Language Equation Solving via Boolean Automata Manipulation

Wan-Hsuan Lin[1,3], Chia-Hsuan Su[2], and Jie-Hong R. Jiang[1,2]
[1]Department of Electrical Engineering, National Taiwan University, Taiwan
[2]Graduate Institute of Electronics Engineering, National Taiwan University, Taiwan
[3]Department of Computer Science, University of California, Los Angeles, USA

ABSTRACT

Language equations are a powerful tool for compositional synthesis, modeled as the unknown component problem. Given a (sequential) system specification S and a fixed component F, we are asked to synthesize an unknown component X such that whose composition with F fulfills S. The synthesis of X can be formulated with language equation solving. Although prior work exploits partitioned representation for effective finite automata manipulation, it remains challenging to solve language equations involving a large number of states. In this work, we propose variants of Boolean automata as the underlying succinct representation for regular languages. They admit logic circuit manipulation and extend the scalability for solving language equations. Experimental results demonstrate the superiority of our method to the state of the art in solving the unknown component problem.

1 INTRODUCTION

Synthesis through composition is an effective design principle to cope with the ever-increasing system complexity. Compositional synthesis can be cast as the *unknown component problem* [19]: Given a system specification S and a pre-designed module F, we are asked to synthesize the unknown component X such that composing F and X fulfills S as illustrated in Figure 1.[1] This problem can be formulated by the *language equation*:

$$F \bullet X \subseteq S, \tag{1}$$

where $\bullet$ denotes some composition operation. In this work, we focus on the most common synchronous composition [21], which has practical applications in sequential circuit optimization as the equation characterizes the notion of complete sequential flexibility (CSF) of a sub-circuit to be simplified in a design.

According to [21], the solution to Eq. (1) can be derived by

$$X = \overline{[F(i,o,u,v) \cdot (\overline{S}(i,o))_{\uparrow i,o,u,v}]_{\downarrow u,v}}, \tag{2}$$

where "$\cdot$" denotes language intersection, the overline denotes language complement, and "$\uparrow$" and "$\downarrow$" denote lifting and projection operations, respectively, to be detailed later. Essentially, all the operations can be done through finite automata manipulation. However, its computation difficulty stems from the requirement of complementing a nondeterministic finite automata (NFA), which may incur exponential increase in the state number due to the powerset construction.

Unlike patching software errors, hardware design errors can hardly be rectified. Therefore, circuit designers have been very conservative in adopting sequential circuit optimization in industrial designs due to its high verification complexity. While most existing logic synthesis methods focus on combinational optimization, only relatively few focus on sequential system optimization [18]. In [13], binary decision diagrams (BDDs) are exploited to support NFA manipulation for solving X of Eq. (2). Recent work [1] proposes flanked finite automata (FFA), which is a subset of NFA, to compute quotient and inclusion of regular language efficiently. However, some essential operations to solve Eq. (2) are not supported, e.g., lifting, projection, and input progressive operations.

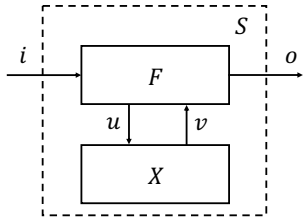

Figure 1: Unknown X under synthesis to be composed with F to fulfill specification S.

The current state-of-the-art approach [13, 18] of language equation solving utilizes a partitioned BDD representation to alleviate the memory blow-up problem of monolithic BDD representation. However, the scalability is still limited. Hence, it remains vital to develop scalable approaches to solving language Eq. (2) involving systems of large state spaces. As And-Inverter Graphs (AIGs) [14] are scalable data structures widely applied in logic synthesis and verification applications, they can be more scalable than BDDs. Inspired by logic circuits representation of NFA for string constraint solving [20], in this work we exploit logic circuits for scalable language equation solving.

Moreover, inspired by [7] using Boolean automata (BA) [4] representation in model checking for regular language constraints, in this work we exploit BA to overcome the complement problem in language equation solving. Essentially, BA offer more succinct representation than NFA in supporting language manipulation due to two characteristics. First, the one-hot state encoding of a BA allows the subset of states being expressed. Second, the transition function of a BA can encode not only existential, as in an NFA, but also universal nondeterminisms in a backward deterministic manner [5, 16]. Thereby, a BA can have exponentially fewer states than an NFA in representing the same regular language [11, 12]. However, to make BA suitable for language equation solving, we have to modify and extend BA in two ways: First, we take advantage of reversed BA (rBA), which accept the reversed language of BA, to allow a sequential circuit representation for regular language manipulation. Second, we propose reversed alternating Boolean automata (rABA), which generalize rBA to accommodate nondeterministic

[1]Note that the unknown component is not necessarily an embedded component. That is, it can have its own inputs and outputs by making them as extra inputs and outputs of the fixed components, respectively, such that these inputs feed directly to u and these outputs are directly driven by v.

universal or existential transitions. This extension is crucial to support nondeterminism arisen from the projection operation and to avoid superset construction in complement operation. We further show how language equation operations can be achieved with rBA and rABA. As rBA and rABA can be represented with sequential circuits, their manipulation can be done scalably. Experimental results show superior performance of the proposed method to the prior approach. Unlike recent developments in language equation solving that remain primarily theoretical [18], we addresses the computation efficiency for practical applications.

The rest of this paper is organized as follows. Section 2 provides the backgrounds of Boolean automata and language operations. Section 3 defines the reversed Boolean automata and presents the conversion between sequential circuits and rBA. Section 4 defines the reversed alternating Boolean automata. Section 5 details language equation solving with the proposed automata representations. Section 6 shows the experimental results, and Section 7 concludes this paper.

2 PRELIMINARIES

2.1 Finite Automata and Boolean Automata

A *nondeterministic finite automaton* [17] A is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A string $\hat{\sigma} = \sigma_1\sigma_2\cdots\sigma_k$, for $\sigma_i \in \Sigma$, is accepted by A if $Q_i = \delta(q_{i-1}, \sigma_i)$ with $q_{i-1} \in Q_{i-1}$ ($Q_0 = \{q_0\}$) and $i = 1, \cdots, k$, and $Q_k \cap F \neq \emptyset$. The set of strings accepted by A is called the language of A, denoted $L(A)$. A language that can be accepted by some finite automata is recognized as a *regular language*. In the sequel, the considered languages are regular languages. Let A' be an automaton with $L(A') = \overline{L(A)}$, where the overline denotes language complement. It is well known that A' can be built from A by the superset construction method, but suffers from an exponential blow-up in the state set. To overcome the complement problem, the variants, *alternating finite automata* (AFA) [5] and *Boolean automata* (BA) [4], are proposed for succinct regular language representation and manipulation.

A *Boolean automaton* [4] A is a 5-tuple $(Q, \Sigma, \delta, f^0, F)$, where $Q = \{q_1, q_2, \ldots, q_n\}$ is the finite nonempty set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \rightarrow \mathcal{B}_Q$, for $\mathcal{B}_Q$ being the set of Boolean functions $\{f(q_1, \ldots, q_n)\}$ over variables $q_1, \ldots, q_n$ (by abusing the notation let variable $q_i = 1$ indicate the presence of q_i), is the transition function, f^0 is the initial function in $\mathcal{B}_Q$, and $F \subseteq Q$ is the set of final states. Let δ be extended from domain $Q \times \Sigma$ to domain $\mathcal{B}_Q \times \Sigma^*$ by defining

$$\begin{aligned} \delta(f, \epsilon) &= f, \\ \delta(f, a) &= f(\delta(q_1, a), \delta(q_2, a), \ldots, \delta(q_n, a)), \\ \delta(f, a\hat{\sigma}) &= \delta(\delta(f, a), \hat{\sigma}), \end{aligned}$$

for $f \in \mathcal{B}_Q$, ϵ denoting the empty string, $a \in \Sigma$, and $\hat{\sigma} \in \Sigma^*$. In the sequel s^0 denotes the vector $(F(q_1), \ldots, F(q_n))$, for $F(q_i) = 1$ if $q_i \in F$ and $F(q_i) = 0$ otherwise. A string $\hat{\sigma} \in \Sigma^*$ is accepted by BA A if $\delta(f^0, \hat{\sigma})(s^0) = 1$. Note that Boolean automata are backward deterministic. The initial state values are defined by s^0, the final state set of the Boolean automaton. When an input sequence $\sigma_1 \ldots \sigma_k$ is read from the last input σ_k to the first input σ_1, every input σ_i uniquely determines the transition function $\delta(q_j, \sigma_i)$ for each state variable q_j to update its value. After finishing reading inputs, the acceptance of inputs is determined by evaluating the initial function f^0 with the current state values.

