpotModular. As can be seen in Figure 7.8, 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 figure 7.9, we see that for other cases we do not see this.

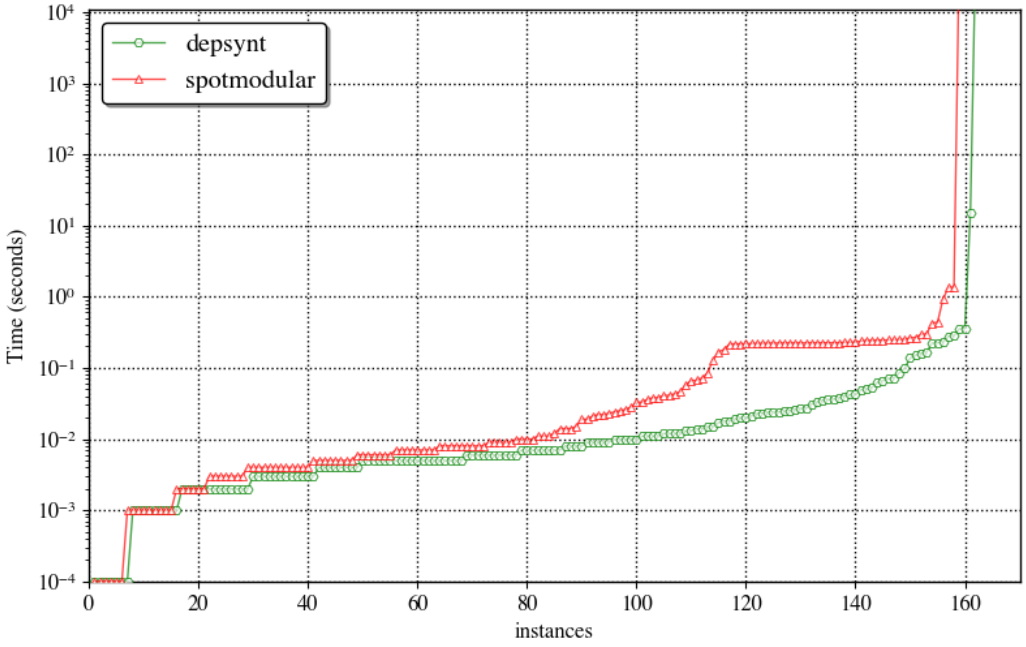


Figure 7.8: Cactus plot comparing DepSynt and SpotModular on 162 benchmarks with at most 3 non-dependent variables.

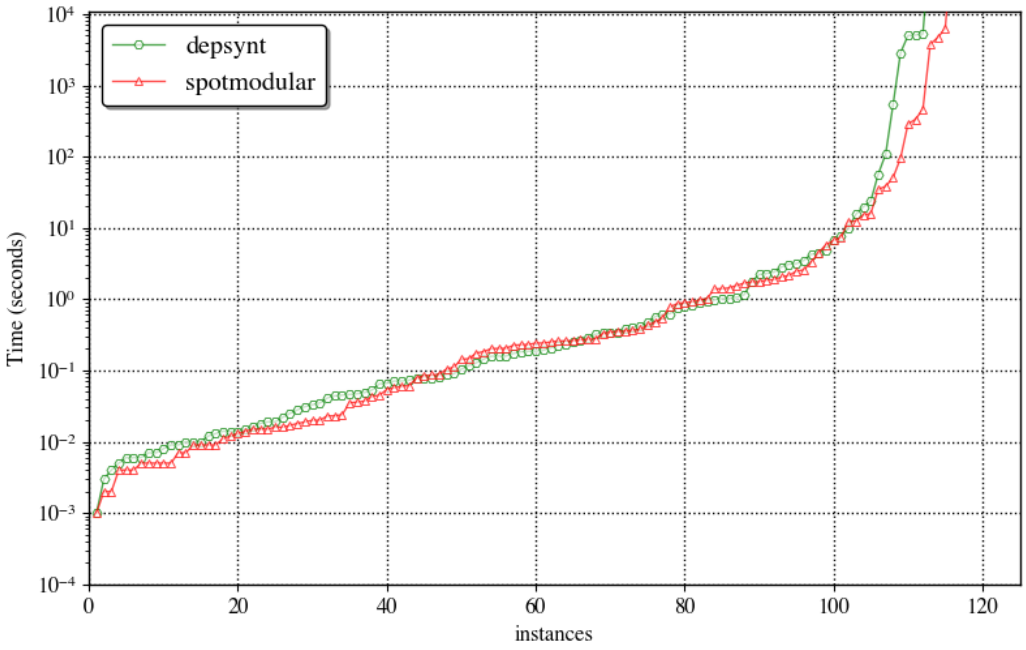


Figure 7.9: Cactus plot comparing DepSynt and SpotModular on 138 benchmarks with more than 3 non-dependent variables.

