Decision Procedures for Inductive Boolean Functions based on Alternating Automata

Abdelwaheb Ayari, David Basin, Felix Klaedtke

Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Germany

Abstract

We show how alternating automata provide decision procedures for the equality of inductively defined Boolean functions and present applications to reasoning about parameterized families of circuits. We use alternating word automata to formalize families of linearly structured circuits and alternating tree automata to formalize families of tree structured circuits. We provide complexity bounds for deciding the equality of function (or circuit) families and show how our decision procedures can be implemented using BDDs. In comparison to previous work, our approach is simpler, has better complexity bounds, and, in the case of tree structured families, is more general.

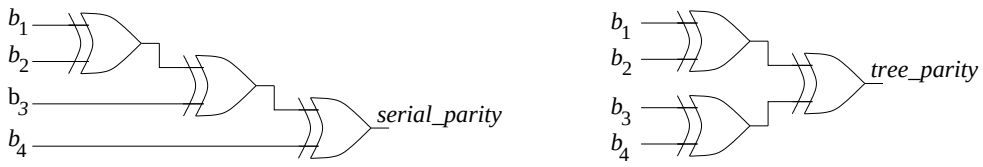
Keywords: Alternating Automaton, Inductive Boolean Function, Verification, Parameterized Circuit

1 Introduction

Reasoning about parametric system descriptions is important in building scalable systems and generic designs. In hardware verification, such reasoning arises in the verification of parametric combinational circuit families, for example, proving that circuits in one family are equivalent to circuits in another, for every parameter value. Another application of parametric reasoning is in establishing properties of sequential circuits where time is the parameter considered. In this paper we present a new approach to formalizing parametric descriptions and reasoning about them.

The starting point for our research is the work of Gupta and Fisher [8,9]. They developed a formalism for describing circuit families using one of two kinds of inductively defined Boolean functions. The first, called *Linearly Inductive Boolean Functions*, or LIFs, formalizes families of linearly structured circuits. The second, called *Exponentially Inductive Boolean Functions*, or EIFs, models

families of tree structured circuits. As simple examples, consider the following linear (serial) and tree structured 4-bit parity circuits:



A LIF describing the general case of the linear circuit is given by the equations (we will formally introduce slightly different syntax in §3 and §4):

$$\begin{aligned} \mathit{serial_parity}^1(b_1) &= b_1 \\ \mathit{serial_parity}^n(b_1,\ldots,b_n) &= b_n \oplus \mathit{serial_parity}^{n-1}(b_1,\ldots,b_{n-1}) \qquad \text{for } n > 1 \ . \end{aligned}$$

Similarly, an EIF describing the family of tree-structured parity circuits is:

$$\begin{aligned} \mathit{tree_parity}^1(b_1) &= b_1 \\ \mathit{tree_parity}^{2^n}(b_1,\ldots,b_{2^n}) &= \mathit{tree_parity}^{2^{n-1}}(b_1,\ldots,b_{2^{n-1}}) \oplus \\ &\quad\ \mathit{tree_parity}^{2^{n-1}}(b_{2^{n-1}+1},\ldots,b_{2^n}) \qquad \text{for } n \geq 1 \ . \end{aligned}$$

Gupta and Fisher developed algorithms to translate these descriptions into data-structures that generalize BDDs (roughly speaking, their data-structures have additional pointers between BDDs, which formalize recursion). The resulting data-structures are canonical: different descriptions of the same family are converted into identical data-structures. This yields decision procedures both for the equality of LIFs and for EIFs.

Motivated by their ideas, we take a different approach to these decision problems. We show how LIFs and EIFs can be translated, respectively, into alternating word and tree automata, whereby the decision problems are solvable by automata calculations. For LIFs, both the translation and the decision procedure are quite direct and may be implemented and analyzed using standard algorithms and results for word automata. For EIFs, the situation is more complex since input is given by trees where only leaves are labeled by data and we are only interested in the equality of complete trees. Here, we decide equality using a procedure that determines whether a tree automaton accepts a complete leaf-labeled tree.

The use of alternating automata has a number of advantages. First, it gives us a simple view of (and leads to simpler formalisms for) LIFs and EIFs based on standard results from automata theory. For example, the expressiveness of these languages directly falls out of our translations: LIFs describe regular languages on words and EIFs describe regular languages on trees (modulo the complexities alluded to above). Moreover, as we will see, the converse also

holds, namely, LIFs and EIFs can formalize any circuit family whose behavior is regular in the language-theoretic sense. Second, it provides a handle on the complexity of the problems. For LIFs we show that the equality problem is PSPACE-complete and for EIFs it is EXPSPACE-complete. The result for LIFs represents a doubly exponential improvement over the previous results of Gupta and Fisher; our results for EIFs are, to our knowledge, the first published bounds for this problem. Finally, the use of alternating automata provides a basis for adapting data-structures recently developed in the MONA project [12]; there, BDDs are used to represent automata and can often exponentially compress the representation of the transition function. We show that the use of BDDs to represent alternating automata offers similar advantages and plays an important rôle in the practical use of these techniques.

We proceed as follows. In §2 we provide background material on word and tree automata. In §3 and §4 we formalize LIFs and EIFs and explain our decision procedures. In §5 we make comparisons and in §6 we draw conclusions and discuss future work. The appendix contains additional proof details.

2 Background

Boolean Logic The set $B(V)$ of *Boolean formulae (over V)* is built from the constants 0 and 1, variables $v \in V$, and the connectives $\neg$, $\vee$, $\wedge$, $\leftrightarrow$, and $\oplus$. $B_+(V)$ is the set of the *positive Boolean formulae (over V)*, i.e. the set of Boolean formulae built from 0, 1, $v \in V$, and the connectives $\vee$ and $\wedge$. For $\beta \in B(V)$, $\beta[v_1/\alpha_1, \ldots, v_n/\alpha_n]$ denotes the formula where the variables $v_i \in V$ are simultaneously replaced by the formulae $\alpha_i \in B(V)$.

Boolean formulae are interpreted in the set $\mathbb{B} = \{0, 1\}$ of *truth values*. A *valuation* is a function $\sigma : V \to \mathbb{B}$ that is homomorphically extended to $B(V)$. For $\sigma : V \to \mathbb{B}$ and $\beta \in B(V)$, we write $\sigma \models \beta$ if $\sigma(\beta) = 1$. We will sometimes identify a subset M of V with the valuation $\sigma_M : V \to \mathbb{B}$, where $\sigma_M(v) = 1$ iff $v \in M$. For example, for the formula $v_1 \oplus v_2$, we have $\{v_1\} \models v_1 \oplus v_2$ but $\{v_1, v_2\} \not\models v_1 \oplus v_2$.

Words and Trees Σ^* is the set of all words over the alphabet Σ. We write λ for the empty word and Σ^+ for $\Sigma^* \setminus \{\lambda\}$. For a a letter not occurring in Σ, we write Σ_a for $\Sigma \cup \{a\}$. Concatenation of $u, v \in \Sigma^*$ is written as juxtaposition uv. The length of $u \in \Sigma^*$ is denoted by $|u|$ and u^{R} denotes the reversal of u.

A *Σ-labeled tree* is a function t where the range of t is Σ and the domain of t, $\mathrm{dom}(t)$ for short, is a finite subset of $\mathbb{N}^*$, where (i) $\mathrm{dom}(t)$ is prefix closed, i.e. if $ui \in \mathrm{dom}(t)$ with $i \in \mathbb{N}$, then $u \in \mathrm{dom}(t)$, and (ii) if $ui \in \mathrm{dom}(t)$

then $uj \in \mathrm{dom}(t)$, for all $j < i$. The elements of $\mathrm{dom}(t)$ are called *nodes* and $\lambda \in \mathrm{dom}(t)$ is called the *root*. The node $ui \in \mathrm{dom}(t)$ is a *successor* of u. A node is an *inner node* if it has successors and is a *leaf* otherwise. The *height* of t is $|t| = \max(\{0\} \cup \{|u| + 1 \mid u \in \mathrm{dom}(t)\})$. The *depth* of a node $u \in \mathrm{dom}(t)$ is the length of u.

A $\Sigma_{\#}$-labeled tree is Σ-*leaf-labeled* when the leaves are labeled with letters in Σ and the inner nodes are labeled with the dummy symbol #. A tree is *complete* if all its leaves have the same depth. The *frontier* of t is the word $\mathrm{front}(t) \in \Sigma^*$, where the ith letter is the label of the ith leaf of t and the leaves are lexicographically ordered. We call t *binary* if every inner node $u \in \mathrm{dom}(t)$ has exactly two successors. $\Sigma^{\mathrm{T}*}$ denotes the set of all binary Σ-labeled trees and $\Sigma^{\mathrm{T}+}$ is $\Sigma^{\mathrm{T}*}$ without the empty tree.

Nondeterministic Automata A *nondeterministic word automaton (NWA)* $\mathcal{A}$ is a tuple $(\Sigma, Q, q_0, F, \delta)$, where Σ is a finite alphabet, Q is a nonempty finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function. A *run* of $\mathcal{A}$ on a word $w = a_1 \dots a_n \in \Sigma^*$ is a word $\pi = s_1 \dots s_{n+1} \in Q^+$ with $s_1 = q_0$ and $s_{i+1} \in \delta(s_i, a_i)$ for $1 \leq i \leq n$. π is *accepting* if $s_{n+1} \in F$. A word w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w; $L(\mathcal{A})$ denotes the set of accepted words.

A *nondeterministic (top-down, binary) tree automaton (NTA)* is defined analogously: $\mathcal{A}$ is a tuple $(\Sigma, Q, q_0, F, \delta)$, where Σ, Q, q_0 and F are as before, and the transition function is $\delta : Q \times \Sigma \to \mathcal{P}(Q \times Q)$. A *run* of a NTA $\mathcal{A}$ on a tree $t \in \Sigma^{\mathrm{T}*}$ is a tree $\pi \in Q^{\mathrm{T}+}$, where $\mathrm{dom}(\pi) = \{\lambda\} \cup \{ub \mid u \in \mathrm{dom}(t) \text{ and } b \in \{0, 1\}\}$. That is, π's nodes are those of t and the additional leaves $u0, u1 \in \mathrm{dom}(\pi)$, where u is a leaf of t. Moreover, $\pi(\lambda) = q_0$ and for $u \in \mathrm{dom}(t)$, $(\pi(u0), \pi(u1)) \in \delta(\pi(u), t(u))$. The run π is *accepting* if $\pi(u) \in F$ for each leaf $u \in \mathrm{dom}(\pi)$. A tree t is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on t; $L(\mathcal{A})$ denotes the set of accepted trees.

NWAs and NTAs recognize the regular word and tree languages and are effectively closed under intersection, union, complement and projection. For a detailed account of regular word and tree languages see [13] and [6,7] respectively.

Alternating Automata *Alternating automata* were introduced for words in [3,4] and for trees in [18]. We use the definition of alternating automata for words from [19] and generalize it to (binary) trees.

An *alternating word automaton (AWA)* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ whose

first four components are as before and the transition function is of the form $\delta : Q \times \Sigma \to B_+(Q)$. The same holds for *alternating tree automata (ATA)* except that the transition function is of the form $\delta : Q \times \Sigma \to B_+(Q \times \{0,1\})$. We write q^b for $(q,b) \in Q \times \{0,1\}$.

We will only define a run for an ATA; the restriction to AWAs is straightforward. For an ATA, a *run* π of $\mathcal{A}$ on $t \in \Sigma^{\mathrm{T}*}$ is a $Q \times \{0,1\}^*$-labeled tree, with $\pi(\lambda) = (q_0, \lambda)$. Moreover, for each node $w \in \mathrm{dom}(\pi)$, with $\pi(w) = (q,u)$,

$$\{p_0^{b_0}, \ldots, p_{r-1}^{b_{r-1}}\} \models \delta(q, t(u)) \,,$$

where r is the number of successors of w and $\pi(wk) = (p_k, ub_k)$, for $0 \leq k < r$. π is *accepting* if for every leaf w in π, with $\pi(w) = (p,u)$, u is a leaf in t implies $p \in F$. The *tree language accepted by* $\mathcal{A}$ is $L(\mathcal{A}) = \{t \in \Sigma^{\mathrm{T}*} \mid \mathcal{A} \text{ accepts } t\}$. If there exists an accepting run of $\mathcal{A}' = (\Sigma, Q, q, F, \delta)$ for $q \in Q$ on t, then we say that $\mathcal{A}$ *accepts* t *from* q. We use the same terminology for AWAs.

It is straightforward to construct an alternating automaton from a nondeterministic automaton of the same size. Conversely, given an AWA one can construct an equivalent NWA with at most exponentially more states [3,4,19]. The states of the nondeterministic automaton are the interpretations of the Boolean formulae of the alternating automaton's transition function. This construction can be generalized to tree automata. Hence alternation does not increase the expressiveness of word and tree automata. However, as we will see, it does enhance their ability to model problems.

Sometimes it is convenient to have ATAs with an initial positive Boolean formula instead of a single initial state. Such an automaton $\mathcal{A}$ can be converted into an equivalent automaton $\mathcal{A}'$ with a single initial state by adding a new state q'. In particular, the transition function δ' of $\mathcal{A}'$ for q' is $\delta'(q', a) = \iota[q_1/\delta(q_1, a), \ldots, q_n/\delta(q_n, a)]$, where ι is the initial Boolean formula, δ is the transition function, and $\{q_1, \ldots, q_n\}$ is the set of states of $\mathcal{A}$.