Table 1: Transition table of example BA.

	a	b
q_1	1	q_1
q_2	$q_1 \vee \neg q_2$	1

When AFA and BA are compared, we note that the former is a BA with the initial function being restricted to a projection function, i.e., $f^0(q_1, \ldots, q_n) = q_i$ for some q_i. On the other hand, when NFA and BA are compared, unlike an NFA with only existential nondeterministic transitions, a BA can encode both universal and existential nondeterminisms. For an NFA, the transition function maps a state and an input to a subset of states, which in the BA notation corresponds to a disjunction of state variables. In contrast, for a BA, the current state set can be expressed by any Boolean function over the state variables. Hence, an NFA is a BA with only disjunction in transition functions and the initial function being the initial state. The generality of BA makes its state number exponentially smaller than that of NFA representing the same regular language [11, 12]. Essentially, converting an n-state BA (AFA) to an equivalent deterministic finite automaton (DFA) and NFA in the worst case requires 2^{2^n} and $2^n + 1$ states, respectively [5, 8].

Example 1. *Consider the BA $A = (Q, \Sigma, \delta, f^0, F)$, with $Q = \{q_1, q_2\}$, $\Sigma = \{a, b\}$, δ specified in Table 1, $f^0 = q_1 \wedge \neg q_2$, and $F = \emptyset$. Assume we intend to determine whether input string ab is accepted.*

To verify whether A accepts ab, we check if $\delta(f^0, ab)(s^0) = 1$. There are two strategies for the checking. We can read the input string in a forward or backward manner. With forward reading, we check $\delta(q_1 \wedge \neg q_2, ab)(s^0)$ by substituting state transition functions to the initial function and evaluating the resulted function by s^0. That is,

$$\begin{aligned} &\delta(q_1 \wedge \neg q_2, ab)(s^0) \\ &= \delta(\delta(q_1, a) \wedge \neg\delta(q_2, a), b)(s^0) \\ &= \delta(1 \wedge \neg(q_1 \vee \neg q_2, b)(s^0) \\ &= \delta(\neg q_1 \wedge q_2, b)(s^0) \\ &= \delta(\neg\delta(q_1, b) \wedge \delta(q_1, b), b)(s^0) \\ &= \delta(\neg q_1 \wedge 1)(00) \\ &= 1, \end{aligned}$$

which indicates ab is accepted.

On the other hand, with backward reading, we check $\delta(q_1 \wedge \neg q_2, ab)(s^0)$ by updating the state values and evaluating the initial function by the resulted state values. That is,

1. *Initially, we have state value $s^0 = (0, 0)$.*
2. *After reading b, we have $s^1 = (\delta(q_1, b)(s^0), \delta(q_2, b)(s^0)) = (0, 1)$.*
3. *After reading a, we have $s^2 = (\delta(q_1, a)(s^1), \delta(q_2, a)(s^1)) = (1, 0)$.*
4. *By evaluating f^0 with s^2, we have $f^0 = 1 \wedge \neg 0 = 1$.*

According to the computation above, A accepts string ab.

2.2 Language Operations

The language equation for the unknown component problem [19] involves language operations, including *complement*, *intersection*, *lifting*, *projection*, *prefix-close*, and *input-progressive* as we define below.

2.2.1 Complement Operation. Given a regular language L over an alphabet Σ, the complement of L, denoted $\overline{L}$, is

$$\text{Complement}(L) = \Sigma^* \setminus L.$$

2.2.2 Intersection Operation. Given regular languages L_1 and L_2, the intersection language of L_1 and L_2, denoted $L_1 \cdot L_2$, is

$$\text{Intersection}(L_1, L_2) = L_1 \cap L_2.$$

2.2.3 Lifting Operation. Given a regular language L over an alphabet Σ and an alphabet Ξ, the lifting of language L to $\Sigma \times \Xi$, denoted $L_{\uparrow(\Sigma,\Xi)}$, is

$$\text{Lifting}(L, \Sigma, \Xi) = \{(\sigma_1, \xi_1)(\sigma_2, \xi_2) \cdots (\sigma_k, \xi_k) | \sigma_1\sigma_2 \cdots \sigma_k \in L, \xi_i \in \Xi\}.$$

2.2.4 Projection Operation. Given a regular language L over alphabet $\Sigma \times \Xi$, the projection of language L to Ξ, denoted $L_{\downarrow\Xi}$, is

$$\text{Projection}(L, \Xi) = \{\xi_1\xi_2 \cdots \xi_k \mid (\sigma_1, \xi_1)(\sigma_2, \xi_2) \cdots (\sigma_k, \xi_k) \in L\}.$$

2.2.5 Prefix-Close Operation. A regular language L over Σ is called *prefix-closed* if

$$\forall \hat{\sigma} \in L,\ \text{Prefix}(\hat{\sigma}) \subseteq L,$$

where $\text{Prefix}(\hat{\sigma}) = \{\hat{\alpha} \in \Sigma^* \mid \exists \hat{\beta} \in \Sigma^* \text{ such that } \hat{\alpha}\hat{\beta} = \hat{\sigma}\}$. Given a regular language L, the largest prefix-closed sub-language of L is

$$\text{PrefixClose}(L) = \bigcup_{\text{prefix-closed } L' \subseteq L} L'.$$

2.2.6 Input-Progressive Operation. A regular language L over alphabet $\Sigma \times \Xi$ is called Σ-*progressive* if, for any string $\hat{\alpha} \in L$, the concatenation $\hat{\alpha}(\sigma, \xi)$ is also in L for every $\sigma \in \Sigma$ and some $\xi \in \Xi$. Given a regular language L over alphabet $\Sigma \times \Xi$, the largest Σ-progressive sub-language of L is denoted $\text{Progressive}(L, \Sigma)$.

3 REVERSED BOOLEAN AUTOMATA AND LOGIC CIRCUIT REPRESENTATION

Inspired by the bit-wise representation of reversed alternating finite automata (r-AFA) [16], for efficient logic circuit representation, we introduce reversed Boolean automata, which accept the reversed languages of BA, as follows. A *reversed Boolean automaton* (rBA) is a 5-tuple $(Q, \Sigma, \delta, f^0, F)$, same as a BA except for the following difference. For a state variable valuation $s \in \mathbb{B}^{|Q|}$ and an input $a \in \Sigma$, we extend the notation of δ and let $\delta(s, a)$ denote $(\delta(q_1, a)(s), ..., \delta(q_n, a)(s))$. For $\hat{\sigma} \in \Sigma^*$, we have

$$\begin{aligned} \delta(s, \epsilon) &= s, \\ \delta(s, \hat{\sigma}a) &= \delta(\delta(s, \hat{\sigma}), a). \end{aligned}$$

Therefore, an rBA accepts a string $\hat{\sigma}$ if $f^0(\delta(s^0, \hat{\sigma})) = 1$. Unlike a BA that expresses the current state set by a Boolean function, an n-state rBA express the current state set by a bit-vector $(q_1, \ldots, q_n)$, where $q_i = 1$ (0) indicates that state q_i is (not) in the current state set. This bit-vector representation of the current state set naturally corresponds to a sequential circuit representation for efficient rBA manipulation.