Summary

Overall, we answer both the research questions we started with. Indeed there are several benchmarks with dependent variables, and using our pipeline does give performance benefits when the number of non-dependent variables is low. In summary, our recipe would be to first run our poly-time check to see if there are dependents and if there aren't too many non-dependents, use our approach and otherwise switch to any existing method.

8 Conclusion

In this thesis, we have introduced the notion of dependent variables in LTL formulas and specifically reactive synthesis benchmarks. By using two methods: formula-based and automata-based. We managed to show that dependent variables in reactive synthesis benchmarks are prevalent. We also described a synthesis framework that utilizes the dependent variables and constructed a SPOT-based synthesis tool that implements this framework. Our evaluation showed that on some benchmarks dependent variables can be deployed for better synthesis. This work covers many aspects, starting from initial definition to characterization, algorithms, and implementations. We focused on one definition of dependent variables although the other definitions noted in the paper that are more restrictive can be of some use. We believe that similar synthesis techniques can be deployed for such variants as well. In addition, the variant of dependency that we explored was not concerned with finding the minimal set of variables upon the set X is dependent on. This task is challenging computational-wise as well and can be thought of as future work. We are also interested to learn whether formula-based synthesis may still be an approach to follow, instead of the automata one that we pursued.

8.1 Future work

This research can be developed and continued in various directions in reactive synthesis and linear temporal logic. We suggest a few topics that we think should be investigated further:

- **Formula approach** - We have introduced a naive approach for dependency based on the formula definition which did not scale well, especially compared to the automaton approach. We think this approach can be researched further and bring better results, for exam-

ple, finding dependency using the formula structure without checking emptiness, which is an EXP time problem.

- **Further dependency notion** - Additional definitions and notions of the dependency concept can be researched, which eventually can include and work on more benchmarks. For example, the ability to define a dependency on the LTL syntax would have a great benefit since it avoids constructing NBA with dependent variables and can be combined with a variety of synthesis algorithms and tools agnostically.

- **Unateness** - This research elevated the concept of dependency from the Boolean Functional Synthesis field. An additional concept that can be elevated from the Boolean Functional Synthesis field is unateness. The Unate concept is defined on Boolean variables and a Boolean formula, such that, a variable is Unate if changing its value from False to True over an assignment does not impact the satisfaction.

- **Further Applications** - This research showed an application of dependency in the field of reactive synthesis, yet, the dependency concept is defined over a linear temporal logic. Therefore, the dependency concepts can be utilized for additional fields that use linear temporal logic, for example, counting models of Linear-Time Temporal logic [20].

- **Utilization of dependency agnostically to the synthesis tool** - In this research, we showed utilization of dependency in the reactive synthesis field, but, the utilization can be improved as well. Currently, the dependency concepts can be utilized only when nondeterministic buchi automata(NBA) is synthesized which is constructed directly from the specification. Not all the synthesizing tools use NBA, for example, Strix [26], the usage of dependency in reactive synthesis should be easily applied to any reactive synthesis tool.

- **Explore heuristics for a maximal set of dependency** - This research shows an algorithm to find a maximal set of dependent variables, the set is maximal in terms of there is no larger dependency set with the same variables. Yet, the order of the tested variables in the algorithm does matter in terms of the size of the dependency set, currently, our algorithm uses an arbitrary order of the variables. The search process of dependent variables can be improved with better heuristics for the order of the variables.

Bibliography

[1] AKSHAY, S., ARORA, J., CHAKRABORTY, S., KRISHNA, S. N., RAGHUNATHAN, D., AND SHAH, S. Knowledge compilation for boolean functional synthesis. In *Formal Methods in Computer Aided Design, FMCAD* (2019).

[2] AKSHAY, S., BASA, E., CHAKRABORTY, S., AND FRIED, D. On dependent variables in reactive synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS* (2024).

[3] AKSHAY, S., AND CHAKRABORTY, S. Synthesizing skolem functions: A view from theory and practice. In *Handbook of Logical Thought in India*. 2022.