3 Linearly Inductive Boolean Functions

3.1 Definition of LIFs

We now define linearly inductive Boolean functions. Our definition differs slightly from [8–10], however they are equivalent (see §5).

Syntax Let the two sets $V = \{v_1, \ldots, v_r\}$ and $F = \{f_1, \ldots, f_s\}$ be fixed for the remainder of this paper.

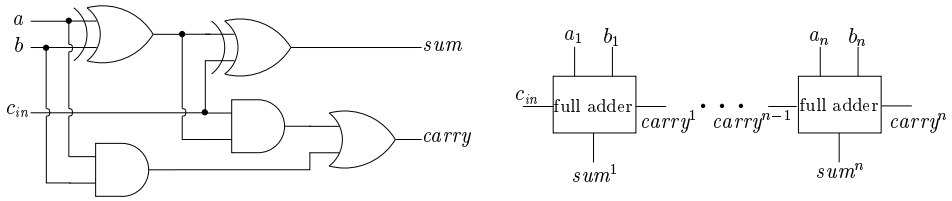


Fig. 1. Full adder (left) and n-bit ripple-carry adder (right)

A *LIF expression* (over V and F) is a pair (α, β), with $\alpha \in B(V)$ and $\beta \in B(V \uplus F)$. The formulae α and β formalize the base and step case of a recursive definition. A *LIF system* (over V and F) is a pair $\mathcal{S} = (\mathsf{E}, \eta)$, where E is a set of LIF expressions over V and F and $\eta : F \to \mathsf{E}$. That is, η assigns to each $f \in F$ a LIF expression $(\alpha, \beta) \in \mathsf{E}$. We will write (α_f, β_f) for $\eta(f) = (\alpha, \beta)$ and omit V and F when they are clear from the context.

Semantics For a word $x = x_1 \dots x_n \in (\mathbb{B}^r)^*$ we use the following notation: $x_{i,j} \in \mathbb{B}$, for $1 \leq i \leq n$ and $1 \leq j \leq r$, denotes the jth coordinate of the ith letter, i.e. $x_i = (x_{i,1}, \dots, x_{i,r})$. Let $\mathcal{S}$ be a LIF system. An *evaluation* of $\mathcal{S}$ on $x \in (\mathbb{B}^r)^+$ is a word $y \in (\mathbb{B}^s)^+$, with $|x| = |y|$, such that for $1 \leq k \leq s$,

$$y_{1,k} = 1 \qquad \text{iff} \qquad \{v_l \mid 1 \leq l \leq r \text{ and } x_{1,l} = 1\} \models \alpha_{f_k} \,,$$

and for all i, with $1 < i \leq |x|$,

$$\begin{aligned} y_{i,k} = 1 \qquad \text{iff} \qquad & \{v_l \mid 1 \leq l \leq r \text{ and } x_{i,l} = 1\} \cup \\ & \{f_l \mid 1 \leq l \leq s \text{ and } y_{i-1,l} = 1\} \models \beta_{f_k} \,. \end{aligned}$$

It is straightforward to show that an evaluation exists and is uniquely defined. Hence, $f_k \in F$ together with $\mathcal{S}$ determine a function $f_k^{\mathcal{S}} : (\mathbb{B}^r)^+ \to \mathbb{B}$. Namely, for $x \in (\mathbb{B}^r)^+$, $f_k^{\mathcal{S}}(x) = y_{|x|,k}$. We call $f_k^{\mathcal{S}}$ the *LIF* of $\mathcal{S}$ and f_k and omit $\mathcal{S}$ when it is clear from the context.

Examples We present three simple examples. First, for $V = \{x\}$ and $F = \{serial_parity\}$, the following LIF system $\mathcal{S}_{sp}$ formalizes the family of linear parity circuits given in the introduction.

$$\alpha_{serial_parity} = x \qquad\qquad \beta_{serial_parity} = x \oplus serial_parity$$

In particular, $serial_parity^{\mathcal{S}_{sp}}$ applied to $b_1 \dots b_n \in \mathbb{B}^+$ equals the function $serial_parity^n(b_1, \dots, b_n)$ from the introduction.

The second LIF system $\mathcal{S}_{rca}$ over $V = \{a, b, c_{in}\}$ and $F = \{sum, carry\}$ formalizes the family of ripple-carry adders pictured in Figure 1.

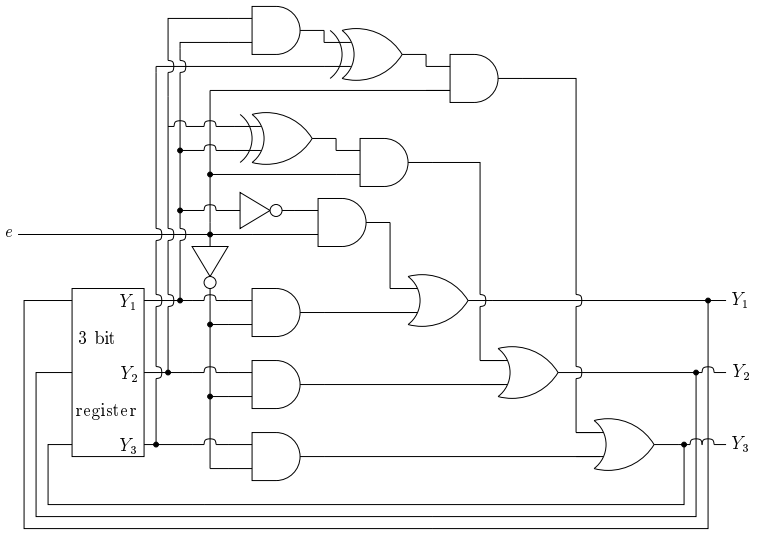


Fig. 2. 3-bit counter

$$\alpha_{sum} = (a \oplus b) \oplus c_{in} \qquad \beta_{sum} = (a \oplus b) \oplus carry$$
$$\alpha_{carry} = ((a \oplus b) \wedge c_{in}) \vee (a \wedge b) \qquad \beta_{carry} = ((a \oplus b) \wedge carry) \vee (a \wedge b)$$

Here $sum^{\mathcal{S}_{rca}}$ [respectively $carry^{\mathcal{S}_{rca}}$] represents the adder's nth output bit [respectively carry bit].

The third example shows how to describe a sequential circuit by a LIF system. The LIF system $\mathcal{S}_{cnt3}$ over $V = \{e\}$ and $F = \{Y_1, Y_2, Y_3\}$ describes a 3-bit counter (Figure 2) with an enable bit. The initial state of the counter is $(0, 0, 0)$, which is described by the base cases of the LIF expressions of Y_1, Y_2, and Y_3.

$$\alpha_{Y_1} = 0 \qquad \beta_{Y_1} = (\neg e \wedge Y_1) \vee (e \wedge \neg Y_1)$$
$$\alpha_{Y_2} = 0 \qquad \beta_{Y_2} = (\neg e \wedge Y_2) \vee (e \wedge (Y_1 \oplus Y_2))$$
$$\alpha_{Y_3} = 0 \qquad \beta_{Y_3} = (\neg e \wedge Y_3) \vee (e \wedge ((Y_1 \wedge Y_2) \oplus Y_3))$$

$Y_i^{\mathcal{S}_{cnt3}}(x)$, with $x \in \mathbb{B}^+$, is the value of the ith output bit at time $|x|$ of the 3-bit counter, where x encodes the enable input signal.

3.2 Equivalence of LIF Systems and AWAs

A function $g : (\mathbb{B}^r)^+ \to \mathbb{B}$ is *LIF-representable* if there exists a LIF system $\mathcal{S}$ and a $f \in F$, where $g(w) = f^{\mathcal{S}}(w)$ for all $w \in (\mathbb{B}^r)^+$. A language $L \subseteq (\mathbb{B}^r)^+$ is *LIF-representable* if its characteristic function $g : (\mathbb{B}^r)^+ \to \mathbb{B}$, where $g(w) = 1$ iff $w \in L$, is LIF-representable. Gupta and Fisher have shown in [8,11] that any LIF-representable language is regular. They prove that their data-structure for representing a LIF system corresponds to a minimal deterministic automaton that accepts the language $\{x^{\mathrm{R}} \mid x \in (\mathbb{B}^r)^+ \text{ and } f^{\mathcal{S}}(x) = 1\}$.

We present here a simpler proof of regularity by showing that LIF systems directly correspond to AWAs. We also prove a weakened form of the converse: almost all regular languages are LIF-representable. The weakening though is trivial and concerns the empty word, and if we consider languages without the empty word we have an equivalence.[1] Hence, for the remainder of this section, *we consider only automata (languages) that do not accept (include) the empty word* λ.

For technical reasons we will work with LIF systems in a kind of negation normal form. A Boolean formula $\beta \in B(X)$ is *positive in* $Y \subseteq X$ if negations occur only directly in front of the Boolean variables $v \in X \setminus Y$ and, furthermore, the only other connectives used are $\wedge$ and $\vee$. A LIF system $\mathcal{S}$ is in *normal form* if β_f is positive in F, for each $f \in F$.

Lemma 1 *Let $\mathcal{S}$ be a LIF system over V and F. Then there is a LIF system $\mathcal{S}'$ over V and $F' = F \uplus \{\overline{f} \mid f \in F\}$ in normal form where, for all $f \in F$ and $x \in (\mathbb{B}^r)^+$,*

$$f^{\mathcal{S}'}(x) = f^{\mathcal{S}}(x) \qquad \textit{and} \qquad \overline{f}^{\mathcal{S}'}(x) = 1 \textit{ iff } f^{\mathcal{S}}(x) = 0\,.$$

PROOF. Without loss of generality, we assume that for $\beta \in B(X)$ only the connectives $\neg$, $\vee$, and $\wedge$ occur. The other connectives can be eliminated as standard, which may lead to exponentially larger formulae. (We will return to this point in the proof of Theorem 3.) By $\mathrm{nnf}(\beta)$ we denote the negation normal form of $\beta \in B(X)$.

By using the same idea as in [4], it is easy to construct a LIF system $\mathcal{S}'$ by introducing for each $f \in F$ a new variable $\overline{f}$ that "simulates" $\neg f$. Let $\mathcal{S} = (\mathsf{E}, \eta)$. For $f \in F$, with $\eta(f) = (\alpha, \beta)$, the mapping η' of the LIF system $\mathcal{S}'$ is defined by $\eta'(f) = (\alpha, \gamma)$ and $\eta'(\overline{f}) = (\neg\alpha, \overline{\gamma})$, where γ and $\overline{\gamma}$ are obtained from $\mathrm{nnf}(\beta)$, respectively $\mathrm{nnf}(\neg\beta)$, by replacing the sub-formulae $\neg f_i$ by $\overline{f}_i$. The claim follows by an induction over the length of the input. □

We now prove that LIF-representable languages and (λ-free) regular languages coincide.

Theorem 2 *LIF systems are equivalent to AWAs. In particular:*

[1] We can easily redefine LIFs to define functions over $(\mathbb{B}^r)^*$. However, following Gupta and Fisher we avoid this as the degenerate base case (0 length input) is ill-suited for modeling parametric circuits. Ignoring the empty word is immaterial for our complexity and algorithmic analysis.

(i) Given an AWA $\mathcal{A} = (\mathbb{B}^r, Q, q_0, F, \delta)$, there is a LIF system $\mathcal{S}$ in normal form over $V = \{v_1, \ldots, v_r\}$ and Q such that for all $x \in (\mathbb{B}^r)^+$ and $q \in Q$,

$$q^{\mathcal{S}}(x) = 1 \qquad \textit{iff} \qquad \mathcal{A} \textit{ accepts } x^{\mathrm{R}} \textit{ from } q\,.$$

(ii) Given a LIF system $\mathcal{S}$ in normal form over V and F, there exists an AWA $\mathcal{A}$ with states $F \uplus \{q_{base}, q_{step}\}$ such that for all $x \in (\mathbb{B}^r)^+$ and $f \in F$,

$$\mathcal{A} \textit{ accepts } x \textit{ from } f \qquad \textit{iff} \qquad f^{\mathcal{S}}(x^{\mathrm{R}}) = 1\,.$$

PROOF. (i) We encode each $b \in \mathbb{B}^r$ by a formula $\gamma_b \in B(V)$. For example, $(0,1,1,0) \in \mathbb{B}^4$ is encoded as the Boolean formula $\gamma_{(0,1,1,0)} = \neg v_1 \wedge v_2 \wedge v_3 \wedge \neg v_4$. The LIF expression for q in $\mathcal{S}$ is given by

$$\alpha_q = \bigvee_{b \in \mathbb{B}^r} \Big(\gamma_b \wedge B(q,b)\Big) \qquad\qquad \beta_q = \bigvee_{b \in \mathbb{B}^r} \Big(\gamma_b \wedge \delta(q,b)\Big)$$

with $B(q,b) = 1$ iff $F \models \delta(q,b)$. Here, the Boolean formula β_q simulates the transition from the state q on a non-final letter of the input word. The final state set F is simulated by the Boolean formula α_q, i.e., $F \models \delta(q,b)$ iff $\{v_i \mid b_i = 1\} \models \alpha_q$.