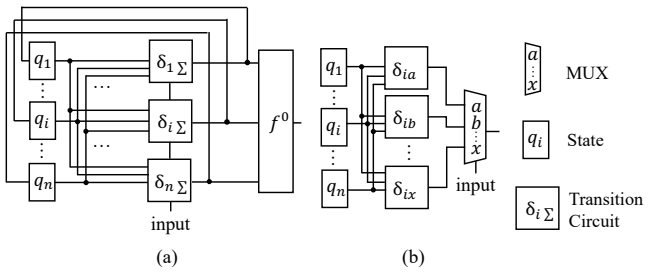

Figure 2: (a) Sequential circuit representation of rBA. (b) Transition function circuit $\delta_{i\Sigma}$ of state q_i.

The logic circuit of an rBA is of the form shown in Figure 2. The box labelled $\delta_{i\Sigma}$ in Figure 2(a) denotes the transition function for state q_i under a given input, and its detailed realization with a multiplexer for selection with respect to the input value is shown in Figure 2(b). The box f^0 in Figure 2(a) implements the initial function f^0. The boxes labelled $q_1, \ldots, q_n$ are latches that store the current state values and are updated according to the transition functions. The latches are initialized to values s^0. The circuit outputs 1 for an input string if and only if the underlying rBA accepts the input string.

In addition to the above rBA to sequential circuit conversion, we may extract an rBA from a sequential circuit, not necessary in the form of Figure 2, as follows. Consider a sequential circuit $C = (I, O, L, L_F, f_L, f_O)$ with input alphabet $I = \mathbb{B}^{|\vec{x}|}$, i.e., the set of valuations of the input variables $\vec{x}$, output alphabet O, i.e., the set of valuations of the output variables, the set of latches $L = \{l_1, \ldots, l_n\}$, the subset of latches $L_F \subseteq L$ whose initial values are 1, the set of transition functions $f_L = \{f_{l_1}, \ldots, f_{l_n}\}$, where $f_{l_j} : I \to \mathcal{B}_{\mathcal{L}}$, for $\mathcal{B}_{\mathcal{L}}$ being the set of Boolean functions over variables $\vec{l} = (l_1, \ldots, l_n)$, and the set of output functions $f_O = \{f_{o_1}, \ldots, f_{o_k}\}$, where $f_{o_j} : I \to \mathcal{B}_{\mathcal{L}}$.[2] We can construct an rBA $A = (Q, \Sigma, \delta, f^0, F)$ with $\Sigma = I \times O$ such that $L(A)$ describes the behavior of C as follows. Let $Q = L \cup \{q'\}$ for q' being a fresh new variable. Let q' have value 1 after reading in string $\hat{\sigma} \in \Sigma^*$ if and only if string $\hat{\sigma}$ is a valid input-output sequence for C. Therefore, we set $f^0 = q'$ and $F = L_F \cup \{q'\}$. For $i \in I$, $o \in O$, we define the transition function as

$$\delta(q, (i, o)) = \begin{cases} q \wedge \bigwedge_j (o[o_j] \equiv f_{o_j}(i)), & \text{if } q = q', \\ f_{l_j}(i), & \text{if } q = l_j, \end{cases}$$

where $o[o_j]$ denotes the o_j-bit value in the output value vector o. Then, the tuple of A is $(L \cup \{q'\}, I \times O, \delta, q', L_F \cup \{q'\})$. Note that given a sequential circuit with input alphabet I, the language

[2] Notice that the transition function $f_{l_j} : L \times I \to \mathbb{B}$ of latch l_j can be alternatively viewed as a mapping from an input $i \in I$ to the Boolean function $f_{l_j}(\vec{l}, \vec{x})|_{\vec{x}=i}$, i.e., the function resulting from substituting $\vec{x}$ with i in $f_{l_j}(\vec{l}, \vec{x})$. Similarly, the output function $f_{o_j} : L \times I \to \mathbb{B}$ can be alternatively viewed as a mapping from an input $i \in I$ to a Boolean function $f_{o_j}(\vec{l}, \vec{x})|_{\vec{x}=i}$.

that describes the behavior of the circuit is prefix-closed and I-progressive.

4 REVERSED ALTERNATING BOOLEAN AUTOMATA

Motivated by the fact that certain language operations, such as projection, may result in nondeterministic transitions, we generalize rBA to accommodate both existential and universal nondeterminisms. Thereby, its complement can be done without determinization to circumvent the state-explosion problem. A *reversed alternating Boolean automaton (rABA)* A is a 7-tuple $(Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$, where Q is a finite set of states, Σ is the input alphabet, γ is the number of nondeterministic transitions for giving an input, $\Delta : \Sigma \to \{\forall, \exists\}$ maps an input to a type of nondeterminism, δ is the transition relation with $\delta(q, a) \in \mathcal{B}_Q^\gamma$, $f^0 \in \mathcal{B}_Q$ is the initial function, and $F \subseteq Q$ is the final state set. If different inputs have different numbers of nondeterministic transitions, we let the γ parameter be the maximal number of nondeterministic transitions among all $a \in \Sigma$, and padding $\delta(q, a)$ with the last transition until the number of the transitions is γ, $\forall q \in Q, a \in \Sigma$. Note that the γ parameter is used to ensure that the number of nondeterministic state transitions is the same over all states under different inputs.

For $Q = \{q_1, q_2, ..., q_n\}$, $s \in \mathbb{B}^{|Q|}$, and $\delta(q_i, a) = (f_{i,1}, \ldots, f_{i,\gamma})$, let $\delta(s, a) = \{(f_{1,i}(s), \ldots, f_{n,i}(s)) \mid i = 1, \ldots, \gamma\}$. We define how A recognizes a pair of a state vector $s \in \mathbb{B}^{|Q|}$ and a string as follows. First, A recognizes (s, ϵ) if $f^0(s) = 1$. Second, for $a \in \Sigma$, $\hat{\sigma} \in \Sigma^*$, and $\Delta(a) = \forall$, A recognizes $(s, a\hat{\sigma})$ if A recognizes every $(s', \hat{\sigma})$ where $s' \in \delta(s, a)$. Third, for $a \in \Sigma$, $\hat{\sigma} \in \Sigma^*$, and $\Delta(a) = \exists$, A recognizes $(s, a\hat{\sigma})$ if A recognizes some $(s', \hat{\sigma})$ where $s' \in \delta(s, a)$. Lastly, A accepts $\hat{\sigma}$ if A recognizes $(s^0, \hat{\sigma})$.

By the above definition, the transitions of state vectors of A with respect to an input sequence $\hat{\sigma}$ can be visualized as a tree $T_A(\hat{\sigma})$. Each tree node is labeled with the current state vector, and the root is labeled with s^0. For string $\hat{\sigma} = \sigma_1\sigma_2 \ldots \sigma_k$ and a node s at level i, the children of s are $\delta(s, \sigma_i)$. Let $T_A(\hat{\sigma}).nodes$ and $T_A(\hat{\sigma}).leaves$ denote nodes and leaves of the tree, respectively. Considering string $\hat{\sigma}$, we define $T_A(\hat{\sigma}).accepted$ to be the set of nodes labeled with the recognized state vectors. The node s at level i can be added to $T_A(\hat{\sigma}).accepted$ if one of the following conditions is satisfied:

- $s \in T_A(\hat{\sigma}).leaves$ and $f^0(s) = 1$.
- For $\Delta(\sigma_{i+1}) = \exists$, at least one child of s is in $T_A(\hat{\sigma}).accepted$.
- For $\Delta(\sigma_{i+1}) = \forall$, all children of s are in $T_A(\hat{\sigma}).accepted$.