[4] AKSHAY, S., CHAKRABORTY, S., GOEL, S., KULAL, S., AND SHAH, S. What's hard about boolean functional synthesis? In *Computer Aided Verification - 30th International Conference, CAV* (2018).

[5] AKSHAY, S., CHAKRABORTY, S., GOEL, S., KULAL, S., AND SHAH, S. Boolean functional synthesis: hardness and practical algorithms. *Formal Methods Syst. Des.* (2021).

[6] AMRAM, G., BANSAL, S., FRIED, D., TABAJARA, L. M., VARDI, M. Y., AND WEISS, G. Adapting behaviors via reactive synthesis. In *Computer Aided Verification - 33rd International Conference, CAV* (2021).

[7] BIERE, A. The AIGER And-Inverter Graph (AIG) format version. Tech. rep., Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[8] BLAHOUDEK, F., DURET-LUTZ, A., AND STREJČEK, J. Can complement generalized Büchi automata via improved semi-

determinization. In *Proceedings of the 32nd International Conference on Computer-Aided Verification* (2020).

[9] BLOEM, R., CHATTERJEE, K., AND JOBSTMANN, B. Graph games and reactive synthesis. In *Handbook of Model Checking*. 2018.

[10] BLOEM, R., JOBSTMANN, B., PITERMAN, N., PNUELI, A., AND SA'AR, Y. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.* (2012).

[11] BRYANT, R. E. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)* (1995).

[12] BÜCHI, J. R. On a decision method in restricted second order arithmetic. *The Collected Works of J. Richard Büchi* (1990).

[13] CHAKRABORTY, S., FRIED, D., TABAJARA, L. M., AND VARDI, M. Y. Functional synthesis via input-output separation. In *Formal Methods in Computer Aided Design (FMCAD)* (2018).

[14] CHURCH, A. Logic, arithmetic, and automata. In *International Congress of Mathematicians* (1962).

[15] COURCOUBETIS, C., VARDI, M., WOLPER, P., AND YANNAKAKIS, M. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* (1992).

[16] DARWICHE, A. Decomposable negation normal form. *J. ACM* (2001).

[17] FAYMONVILLE, P., FINKBEINER, B., AND TENTRUP, L. Bosy: An experimentation framework for bounded synthesis. In *Computer Aided Verification: 29th International Conference, CAV* (2017).

[18] FINKBEINER, B., GEIER, G., AND PASSING, N. Specification decomposition for reactive synthesis. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings* (2021).

[19] FINKBEINER, B., AND SCHEWE, S. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.* (2013).

[20] FINKBEINER, B., AND TORFAH, H. Counting models of linear-time temporal logic. In *Language and Automata Theory and Applications* (2014).

[21] GOLIA, P., ROY, S., AND MEEL, K. S. Manthan: A data-driven approach for boolean function synthesis. *Computer Aided Verification* (2020).

[22] GOLIA, P., SLIVOVSKY, F., ROY, S., AND MEEL, K. S. Engineering an efficient boolean functional synthesis engine. *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (2021).

[23] HUTH, M., AND RYAN, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2004.

[24] JACOBS, S., BLOEM, R., COLANGE, M., FAYMONVILLE, P., FINKBEINER, B., KHALIMOV, A., KLEIN, F., LUTTENBERGER, M., MEYER, P. J., MICHAUD, T., SAKR, M., SICKERT, S., TENTRUP, L., AND WALKER, A. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results.

[25] MEALY, G. H. A method for synthesizing sequential circuits. *The Bell System Technical Journal* (1955).

[26] MEYER, P. J., SICKERT, S., AND LUTTENBERGER, M. Strix: Explicit reactive synthesis strikes back! In *Computer Aided Verification: 30th International Conference, CAV* (2018).

[27] MICHAUD, T., AND COLANGE, M. Reactive synthesis from ltl specification with spot. In *Proceedings of the 7th Workshop on Synthesis, SYNT@ CAV* (2018).

[28] PARYS, P. Parity games: Zielonka's algorithm in quasi-polynomial time. In *44th International Symposium on Mathematical Foundations of Computer Science, MFCS* (2019).

[29] PNUELI, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, sfcs* (1977).

[30] PNUELI, A., AND ROSNER, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989).

[31] RABE, M. N., AND SESHIA, S. A. Incremental determinization. In *Theory and Applications of Satisfiability Testing - SAT* (2016).