We prove (i) by induction over the length of $x \in (\mathbb{B}^r)^+$. If $|x| = 1$, then the equivalence follows from the definition of α_q, for any $q \in Q$. Assume that (i) is true for the word x, i.e., for each $q_k \in Q$, $\mathcal{A}$ accepts x^{R} from q_k iff $q_k^{\mathcal{S}}(x) = 1$. Let n be the length of x and let y be an evaluation of $\mathcal{S}$ on x with $y_n = (y_{n,1}, \ldots, y_{n,|Q|})$. It holds that $q_k^{\mathcal{S}}(x) = 1$ iff $c_k = 1$. We prove (i) for xb with $b = (b_1, \ldots, b_r) \in \mathbb{B}^r$. As defined, for each $q \in Q$, we have $q^{\mathcal{S}}(xb) = 1$ iff

$$\{v_l \mid 1 \le l \le r \text{ and } b_l = 1\} \cup \{q_l \mid 1 \le l \le |Q| \text{ and } y_{n,l} = 1\} \models \bigvee_{b' \in \mathbb{B}^r} \Big(\gamma_{b'} \wedge \delta(q,b')\Big)\,.$$

By the induction hypothesis, we obtain

$$\{q_l \mid \mathcal{A} \text{ accepts } x^{\mathrm{R}} \text{ from } q_l, \text{ for } 1 \le l \le |Q|\} \models \delta(q_k, b)\,.$$

From this we can easily construct an accepting run of $\mathcal{A}$ from q on $(xb)^{\mathrm{R}}$. The other direction holds by the definition of an accepting run.

(ii) For an arbitrary $g \in F$, let $\mathcal{A} = (\mathbb{B}^r, F \uplus \{q_{base}, q_{step}\}, g, \{q_{base}\}, \delta)$ with $\delta(q_{base}, b) = 0$, $\delta(q_{step}, b) = q_{base} \vee q_{step}$, and for $f \in F$

$$\delta(f, (b_1, \ldots, b_r)) = \Big(q_{step} \wedge \beta_f[v_1/b_1, \ldots, v_r/b_r]\Big) \vee \begin{cases} q_{base} & \text{if } \{v_i \mid b_i = 1\} \models \alpha_f, \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, when $\mathcal{A}$ is in state $f \in F$ and reads $(b_1, \ldots, b_r) \in \mathbb{B}^r$ it guesses if the base case is reached. When this is the case, the next state is q_{base} iff

$\{v_i \mid b_i = 1\} \models \alpha_f$. Otherwise, if the base case is not reached, the AWA proceeds according to the step case given by the Boolean formula β_f of the LIF system. The equivalence is proved in a similar way to (i). □

Note that if a LIF expression only contains the connectives $\neg$, $\wedge$ and $\vee$, then, following the proof of Lemma 1, a normal form can be obtained in polynomial time. Moreover, if V has a fixed size, the AWA $\mathcal{A}$ of Theorem 2(ii) can be constructed in linear time in $|F|$ since the size of the alphabet $\mathbb{B}^r$ is a constant. However, if we allow V to vary, then the size of the alphabet of the AWA constructed can be exponentially larger than the lengths of the formulae of the given LIF system.

3.3 Deciding LIF Equality

Given LIF systems $\mathcal{S}$ over V and F, and $\mathcal{T}$ over V and G, and function symbols $f \in F$ and $g \in G$, the *equality problem for LIFs* is to decide whether $f^{\mathcal{S}} = g^{\mathcal{T}}$. We first show that this problem is PSPACE-complete and afterwards show how, using BDDs, the construction in Theorem 2 provides the basis for an efficient implementation.

Theorem 3 *The equality problem for LIFs is PSPACE-complete.*

PROOF. We reduce the emptiness problem for AWAs, which is PSPACE-hard [14,19], to the equality problem for LIFs. Given an AWA $\mathcal{A}$ with initial state q_0, by Theorem 2(i) we can construct an equivalent LIF system $\mathcal{S}$ in polynomial time. Let the LIF system $\mathcal{T}$ be given by the formulae $\alpha_g = 0$ and $\beta_g = 0$. Then $q_0^{\mathcal{S}} = g^{\mathcal{T}}$ iff $L(\mathcal{A}) = \emptyset$.

Theorem 2(ii) cannot be used to show that the problem is in PSPACE because, as explained in the previous section, both the normal form and the size of the alphabet of the two constructed AWAs can be exponentially larger than the lengths of the LIF expressions. Hence, we instead give a direct proof. Let the LIF system $\mathcal{S}$ over V and F, and the LIF system $\mathcal{T}$ over V and G, and function symbols $f \in F$ and $g \in G$ be a problem instance of the equality problem for LIFs. The following Turing machine $\mathcal{M}$ accepts the problem instance in PSPACE iff a word $x = x_1 \ldots x_n \in (\mathbb{B}^r)^+$ exists with $f^{\mathcal{S}}(x) \neq g^{\mathcal{T}}(x)$. Let $y = y_1 \ldots y_n \in (\mathbb{B}^{|F|})^+$ be the evaluation of $\mathcal{S}$ on x and $y'_1 \ldots y'_n \in (\mathbb{B}^{|G|})^+$ be the evaluation of $\mathcal{T}$ on x. $\mathcal{M}$ guesses in the ith step $x_i \in \mathbb{B}^r$ and calculates $y_i = (y_{i,1}, \ldots, y_{i,|F|})$ and $y'_i = (y'_{i,1}, \ldots, y'_{i,|G|})$ of the evaluations. If $y_{i,k} \neq y'_{i,l}$ then $\mathcal{M}$ accepts the instance and otherwise $\mathcal{M}$ continues with the $(i+1)$st step. Note that for the ith step only y_{i-1} and y'_{i-1} and x_i are required to calculate y_i and y'_i. Hence $\mathcal{M}$ runs in polynomial space since, in the ith step,

it only requires space $|V|$ to store x_i and space $2(|F| + |G|)$ to store y_{i-1}, y_i, x'_{i-1}, and y'_i. $\mathcal{M}$ needs linear time in the size of the LIF expressions of $\mathcal{S}$ and $\mathcal{T}$ to calculate y_i and y'_i from x_i, y_{i-1} and y'_{i-1}. Since PSPACE is closed under nondeterminism and complementation, the equality problem for LIFs is in PSPACE. □

Although the machinery of alternating automata may appear a bit heavy, it leads to simple translations as there is a direct correspondence between function symbols in a LIF system and states in the corresponding AWA. This would not be possible using nondeterministic automata. Since the emptiness problem for NWAs is LOGSPACE-complete and the equality problem for LIFs is PSPACE-complete, a translation of a LIF system to a nondeterministic automata leads, in general, to an exponential blow-up in the state space.

Implementation Gupta and Fisher formalize LIFs using a data-structure based on multi-terminal BDDs where terminal nodes are both the constants 0 and 1 as well as pointers to other BDDs. They prove that each LIF system has a representation that can be obtained in $O(2^{2^{|F|}}2^{|V|})$ time and space in the worst-case. The representation can be made canonical in time $O(n^2)$ where n is size of their representation; in the worst-case n is $2^{2^{|F|}}$ as the following example shows.

For $n \geq 1$, let L_n be the set of words $w \in \{0,1\}^*$ where the nth letter of w is 1. We can define a LIF system $\mathcal{S}_n$ over $V = \{x\}$ and $F = \{f_1, \ldots, f_{\lceil \log n \rceil}, g, h\}$ such that $L_{n+1} = \{w \in \{0,1\}^* \mid h(w) = 1\}$. For example, the LIF system $\mathcal{S}_8$ is given by

$$\begin{array}{ll}
\alpha_{f_1} = 0 & \beta_{f_1} = \neg f_1 \\
\alpha_{f_2} = 0 & \beta_{f_2} = f_1 \oplus f_2 \\
\alpha_{f_3} = 0 & \beta_{f_3} = (f_1 \wedge f_2) \oplus f_3 \\
\alpha_g = 0 & \beta_g = g \vee (\neg x \wedge f_1 \wedge f_2 \wedge f_3 \wedge \neg g \wedge \neg h) \\
\alpha_h = 0 & \beta_h = h \vee (x \wedge f_1 \wedge f_2 \wedge f_3 \wedge \neg g \wedge \neg h)
\end{array}$$

A theorem by Gupta and Fisher in [8, Theorem 3] states that their canonical representation of a LIF coincides with the minimal deterministic word automaton that accepts the reverse language of the LIF. In this example, the minimal deterministic word automaton that accepts L_n^{R} has at least $O(2^n)$ states.

Given the canonical representation of the LIF systems it is easy to decide whether two LIFs are equal. But the worst-case complexity of Gupta and Fisher's procedure is double exponential. This is depicted graphically in the left half of Figure 3. In the following we will explain the right half of Figure 3,

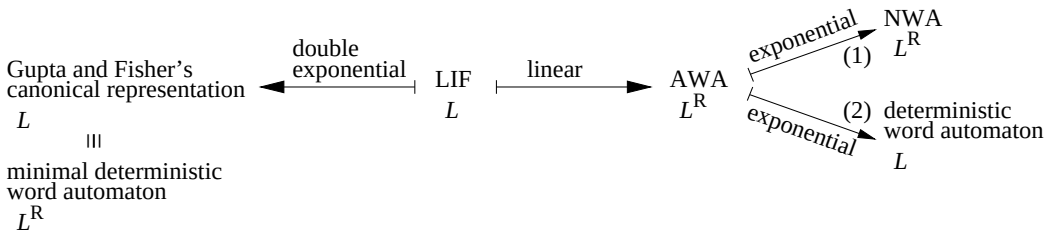


Fig. 3. Decision procedures for the LIF equality problem.

which represents a practical alternative to the PSPACE decision procedure given in the proof of Theorem 3. Recall that in the proof of Theorem 3 we did not use the mapping from LIFs to AWAs given by Theorem 2(ii) due to the possible exponential blow-up when normalizing the LIF system, and the exponential blow-up in representing the AWA's alphabet. We now describe an alternative where Theorem 2(ii) is employed and these blow-ups can sometimes be avoided by using BDDs. That is, we use alternating word automata to give a decision procedure that runs in exponential time (an exponential improvement over Gupta and Fisher's decision procedure) and that often will run in polynomial space.

Since the size of $\mathcal{A}$'s alphabet ($\mathbb{B}^r$) is exponential in $|V|$, we use the same idea that is used in the MONA system and that Gupta and Fisher employed for their representation of LIFs: instead of explicitly representing the exponentially large alphabet, we use BDDs to represent the transition function. For example, Figure 4 depicts the representation of the AWA $\mathcal{A} = (\mathbb{B}, \{q_0, q_1, q_2\}, q_0, \{q_2\}, \delta)$ with the transition function

$$\begin{aligned} \delta(q_0, 0) &= q_1 \wedge q_2 & \qquad \delta(q_0, 1) &= q_1 \\ \delta(q_1, 0) &= q_1 \wedge q_2 & \qquad \delta(q_1, 1) &= q_1 \vee q_2 \\ \delta(q_2, 0) &= 1 & \qquad \delta(q_2, 1) &= q_1. \end{aligned}$$

The solid [respectively dashed] lines correspond to the variable assignment 1 [respectively 0]. For example, the state q_0 has a pointer to a BDD whose first node (labeled a) encodes the alphabet; the solid line from this node points to a BDD representing $\delta(q_0, 1) = q_1$ and the dashed line points to a BDD representing $\delta(q_0, 0) = q_1 \wedge q_2$.

To decide if an AWA $\mathcal{A}$ accepts the empty language there are now two possible constructions (depicted at the right in Figure 3):

(1) We can construct an equivalent NWA and test if it accepts the empty language.
(2) We can construct a deterministic word automaton that accepts the reverse language of $\mathcal{A}$ and test this language for emptiness.

As a language L is empty iff its reversal L^{R} is empty, both constructions

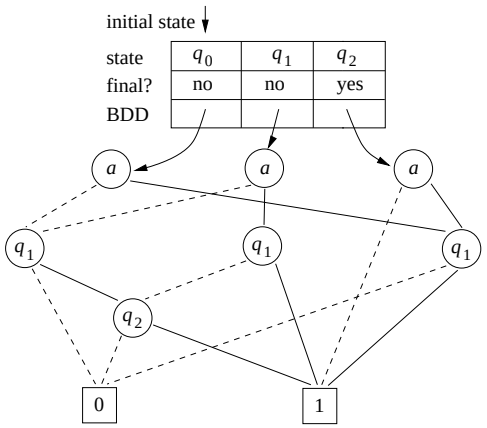


Fig. 4. Representation of an AWA.

return the same results. Both build, in worst case, an automaton that has exponentially more states than $\mathcal{A}$.

We only describe (2) in more detail, the construction of the deterministic word automata since it has the following advantages: the deterministic word automaton recognizes the same language as the LIF, and the constructed deterministic word automaton can be minimized in polynomial time to get a canonical representation for the LIF. The canonical representation can then be used to decide equality. However, in practice it is often more efficient to reduce equality to testing emptiness of the language accepted by an AWA as described below.

