Abstraction Refinement for Emptiness Checking of Alternating Data Automata

Radu Iosif(✉) and Xiao Xu

CNRS, Verimag, Université de Grenoble Alpes, Grenoble, France
{Radu.Iosif,Xiao.Xu}@univ-grenoble-alpes.fr

Abstract. Alternating automata have been widely used to model and verify systems that handle data from finite domains, such as communication protocols or hardware. The main advantage of the alternating model of computation is that complementation is possible in linear time, thus allowing to concisely encode trace inclusion problems that occur often in verification. In this paper we consider alternating automata over infinite alphabets, whose transition rules are formulae in a combined theory of Booleans and some infinite data domain, that relate past and current values of the data variables. The data theory is not fixed, but rather it is a parameter of the class. We show that union, intersection and complementation are possible in linear time in this model and, though the emptiness problem is undecidable, we provide two efficient semi-algorithms, inspired by two state-of-the-art abstraction refinement model checking methods: lazy predicate abstraction [8] and the IMPACT semi-algorithm [17]. We have implemented both methods and report the results of an experimental comparison.

1 Introduction

The language inclusion problem is recognized as being central to verification of hardware, communication protocols and software systems. A property is a specification of the correct executions of a system, given as a set $\mathcal{P}$ of executions, and the verification problem asks if the set $\mathcal{S}$ of executions of the system under consideration is contained within $\mathcal{P}$. This problem is at the core of widespread verification techniques, such as automata-theoretic model checking [23], where systems are specified as finite-state automata and properties defined using Linear Temporal Logic [21]. However the bottleneck of this and other related verification techniques is the intractability of language inclusion (PSPACE-complete for finite-state automata over finite alphabets).

Alternation [3] was introduced as a generalization of nondeterminism, introducing universal, in addition to existential transitions. For automata over finite alphabets, the language inclusion problem can be encoded as the emptiness problem of an alternating automaton of linear size. Moreover, efficient exploration techniques based on antichains are shown to perform well for alternating automata over finite alphabets [5].

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 93–111, 2018.
https://doi.org/10.1007/978-3-319-89963-3_6

Using finite alphabets for the specification of properties and models is however very restrictive, when dealing with real-life computer systems, mostly because of the following reasons. On one hand, programs handle data from very large domains, that can be assumed to be infinite (64-bit integers, floating point numbers, strings of characters, etc.) and their correctness must be specified in terms of the data values. On the other hand, systems must respond to strict deadlines, which requires temporal specifications as timed languages [1].

Although being convenient specification tools, automata over infinite alphabets lack the decidability properties ensured by finite alphabets. In general, when considering infinite data as part of the input alphabet, language inclusion is undecidable and, even complementation becomes impossible, for instance, for timed automata [1] or finite-memory register automata [13]. One can recover theoretical decidability, by restricting the number of variables (clocks) in timed automata to one [20], or forbidding relations between current and past/future values, as with symbolic automata [24]. In such cases, also the emptiness problem for the alternating versions becomes decidable [4,14].

In this paper, we present a new model of alternating automata over infinite alphabets consisting of pairs (a, ν) where a is an input event from a finite set and ν is a valuation of a finite set $\mathbf{x}$ of variables that range over an infinite domain. We assume that, at all times, the successive values taken by the variables in $\mathbf{x}$ are an observable part of the language, in other words, there are no hidden variables in our model. The transition rules are specified by a set of formulae, in a combined first-order theory of Boolean control states and data, that relate past with present values of the variables. We do not fix the data theory a priori, but rather consider it to be a parameter of the class.

A run over an input word $(a_1, \nu_1) \dots (a_n, \nu_n)$ is a sequence $\phi_0(\mathbf{x}_0) \Rightarrow \phi_1(\mathbf{x}_0, \mathbf{x}_1) \Rightarrow \dots \Rightarrow \phi_n(\mathbf{x}_0, \dots, \mathbf{x}_n)$ of rewritings of the initial formula by substituting Boolean states with time-stamped transition rules. The word is accepted if the final formula $\phi_n(\mathbf{x}_0, \dots, \mathbf{x}_n)$ holds, when all time-stamped variables $\mathbf{x}_1, \dots, \mathbf{x}_n$ are substituted by their values in $\nu_1, \dots, \nu_n$, all non-final states replaced by false and all final states by true.