$\hat{\sigma} \in L(A)$ if and only if $s^0 \in T_A(\hat{\sigma}).accepted$. Note that the tree is a complete γ-ary tree whose depth is the length of the input string.

The following example illustrates how rABA works.

Example 2. *Consider the rABA $A = (Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$, with $Q = \{q_1, q_2\}$, $\Sigma = \{a, b\}$, $\gamma = 2$, $\Delta(a) = \exists, \Delta(b) = \forall$, δ specified in Table 2, $f^0 = q_1 \wedge q_2$, and $F = \emptyset$. Assume we intend to determine whether string ab is accepted.*

To verify whether A accepts ab, we check if A recognizes $(s^0, ab) = (00, ab)$. First, we have $\delta(00, a) = 11|01$, where "$s_1|s_2$" denotes existential nondeterministic branching into state vectors s_1 and s_2. Then, we check whether A recognizes $(11, b)$ or $(01, b)$. Since $\delta(01, b) = 11\&01$, where "$s_1\&s_2$" denotes universal nondeterministic branching into state vectors s_1 and s_2, and $f^0(0, 1) = 0$, A does not recognizes $(01, b)$. For $\delta(11, b) = 11\&11$ and $f^0(1, 1) = 1$, A recognizes $(11, b)$. Therefore, A accepts string ab. The corresponding transition tree $T_A(ab)$ is shown in Figure 3, where nodes in $T_A(ab).accepted$ are doubly circled.

Table 2: Transition table of example rABA.

	a	b
q_1	$(1, q_1)$	$(1, q_1)$
q_2	$(q_1 \vee \neg q_2, 1)$	$(1, 1)$

figure/tree1.pdf

Figure 3: Transition tree for example rABA under string ab. $T_A(ab)$. The nodes $s \in T_A(ab).accepted$ are doubled circled.

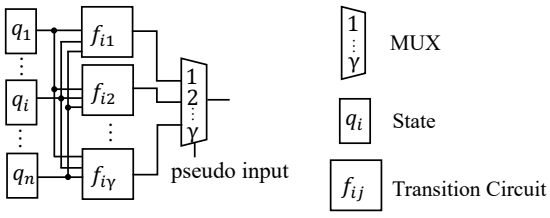

Figure 4: Circuit for nondeterministic Boolean function choices of transition function $\delta_{i,a}$ of state q_i under input a.

Similar to the logic circuit representation of rBA, we can represent rABA with a logic circuit. Specifically, the nondeterministic choices of transition function $\delta_{i,a}$ for state q_i under input a can be represented by the circuit shown in Figure 4, where the multiplexer selects a function in $\{f_{i,1}, \ldots f_{i,\gamma}\}$ according to the value $j \in \{1, \ldots, \gamma\}$ of the pseudo-input. Also the pseudo-input is quantified according to the type of nondeterminism $\Delta(a)$.

4.1 Conversion from Reversed Alternating Boolean Automata to Reversed Boolean Automata

For the purpose of language equation solving, an rABA has existential or universal nondeterminism, but not both. Under this

assumption, Algorithm 1 shows the steps determinizing an rABA to an equivalent rBA. The construction in a way is a superset construction because the state set of the A_D consists of all valuations of the state set of A, i.e., $|Q_D| = 2^{|Q|}$. We note that this procedure is not required in our language solving for X derivation, but can be used in verifying the validity of X.

Algorithm 1 *rABA_Determinize(A)*

Input: rABA $A = (Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$, where $\Delta(a)$ is the same for all $a \in \Sigma$
Output: rBA $A_D = (Q_D, \Sigma, \delta_D, f_D^0, F_D)$ such that $L(A_D) = L(A)$
1: $Q_D = \{q_s \mid s \in \mathbb{B}^{|Q|}\}$
2: $\delta_D(q_s, a) = \bigvee_{s \in \delta(s',a)} q_{s'}$.
3: **if** $for\ each\ a \in \Sigma, \Delta(a) = \forall$
4: $\quad f_D^0 = \neg(\bigvee_{f^0(s)=0} q_s)$
5: **else if** $for\ each\ a \in \Sigma, \Delta(a) = \exists$
6: $\quad f_D^0 = \bigvee_{f^0(s)=1} q_s$
7: $F_D = \{q_{s^0}\}$, where $s^0 = (F(q_1), \ldots, F(q_n))$.
8: **return** $A_D = (Q_D, \Sigma, \delta_D, f_D^0, F_D)$

The following proposition is useful for showing the correctness of Algorithm 1.

Proposition 1. *Given an rABA A with only one kind of transition property and the corresponding rBA A_D constructed by Algorithm 1, $\delta_D(s_D^0, \hat{\sigma})[q_s] = 1$ if and only if $s \in T_A(\hat{\sigma}).leaves$, where state vector $s \in \mathbb{B}^{|Q|}$, $q \in Q$, and $s[q_i]$ denotes the value of q_i in the state vector s.*

Proof. We prove the proposition by induction on the length of $|\hat{\sigma}|$. The base case $|\hat{\sigma}| = 0$ is trivial by the definition. For the induction step that $|\hat{\sigma}| = k$, we assume $\delta_D(s_D^0, \hat{\sigma})[q_s] = 1 \Leftrightarrow s \in T_A(\hat{\sigma}).leaves$. We consider the case $\hat{\sigma} = \hat{\sigma}'a$ with $|\hat{\sigma}'| = k$. By definition, we have

$$s \in T_A(\hat{\sigma}).leaves \Leftrightarrow \exists s' \in T_A(\hat{\sigma}').leaves,\ \delta(s', a) = s,$$

and

$$\delta_D(s_D^0,\hat{\sigma})[q_s] = 1 \Leftrightarrow$$
$$\exists s'' \in \mathbb{B}^{|Q_D|},\ \delta(s'', a) = s \text{ and } \delta_D(s_D^0, \hat{\sigma}')[q_{s''}] = 1.$$

Therefore, we can prove $\delta_D(s_D^0, \hat{\sigma})[q_s] = 1 \Leftrightarrow s \in T_A(\hat{\sigma}).leaves$ by showing

$$\exists s' \in T_A(\hat{\sigma}').leaves,\ \delta(s', a) = s \Leftrightarrow$$
$$\exists s'' \in \mathbb{B}^{|Q_D|},\ \delta(s'', a) = s \text{ and } \delta_D(s_D^0, \hat{\sigma}')[q_{s''}] = 1.$$

The equation is true by induction hypothesis, leading to $\delta_D(s_D^0, \hat{\sigma})[q_s] = 1 \Leftrightarrow s \in T_A(\hat{\sigma}).leaves$. □

With the above proposition, the correctness of Algorithm 1 can be established in the following theorem.

Theorem 1. *Given an rABA $A = (Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$ with only universal or existential transitions, we can construct an rBA A_D by Algorithm 1 such that $L(A_D) = L(A)$.*

Proof. We show the correctness of two cases separately. Consider the first case for rABA with only universal nondeterminism. Since all transitions of A are universal, $\hat{\sigma}$ is accepted by A if and only if $T_A(\hat{\sigma}).leaves \subseteq T_A(\hat{\sigma}).accepted$. According to Proposition 1 and the fact that $\hat{\sigma}$ is accepted by A_D if and only if $f_D^0(\delta_D(s_D^0, \hat{\sigma})) = 1$, we have $L(A_D) = L(A)$.