Given the AWA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ we can construct a deterministic word automaton $\mathcal{B}$ that accepts the reverse language of $\mathcal{A}$ as follows. Let $\mathcal{B} = (\Sigma, \mathcal{P}(Q), F, \delta', \{P \subseteq Q \mid q_0 \in P\})$, where the transition function $\delta' : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ is defined as $\delta'(P, a) = \{q \in Q \mid P \models \delta(q, a)\}$. Since we are only interested in testing whether the AWA $\mathcal{A}$ accepts the empty language, we can construct the deterministic word automaton $\mathcal{B}$ "on-the-fly", that is, we construct the reachable state space of $\mathcal{B}$ only as necessary to answer the emptiness question. Figure 5 shows an algorithm that builds this reachable state space starting from $\mathcal{B}$'s initial state F. If a final state of $\mathcal{B}$ is reached, i.e., a subset of Q containing q_0, the algorithm returns that $\mathcal{A}$ does not accept the empty language. If none of the reachable states of $\mathcal{B}$ is final, the algorithm returns that $\mathcal{A}$ accepts the empty language.

To analyze the complexity, observe that the **while**-loop is iterated at most $2^{|Q|}$-times and the calculation in each iteration requires $O(2^{|Q|}|\Sigma|)$-time. Hence the worst-case running time is $O(|\Sigma|2^{2|Q|})$. We need two vectors of the length $2^{|Q|}$ to represent the sets *Current* and *Processed*. Hence the required space is the maximum of $O(2^{|Q|})$ and the size of the representation of the AWA $\mathcal{A}$. The sets $\mathit{Current}, \mathit{Processed} \subseteq \mathcal{P}(Q)$ can be encoded as BDDs. This use of BDDs to represent the characteristic function of the set will sometimes achieve an

INPUT: AWA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$

Current := $\{F\}$;
Processed := $\emptyset$;
while *Current* $\neq \emptyset$ **do begin**
  if *Current* $\cap \{P \subseteq Q \mid q_0 \in P\} \neq \emptyset$ **then return** *"nonempty"*;
  else begin
    Processed := *Processed* $\cup$ *Current*;
    $X := \emptyset$;
    for each $P \in$ *Current* **do** $X := X \cup \big\{\{p \in Q \mid P \models \delta(p, a) \text{ and } a \in \Sigma\}\big\}$;
    Current := $X \setminus$ *Processed*;
  end else;
end while;
return *"empty"*;

Fig. 5. Decision procedure for the emptiness problem for AWAs.

exponential savings in space.

The reduction of LIF equality to the emptiness problem for AWAs is straightforward. From the LIF systems $\mathcal{S}$ over V and F, and $\mathcal{T}$ over V and G, we construct the LIF system $\widetilde{\mathcal{S}}$ over V and $\{\widetilde{f}\} \uplus F \uplus G$ with the additional LIF expression $\alpha_{\widetilde{f}} = \neg(\alpha_f \leftrightarrow \alpha_g)$ and $\beta_{\widetilde{f}} = \neg(\beta_f \leftrightarrow \beta_g)$. We then normalize $\widetilde{\mathcal{S}}$ and use Theorem 2(ii) to construct the AWA $\mathcal{A}$ with the initial state $\widetilde{f}$. By construction, $L(\mathcal{A}) \neq \emptyset$ iff $\widetilde{f}^{\widetilde{\mathcal{S}}}(x) = 1$ for some $x \in (\mathbb{B}^r)^+$ iff $f^{\mathcal{S}} \neq g^{\mathcal{T}}$.

Despite its worse space complexity, our algorithm based on BDD-represented AWAs may give better results in practice than our PSPACE decision procedure. This depends on whether the BDDs used require polynomial or exponential space. If the space required is polynomial, then the resulting AWA and its emptiness test requires only polynomial space. In the exponential case, as there are only $2|F| + 2$ states, the emptiness test requires $O(2^{|V|+2|F|})$ space and $O(2^{|V|+4|F|})$ time. This case also represents an exponential improvement over Gupta and Fisher's results, both in time and space.

4 Exponentially Inductive Boolean Functions

The structure of this section parallels that of §3. After defining EIFs, we show that their equality problem can be decided using tree automata. The decision procedure however is not as direct as it is for LIFs. One problem is that inputs to EIFs are words not trees. We solve this by labeling the interior nodes of trees with a dummy symbol. However, the main problem is that the word length must be a power of two. This restriction cannot be checked by

tree automata and we solve this by deciding separately if a tree automaton accepts a complete tree.

4.1 Definition of EIFs

Syntax An *EIF expression* (over V and F) is a pair (α, β), with $\alpha \in B(V)$ and $\beta \in B(F \times \{0,1\})$. We write f^0 [respectively f^1] for the variable $(f,0)$ [respectively $(f,1)$] in $F \times \{0,1\}$. An *EIF system* (over V and F) is a pair $\mathcal{S} = (\mathsf{E}, \eta)$, where E and η are defined as for a LIF system. Similarly to LIF systems, we write (α_f, β_f) for $\eta(f) = (\alpha, \beta)$.

Semantics We define the semantics of an EIF system in a similar way to LIF systems. For a tree $t \in (\mathbb{B}^r)^{\mathrm{T}*}$, we write $t(u)_j$ to denote the jth coordinate of $t(u)$. Let $\mathcal{S}$ be an EIF system over $V = \{v_1, \ldots, v_r\}$ and $F = \{f_1, \ldots, f_s\}$. An *evaluation* of $\mathcal{S}$ on a word $x = x_1 \ldots x_{2^n} \in (\mathbb{B}^r)^+$ is a complete binary $\mathbb{B}^s$-labeled tree y of height $n+1$ such that for $1 \leq k \leq s$ and $u \in \mathrm{dom}(y)$:

(i) If u is the ith leaf of $\mathrm{dom}(y)$ (where the leaves are ordered lexicographically), then

$$y(u)_k = 1 \quad \text{iff} \quad \{v_l \mid 1 \leq l \leq r \text{ and } x_{i,l} = 1\} \models \alpha_{f_k} .$$

(ii) If $u \in \mathrm{dom}(y)$ is an inner node, then

$$y(u)_k = 1 \quad \text{iff} \quad \{f_i^b \mid 1 \leq l \leq s \text{ and } y(ub)_l = 1\} \models \beta_{f_k} .$$

Let $\Sigma^{2+} = \{w \in \Sigma^* \mid n \in \mathbb{N} \text{ and } |w| = 2^n\}$. As with LIFs, the evaluation y is uniquely defined; hence $f_k \in F$ and $\mathcal{S}$ together define a function $f_k^{\mathcal{S}} : (\mathbb{B}^r)^{2+} \to \mathbb{B}$, namely $f_k^{\mathcal{S}}(x) = y(\lambda)_k$. The notion *EIF-representable* is defined analogously to LIF-representable.

For example, the tree implementation of the parameterized parity circuit from the introduction is described by the EIF system $\mathcal{S}_{tp}$ over $V = \{x\}$ and $F = \{\mathit{tree_parity}\}$ with the EIF expression

$$\alpha_{\mathit{tree_parity}} = x \qquad \beta_{\mathit{tree_parity}} = \mathit{tree_parity}^0 \oplus \mathit{tree_parity}^1 .$$

Here the value of the EIF $\mathit{tree_parity}^{\mathcal{S}_{tp}}$ applied to a word $w = b_1 \ldots b_{2^n} \in \mathbb{B}^+$ is the value of the function parity^{2^n} applied to $(b_1, \ldots, b_{2^n})$.

As a second example, and one less trivial, we present a family of circuits used to calculate the propagate and generate bits for a carry lookahead adder. The members of this family can be used to calculate the sum of two binary

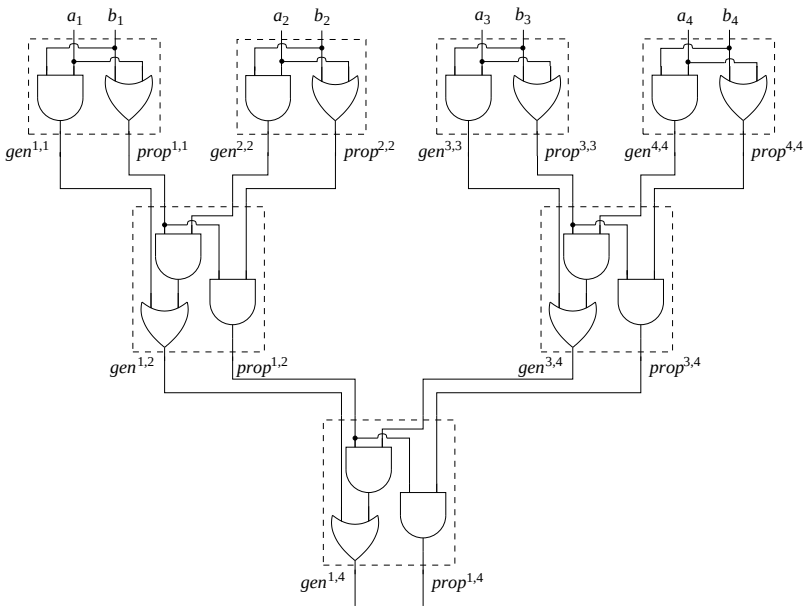


Fig. 6. Circuit for $prop^{1,4}$ and $gen^{1,4}$.

numbers by calculating the carry bits in parallel; a detailed description of a carry lookahead adder can be found in [5].

In particular, let $a_1 \dots a_{2^n} \in \mathbb{B}^{2+}$ and $b_1 \dots b_{2^n} \in \mathbb{B}^{2+}$ be the binary representations of the two natural numbers a and b. And let $s_1 \dots s_{2^n} s_{2^n+1}$ be a binary representation of $s = a + b$. The bits of s can be calculated with the functions $prop^{i,i'}, gen^{i,i'} : (\mathbb{B}^2)^+ \to \mathbb{B}$ with $prop^{i,i'}(\binom{a_i}{b_i} \dots \binom{a_{i'}}{b_{i'}}) = 1$ iff a carry bit is propagated from digit i to the digit i', and $gen^{i,i'}(\binom{a_i}{b_i} \dots \binom{a_{i'}}{b_{i'}}) = 1$ iff a carry bit is generated from digit i to digit i'. For $1 \leq i \leq 2^n$, $s_i = a_i \oplus b_i \oplus gen^{1,i}(\binom{a_1}{b_1} \dots \binom{a_i}{b_i})$ and $s_{2^n+1} = gen^{1,2^n+1}(\binom{a_1}{b_1} \dots \binom{a_{2^n}}{b_{2^n}})$. $prop^{i,i'}$ is needed to calculate $gen^{i,i'}$. A circuit for $prop^{1,4}$ and $gen^{1,4}$ is depicted in Figure 6.

The EIF system $\mathcal{S}_{cla}$ over $V = \{a, b\}$ and $F = \{prop, gen\}$ given by the following EIF expressions represents the functions $prop^{1,2^n}$ and $gen^{1,2^n}$ for any $n \geq 0$:

$$\begin{aligned} \alpha_{prop} &= a \vee b & \beta_{prop} &= prop^0 \wedge prop^1 \\ \alpha_{gen} &= a \wedge b & \beta_{gen} &= (gen^0 \wedge prop^1) \vee gen^1 \,. \end{aligned}$$

If we have additionally a carry-in bit c_{in}, we can extend $\mathcal{S}_{cla}$ to the EIF system over $V = \{a, b, c_{in}\}$ and $F = \{prop, gen, gen_0\}$ with the additional EIF expression

$$\alpha_{gen_0} = (a \wedge b) \vee ((a \vee b) \wedge c_{in} \qquad \beta_{gen_0} = (gen_0^0 \wedge prop^1) \vee gen^1 \,.$$

Let $s_1 \dots s_{2^n} s_{2^n+1}$ denote the binary representation of the sum of $a + b + c_{in}$. We have that

$$s_{2^n+1} = gen_0^{\mathcal{S}_{cla}}(\begin{pmatrix} a_1 \\ b_1 \\ c_{in} \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ - \end{pmatrix} \dots \begin{pmatrix} a_{2^n} \\ b_{2^n} \\ - \end{pmatrix}),$$

where the symbol "$-$" denotes an arbitrary value of $\mathbb{B}$ (i.e., $gen_0^{\mathcal{S}_{cla}}$ does not depend on these values).

4.2 Equivalence of EIF Systems and ATAs

Using ATAs we can characterize the EIF-representable functions. To interpret a word in the domain of an EIF as a tree, we identify a word $b_1 \dots b_{2^n} \in \Sigma^*$ with the complete tree $t \in \Sigma_{\#}^{\mathrm{T}*}$, where $\mathrm{front}(t) = b_1 \dots b_{2^n}$ and all inner nodes are labeled with the dummy symbol $\#$.

Normal forms for EIF systems can be defined and obtained as for LIF systems and the proof of Theorem 2 can, with minor modifications, be generalized to EIFs.