[32] RABIN, M. O., AND SCOTT, D. Finite automata and their decision problems. *IBM Journal of Research and Development* (1959).

[33] REICHL, F.-X., SLIVOVSKY, F., AND SZEIDER, S. Certified DQBF solving by definition extraction. In *Proc. of SAT* (2021).

[34] SAFRA, S. On the complexity of omega-automata. In *Proceedings of the 29th FOCS* (1988).

[35] SHAH, P., BANSAL, A., AKSHAY, S., AND 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).

[36] SOOS, M., AND MEEL, K. S. Arjun: An efficient independent support computation technique and its applications to counting and sampling. In *ICCAD* (2022).

[37] STASIO, A. D., MURANO, A., PERELLI, G., AND VARDI, M. Y. Solving parity games using an automata-based algorithm. In *Implementation and Application of Automata - 21st International Conference, CIAA* (2016).

[38] TSEITIN, G. S. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983).

[39] TURING, A. Intelligent machinery. In *Collected Works of A.M. Turing: Mechanical Intelligence*. 1992.

[40] VARDI, M., AND WOLPER, P. Reasoning about infinite computations. *Information and Computation* (1994).

[41] VARDI, M. Y., AND WOLPER, P. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences* (1986).

[42] YANG, J., CHAKRABORTY, S., AND MEEL, K. S. Projected model counting: Beyond independent support. *CoRR* (2021).

[43] ZIELONKA, W. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science* (1998).

תוכן עניינים

תקציר

עבור נוסחה לוגית טמפורלית לינארית על גבי משתני קלט/פלט, סינתזה תגובתית דורשת לתכנן מכונה דטרמניסטית הנקראת מכונת מאלי/טרנסדיוצר כך שהיא מספקת השמה למשתני הפלט בכל נקודת זמן בהינתן השמה למשתני הקלט עד אותה נקודת זמן, בצורה כזו שהנוסחה הלוגית טמפורלית מסופקת. בתזה הזו, אנו חוקרים את הרעיון של משתנים תלויים בהקשר של סינתזה תגובתית בהשראת רעיון מוצלח של משתנים תלויים בסינתזה של לוגיקה בוליאנית. אנו מגדירים את המשתנים תלויים כמשתני פלט שיש להם השמה יחידה ויחודית בכל נקודת זמן בהינתן היסטוריה של השמות עבור שאר המשתנים בנוסחה עד אותה נקודת זמן. אנו חוקרים שלוש גישות שקולות למשתנים תלויים בנוסחה לוגית טמפורלית לינארית: ספיקות של נוסחה לוגית טמפורלית לינארית, תנאים על גבי האוטומט בוקי הלא־דטרמניסטי של הנוסחה הטמפורלית, השפת־ω של הנוסחה.

תזה זו מראה שמשתנים תלויים הם תופעה נפוצה בנוסחאות טמפורלית ומציגה גישה חדשנית בסניתזה תגובתית שמשתמשת בהגדרות ואלגוריתמים של משתנים תלויים. על מנת לממש את הרעיונות האלו בצורה פרקטית, אנחנו משתמשים בספריית קוד נקראת "ספוט" המאפשר עבודה על אוטומטים, נוסחאות לוגיות טמפורליות ואלגוריתמי סינתזה וכתבנו קוד מותאם משלנו למימוש האלגוריתמים ב־C++. על פי ניסויים שהרצנו, בנוסחאות טמפורליות עם מספר מועט של משתנים בלתי־תלויים ־ לכל היותר 3, השיטה שלנו פועלת בצורה טובה יותר מהכלים העדכניים ביותר בתחום הסינתזה התגובתית. תזה זו תורמת הן בפן התיאורטי, שבו מוצגים הגדרות, רעיונות ואלגוריתמים חדשניים בצורה מדוקדקת, והן בפן המעשי, שבו פיתחנו כלי המממש את האלגוריתמים האלו ובפועל מוביל במדדים במקרים מסויימים לעומת הכלים הקיימים.

האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

משתנים תלויים בלוגיקה לינארית טמפורלית וסינתזה תגובתיות

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר
"מוסמך למדעים" .M.Sc במדעי המחשב
באוניברסיטה הפתוחה
המחלקה למתמטיקה ומדעי המחשב

על־ידי
אליהו בסה

בהנחיית ד"ר דרור פריד

דצמבר 2023