Consider the second case for rABA with only existential nondeterminism. Since all the transitions of A are existential, $\hat{\sigma}$ is accepted by A if and only if $\exists s \in T_A(\hat{\sigma}).leaves, s \in T_A(\hat{\sigma}).accepted$. According to Proposition 1 and the fact that $\hat{\sigma}$ is accepted by A_D if and only if $f_D^0(\delta_D(s_D^0, \hat{\sigma})) = 1$, we have $L(A_D) = L(A)$. □

Algorithm 2 *LangEqSolveFA(S,F)*

Input: NFA S with alphabet $I \times O$ and F with alphabet $I \times O \times U \times V$
Output: the largest prefix-closed and input-progressive solution X in DFA
1: S := Complete(S)
2: S := Determinize(S)
3: S := Complement(S)
4: S := Lifting($S, I \times O, U \times V$)
5: X := Intersection(Complete(F),S)
6: X := Projection($X, U \times V$)
7: X := Determinize(X)
8: X := Complement(X)
9: X := PrefixClose(X)
10: X := Progressive(X,U)
11: **return** X

5 LANGUAGE EQUATION SOLVING

In [21], the solution X to Eq. (1) is characterized by Eq. (2) and computed by the procedure of Algorithm 2.

The procedure takes as input a specification NFA S with alphabet $I \times O$ and a fixed component NFA F with alphabet $I \times V \times U \times O$. It returns a DFA X, the largest prefix-closed and input-progression solution to Eq. (1). Following Eq. (2), in lines 1-3, S is first completed and determinized to ensure any current state under any input has exactly one next state. Then, S is complemented. In lines 4-5, the input alphabet of S is lifted to $I \times V \times U \times O$ in order to construct product automaton X of F and S. In line 6, the input alphabet of X is projected to $V \times U$, which may result in nondeterministic transitions. In lines 7-8, X is deteminized and complemented. Finally, to make X implementable with a sequential circuit, X is made prefix-closed and U-progressive in lines 9 and 10, respectively.

Solving the language equation by NFA/DFA representation suffers from the scalability problem, even using the partitioned BDD representation [13], as the algorithm may fail when an NFA/DFA has a large number of states. Particularly, complementing an NFA requires superset construction and may result in an exponential blow-up in the number of states. In this work, we utilize rBA, a more compact representation for regular languages, to solve language equation. Moreover, we employ rABA to allow complement operation without superset construction. The modified procedure is shown in Algorithm 3, which is the same as Algorithm 2 except for two differences: First, the underlying representations are different. The modified procedure takes a specification rBA S and a fixed component rBA F as input, and returns an rABA X as output. Second, there is no need to perform Complete and Determinize operations. Below we elaborate each operation in Algorithm 3.

Algorithm 3 *LangEqSolveBA(S,F)*

Input: rBA S with alphabet $I \times O$ and F with alphabet $I \times O \times U \times V$
Output: largest prefix-closed and input-progressive solution X in rABA

```
S := Complement(S)
S := Lifting(S, I × O, U × V)
X_Int := Intersection(F,S)
X := Projection(X_Int, U × V)
X := Complement(X)
X := PrefixClose(X)
X := Progressive(X,U)
return X
```

5.1 Complement Operation

Given an rBA $A = (Q, \Sigma, \delta, f^0, F)$, its complement rBA is $A_{Com} = (Q, \Sigma, \delta, \neg f^0, F)$ such that $L(A_{Com}) = \text{Complement}(L(A))$. On the other hand, given an rABA $A = (Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$, we can construct its complement rABA $A_{Com} = (Q, \Sigma, \gamma, \Delta_{Com}, \delta, \neg f^0, F)$ such that $L(A_{Com}) = \text{Complement}(L(A))$, where

$$\Delta_{Com}(a) = \begin{cases} \exists, & \text{if } \Delta(a) = \forall \\ \forall, & \text{if } \Delta(a) = \exists \end{cases}.$$

The correctness of the above construction can be reasoned as follows. For $\hat{\sigma} \in \Sigma^*$ and $|\hat{\sigma}| = k$, we can construct the same transition tree $T(\hat{\sigma})$ for both $T_A(\hat{\sigma})$ and $T_{A_{Com}}(\hat{\sigma})$ because the state transition of A and A_{Com} are the same. Note that if $T(\hat{\sigma}).root \in T_A(\hat{\sigma}).accepted \Leftrightarrow T(\hat{\sigma}).root \notin T_{A_{Com}}(\hat{\sigma}).accepted$ holds for every $\hat{\sigma} \in \Sigma^*$, then $L(A_{Com}) = \text{Complement}(L(A))$. In fact, we can show that $\forall s \in T(\hat{\sigma})$, $s \in T_{A_{Com}}(\hat{\sigma}).accepted \Leftrightarrow s \notin T_A(\hat{\sigma}).accepted$. We prove this by a simple induction on the level of s.

The base case that s is a leaf is trivial since $f^0_{Com} = \neg f^0$. For the induction step, we assume the statement holds for s at level $t < k$. For s at level $t-1$, there are two cases. First, we consider the case that $\Delta_A(\sigma_t) = \exists$. If $s \in T_A(\hat{\sigma}).accepted$, there is a child $s' \in T_A(\hat{\sigma}).accepted$. By induction hypothesis, $s' \notin T_{A_{Com}}(\hat{\sigma}).accepted$. Since $\Delta_{A_{Com}} = \forall$, $s \notin T_{A_{Com}}(\hat{\sigma}).accepted$. The case $s \notin T_A(\hat{\sigma}).accepted$ is similar. Next, we consider the case that $\Delta_A(\sigma_t) = \forall$. If $s \in T_A(\hat{\sigma}).accepted$, all of its children are in $T_A(\hat{\sigma}).accepted$. By induction hypothesis, all of its children are not in $T_{A_{Com}}(\hat{\sigma}).accepted$. Therefore, $s \notin T_{A_{Com}}(\hat{\sigma}).accepted$. The case $s \notin T_A(\hat{\sigma}).accepted$ is similar. Hence, we have

$$T_{A_{Com}}(\hat{\sigma}).accepted = T_A(\hat{\sigma}).nodes \setminus T_A(\hat{\sigma}).accepted,$$

leading to $L(A_{Com}) = \text{Complement}(L(A))$.

5.2 Lifting Operation

Given an rBA $A = (Q, I, \delta, f^0, F)$, we can construct an rBA

$$A_{Lift(I \times U)} = (Q, I \times U, \delta_{Lift}, f^0, F)$$

such that $A_{Lift(I \times U)} = \text{Lifting}(L(A), I, U)$, where $\delta_{Lift}(q, (i,u)) = \delta(q, i)$. On the other hand, given an rABA $A = (Q, I, \gamma, \Delta, \delta, f^0, F)$, we can construct an rABA $A_{Lift(I \times U)} = (Q, I \times U, \gamma, \Delta, \delta_{Lift}, f^0, F)$ such that $A_{Lift(I \times U)} = \text{Lifting}(L(A), I, U)$, where $\delta_{Lift}(q, (i,u)) = \delta(q, i)$.

5.3 Intersection Operation

Given two rBA $A_1 = (Q_1, \Sigma, \delta_1, f^0_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, f^0_2, F_2)$, their intersection rBA is $A_{Int} = (Q_1 \cup Q_2, \Sigma, \delta_1 \cup \delta_2, f^0_1 \wedge f^0_2, F_1 \cup F_2)$ such that $L(A_{Int}) = \text{Intersection}(L(A_1), L(A_2))$. On the other hand, given two rABA $A_1 = (Q_1, \Sigma, \gamma, \Delta, \delta_1, f^0_1, F_1)$ and $A_2 = (Q_2, \Sigma, \gamma, \Delta, \delta_2, f^0_2, F_2)$ with the same γ and Δ, we can construct an rABA

$$A_{Int} = (Q_1 \cup Q_2, \Sigma, \gamma, \Delta, \delta_1 \cup \delta_2, f^0_1 \wedge f^0_2, F_1 \cup F_2)$$

such that $L(A_{Int}) = \text{Intersection}(L(A_1), L(A_2))$.