Theorem 4 *EIF systems are equivalent to ATAs if the input trees are restricted to complete leaf-labeled trees. In particular:*

(i) Let $\mathcal{A} = (\mathbb{B}_{\#}^r, Q, q_0, F, \delta)$ be an ATA. There is a normal form EIF system $\mathcal{S}$ over $V = \{v_1, \dots, v_r\}$ and Q, such that for all $q \in Q$ and any complete $\mathbb{B}^r$-leaf-labeled tree $t \in (\mathbb{B}_{\#}^r)^{\mathrm{T}+}$,

$$q^{\mathcal{S}}(\mathrm{front}(t)) = 1 \qquad \textit{iff} \qquad \mathcal{A} \textit{ accepts } t \textit{ from } q\,.$$

(ii) Let $\mathcal{S}$ be a normal form EIF system over V and F. There is an ATA $\mathcal{A}$ with the state set F, such that for any complete $\mathbb{B}^r$-leaf-labeled tree $t \in (\mathbb{B}_{\#}^r)^{\mathrm{T}+}$,

$$\mathcal{A} \textit{ accepts } t \textit{ from } f \qquad \textit{iff} \qquad f^{\mathcal{S}}(\mathrm{front}(t)) = 1\,.$$

PROOF. For (i), we only give the EIF system $\mathcal{S}$ over $V = \{v_1, \dots, v_r\}$ and Q. Let γ_b, for $b \in \mathbb{B}^r$, be the Boolean formula as in the proof of Theorem 2(i). For $q \in Q$ let

$$\alpha_q = \bigvee_{b \in \mathbb{B}^r} (\gamma_b \wedge B(q, b)) \qquad \text{and} \qquad \beta_q = \delta(q, \#)$$

with $B(q, b) = 1$ iff $F \times \{0, 1\} \models \delta(q, b)$. It is straightforward to show by induction over the height of a $\mathbb{B}^r$-leaf-labeled tree $t \in (\mathbb{B}_{\#}^r)^{\mathrm{T}+}$ that $q^{\mathcal{S}}(\mathrm{front}(t)) = 1$ iff $\mathcal{A}$ accepts t from q, for all $q \in Q$.

For (ii), let $\mathcal{S}$ be an EIF system over $V = \{v_1, \ldots, v_r\}$ and F in normal form and let $f_0 \in F$. The equivalent ATA $\mathcal{A}$ is defined as follows: $\mathcal{A} = (\mathbb{B}^r_\#, F, f_0, \emptyset, \delta)$ with $\delta(f, \#) = \beta_f$ and

$$\delta(f, (b_1, \ldots, b_r)) = \begin{cases} 1 & \text{if } \{v_i \in V \mid 1 \leq i \leq r \text{ and } b_i = 1\} \models \alpha_f, \\ 0 & \text{otherwise,} \end{cases}$$

for $f \in F$ and $b_1, \ldots, b_r \in \mathbb{B}$. The claim can be proved by induction over the height of the complete $\mathbb{B}^r$-leaf-labeled trees. □

The following lemma shows that the above restriction to leaf-labeled tree languages does not result in a loss of expressive power.

Lemma 5 *There is an injective function $c : \Sigma^{\mathrm{T}*} \to \Sigma^{\mathrm{T}*}_\#$ such that (i) $c(t)$ is a Σ-leaf-labeled tree and (ii) t is complete iff $c(t)$ is complete. Moreover, if $L \subseteq \Sigma^{\mathrm{T}*}$ is regular then $c(L) = \{c(t) \mid t \in L\}$ is regular.*

PROOF. First, we define the function c. The empty tree is mapped by c to itself. Now, let t be a nonempty tree over Σ and $a_0 \in \Sigma$. The inner nodes of $c(t)$ are the nodes of t and ub is a leaf of $c(t)$ if u is a node of t, i.e. $\mathrm{dom}(c(t)) = \{\lambda\} \cup \{ub \mid u \in \mathrm{dom}(t) \text{ and } b \in \{0,1\}\}$. The labeling of the node u of t can be found at the node of $c(t)$ by passing to u's right successor and from there by left successors down to the frontier. The leaf $0 \ldots 0 \in \mathrm{dom}(c(t))$ is labeled by the dummy symbol a_0. Formally,

$$(c(t))(u) = \begin{cases} \# & \text{if } u \in \mathrm{dom}(t), \\ t(v) & \text{if } u = v01 \ldots 1 \text{ and } u \text{ is a leaf of } c(t), \\ a_0 & \text{otherwise.} \end{cases}$$

Obviously, c is injective and satisfies (i) and (ii).

Let $L \subseteq \Sigma^{\mathrm{T}*}$ be a regular tree language and let $\mathcal{A} = (\Sigma, Q, q_0, \delta, F)$ be an ATA that accepts L. We show that $c(L)$ is regular by constructing an ATA $\mathcal{A}'$ with $L(\mathcal{A}') = c(L)$. Before we define formally $\mathcal{A}'$ we describe intuitively its transition function. If $\mathcal{A}'$ reads the letter $\#$ at node v of $c(t)$ then it guesses the letter $t(v)$ and whether v is a leaf or an inner node in t. If v is an inner node in t then $\mathcal{A}'$ makes a transition according to the guessed letter $t(v)$ and the transition function δ of $\mathcal{A}$. If v is a leaf in t then $\mathcal{A}'$ can only make a transition if $F \times \{0,1\}$ is a model of the Boolean formula of the transition of $\mathcal{A}$ according to the guessed letter $t(v)$. $\mathcal{A}'$ verifies its guess of the label $t(v)$ by checking that the leaf $v10 \ldots 0$ of $c(t)$ is labeled with the letter $t(v)$. Further, $\mathcal{A}'$ checks that the leaf $0 \ldots 0$ of the input tree is labeled with a_0.

Formally, the set of states of $\mathcal{A}'$ is the set $Q \uplus \{q_b \mid b \in \Sigma\} \uplus \{p_b \mid b \in \Sigma\} \uplus \{q_{ok}, q_{\#}, q_{leaf}, s_{a_0}\}$ and the set of final states is $\{q_{ok}\}$. The initial Boolean formula of $\mathcal{A}'$ is $(q_0 \wedge q_{\#} \wedge s_{a_0}) \vee q_{ok}$ if $q_0 \in F$, and $q_0 \wedge q_{\#} \wedge s_{a_0}$ otherwise.

For $a \in \Sigma$, the transition function ρ of $\mathcal{A}'$ is defined

$$\begin{array}{ll}
\rho(q_{ok}, \#) = 0 & \rho(q_{ok}, a) = 0 \\
\rho(q_{\#}, \#) = (q^0_{\#} \wedge q^1_{\#}) \vee (q^0_{leaf} \wedge q^1_{leaf}) & \rho(q_{\#}, a) = 0 \\
\rho(q_{leaf}, \#) = 0 & \rho(q_{leaf}, a) = q^0_{ok} \wedge q^1_{ok} \\
\rho(s_{a_0}, \#) = s^0_{a_0} & \rho(s_{a_0}, a) = \begin{cases} q^0_{ok} \wedge q^1_{ok} & \text{if } a = a_0, \\ 0 & \text{otherwise.} \end{cases}
\end{array}$$

For the states q_b and p_b with $b \in \Sigma$, ρ is defined as

$$\begin{array}{ll}
\rho(p_b, \#) = q^0_b & \rho(p_b, a) = 0 \\
\rho(q_b, \#) = q^1_b & \rho(q_b, a) = \begin{cases} q^0_{ok} \wedge q^1_{ok} & \text{if } b = a, \\ 0 & \text{otherwise.} \end{cases}
\end{array}$$

For $q \in Q$, $\rho(q, a) = 0$ and

$$\rho(q, \#) = \left(q^0_{\#} \wedge q^1_{\#} \wedge \bigvee_{b \in \Sigma} (p^1_b \wedge \delta(q, b))\right) \vee \left(q^0_{leaf} \wedge q^1_{leaf} \wedge \bigvee_{b \in \Sigma} (q^1_b \wedge B(q, b))\right),$$

where $B(q, b) = 1$ iff $F \times \{0, 1\} \models \delta(q, b)$.

It is straightforward to prove $L(\mathcal{A}') = c(L)$. □

Remark 6 *The ATA $\mathcal{A}'$ in the above proof can be constructed in polynomial time.*

4.3 Deciding EIF Equality

The *equality problem for EIFs* is defined similarly to LIFs. We cannot generalize the decision procedure from §3.3 to EIFs since we are only interested in trees of a restricted form: complete leaf-labeled binary trees. Unfortunately completeness is not a regular property, i.e. one recognizable by tree automata, and hence we cannot reduce the problem to an emptiness problem. Instead, we reduce the problem to the *complete tree containment problem (CTCP) for NTAs*, which is to decide whether a given NTA accepts a complete tree.

Lemma 7 *CTCP for NTAs is in PSPACE.*

PROOF. For the NTA $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ we construct the AWA $\mathcal{A}' = (\{1\}, Q, q_0, F, \delta')$ with $\delta'(q,1) = \bigvee_{a\in\Sigma} \bigvee_{(p,p')\in\delta(q,a)} (p \wedge p')$. It is easy to prove that $\mathcal{A}$ accepts a complete tree of height h iff $\mathcal{A}'$ accepts a word of length h. From this follows that CTCP for NTAs is in PSPACE because the emptiness problem for AWAs is in PSPACE [14,19]. □

Theorem 8 *The equality problem for EIFs is EXPSPACE-complete.*

PROOF. First we show that the equality problem for EIFs is in EXPSPACE. Let $\mathcal{S}$ over V and F, and $\mathcal{T}$ over V and G, be EIF systems, and let $f \in F$ and $g \in G$ be given. Let $\widetilde{\mathcal{S}}$ be the EIF system over V and $\{\widetilde{f}\} \uplus F \uplus G$ with the additional EIF expression defined by $\alpha_{\widetilde{f}} = \neg(\alpha_f \leftrightarrow \alpha_g)$ and $\beta_{\widetilde{f}} = \neg(\beta_f \leftrightarrow \beta_g)$. We normalize $\widetilde{S}$ and by Theorem 4(ii) construct an ATA $\mathcal{A}$ with the initial state $\widetilde{f}$, such that $f^{\mathcal{S}} \neq g^{\mathcal{T}}$ iff $\mathcal{A}$ accepts a complete tree. $\mathcal{A}$ has $2|\{\widetilde{f}\} \uplus F \uplus G| + 2$ states and the size of the alphabet $\mathbb{B}_{\#}^{|V|}$ is $2^{|V|} + 1$. From $\mathcal{A}$ we can construct an equivalent NTA $\mathcal{B}$ that has at most $O(2^{2|F|+|G|})$ states. Hence we have reduced the equality problem for EIFs to CTCP for NTAs. The required space for the reduction is $O(2^{|V|} 2^{2|F|+|G|})$. From Lemma 7 it follows that the equality problem is in EXPSPACE.

It remains to show that the equality problem is EXPSPACE-hard. The *complete tree containment problem (CTCP) for ATAs* is to decide whether a given ATA accepts a complete tree. In Appendix A we show that CTAP for ATAs is EXPSPACE-hard (Theorem 11). Here we reduce CTCP for ATAs to the equality problem.

Let $\mathcal{A}$ be a given ATA. By Remark 6 we can construct an ATA $\mathcal{A}'$ with the initial state q_0 that accepts a complete leaf-labeled tree iff $\mathcal{A}$ accepts a complete tree. Let $\mathcal{S}$ be the EIF system that we obtain by the construction on $\mathcal{A}'$ from the proof of Theorem 4(i). Let the EIF system $\mathcal{T}$ be given by the formulae $\alpha_g = \beta_g = 0$. Then, $q_0^{\mathcal{S}} = g^{\mathcal{T}}$ iff $\mathcal{A}'$ accepts a complete leaf-labeled tree iff $\mathcal{A}$ accepts a complete leaf-labeled tree. Note that all the constructions can be accomplished in polynomial time. □

5 Comparisons and Related Work

Gupta and Fisher Our work was motivated by that of Gupta and Fisher [8–10] and we begin by comparing our formalization of LIFs and EIFs with theirs, which we will call LIF_0 and EIF_0.

For each $n \geq 1$, a LIF_0 f is given by a Boolean function, called the n-instance of f and denoted by f^n, where $f^1 : \mathbb{B}^r \to \mathbb{B}$ and $f^n : \mathbb{B}^{r+s} \to \mathbb{B}$ for $n > 1$

(r is the number of n-instance inputs and s is the number of $(n-1)$-instance function inputs). Further it must hold that, for all $m, n > 1$, the m-instance and the n-instance of f are equal, i.e. $f^m = f^n$. By means of the parity function we explain how the value of a LIF_0 is calculated. The n-instances of $serial_parity^n$ (using their notation) are:

$$serial_parity^1 = b^1 \qquad \text{and} \qquad serial_parity^n = b^n \oplus serial_parity^{n-1},$$

for $n > 1$. The *value* of the LIF_0 *serial_parity* on the word $b_1 \ldots b_n \in (\mathbb{B}^r)^+$, written as $serial_parity(b_1, \ldots, b_n)$, is the value of the 1-instance $serial_parity^1$ applied to b_1, for $n = 1$. For $n > 1$, it is the value of the n-instance $serial_parity^n$ applied to b_n and $serial_parity(b_1, \ldots, b_{n-1})$.

The definitions of a "LIF expression" and a "LIF system" correspond to the definition of a "LIF_0". Moreover, the way the "value" of a LIF_0 is calculated corresponds to our definition of "evaluation". Hence both formalisms are equivalent. However, the algorithms, data-structures, and complexity of our approaches are different. As explained in §3.3, we get a double exponential improvement from a construction using nondeterministic Turing machines and an exponential improvement in time and space for the LIF equality problem by reducing the problem to testing the emptiness of alternating word automata.