The Boolean operations of union, intersection and complement can be implemented in linear time in this model, thus matching the complexity of performing these operations in the finite-alphabet case. The price to be paid is that emptiness becomes undecidable, for which reason we provide two efficient semi-algorithms for emptiness, based on lazy predicate abstraction [8] and the Impact method [17]. These algorithms are proven to terminate and return a word from the language of the automaton, if one exists, but termination is not guaranteed when the language is empty.

We have implemented the Boolean operations and emptiness checking semi-algorithms and carried out experiments with examples taken from array logics [2], timed automata [9], communication protocols [25] and hardware verification [22].

Related Work. Data languages and automata have been defined previously, in a classical nondeterministic setting. For instance, Kaminski and Francez [13]

consider languages, over an infinite alphabet of data, recognized by automata with a finite number of registers, that store the input data and compare it using equality. Just as the timed languages recognized by timed automata [1], these languages, called quasi-regular, are not closed under complement, but their emptiness is decidable. The impossibility of complementation here is caused by the use of hidden variables, which we do not allow. Emptiness is however undecidable in our case, mainly because counting (incrementing and comparing to a constant) data values is allowed, in many data theories.

Another related model is that of predicate automata [6], which recognize languages over integer data by labeling the words with conjunctions of uninterpreted predicates. We intend to explore further the connection with our model of alternating data automata, in order to apply our method to the verification of parallel programs.

The model presented in this paper stems from the language inclusion problem considered in [11]. There we provide a semi-algorithm for inclusion of data languages, based on an exponential determinization procedure and an abstraction refinement loop using lazy predicate abstraction [8]. In this work we consider the full model of alternation and rely entirely on the ability of SMT solvers to produce interpolants in the combined theory of Booleans and data. Since determinisation is not needed and complementation is possible in linear time, the bulk of the work is carried out by the solver.

The emptiness check for alternating data automata adapts similar semi-algorithms for nondeterministic infinite-state programs to the alternating model of computation. In particular, we considered the state-of-the-art IMPACT procedure [17] that is shown to outperform lazy predicate abstraction [8] in the nondeterministic case, and generalized it to cope with alternation. More recent approaches for interpolant-based abstraction refinement target Horn systems [10,18], used to encode recursive and concurrent programs [7]. However, the emptiness of alternating word automata cannot be directly encoded using Horn clauses, because all the branches of the computation synchronize on the same input, which cannot be encoded by a finite number of local (equality) constraints. We believe that the lazy annotation techniques for Horn clauses are suited for branching computations, which we intend to consider in a future tree automata setting.

2 Preliminaries

A *signature* $\mathsf{S} = (\mathsf{S}^{\mathrm{s}}, \mathsf{S}^{\mathrm{f}})$ consists of a set S^{s} of *sort symbols* and a set S^{f} of sorted *function symbols*. To simplify the presentation, we assume w.l.o.g. that $\mathsf{S}^{\mathrm{s}} = \{\mathsf{Data}, \mathsf{Bool}\}$[1] and each function symbol $f \in \mathsf{S}^{\mathrm{f}}$ has $\#(f) \geq 0$ arguments of sort Data and return value $\sigma(f) \in \mathsf{S}^{\mathrm{s}}$. If $\#(f) = 0$ then f is a *constant*. We consider constants $\top$ and $\bot$ of sort Bool.

[1] The generalization to more than two sorts is without difficulty, but would unnecessarily clutter the technical presentation.

Let Var be an infinite countable set of *variables*, where each $x \in \mathsf{Var}$ has an associated sort $\sigma(x)$. A *term* t of sort $\sigma(t) = S$ is a variable $x \in \mathsf{Var}$ where $\sigma(x) = S$, or $f(t_1, \ldots, t_{\#(f)})$ where $t_1, \ldots, t_{\#(f)}$ are terms of sort Data and $\sigma(f) = S$. An *atom* is a term of sort Bool or an equality $t \approx s$ between two terms of sort Data. A *formula* is an existentially quantified combination of atoms using disjunction $\vee$, conjunction $\wedge$ and negation $\neg$ and we write $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$.