5.4 Projection Operation

Given an rBA $A = (Q, I \times U, \delta, f^0, F)$, we construct an rABA $A' = (Q, U, \gamma, \Delta', \delta', f^0, F)$ such that $L(A') = \text{Projection}(L(A), U)$, where $\gamma = |I|$, $\forall u \in U$, $\Delta'(u) = \exists$, and $\delta'(q,u) = \{\delta(q,(i,u)) | i \in I\}$. On the other hand, given an rABA $A = (Q, I \times U, \gamma, \Delta, \delta, f^0, F)$, where $\Delta((i,u)) = \exists$, and $\delta(q,(i,u)) = (f_{(i,u),q,1}, \ldots, f_{(i,u),q,\gamma})$, we can construct an rABA $A' = (Q, U, \gamma', \Delta, \delta', f^0, F)$ such that $L(A') = \text{Projection}(L(A), U)$, where $\gamma' = \gamma \times |I|$ and

$$\delta'(q,u) = (f_{(i_1,u),q,1}, \ldots, f_{(i_{|I|},u),q,1}, f_{(i_1,u),q,2}, \ldots, f_{(i_{|I|},u),q,\gamma}).$$

5.5 Prefix-Close Operation

Given an rBA $A = (Q, \Sigma, \delta, f^0, F)$, we can construct the rBA

$$A_{PC} = (Q \cup \{q_{pc}\}, \Sigma, \delta_{pc}, f^0 \wedge q_{pc}, F \cup \{q_{pc}\})$$

such that $L(A_{PC}) = \text{PrefixClose}(L(A))$, where

$$\delta_{pc}(q,a) = \begin{cases} \delta(q,a), & \text{if } q \in Q \\ q_{pc} \wedge f^0, & \text{if } q = q_{pc} \end{cases}.$$

On the other hand, given an rABA $A = (Q, \Sigma, \gamma, \Delta, \delta, f^0, F)$, we can construct the rABA

$$A_{PC} = (Q \cup \{q_{pc}\}, \Sigma, \gamma, \Delta, \delta_{pc}, f^0 \wedge q_{pc}, F \cup \{q_{pc}\})$$

such that $L(A_{PC}) \subseteq L(A)$ is prefix-closed, where δ_{pc}

$$\delta_{pc}(q,a) = \begin{cases} \delta(q,a) & , \text{if } q \in Q \\ (\underbrace{f^0 \wedge q_{pc}, \ldots, f^0 \wedge q_{pc}}_{\gamma}), & \text{if } q = q_{pc} \end{cases}.$$

Note that if $\forall a \in \Sigma$, $\Delta(a) = \forall$, then $L(A_{PC}) = \text{PrefixClose}(L(A))$.

The correctness of the prefix-close operation for rABA can be established as follows. First, we show that $L(A_{PC})$ is prefix-closed, i.e., $\hat{\sigma} \in L(A_{PC}) \Leftrightarrow \text{Prefix}(\hat{\sigma}) \subseteq L(A_{PC})$, by induction on the length of $\hat{\sigma}$. For the base case $|\hat{\sigma}| = 0$, the property holds trivially. For the induction step $|\hat{\sigma}| = k$, we assume $\hat{\sigma} \in L(A_{PC}) \Leftrightarrow \text{Prefix}(\hat{\sigma}) \subseteq L(A_{PC})$ is true. For $|\hat{\sigma}| = k+1$, let $\hat{\sigma} = \hat{\sigma}' a \in L(A_{PC})$. Then, we construct $T_{A_{PC}}(\hat{\sigma})$ and $T_{A_{PC}}(\hat{\sigma}') = T_{A_{PC}}(\hat{\sigma}) \setminus T_{A_{PC}}(\hat{\sigma}).leaves$. We prove $\hat{\sigma}' \in L(A_{PC})$ by showing $(T_{A_{PC}}(\hat{\sigma}).accepted \setminus T_{A_{PC}}(\hat{\sigma}).leaves) \subseteq T_{A_{PC}}(\hat{\sigma}').accepted$. For s in the k^{th} level, if $s \in T_{A_{PC}}(\hat{\sigma}).accepted$, there is a child s' of s such that $(f^0 \wedge q_{pc})(s') = 1$, which implies $q_{pc}(s') = (f^0 \wedge q_{pc})(s) = 1$, leading to $s \in T_{A_{PC}}(\hat{\sigma}').accepted$. By definition, $(T_{A_{PC}}(\hat{\sigma}).accepted \setminus T_{A_{PC}}(\hat{\sigma}).leaves) \subseteq T_{A_{PC}}(\hat{\sigma}').accepted$ holds for level 1 to k. By the induction hypothesis, we have $\text{Prefix}(\hat{\sigma}') \subseteq L(A_{PC})$. Therefore, $\text{Prefix}(\hat{\sigma}) = \text{Prefix}(\hat{\sigma}') \cup \{\hat{\sigma}\} \subseteq L(A_{PC})$, leading to $L(A_{PC})$ is prefix-closed. Besides, by $f^0_{PC}(s) = (f^0 \wedge q_{pc})(s) = 1 \Rightarrow f^0(s) = 1$, we have $L(A_{PC}) \subseteq L(A)$. Since $L(A_{PC}) \subseteq L(A)$, we have $L(A_{PC}) \subseteq \text{PrefixClose}(L(A))$ is prefix-closed.

Moreover, for A_{PC} with only universal nondeterminism, we can establish $L(A_{PC}) = \text{PrefixClose}(L(A))$ by further showing $L(A_{PC}) \supseteq \text{PrefixClose}(L(A))$. We show that $\hat{\sigma} \in \text{PrefixClose}(L(A)) \Rightarrow \hat{\sigma} \in$

$(L(A_{PC}))$ by induction on the length of input $\hat{\sigma}$. For the base case $|\hat{\sigma}| = 0$, the property holds trivially. For the induction step $|\hat{\sigma}| = k$, we assume that $\hat{\sigma} \in \text{PrefixClose}(L(A)) \Rightarrow \hat{\sigma} \in (L(A_{PC}))$ holds. For $|\hat{\sigma}| = k+1$, let $\hat{\sigma} = \hat{\sigma}'a \in \text{PrefixClose}(L(A))$. By the property of prefix-closedness, we have $\hat{\sigma}' \in \text{PrefixClose}(L(A))$. By induction hypothesis, we have $\hat{\sigma}' \in L(A_{PC})$. Then, we construct trees $T_{A_{PC}}(\hat{\sigma})$ and $T_{A_{PC}}(\hat{\sigma}') = T_{A_{PC}}(\hat{\sigma}) \setminus T_{A_{PC}}(\hat{\sigma}).leaves$. Considering $s \in T_{A_{PC}}(\hat{\sigma}).leaves$, its parent s' is a leaf of $T_{A_{PC}}(\hat{\sigma}')$. First, for $\hat{\sigma}' \in L(A_{PC})$ with universal transition property, we know that $s' \in T_{A_{PC}}(\hat{\sigma}').accepted$, i.e., $(f^0 \wedge q_{pc})(s') = q_{pc}(s) = 1$. Second, for each $s \in T_{A_{PC}}(\hat{\sigma}).leaves$, we have $f^0(s) = 1$ since $\hat{\sigma} \in \text{PrefixClose}(L(A)) \subseteq L(A)$. Combining two results, we have $(f^0 \wedge q_{pc})(s) = 1$, leading to $\hat{\sigma} \in L(A_{PC})$. Hence, we have $L(A_{PC}) = \text{PrefixClose}(L(A))$ for A with only universal nondeterminism.