An EIF_0 f has, like a LIF_0, for each $n \geq 0$, an n-instance function f^{2^n}, where $f^1 : \mathbb{B}^r \to \mathbb{B}$ and, for $n > 0$, f^{2^n} is a Boolean combination of three EIF_0s, e, g and h, i.e. $f^{2^n} : \mathbb{B}^3 \to \mathbb{B}$. Further it must hold that $f^{2^m} = f^{2^n}$, for all $m, n > 0$. The *value* of the EIF_0 f on the word $b_1 \ldots b_{2^n} \in (\mathbb{B}^r)^+$, written as $f(b_1, \ldots, b_{2^n})$, is the value of the 0-instance f^1 applied to b_1, if $n = 0$. For $n > 0$, it is the value of the n-instance f^{2^n} applied to the value of the EIF_0 e of the left half of the word, i.e. $e(b_1, \ldots, b_{2^{n-1}})$, and to the values of the EIF_0s g and h of the right half of the word, i.e. $g(b_{2^{n-1}+1}, \ldots, b_{2^n})$ and $h(b_{2^{n-1}+1}, \ldots, b_{2^n})$. The restrictions of the n-instance function of an EIF_0 stems from the data-structure proposed for EIF_0s in [8,9] in order to have a canonical representation.

EIF_0s are strictly less expressive than EIFs; the reason for this is similar to why deterministic top-down tree automata are weaker than nondeterministic top-down tree automata. As an example, consider the function $F : \mathbb{B}^{2+} \to \mathbb{B}$, where $F(x) = 1$ iff $x = 0000$ or $x = 1100$ or $x = 1011$. Suppose that F were EIF_0-representable, that is, assume there is an EIF_0 f representing F. For $b_1, b_2, b_3, b_4 \in \mathbb{B}$, we have, by definition, $f(b_1, b_2, b_3, b_4) = f^4(e(b_1, b_2), g(b_3, b_4), h(b_3, b_4))$, where e, g, h are EIF_0s. In particular,

$$\begin{aligned} 1 &= f(1011) = f^4(e(10), g(11), h(11)) \\ 0 &= f(0011) = f^4(e(00), g(11), h(11)) \end{aligned} \tag{1}$$

and

$$\begin{aligned} 1 &= f(1011) = f^4(e(10), g(11), h(11)) \\ 0 &= f(0111) = f^4(e(01), g(11), h(11)) \,. \end{aligned} \tag{2}$$

From (1) it follows that $e(10) \neq e(00)$, and from (2) it follows that $e(10) \neq e(01)$. Thus, $e(00) = e(01)$ and we obtain the contradiction

$$0 = f(0100) = f^4(e(01), g(00), h(00)) = f^4(e(00), g(00), h(00)) = f(0000) = 1 \,.$$

On the other hand, F is easy to represent as an EIF.

Our results on the complexity of the equality problem for EIFs are, to our knowledge, the first such results given in the literature. Neither we nor Gupta and Fisher have implemented a decision procedure for the equality problem for EIFs or EIF$_0$s.

Approaches Based on Equation Systems There are also similarities between our formalization of LIFs and work of Brzozowski and Leiss on formalizing circuits by equations [3]. A *system of equations* S has the form $X_i = \bigcup_{a \in \Sigma} \{a\}.F_{i,a} \cup \delta_i$ (for $1 \leq i \leq n$), where the $F_{i,a}$ are Boolean functions in the variables X_i, and each δ_i is either $\{\lambda\}$ or $\emptyset$. It is shown in [3], using Boolean automata (a form of alternating automata on words), that a solution to S is unique and regular, i.e., if each X_i is interpreted with a $L_i \subseteq \Sigma^*$ and L_i satisfy the equations in S, then L_i is unique and regular. It is also shown how these systems of equations can be used to model sequential circuits (parameterized circuit families were not considered).

LIF systems offer advantages in describing parameterized circuit families. For example, with LIFs one directly models a circuit's input ports using the variables V. In contrast, with a system of equations, one must use the alphabet $\mathbb{B}^r$ and cannot "mix" input pins and the signals of the internal wiring (and the same holds for outputs) when describing circuits. Another distinction is that descriptions using LIFs cleanly separate the base and step cases of the circuit family, which is not the case with [3].

Note that Brzozowski and Leiss [3] do not provide analogous complexity bounds or consider implementation issues. Our complexity results and the use of BDDs in representing alternating automata should carry over into their setting.

Approaches Based on Monadic Logics The use of BDDs to represent word automata, without alternation, has been explored in [12,15]. There, BDD-represented automata are used to provide a decision procedure for WS1S.

This decision procedure is implemented in the MONA system and also used to formalize and reason about hardware, e.g., [1,2].

WS1S formalize the same class of languages as LIFs, namely regular languages on words. However, this logic is more expressive in the sense that there are regular languages whose representation as automata, and hence also LIFs, are nonelementary larger than the corresponding formulae in WS1S [16]. Conversely, however, there is a simple log space translation of any LIF to an equivalent WS1S formulae, which we illustrate with an example.

Consider the LIF system for the family of ripple carry adders formalized in §3. This can be formalized by the following WS1S predicate.

$$
\begin{aligned}
\forall i.\, \Big(& i = 0 \rightarrow \\
& \quad (Sum(0) \leftrightarrow xor(A(0), xor(B(0), cin))) \wedge \\
& \quad (Carry(0) \leftrightarrow (A(0) \wedge B(0)) \vee (xor(A(0), B(0)) \wedge cin))\Big) \wedge \\
& \Big(0 < i \leq n \rightarrow \\
& \quad (Sum(i) \leftrightarrow xor(A(i), xor(B(i), Carry(i-1)))) \wedge \\
& \quad (Carry(i) \leftrightarrow (A(i) \wedge B(i)) \vee (xor(A(i), B(i)) \wedge Carry(i-1)))\Big)\,,
\end{aligned}
$$

where $n \in \mathbb{N}$ is the size of the circuit and the finite subsets $A, B \subseteq \mathbb{N}$ represent the interpretation of the input pins.

This pattern generalizes to other LIFs. We use the $i = 0$ case to formalize the respective bases cases, and the $i > 0$ case for the step cases. We have also used standard syntactic sugar (e.g., for subtraction by a constant) and free second-order variables represent parameterized collections of inputs and outputs. (Capitalized variables are second-order, i is a first-order variable, and cin is a Boolean variable representing the initial carry-in.)

Given a translation of a LIF system into a monadic formula, we can use the MONA tool to convert it into a (BDD-represented) automaton. Alternatively we can use MONA to establish that it has some property, specified in WS1S.

Although an encoding in WS1S has the advantage of using an existing decision procedure, and a richer language for specifying properties, the complexity of the decision procedure can be considerably worse both in theory and in practice. For example, for a 12-bit counter MONA (version 1.3) needs more than an hour to build the automaton. This is an order of magnitude larger than what is needed for the preliminary tests with an implementation of our emptiness test for AWAs.

6 Conclusions

We have shown that LIFs and EIFs can be understood and analyzed using standard formalisms and results from automata theory. Not only is this conceptually attractive, but we also obtain improved results for the decision problem for LIFs and the first complexity results for EIFs. We have carried out an experimental analysis that shows that, for LIFs, our proposed decision procedure is competitive with or, in some cases, faster than alternatives. Future work involves implementing the decision procedure for EIFs and gaining practical experience with the decision procedures.

Acknowledgments: The authors thank Nils Klarlund and Aarti Gupta for helpful discussions.

References

[1] A. Ayari, D. Basin, and S. Friedrich. Structural and behavioral modeling with monadic logics. In *IEEE International Symposium on Multiple-Valued Logic, ISMVL'99*, pages 142–151, Freiburg, Germany, 1999.

[2] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3):255–288, 1998.

[3] J. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.

[4] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[6] F. Gécseg and M. Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[7] F. Gécseg and M. Steinby. Tree languages. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3. Beyond Words, chapter 1, pages 389–455. Springer-Verlag, 1997.

[8] A. Gupta. *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994.

[9] A. Gupta and A. Fisher. Parametric circuit representation using inductive boolean functions. In C. Courcoubetis, editor, *Computer Aided Verification, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 15–28, 1993.

[10] A. Gupta and A. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proc. of the IEEE International Conference on Computer-Aided Design*, pages 192–199, Los Alamitos, CA, 1993.

[11] A. Gupta and A. Fisher. Tradeoffs in canonical sequential function representations. In *Proceedings of the International Conference on Computer Design : VLSI in Computers and Processors*, pages 111–116, 1994.

[12] J. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, TACAS'95*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110, 1995.

[13] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Conmputation.* Addison-Wesley, 1979.

[14] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata. *Information Processing Letters*, 40(1):25–31, 1991.

[15] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, CSL'97*, volume 1414 of *Lecture Notes in Computer Science*, pages 311–326, 1998.

[16] A. Meyer. Weak monadic second-order theory of successor is not elementary-recursive. In R. Parikh, editor, *Logic Colloquium (Boston, Mass., 1972–1973).*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer, 1975.

[17] A. Monti and A. Roncato. Completeness results concerning systolic tree automata and E0L languages. *Information Processing Letters*, 53:11–16, 1995.

[18] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41(2-3):305–318, 1985.

[19] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, 1996.

A Complete Tree Containment Problem for ATAs

In the appendix, we show that CTCP for ATAs is EXPSPACE-hard, which was used to prove Theorem 8.

Set Systems The following definitions are based on [17] and are adapted for exponential (instead of polynomial) space bounded Turing machines.

Let $[n]$ denote the set $\{0, \ldots, n-1\}$, for $n \in \mathbb{N}$. A *state description* μ of size $n > 0$ is a function from $[n]$ to $\mathcal{P}(W)$, where W is a set of states. Let $P \subseteq W^4$

and let $\mu, \mu' : [n] \to \mathcal{P}(W)$ be two state descriptions. We write $\langle x, y, z\rangle \longrightarrow_P w$ for a tuple (x, y, z, w) in P. μ' is the *successor* of μ, written as $\mu \Longrightarrow_P \mu'$, if

$$\mu'(i) = \{w \mid \langle x, y, z\rangle \longrightarrow_P w \text{ with } x \in \mu(i \ominus 1), y \in \mu(i) \text{ and } z \in \mu(i \oplus 1)\},$$

where $i \in [n]$, $\ominus$ denotes subtraction modulo n, and $\oplus$ denotes addition modulo n. Intuitively, an element $w \in W$ is in the successor μ' at the position $i \in [n]$ if there is a rule $\langle x, y, z\rangle \longrightarrow_P w$ that can be fired, i.e. $x \in \mu(i \ominus 1)$, $y \in \mu(i)$ and $z \in \mu(i \oplus 1)$.

A *set system* $\mathcal{R}$ of size $n \in \mathbb{N}$ is a tuple (p, W, V, P, w_0, μ_0), where (i) p is a polynomial whose coefficients are natural numbers, (ii) W is a finite set of states, (iii) $V \subseteq W$ is the set of final states, (iv) $P \subseteq W^4$ is the transition relation, (v) $w_0 \in W$ is the default state, and (vi) μ_0 is the initial state description of size $2^{p(n)}$ with $\mu_0(i) = \{w_0\}$, for $n \leq i < 2^{p(n)}$. The *size* of $\mathcal{R}$ is $n + |W|$.

Intuitively, a set system $\mathcal{R}$ can be seen as a device with $2^{p(n)}$ registers, where each register contains a set of states. The initial content of the registers is given by μ_0, where the content of the ith register is $\{w_0\}$, if $n \leq i < 2^{p(n)}$, and each register $i \in [n]$ initially contains an arbitrary subset of W. The content of a register changes accordingly to rules in P, where in each step the contents of the registers change simultaneously. We are interested whether some register eventually contains a final state.

A state w is *reachable* if there are state descriptions $\mu_1, \ldots, \mu_h$ of size $2^{p(n)}$ such that $\mu_1 = \mu_0$ and $\mu_i \Longrightarrow_P \mu_{i+1}$, for $1 \leq i < h$, and there is a $j \in [2^{p(n)}]$ with $w \in \mu_h(j)$. In this case we say that $\mu_1, \ldots, \mu_h$ are a *solution* for w and j. The *reachability problem for set systems (RP)* is to decide if some final state of a set system $\mathcal{R}$ is reachable.