We denote by $\mathrm{FV}^{\sigma}(\phi)$ the set of free variables of sort σ in ϕ and write $\mathrm{FV}(\phi)$ for $\bigcup_{\sigma \in \mathsf{S}^{\mathsf{s}}} \mathrm{FV}^{\sigma}(\phi)$. For a variable $x \in \mathrm{FV}(\phi)$ and a term t such that $\sigma(t) = \sigma(x)$, let $\phi[t/x]$ be the result of replacing each occurrence of x by t. For indexed sets $\mathbf{t} = \{t_1, \ldots, t_n\}$ and $\mathbf{x} = \{x_1, \ldots, x_n\}$, we write $\phi[\mathbf{t}/\mathbf{x}]$ for the formula obtained by simultaneously replacing x_i with t_i in ϕ, for all $i \in [1, n]$. The size $|\phi|$ is the number of symbols occuring in ϕ.

An *interpretation* $\mathcal{I}$ maps (1) the sort Data into a non-empty set $\mathsf{Data}^{\mathcal{I}}$, (2) the sort Bool into the set $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$, where $\top^{\mathcal{I}} = \mathtt{true}$, $\bot^{\mathcal{I}} = \mathtt{false}$, and (3) each function symbol f into a total function $f^{\mathcal{I}} : (\mathsf{Data}^{\mathcal{I}})^{\#(f)} \rightarrow \sigma(f)^{I}$, or an element of $\sigma(f)^{I}$ when $\#(f) = 0$. Given an interpretation $\mathcal{I}$, a *valuation* ν maps each variable $x \in \mathsf{Var}$ into an element $\nu(x) \in \sigma(x)^{\mathcal{I}}$. For a term t, we denote by $t^{\mathcal{I}}_{\nu}$ the value obtained by replacing each function symbol f by its interpretation $f^{\mathcal{I}}$ and each variable x by its valuation $\nu(x)$. For a formula ϕ, we write $\mathcal{I}, \nu \models \phi$ if the formula obtained by replacing each term t in ϕ by the value $t^{\mathcal{I}}_{\nu}$ is logically equivalent to true.

A formula ϕ is *satisfiable* in the interpretation $\mathcal{I}$ if there exists a valuation ν such that $\mathcal{I}, \nu \models \phi$, and *valid* if $\mathcal{I}, \nu \models \phi$ for all valuations ν. The *theory* $\mathbb{T}(\mathsf{S}, \mathcal{I})$ is the set of valid formulae written in the signature S, with the interpretation $\mathcal{I}$. A *decision procedure* for $\mathbb{T}(\mathsf{S}, \mathcal{I})$ is an algorithm that takes a formula ϕ in the signature S and returns yes iff $\phi \in \mathbb{T}(\mathsf{S}, \mathcal{I})$.

Given formulae φ and ψ, we say that ϕ *entails* ψ, denoted $\phi \models^{\mathcal{I}} \psi$ iff $\mathcal{I}, \nu \models \varphi$ implies $\mathcal{I}, \nu \models \psi$, for each valuation ν, and $\phi \Leftrightarrow^{\mathcal{I}} \psi$ iff $\phi \models^{\mathcal{I}} \psi$ and $\psi \models^{\mathcal{I}} \phi$. We omit mentioning the interpretation $\mathcal{I}$ when it is clear from the context.

3 Alternating Data Automata

In the rest of this section we fix an interpretation $\mathcal{I}$ and a finite alphabet Σ of *input events*. Given a finite set $\mathbf{x} \subset \mathsf{Var}$ of variables of sort Data, let $\mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}}$ be the set of valuations of the variables $\mathbf{x}$ and $\Sigma[\mathbf{x}] = \Sigma \times (\mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}})$ be the set of *data symbols*. A *data word* (word in the sequel) is a finite sequence $(a_1, \nu_1)(a_2, \nu_2) \ldots (a_n, \nu_n)$ of data symbols, where $a_1, \ldots, a_n \in \Sigma$ and $\nu_1, \ldots, \nu_n : \mathbf{x} \rightarrow \mathsf{Data}^{\mathcal{I}}$ are valuations. We denote by ε the empty sequence, by Σ^* the set of finite sequences of input events and by $\Sigma[\mathbf{x}]^*$ the set of data words over $\mathbf{x}$.