5.6 Input-Progressive Operation

After the prefix-close operation, the input-progressive operation is applied to make the trimmed language valid for deterministic hardware realization. Given a finite automaton X with input alphabet $U \times V$, we can derive an new automaton with the largest U-progressive language of $L(X)$ by applying the algorithm proposed by Yevtushenko *et al.* [21]. The algorithm iteratively deletes invalid states that cannot reach accepting states under input set $\{(u,v) \mid v \in V\}$ with fixed $u \in U$. The procedure is terminated when no more state can be deleted during the iteration or the initial state is deleted.

To perform the input-progressive operation on rBA/rABA, we modify the prior algorithm [21] as follows. Since the acceptance of strings are decided by values of all states under rBA/rABA representation, we consider the transition of all states rather than a single state. The state vector $s \in \mathbb{B}^{|Q|}$ is valid if and only if s satisfies the formula

$$\chi = \forall u \; \exists v \; f^0(\delta(s,(u,v))), \tag{3}$$

where $u \in U$ and $v \in V$. That is, χ of Eq. (3) is the characteristic function of the set of valid state vectors. Instead of deleting invalid state vectors, we modify the initial function to make only valid state vectors acceptable. Then, we update the initial function f^0 to

$$f^0 := f^0 \wedge \chi. \tag{4}$$

Note that quantifier elimination is required to update the initial function. Therefore, only input strings that transition to valid state vectors are accepted by the automaton. The steps that deriving χ and updating f^0 are repeated until the valid state set of the current iteration χ_{new} equals that of the previous iteration χ_{old}, that is, a fixed-point is reached.

Algorithm 4 *Progressive(X,U)*

Input: prefix-closed rBA/rABA X and input set U
Output: prefix-closed and U-progressive rBA/rABA X

1: $\chi_{old} := 1$
2: $\chi_{new} := \forall u \; \exists v \; f^0(\delta(s,(u,v)))$
3: **while** $\chi_{new} \neq \chi_{old}$
4: $\quad f_0 := f_0 \wedge \chi_{new}$
5: $\quad \chi_{old} := \chi_{new}$
6: $\quad \chi_{new} := \forall u \; \exists v \; f^0(\delta(s,(u,v)))$
7: **return** X

Algorithm 4 trims rBA/rABA to satisfy the input-progressive property. In line 2, the characteristic function χ_{new} is initialized by Eq. (3). Then, line 4 updates the initial function by Eq. (4), and line 5 records χ_{old}, the valid state set of the previous iteration. Next, line 6 updates χ_{new} according to Eq. (3) with the new initial function. The while-loop of line 3 repeats until χ_{new} and χ_{old} are equivalent.

6 EXPERIMENTAL RESULTS

The proposed algorithm was implemented in C++ within the ABC system [2]. All the experiments were conducted on a Linux server with Intel Xeon Silver 4210 CPUs of 2.20 GHz and 126 GB RAM. The experiments were conducted on the ISCAS [3] and ITC [6] benchmarks, which were converted to the unknown component instances by the latch-split operation in BALM [13], implemented in the MVSIS system [9]. The latch-split operation transforms a given sequential circuit S into two sub-circuits F and R. Then S is treated as the specification, and F the fixed component containing a subset of the latches in S. Accordingly, R is a particular solution to the unknown part X and contains the rest of the latches. Given S and F, we compute the complete sequential flexibility X, which is the largest prefix-closed and input progressive solution of Eq. (1). We optionally use synthesis script `scleanup; ssweep;` of ABC for optimizing S and F.

The experimental results of BALM and our method are shown in Table 3, where $\#i$, $\#o$, $\#l$, $\#l_F$, and $\#l_R$ denote the input number, the output number, the total latch number in S, the latch number in F, and the latch number in rest part R, respectively. Our method is compared with the state-of-the-art method [13] in terms of solution size ($|Q|$ of X) and runtime in Table 3. An entry "TO" in the table indicates that the case cannot be solved within the timeout limit of 3600 seconds. Note that we performed quantifier elimination by using command `qvar` of ABC.

The results suggest that the rBA/rABA-based method outperforms the partitioned representation by solving 4 more cases among the considered 31 cases. As observed, the runtime and solution size of [13] increase drastically for large $\#i$, $\#o$, $\#l$, $\#l_F$ values. When the state number of X is too large, the partitioned representation fails to solve the problem. As our method does not require superset construction in the unknown component X derivation, the state number of X equals $\#l + \#l_F + 3$ (for 2 additional states introduced in extracting rBA from sequential circuits (of S and F) and 1 extra state introduced by the prefix-close operation). Thus, when S and F are simplified, the state number of X can be reduced. The solution sizes of [13] are on average 100× larger than those of ours, showing that rBA/rABA is a more powerful representation for language equation solving. For the cases solved by both methods, our algorithm is on average 740× faster than [13]. For all the instances, our algorithm can complete the operations of lines 1-6 in Algorithm 3 in less than 0.03 seconds. The most time-consuming part among the language operations of our algorithm is the input progressive operation, which requires quantifier elimination to convert a quantified formula to a quantifier-free one for initial function update. The quantifier elimination procedure timed out on 5 of the instances. Our method outperforms the partitioned representation by solving 9 more cases among the considered 36 cases.

Table 3: Results for comparison between [13] and ours.

Name	#i/#o/#l	$\#l_F/\#l_R$	[13]		Ours					
					w/o optimization			w/ optimization		
			$\|Q\|$	Time (s)	$\|Q\|$	Mem (MB)	Time (s)	$\|Q\|$	Mem (MB)	Time (s)
s208	10/1/8	4/4	184	0.10	16	34.84	0.32	12	36.76	0.34
s298	3/6/14	8/6	219	0.40	25	34.60	0.08	24	37.11	0.08
s344	9/11/15	6/9	9800	181.54	24	34.26	0.06	23	37.26	0.09
s349	9/11/15	6/9	38266	743.84	24	34.21	0.08	23	37.33	0.08
s382	3/6/21	8/13	17730	35.48	32	34.46	0.09	31	37.29	0.10
s386	7/7/6	4/2	15	0.02	13	34.93	0.20	12	38.02	0.18
s400	3/6/21	8/13	17730	34.85	32	34.78	0.09	31	37.45	0.09
s420	19/2/16	9/7	244	0.70	28	999.96	413.88	13	37.19	0.74
s444	3/6/21	6/15	8866	15.57	30	34.65	0.07	29	37.68	0.09
s510	19/7/6	4/2	49	0.05	13	34.55	0.11	12	37.48	0.10
s526	3/6/21	12/9	—	TO	36	35.76	0.12	35	38.18	0.13
s713	35/23/19	6/13	—	TO	—	—	TO	—	—	TO
s820	18/19/5	3/2	28	0.04	11	36.00	0.45	10	38.66	0.35
s832	18/19/5	3/2	28	0.05	11	35.92	0.86	10	39.14	0.87
s838	35/2/32	11/21	82	1.26	—	—	TO	—	—	TO
s953	16/23/30	7/23	505	244.74	40	35.43	0.15	38	37.44	0.15
s1196	14/14/19	9/10	—	TO	—	—	TO	—	—	TO
s1238	14/14/19	9/10	—	TO	—	—	TO	—	—	TO
s1423	17/5/75	21/54	—	TO	—	—	TO	—	—	TO
s1488	8/19/7	4/3	49	0.08	14	92.14	18.47	12	52.06	3.31
s1494	8/19/7	4/3	49	0.07	14	107.93	19.34	12	47.28	3.27
s9234	36/39/211	41/170	—	TO	255	39.44	1.87	173	41.28	1.66
b01	4/2/5	3/2	34	0.00	11	34.05	0.08	10	37.34	0.07
b02	3/1/4	3/1	31	0.00	10	33.61	0.07	9	36.27	0.09
b04	13/8/66	16/50	—	TO	85	36.67	0.20	85	39.20	0.22
b05	3/36/34	9/25	532	19.60	46	36.43	0.79	46	39.23	0.72
b07	3/8/49	9/40	3927	2.80	61	35.57	0.17	49	38.62	0.18
b08	11/4/21	6/15	85229	385.63	30	34.92	0.10	29	37.59	0.12
b09	3/1/28	9/19	—	TO	40	35.40	0.12	39	37.73	0.12
b10	13/6/17	8/9	142797	1026.54	28	34.95	0.12	27	37.62	0.13
b11	9/6/31	12/19	—	TO	46	36.00	0.22	45	38.72	0.24
b13	12/10/53	11/42	—	TO	67	35.00	0.17	58	37.45	0.17
b14	34/54/245	45/200	—	TO	293	45.07	2.29	263	51.06	2.70
b15	37/70/449	49/400	—	TO	501	53.80	3.47	467	62.80	7.15
b20	34/22/490	40/450	—	TO	533	71.66	6.25	472	61.38	4.88
b21	34/22/490	40/450	—	TO	533	71.85	6.16	472	61.35	5.25
Number of solved cases			22		31			31		