Theorem 9 *RP is EXPSPACE-hard.*

PROOF. We reduce the word problem for exponential space bounded Turing machines [2] to RP. Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ be a $2^{p(n)} - 1$ space bounded

[2] We use the definition of a Turing machine from [13], with a one-way infinite tape. A (deterministic) Turing machine $\mathcal{M}$ is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, \#, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, Γ is the tape alphabet, $\Sigma \subseteq \Gamma \setminus \{\#\}$ is the input alphabet, $\# \in \Gamma$ is the blank symbol, and δ is the transition function, i.e., a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$. We assume that $Q \cap \Gamma = \emptyset$ and that $\delta(q, a)$ is undefined, for $q \in F$ and $a \in \Gamma$.
In the following, we assume that $Q \cap \Gamma = \emptyset$. A configuration C is a word $\alpha_1 q b \alpha_2$, where $q \in Q$, $b \in \Gamma$, and $\alpha_1, \alpha_2 \in \Gamma^*$. A computation on $w \in \Sigma^*$ is a finite sequence of configurations $C_0, \ldots, C_m$, where $C_0 = q_0 w$ and C_{i+1} is the successor configuration (defined as expected) of C_i, for $0 \leq i < m$. $C_0, \ldots, C_m$ is accepting if $q \in F$, for

Turing machine, where p is a polynomial with natural numbers coefficients. Further, let $w = a_1 \dots a_n \in \Sigma^*$ be an input word. We define a set system $\mathcal{R} = (p, W, V, P, w_0, \mu_0)$ of size n such that some $v \in V$ is reachable iff $\mathcal{M}$ accepts w.

Let $W = \{\$\} \cup \{(q, a) \mid q \in Q \uplus \{\square\} \text{ and } a \in \Gamma\}$. A configuration $C = b_1 \dots b_{k-1} q b_k b_{k+1} \dots b_m$ can be represented as the state description $\mu_C : [2^{p(n)}] \to \mathcal{P}(W)$ with

$$\mu_C(i) = \begin{cases} \{\$\} & \text{if } i = 0, \\ \{(q, b_i)\} & \text{if } i = k, \\ \{(\square, \#)\} & \text{if } i > m, \\ \{(\square, b_i)\} & \text{otherwise.} \end{cases}$$

We define the transition relation $P \subseteq W^4$ of $\mathcal{R}$ such that P simulates the transition function of $\mathcal{M}$. Let $x, y, z \in \Gamma$ and $q \in Q$. For $u \in Q \cup \{\square\}$,

$$\langle (u, x), \$, (\square, z) \rangle \longrightarrow_P \ \$ \qquad \text{and} \qquad \langle (\square, x), \$, (u, z) \rangle \longrightarrow_P \ \$ \, .$$

Furthermore, we have the following three transitions in P:

$$\langle \$, (\square, y), (\square, z) \rangle \longrightarrow_P (\square, y) \, , \qquad \langle (\square, x), (\square, y), \$ \rangle \longrightarrow_P (\square, y) \, , \qquad \text{and}$$
$$\langle (\square, x), (\square, y), (\square, z) \rangle \longrightarrow_P (\square, y) \, .$$

Finally, for $\delta(q, b) = (p, c, X)$, where $X = R$ or $X = L$, we have the transitions $\langle (\square, x), (q, b), (\square, y) \rangle \longrightarrow_P (\square, c)$ and the transitions:

$X = R$	$X = L$
$\langle \$, (q, b), (\square, z) \rangle \longrightarrow_P (\square, c)$	$\langle \$, (q, b), (\square, z) \rangle \longrightarrow_P (q, b)$
$\langle (\square, x), (q, b), \$ \rangle \longrightarrow_P (q, b)$	$\langle (\square, x), (q, b), \$ \rangle \longrightarrow_P (\square, c)$
$\langle \$, (\square, y), (q, b) \rangle \longrightarrow_P (\square, y)$	$\langle \$, (\square, y), (q, b) \rangle \longrightarrow_P (p, y)$
$\langle (q, x), (\square, y), \$ \rangle \longrightarrow_P (p, y)$	$\langle (q, b), (\square, y), \$ \rangle \longrightarrow_P (\square, y)$
$\langle (\square, x), (\square, y), (q, b) \rangle \longrightarrow_P (\square, y)$	$\langle (\square, x), (\square, y), (q, b) \rangle \longrightarrow_P (p, y)$
$\langle (q, b), (\square, y), (\square, z) \rangle \longrightarrow_P (p, y)$	$\langle (q, b), (\square, y), (\square, z) \rangle \longrightarrow_P (\square, y)$

It is straightforward to show that C' is a successor configuration of C of $\mathcal{M}$ iff $\mu_C \Longrightarrow_P \mu_{C'}$. Hence, defining the initial state description μ_0 as $\mu_{C_0(w)}$ and the set of final states V as $\{(q, a) \mid q \in F \text{ and } a \in \Gamma\}$, if follows that $\mathcal{M}$ accepts w iff some final state is reachable. □

$\overline{C_m = \alpha_1 q b \alpha_2}$.

Alternating Tree Automata over Signatures To simplify our subsequent proofs we endow alternating tree automata with a signature instead of an alphabet. A *signature* Γ is a pair (Σ, σ), where Σ is a nonempty finite alphabet and $\sigma : \Sigma \to \mathbb{N}$ assigns each letter its arity. Σ_n denotes the set of n-ary letters. We assume that there is some $n \geq 2$, where $\Sigma_n \neq \emptyset$.

A Σ-labeled tree t *respects* σ if each node $u \in \mathrm{dom}(t)$ has exactly $\sigma(t(u))$ successors. T_Γ denotes the set of all Σ-labeled trees that respect σ. We represent trees in T_Γ as terms in the standard way. Moreover, we assume that any Σ-labeled tree respects σ.

An *alternating tree automaton over* Γ (Γ-ATA) is a tuple (Q, q_0, δ), where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \to B_+(Q \times \mathbb{N})$. The transition function δ satisfies the condition $\delta(q, a) \in B_+(Q \times [n])$, for $q \in Q$ and $a \in \Sigma_n$. We write q^c, for $(q, c) \in Q \times \mathbb{N}$.

A *run* π of $\mathcal{A}$ on a Σ-labeled tree t is a $Q \times \mathrm{dom}(t)$-labeled tree with $\pi(\lambda) = (q_0, \lambda)$ and $\{q^c \mid w \in \mathrm{dom}(\pi) \text{ and } \pi(w) = (q, uc)\} \models \delta(q, t(u))$, for $u \in \mathrm{dom}(t)$. Note that $\delta(q, t(u)) = 1$, for each node $w \in \mathrm{dom}(\pi)$ with $\pi(w) = (q, u)$ and u a leaf in t. A Σ-labeled tree t is *accepted* by $\mathcal{A}$ if there exists a run on t. $L(\mathcal{A})$ is the set of all Σ-labeled trees accepted by $\mathcal{A}$. As with ATAs, we sometimes use an initial Boolean formula for Γ-ATAs instead of an initial state.

The complete tree containment problem (CTCP) for Γ-ATAs is defined analogously as for ATAs.

Lemma 10 *CTCP for Γ-ATAs can be reduced in polynomial time to CTCP for ATAs.*

PROOF. Let $\mathcal{A} = (Q, q_0, \delta)$ be a Γ-ATA, with $\Gamma = (\Sigma, \sigma)$, and let Ω be the alphabet $\Sigma \uplus \{\#_0, \#_2\}$. We proceed in four steps: (i) We define a tree homomorphism h that translates trees over the signature Γ into binary Ω-labeled trees; (ii) we construct an ATA $\mathcal{B}_0$ that accepts the image of h, i.e. $L(\mathcal{B}_0) = h(T_\Gamma)$; (iii) we transform $\mathcal{B}_0$ into an ATA $\mathcal{B}_1$ such that $L(\mathcal{B}_1) = h(L(\mathcal{A}))$; finally, (iv) we transform $\mathcal{B}_1$ into an ATA $\mathcal{B}_2$ that accepts a complete tree iff $\mathcal{A}$ does.

(i) Homomorphism h**:** Let $d = \left(\max \bigcup_{a \in \Sigma} \sigma(a)\right) - 1$, and let $\overline{U} = \{u \in \{0,1\}^* \mid |u| \leq d\}$, $\mathrm{Pos}(f) = \{1^i 0^{d-i} \mid i < \sigma(f)\}$, for $f \in \Sigma_n$ with $n > 0$.

Furthermore, let t_f be the binary tree, with $\mathrm{dom}(t_f) = \overline{U}$ and

$$t_f(u) = \begin{cases} f & \text{if } u = \lambda, \\ x_i & \text{if } u = 1^i 0^{d-1} \text{ and } u \in \mathrm{Pos}(f), \\ \#_2 & \text{if } 0 < |u| < d, \\ \#_0 & \text{otherwise,} \end{cases}$$

where the x_is are variables. Finally, the homomorphism h is defined as $h(a) = a$, if $a \in \Sigma_0$, and $h(a) = t_a$ otherwise. h is homomorphically extended to trees in T_Γ, i.e. $h : T_\Gamma \to \Omega^{\mathrm{T}*}$.

(ii) Construction of $\mathcal{B}_0$: Let $\mathcal{B}_0 = (\Omega, Q_0, \iota_0, \delta_0, F_0)$, where $Q_0 = \{s_u^f \mid f \in \Sigma,\ u \in \overline{U}\} \cup \{q_{ok}\}$, $\iota_0 = \bigvee_{f \in \Sigma} s_\lambda^f$, and $F_0 = \{q_{ok}\}$. For $x \in \Omega$ and $s_u^f \in Q_0$, the transition function δ_0 is defined by:

$$\delta_0(s_u^f, x) = \begin{cases} \left(s_0^f\right)^0 \wedge \left(s_1^f\right)^1 & \text{if } x = f, \\ \left(s_{u0}^f\right)^0 \wedge \left(s_{u1}^f\right)^1 & \text{if } x = \#_2,\ |u| < d \text{ and } u0, u1 \notin \mathrm{Pos}(f), \\ \bigvee_{g,h \in \Sigma} \left((s_\lambda^g)^0 \wedge (s_\lambda^h)^1\right) & \text{if } x = \#_2 \text{ and } u0, u1 \in \mathrm{Pos}(f), \\ \bigvee_{g \in \Sigma} \left((s_\lambda^g)^0 \wedge (s_{u1}^f)^0\right) & \text{if } x = \#_2,\ u0 \in \mathrm{Pos}(f) \text{ and } u1 \notin \mathrm{Pos}(f), \\ q_{ok}^0 \wedge q_{ok}^1 & \text{if } x = \#_0,\ u \notin \mathrm{Pos}(f) \text{ and } |u| = d, \\ 0 & \text{otherwise.} \end{cases}$$

By structural induction over trees, we can show that $L(\mathcal{B}_0) = h(T_\Gamma)$.

(iii) Construction of $\mathcal{B}_1$: Let $U = \{u \in \{0,1\}^* \mid |u| < d\}$, and let $\mathcal{B}_1 = (\Omega, Q_1, \iota_1, \delta_1, F_0)$, where $Q_1 = Q_0 \uplus \{q_u \mid q \in Q \text{ and } u \in U\}$ and $\iota_1 = q_0 \wedge \iota_0$. For $\theta \in B_+(Q \times [d])$, we denote by $\eta(\theta)$ the Boolean formula obtained from θ by replacing each occurrence of q^i by q_u^b, where $bu = 1^i 0^{d-i}$, for $u \in U$ and $b \in \{0,1\}$. The transition function δ_1 is defined by:

$$\delta_1(p, x) = \begin{cases} q_{ok}^1 \wedge q_{ok}^1 & \text{if } x \in \Sigma_0, p \in Q, \text{ and } \delta(p, x) = 1, \\ \eta(\delta(p, x)) & \text{if } x \in \Sigma_n \text{ and } n > 0, \\ q_u^b & \text{if } p = q_{bu} \text{ and } bu \in U, \\ \delta_0(p, x) & \text{if } p \in Q_0, \\ 0 & \text{otherwise.} \end{cases}$$

By a structural induction over trees, we can prove that if $t \in T_\Gamma$ is accepted from a state $q \in Q$ by $\mathcal{A}$ then the tree $h(t) \in \Omega^{\mathrm{T}*}$ is accepted from q by $\mathcal{B}_1$ and thus $h(L(\mathcal{A})) \subseteq L(\mathcal{B}_1)$. Conversely, if $t' \in \Omega^{\mathrm{T}*}$ is accepted from a state $q \in Q$ by $\mathcal{B}_1$ then there is a tree $t \in T_\Gamma$ accepted from q by $\mathcal{A}$, with $t' = h(t)$. The proof of this is by induction on the number of the states in Q occurring along a run of t' in $\mathcal{B}_1$. Thus $L(\mathcal{B}_1) \subseteq h(L(\mathcal{A}))$ and so $L(\mathcal{B}_1) = h(L(\mathcal{A}))$.

(iv) Construction of $\mathcal{B}_2$: Let $\mathcal{B}_2 = (\Omega, Q_1, \iota_1, \delta_2, F_1)$, where $\delta_2(s_u^f, f) = 1$

if $f \in \Sigma$ and $u \in \overline{U} \setminus \mathrm{Pos}(f)$ and $|u| = d$, and otherwise δ_2 is identical to δ_1. A tree t is accepted by $\mathcal{B}_2$ iff there are nodes $u_1, \ldots, u_n$ such that if we replace the subtrees at the nodes u_i in t with $\#_0$ then we obtain a tree in $L(\mathcal{B}_1)$. Additionally, note that if $t \in L(\mathcal{A})$ is a complete tree then all leaves of $h(t)$, except those labeled with $\#_0$, have the same depth. Thus $L(\mathcal{A})$ contains a complete tree iff $L(\mathcal{B}_2)$ contains a complete tree. □

Complete Tree Containment Problem We now have the ingredients to prove:

Theorem 11 *CTCP for ATAs is EXPSPACE-hard.*

PROOF. Let $\mathcal{R} = (p, W, V, P, w_0, \mu_0)$ be a set system of size n. We make the simplification that $p(n)$ equals n; the proof can be easily generalized for an arbitrary polynomial. Let Γ be the signature with $\Sigma_0 = \{0, 1\}$, $\Sigma_3 = W$, $\Sigma_n = \{\hat{w} \mid w \in W\}$, and $\Sigma_m = \emptyset$, for $m \in \mathbb{N} \setminus \{0, 3, n\}$. Let $T \subseteq T_\Gamma$ be the set of trees t that satisfy (i) $t(\lambda) = w$ or $t(\lambda) = \hat{w}$, for $w \in W$, (ii) all leaves are labeled with 0 or 1, and (iii) all inner nodes are labeled with w or $\hat{w}$, with $w \in W$. Moreover, if $t(u) = w$ then u has three successors; otherwise, if $t(u) = \hat{w}$ then u has n successors, all of which are leaves.