This definition generalizes the classical notion of words from a finite alphabet to the possibly infinite alphabet $\Sigma[\mathbf{x}]$. Clearly, when $\mathsf{Data}^{\mathcal{I}}$ is sufficiently large or infinite, we can map the elements of Σ into designated elements of $\mathsf{Data}^{\mathcal{I}}$ and use a special variable to encode the input events. However, keeping Σ explicit

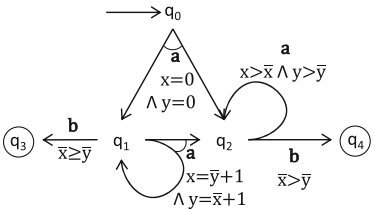
in the following simplifies several technical points below, without cluttering the presentation.

Given sets of variables $\mathbf{b}, \mathbf{x} \subset \mathsf{Var}$ of sort Bool and Data, respectively, we denote by $\mathsf{Form}(\mathbf{b}, \mathbf{x})$ the set of formulae ϕ such that $\mathrm{FV}^{\mathsf{Bool}}(\phi) \subseteq \mathbf{b}$ and $\mathrm{FV}^{\mathsf{Data}}(\phi) \subseteq \mathbf{x}$. By $\mathsf{Form}^{+}(\mathbf{b}, \mathbf{x})$ we denote the set of formulae from $\mathsf{Form}(\mathbf{b}, \mathbf{x})$ in which each Boolean variable occurs under an even number of negations.

An *alternating data automaton* (ADA or automaton in the sequel) is a tuple $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$, where:

- $\mathbf{x} \subset \mathsf{Var}$ is a finite set of variables of sort Data,
- $Q \subset \mathsf{Var}$ is a finite set of variables of sort Bool (*states*),
- $\iota \in \mathsf{Form}^{+}(Q, \emptyset)$ is the *initial configuration*,
- $F \subseteq Q$ is a set of *final states*, and
- $\Delta : Q \times \Sigma \to \mathsf{Form}^{+}(Q, \overline{\mathbf{x}} \cup \mathbf{x})$ is a *transition function*, where $\overline{\mathbf{x}}$ denotes $\{\overline{x} \mid x \in \mathbf{x}\}$.

In each formula $\Delta(q, a)$ describing a transition rule, the variables $\overline{\mathbf{x}}$ track the previous and $\mathbf{x}$ the current values of the variables of $\mathcal{A}$. Observe that the initial values of the variables are left unconstrained, as the initial configuration does not contain free data variables. The size of $\mathcal{A}$ is defined as $|\mathcal{A}| = |\iota| + \sum_{(q,a) \in Q \times \Sigma} |\Delta(q, a)|$.



(a)

$$\Delta(q_0, a) \equiv q_1 \wedge q_2 \wedge x \approx 0 \wedge y \approx 0$$
$$\Delta(q_1, a) \equiv q_1 \wedge q_2 \wedge x \approx \overline{y} + 1 \wedge y \approx \overline{x} + 1$$
$$\Delta(q_1, b) \equiv q_3 \wedge \overline{x} \geq \overline{y}$$
$$\Delta(q_2, a) \equiv q_2 \wedge x > \overline{x} \wedge y > \overline{y}$$
$$\Delta(q_2, b) \equiv q_4 \wedge \overline{x} > \overline{y}$$

(b)

Fig. 1. Alternating data automaton example

Example. Figure 1(a) depicts an ADA with input alphabet $\Sigma = \{a, b\}$, variables $\mathbf{x} = \{x, y\}$, states $Q = \{q_0, q_1, q_2, q_3, q_4\}$, initial configuration q_0, final states $F = \{q_3, q_4\}$ and transitions given in Fig. 1(b), where missing rules, such as $\Delta(q_0, b)$, are assumed to be $\bot$. Rules $\Delta(q_0, a)$ and $\Delta(q_1, a)$ are universal and there are no existential nondeterministic rules. Rules $\Delta(q_1, a)$ and $\Delta(q_2, a)$ compare past $(\overline{x}, \overline{y})$ with present (x, y) values, $\Delta(q_0, a)$ constrains the present and $\Delta(q_1, b)$, $\Delta(q_2, b)$ the past values, respectively. □