Table 4: Results of our method (w/ optimization) for different latch-split options for F and R on circuit b04.

$\#l_F/\#l_R$	$\|Q\|$	Mem (MB)	Time (s)
6/60	75	39.003	0.17
16/50	85	39.328	0.23
26/40	95	39.898	0.34
36/30	105	40.434	0.54
46/20	115	564.792	57.66
56/10	—	—	TO

To study the effect of latch-split, we performed a case study on circuit b04. The results are shown in Table 4, which compares the runtime required for different latch numbers placed in F and R. From the table, it can be observed that runtime increases in accordance with the latch number in F, and thus the state number $|Q|$ of X. It indicates that, for larger $\#l_F$, a circuit with more state bits is required to encode the rABA of X, and thus the quantifier elimination process of the input-progressive operation would take more time. We note that by the latch-split method and the input-progressive operation, the variables to be eliminated are those (with $\#i + \#l_F$ bits) that encode alphabet U, those (with $\#l_R$ bits) encode V, and the pseudo-inputs (with $\#o$ bits) of X for encoding deterministic choices. Therefore, the total number of variables to be eliminated is $\#i + \#l_F + \#l_R + \#o$, which is fixed for a circuit regardless of how the latches are split in F and R.

7 CONCLUSIONS

This paper has proposed variants of Boolean automata, rBA and rABA, amenable for logic circuit representation that supports scalable regular language manipulation. Language operations based on rBA and rABA have been devised. Experimental results have shown that our method outperforms the state of the art. In this work, we mainly rely on ABC for quantifier elimination in the input-progressive operation. As there are recent Boolean function synthesis tools [10, 15] being developed, we plan to take advantage of them for potential improvements. Also, we would like to explore more applications using our developed Boolean automata.

ACKNOWLEDGMENTS

This work was supported in part by the Ministry of Science and Technology of Taiwan under grants MOST 108-2221-E-002-144-MY3, 110-2224-E-002-011, and 111-2119-M-002-012.

REFERENCES

[1] Florent Avellaneda, Silvano Dal Zilio, and Jean-Baptiste Raclet. 2016. Solving Language Equations Using Flanked Automata. In *International Symposium on Automated Technology for Verification and Analysis*. 106–121.
[2] Robert K. Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of the International Conference on Computer Aided Verification*. 24–40.
[3] Franc Brglez, David Bryan, and Krzysztof Kozminski. 1989. Combinational profiles of sequential benchmark circuits. In *IEEE International Symposium on Circuits and Systems,*. 1929–1934.
[4] Janusz A. Brzozowski and Ernst Leiss. 1980. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science* 10, 1

(1980), 19–35.
[5] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. 1981. Alternation. *J. ACM* 28, 1 (1981), 114–133.
[6] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. 2000. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers* 17, 3 (2000), 44–53.
[7] Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. *CoRR* abs/1708.09073 (2017). arXiv:1708.09073 http://arxiv.org/abs/1708.09073
[8] A. Fellah, H. Jürgensen, and S. Yu. 1990. Constructions for alternating finite automata. *International Journal of Computer Mathematics* 35, 1-4 (1990), 117–132.
[9] Minxi Gao, Jie-Hong R. Jiang, Yunjian Jiang, Yinghua Li, Alan Mishchenko, Subarna Sinha, Tiziano Villa, and Robert Brayton. 2002. Optimization of multi-valued multi-level networks. In *Proceedings of IEEE International Symposium on Multiple-Valued Logic*. 168–177. https://ptolemy.berkeley.edu/projects/embedded/mvsis/
[10] Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. 2020. Manthan: A Data-Driven Approach for Boolean Function Synthesis. In *Proceedings of International Conference on Computer Aided Verification*. 611–633.
[11] Michal Hospodár, Galina Jirásková, and Ivana Krajňáková. 2018. Operations on Boolean and alternating finite automata. In *Proceedings of International Computer Science Symposium in Russia*.
[12] Galina Jirásková. 2012. Descriptional complexity of operations on alternating and Boolean automata. In *Proceedings of International Computer Science Symposium in Russia*.
[13] Alan Mishchenko, Robert Brayton, Roland Jiang, Tiziano Villa, and Nina Yevtushenko. 2005. Efficient solution of language equations using partitioned representations. In *Proceedings of Design, Automation and Test in Europe Conference*. 418–423.
[14] Alan Mishchenko, Satrajit Chatterjee, Jie-Hong R. Jiang, and Robert K. Brayton. 2005. FRAIGs: A unifying representation for logic synthesis and verification. *ERL Technical Report, UC Berkeley* (2005).
[15] Markus N. Rabe. 2019. Incremental determinization for quantifier elimination and functional synthesis. In *Proceedings of International Conference on Computer Aided Verification*. 84–94.
[16] Kai Salomaa, Xiuming Wu, and Sheng Yu. 2000. Efficient Implementation of Regular Languages Using Reversed Alternating Finite Automata. *Theor. Comput. Sci.* 231, 1 (2000), 103–111.
[17] Michael Sipser. 1996. *Introduction to the Theory of Computation*. International Thomson Publishing.
[18] Tiziano Villa, Alexandre Petrenko, Nina Yevtushenko, Alan Mishchenko, and Robert Brayton. 2015. Component-based design by solving language equations. *Proc. IEEE* 103, 11 (2015), 2152–2167.
[19] Tiziano Villa, Nina Yevtushenko, Robert K. Brayton, Alan Mishchenko, Alexandre Petrenko, and Alberto Sangiovanni-Vincentelli. 2014. *The Unknown Component Problem: Theory and Applications*. Springer Publishing Company, Incorporated.
[20] Hung-En Wang, Tzung-Lins Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R Jiang. 2016. String analysis via automata manipulation with logic circuit representation. In *Proceedings of International Conference on Computer Aided Verification*. 241–260.
[21] Nina Yevtushenko, Tiziano Villa, Robert K Brayton, Alex Petrenko, and Alberto L Sangiovanni-Vincentelli. 2000. Sequential synthesis by language equation solving. In *Proceedings of International Workshop on Logic Synthesis*.