A solution $\mu_1, \ldots, \mu_h$ for $w \in W$ and $j \in [2^n]$ can be represented as a complete tree $t \in T$ of height $h + 2$ with $t(\lambda) = w$ or $t(\lambda) = \hat{w}$, and for $u \in \mathrm{dom}(t)$,

- if $t(u) = w$ then $\langle x, y, z\rangle \longrightarrow_P w$, where either $t(u0) = x$, $t(u1) = y$, and $t(u2) = z$ or $t(u0) = \hat{x}$, $t(u1) = \hat{y}$ and $t(u2) = \hat{z}$;
- if $t(u) = \hat{w}$ then $w \in \mu_0(j \oplus k)$, where $k = \sum_{i=1}^{m} k_i - 1$, for $u = k_1 \ldots k_m$, and the labels of the n successors of u are the binary representation of $j \oplus k$.

It is straightforward to show by induction over h that such a tree t exists.

By Lemma 10 it is sufficient to construct an Γ-ATA $\mathcal{A}$ in time polynomial in the size of $\mathcal{R}$ such that $\mathcal{A}$ accepts a complete tree iff $\mathcal{R}$ has a solution for some $v \in V$. We construct such an $\mathcal{A}$ below using of two auxiliary Γ-ATAs $\mathcal{A}_1$ and $\mathcal{A}_2$.

Let $\mathcal{A}_1 = (\{q_w \mid w \in W\} \uplus \{p, \overline{p}\}, \iota_1, \delta_1)$ be the Γ-ATA with the initial Boolean formula $\iota_1 = \bigvee_{v \in V} q_v$, and the transition function δ_1 defined by:

$$\delta_1(q,a) = \begin{cases} \bigvee_{\langle x,y,z\rangle \longrightarrow_P w} (q_x^0 \wedge q_y^1 \wedge q_z^2) & \text{if } q = q_a, \\ \bigwedge_{k \in [n]} (p^k \vee \overline{p}^k) & \text{if } q = q_w \text{ and } a = \hat{w}, \\ 1 & \text{if } (q = p \text{ and } a = 1) \text{ or} \\ & \qquad (q = \overline{p} \text{ and } a = 0), \\ 0 & \text{otherwise.} \end{cases}$$

From the definition of δ_1 it immediately follows that:

Proposition 12 *$L(\mathcal{A}_1) \subseteq T$. Moreover, $\mathcal{A}_1$ accepts $t \in T$ iff (1) $t(\lambda) = v$ or $t(\lambda) = \hat{v}$, with $v \in V$, and (2) if $t(u) = w$ then $\langle x, y, z\rangle \longrightarrow_P t(u)$, where $t(u0) = x$, $t(u1) = y$, $t(u2) = z$ or $t(u0) = \hat{x}$, $t(u1) = \hat{y}$, $t(u2) = \hat{z}$.*

Let $\mathcal{A}_2 = (\{p, \overline{p}\} \cup Q, \iota_2, \delta_2)$, where $Q = \{q_0, \ldots, q_{n-1}\} \cup \{\overline{q}_0, \ldots, \overline{q}_{n-1}\}$ and $\iota_2 = \bigwedge_{k \in [n]} (q_k \vee \overline{q}_k)$. In the following, we define the transition function δ_2.

For $a \in \Sigma$, let $\delta_2(p,a) = \delta_1(p,a)$ and $\delta_2(\overline{p},a) = \delta_1(\overline{p},a)$. For $b \in \{0,1\}$ and $q \in Q$, let $\delta_2(q,b) = 0$. To define the transition for $\delta_2(q_k, \hat{w})$ and $\delta_2(\overline{q}_k, \hat{w})$ we employ the following auxiliary definition. For $i \in [2^n]$ and $w \in W$, let

$$\gamma(i,w) = \begin{cases} \bigwedge_{k \in [n]} \mathrm{bit}(i,k) & \text{if } w \in \mu_0(i), \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathrm{bit}(i,k) = p^k$ if the kth bit of the binary representation of i is 1 and $\mathrm{bit}(i,k) = \overline{p}^k$ otherwise. We now define

$$\delta_2(q_k, \hat{w}) = p^k \wedge (\beta \vee \bigvee_{i \leq m} \gamma(w,i)) \qquad \text{and} \qquad \delta_2(\overline{q}_k, \hat{w}) = \overline{p}^k \wedge (\beta \vee \bigvee_{i \leq m} \gamma(w,i)),$$

where $m = \min\{i \mid n \leq 2^i\}$ and $\beta = p^{n-1} \vee \cdots \vee p^m$, for $w = w_0$, and $\beta = 0$ otherwise. Note that the subformula $\beta \vee \bigvee_{i \leq m} \gamma(w,i)$ checks if there is an $i \in [2^n]$ with $w \in \mu_0(i)$, since $\mu_0(i) = \{w_0\}$, for all $i > n$.

For $w \in W$ and $q \in Q$, let $\delta_2(q,w) = L(q) \wedge M(q) \wedge R(q)$ with

$$L(q) = \begin{cases} (q_k^0 \wedge \bigvee_{i \in [k]} \overline{q}_i^0) \vee (\overline{q}_k^0 \wedge \bigwedge_{i \in [k]} q_i^0) & \text{if } q = q_k, \\ (q_k^0 \wedge \bigwedge_{i \in [k]} q_i^0) \vee (\overline{q}_k^0 \wedge \bigvee_{i \in [k]} \overline{q}_i^0) & \text{if } q = \overline{q}_k, \end{cases}$$

$$M(q) = q^1,$$

$$R(q) = \begin{cases} (q_k^2 \wedge \bigvee_{i \in [k]} q_i^2) \vee (\overline{q}_k^2 \wedge \bigwedge_{i \in [k]} \overline{q}_i^2) & \text{if } q = q_k, \\ (q_k^2 \wedge \bigwedge_{i \in [k]} \overline{q}_i^2) \vee (\overline{q}_k^2 \wedge \bigvee_{i \in [k]} q_i^2) & \text{if } q = \overline{q}_k. \end{cases}$$

A set $J \subseteq Q$ is *consistent* iff for all $k \in [n]$ it holds that $q_k \in J$ iff $\overline{q}_k \notin J$. Let $\mathrm{num}(J)$ denote the number whose kth bit is 1 in its binary representation iff $q_k \in J$. Note that $\mathrm{num}(J) \in [2^n]$. Now, to complete the proof we require the following proposition, whose proof we give afterwards.

Proposition 13 *$L(\mathcal{A}_2) \subseteq T$. Let π be a run of $\mathcal{A}_2$ on $t \in T$ and let $J(u) = \{q \in Q \mid (q,u) \in \mathrm{dom}(\pi)\}$, where u is an inner node of t. π is accepting iff for every inner node of t, $J(u)$ is consistent and moreover:*

(1) $\mathrm{num}(J(u)) = \mathrm{num}(J(\lambda)) \oplus (\sum_{i=1}^{m} k_i - 1)$, *where $u = k_1 \ldots k_m$, and*
(2) if $t(u) = \hat{w}$, with $w \in W$, then $w \in \mu_0(\mathrm{num}(J(u)))$ and the labels of the n successors of u are the binary representation of $\mathrm{num}(J(u))$.

From the Propositions 12 and 13 it follows that a complete tree is in $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ iff it encodes a solution for some final state of $\mathcal{R}$. It is straightforward to construct the Γ-ATA $\mathcal{A}$ with $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. □

To show Proposition 13, we first establish a preliminary lemma.

Lemma 14 *Let $J, K_0, K_1, K_2 \subseteq Q$ be consistent sets with $K_0 \times \{0\} \models \bigwedge_{q \in J} L(q)$, $K_1 \times \{1\} \models \bigwedge_{q \in J} M(q)$, and $K_2 \times \{2\} \models \bigwedge_{q \in J} R(q)$. Then for $i \in \{0,1,2\}$,*

$$\mathrm{num}(K_i) = (\mathrm{num}(J) + 1) \ominus i\,.$$

PROOF. For $i = 1$, $K_1 = J$ by the definition of $M(q)$. Thus, $\mathrm{num}(K_1) = \mathrm{num}(J)$. In the following, we prove the claim for $i = 0$ (the claim for $i = 2$ can be proved analogously). We make the following observations for $k \in [n]$:

(a) Let $q_k \in J$ and $\overline{q}_k \in K_0$. From the definition of $L(q_k)$, $q_i \in K_0$, for all $i \in [k]$. From the definitions of $L(q_i)$ and $L(\overline{q}_i)$ and because $q_i \in K_0$, $\overline{q}_i \in J$ (otherwise J could not be a consistent set). Thus if $q_k \in J$ and $\overline{q}_k \in K_0$ then $\overline{q}_i \in J$ and $q_i \in K_0$, for all $i \in [k]$.
(b) For $\overline{q}_k \in J$ and $q_k \in K_0$, it must be the case that $\overline{q}_i \in J$ and $q_i \in K_0$. This can be proved analogously to (a).

Let k be the smallest member of $[n]$ such that $q_k \in J$ iff $q_k \in K_0$. By the definitions of $L(q_k)$ and $L(\overline{q}_k)$, for $k \in [n]$, we have that (1) if $q_k \in J$ then there is an $i \in [k]$ with $\overline{q}_i \in K_0$, and (2) if $\overline{q}_k \in J$ then there is an $i \in [k]$ with $q_i \in K_0$. Since k is minimal, $q_{k_0} \in J$ iff $\overline{q}_{k_0} \in K_0$, for some $k_0 \in [k]$.

Let $b_{n-1} \ldots b_0$ be the binary representation of $\mathrm{num}(J)$ and let $b'_{n-1} \ldots b'_0$ be the binary representation of $\mathrm{num}(K_0)$, with $b_i, b'_i \in \{0,1\}$ for $i \in [n]$. Let $k_0 \in [n]$ be maximal with $q_{k_0} \in J$ iff $q_{k_0} \notin K_0$.

Case 1: $q_{k_0} \in J$ *and* $\overline{q}_{k_0} \in K_0$. Then by (a), $\overline{q}_i \in J$ and for all $i \in [k_0]$, $q_i \in K_0$. The binary representation of $\mathrm{num}(J)$ is $b_{n-1} \dots b_{k_0+1} 10 \dots 0$ and $b_{n-1} \dots b_{k_0+1} 00 \dots 0$ is the binary representation of $\mathrm{num}(K_0)$.

Case 2: $\overline{q}_{k_0} \in J$ *and* $q_{k_0} \in K_0$. Then by (b), $\overline{q}_i \in J$ and for all $i \in [k_0]$, $q_i \in K_0$. The binary representation of $\mathrm{num}(J)$ is $b_{n-1} \dots b_{k_0+1} 00 \dots 0$ and $b_{n-1} \dots b_{k_0+1} 11 \dots 1$ is the binary representation of $\mathrm{num}(K_0)$.

It both cases, $\mathrm{num}(K_0) = \mathrm{num}(J) \ominus 1$. □

PROOF of Proposition 13. The claim $L(\mathcal{A}_2) \subseteq T$ follows directly from the definition of the transition function and the right to left direction follows straightforwardly by checking that a run π with the stated conditions is accepting.

For the left to right direction, let π be an accepting run of $\mathcal{A}_2$ on t. We show that $J(u)$ is consistent. First, an induction over the length of u shows that $q_k \in J(u)$ or $\overline{q}_k \in J(u)$, for all $k \in [n]$. By the definition of the initial Boolean formula, either q_k or $\overline{q}_k$ is in $J(\lambda)$. Let $|u| > 0$ and $t(u) \in W$. From the definition of the transition function, if q_k or $\overline{q}_k$ is in $J(u)$ then either q_k or $\overline{q}_k$ is in $J(ui)$, for $0 \leq i \leq 2$. Second, assume that $q_k, \overline{q}_k \in J(u)$ for some $k \in [n]$, i.e., there are nodes w, w' in π with $\pi(w) = (q_k, u)$ and $\pi(w') = (\overline{q}_k, u)$. We obtain a contradiction since the leaf $vk \in \mathrm{dom}(t)$, where v is a suffix of u, must be labeled with 0 and 1 by the definition of $M(q_k)$ and $M(\overline{q}_k)$. Thus $J(u)$ is consistent.

A simple induction over the length of u shows (1). The base case is obvious and the step case follows directly from Lemma 14. (2) follows from the definition of the transition function. □