Formally, let $\mathbf{x}_k = \{x_k \mid x \in \mathbf{x}\}$, for any $k \geq 0$, be a set of time-stamped variables. For an input event $a \in \Sigma$ and a formula ϕ, we write $\Delta(\phi, a)$ (respectively $\Delta^k(\phi, a)$) for the formula obtained from ϕ by simultaneously replacing each state $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi)$ by the formula $\Delta(q, a)$ (respectively $\Delta(q, a)[\mathbf{x}_k/\overline{\mathbf{x}}, \mathbf{x}_{k+1}/\mathbf{x}]$,

for $k \geq 0$). Given a word $w = (a_1, \nu_1)(a_2, \nu_2) \ldots (a_n, \nu_n)$, the *run* of $\mathcal{A}$ over w is the sequence of formulae:

$$\phi_0(Q) \Rightarrow \phi_1(Q, \mathbf{x}_0 \cup \mathbf{x}_1) \Rightarrow \ldots \Rightarrow \phi_n(Q, \mathbf{x}_0 \cup \ldots \cup \mathbf{x}_n)$$

where $\phi_0 \equiv \iota$ and, for all $k \in [1, n]$, we have $\phi_k \equiv \Delta^k(\phi_{k-1}, a_k)$. Next, we slightly abuse notation and write $\Delta(\iota, a_1, \ldots, a_n)$ for the formula $\phi_n(\mathbf{x}_0, \ldots, \mathbf{x}_n)$ above. We say that $\mathcal{A}$ *accepts* w iff $\mathcal{I}, \nu \models \Delta(\iota, a_1, \ldots, a_n)$, for some valuation ν that maps:(1) each $x \in \mathbf{x}_k$ to $\nu_k(x)$, for all $k \in [1, n]$, (2) each $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi_n) \cap F$ to $\top$ and (3) each $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi_n) \setminus F$ to $\bot$. The language of $\mathcal{A}$ is the set $L(\mathcal{A})$ of words from $\Sigma[\mathbf{x}]^*$ accepted by $\mathcal{A}$.

Example. The following sequence is a non-accepting run of the ADA from Fig. 1 on the word $(a, \langle 0, 0\rangle), (a, \langle 1, 1\rangle), (b, \langle 2, 1\rangle)$, where $\mathsf{Data}^{\mathcal{I}} = \mathbb{Z}$ and the function symbols have standard arithmetic interpretation:

$$q_0 \overset{(a,\langle 0,0\rangle)}{\Longrightarrow} q_1 \wedge q_2 \wedge x_1 \approx 0 \wedge y_1 \approx 0 \overset{(a,\langle 1,1\rangle)}{\Longrightarrow} \underbrace{q_1 \wedge q_2 \wedge x_2 \approx y_1 + 1 \wedge y_2 \approx x_1 + 1}_{q_1} \wedge \underbrace{q_2 \wedge x_2 > x_1 \wedge y_2 > y_1}_{q_2} \wedge x_1 \approx 0 \wedge y_1 \approx 0 \overset{(b,\langle 2,1\rangle)}{\Longrightarrow}$$

$$\underbrace{q_3 \wedge x_2 \geq y_2}_{q_1} \wedge \underbrace{q_4 \wedge x_2 > y_2}_{q_2} \wedge x_2 \approx y_1 + 1 \wedge\ y_2 \approx x_1 + 1 \wedge \underbrace{q_4 \wedge x_2 > y_2}_{q_2} \wedge x_2 > x_1 \wedge y_2 > y_1 \wedge x_1 \approx 0 \wedge y_1 \approx 0$$

□

In this paper we tackle the following problems:

1. *Boolean closure*: given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, both with the same set of variables $\mathbf{x}$, do there exist automata $\mathcal{A}_\cup$, $\mathcal{A}_\cap$ and $\overline{\mathcal{A}_1}$ such that $L(\mathcal{A}_\cup) = \mathcal{A}_1 \cup \mathcal{A}_2$, $L(A_\cap) = \mathcal{A}_1 \cap \mathcal{A}_2$ and $L(\overline{\mathcal{A}_1}) = \Sigma[\mathbf{x}]^* \setminus L(\mathcal{A}_1)$?
2. *emptiness*: given an automaton $\mathcal{A}$, is $L(\mathcal{A}) = \emptyset$?

It is well known that other problems, such as *universality* (given automaton $\mathcal{A}$ with variables $\mathbf{x}$, does $L(\mathcal{A}) = \Sigma[\mathbf{x}]^*$?) and *inclusion* (given automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same set of variables, does $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?) can be reduced to the above problems. Observe furthermore that we do not consider cases in which the sets of variables in the two automata differ. An interesting problem in this case would be: given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, with variables $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively, such that $\mathbf{x}_1 \subseteq \mathbf{x}_2$, does $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)\downarrow_{\mathbf{x}_1}$, where $L(\mathcal{A}_2)\downarrow_{\mathbf{x}_1}$ is the projection of the set of words $L(\mathcal{A}_2)$ onto the variables $\mathbf{x}_1$? This problem is considered as future work.

3.1 Boolean Closure

Given a set Q of Boolean variables and a set $\mathbf{x}$ of variables of sort Data, for a formula $\phi \in \mathsf{Form}^+(Q, \mathbf{x})$, with no negated occurrences of the Boolean variables, we define the formula $\overline{\phi} \in \mathsf{Form}^+(Q, \mathbf{x})$ recursively on the structure of ϕ:

$$\begin{array}{rlrl} \overline{\phi_1 \vee \phi_2} & \equiv \overline{\phi_1} \wedge \overline{\phi_2} & \overline{\phi_1 \wedge \phi_2} & \equiv \overline{\phi_1} \vee \overline{\phi_2} \\ \overline{\neg\phi} & \equiv \neg\overline{\phi} \text{ if } \phi \text{ not atom} & \overline{\phi} & \equiv \phi \text{ if } \phi \in Q \\ \overline{\phi} & \equiv \neg\phi \text{ if } \phi \notin Q \text{ atom} & & \end{array}$$

We have $|\overline{\phi}| = |\phi|$, for every formula $\phi \in \mathsf{Form}^+(Q, \mathbf{x})$.

In the following let $\mathcal{A}_i = \langle \mathbf{x}, Q_i, \iota_i, F_i, \Delta_i \rangle$, for $i = 1, 2$, where w.l.o.g. we assume that $Q_1 \cap Q_2 = \emptyset$. We define:

$$\mathcal{A}_\cup = \langle \mathbf{x}, Q_1 \cup Q_2, \iota_1 \vee \iota_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle$$
$$\mathcal{A}_\cap = \langle \mathbf{x}, Q_1 \cup Q_2, \iota_1 \wedge \iota_2, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \rangle$$
$$\overline{\mathcal{A}_1} = \langle \mathbf{x}, Q_1, \overline{\iota_1}, Q_1 \setminus F_1, \overline{\Delta_1} \rangle$$

where $\overline{\Delta_1}(q, a) \equiv \overline{\Delta_1(q, a)}$, for all $q \in Q_1$ and $a \in \Sigma$. The following lemma shows the correctness of the above definitions:

Lemma 1. *Given automata $\mathcal{A}_i = \langle \mathbf{x}, Q_i, \iota_i, F_i, \Delta_i \rangle$, for $i = 1, 2$, such that $Q_1 \cap Q_2 = \emptyset$, we have $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, $L(\mathcal{A}_\cap) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ and $L(\overline{\mathcal{A}_1}) = \Sigma[\mathbf{x}]^* \setminus L(\mathcal{A}_1)$.*

It is easy to see that $|\mathcal{A}_\cup| = |\mathcal{A}_\cap| = |\mathcal{A}_1| + |\mathcal{A}_2|$ and $|\overline{\mathcal{A}}| = |\mathcal{A}|$, thus the automata for the Boolean operations, including complementation, can be built in linear time. This matches the linear-time bounds for intersection and complementation of alternating automata over finite alphabets [3].

4 Antichains and Interpolants for Emptiness

The emptiness problem for ADA is undecidable, even in very simple cases. For instance, if $\mathsf{Data}^{\mathcal{I}}$ is the set of positive integers, an ADA can simulate an Alternating Vector Addition System with States (AVASS) using only atoms $x \geq k$ and $x = \overline{x} + k$, for $k \in \mathbb{Z}$, with the classical interpretation of the function symbols on integers. Since reachability of a control state is undecidable for AVASS [15], ADA emptiness is undecidable.

Consequently, we give up on the guarantee for termination and build semi-algorithms that meet the requirements below:

(i) given an automaton $\mathcal{A}$, if $L(\mathcal{A}) \neq \emptyset$, the procedure will terminate and return a word $w \in L(\mathcal{A})$, and
(ii) if the procedure terminates without returning such a word, then $L(\mathcal{A}) = \emptyset$.

Let us fix an automaton $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$ whose (finite) input event alphabet is Σ, for the rest of this section. Given a formula $\phi \in \mathsf{Form}^+(Q, \mathbf{x})$ and an input event $a \in \Sigma$, we define the *post-image* function $\mathsf{Post}_{\mathcal{A}}(\phi, a) \equiv \exists \overline{\mathbf{x}}. \Delta(\phi[\overline{\mathbf{x}}/\mathbf{x}], a) \in \mathsf{Form}^+(Q, \mathbf{x})$, mapping each formula in $\mathsf{Form}^+(Q, \mathbf{x})$ to a formula defining the effect of reading the event a. We generalize the post-image function to finite sequences of input events, as follows:

$$\mathsf{Post}_{\mathcal{A}}(\phi, \varepsilon) \equiv \phi \qquad \mathsf{Post}_{\mathcal{A}}(\phi, ua) \equiv \mathsf{Post}_{\mathcal{A}}(\mathsf{Post}_{\mathcal{A}}(\phi, u), a)$$
$$\mathsf{Acc}_{\mathcal{A}}(u) \equiv \mathsf{Post}_{\mathcal{A}}(\iota, u) \wedge \bigwedge_{q \in Q \setminus F} (q \to \bot), \text{ for any } u \in \Sigma^*$$

Then the emptiness problem for $\mathcal{A}$ becomes: does there exist $u \in \Sigma^*$ such that the formula $\mathsf{Acc}_{\mathcal{A}}(u)$ is satisfiable? Observe that, since we ask a satisfiability

query, the final states of $\mathcal{A}$ need not be constrained[2]. A naïve semi-algorithm enumerates all finite sequences and checks the satisfiability of $\mathsf{Acc}_{\mathcal{A}}(u)$ for each $u \in \Sigma^*$, using a decision procedure for the theory $\mathbb{T}(\mathsf{S}, \mathcal{I})$.

Since no Boolean variable from Q occurs under negation in ϕ, it is easy to prove the following monotonicity property: given two formulae $\phi, \psi \in \mathsf{Form}^+(Q, \mathbf{x})$ if $\phi \models \psi$ then $\mathsf{Post}_{\mathcal{A}}(\phi, u) \models \mathsf{Post}_{\mathcal{A}}(\psi, u)$, for any $u \in \Sigma^*$. This suggest an improvement of the above semi-algorithm, that enumerates and stores only a set $U \subseteq \Sigma^*$ for which $\{\mathsf{Post}_{\mathcal{A}}(\phi, u) \mid u \in U\}$ forms an *antichain*[3] w.r.t. the entailment partial order. This is because, for any $u, v \in \Sigma^*$, if $\mathsf{Post}_{\mathcal{A}}(\iota, u) \models \mathsf{Post}_{\mathcal{A}}(\iota, v)$ and $\mathsf{Acc}_{\mathcal{A}}(uw)$ is satisfiable for some $w \in \Sigma^*$, then $\mathsf{Post}_{\mathcal{A}}(\iota, uw) \models \mathsf{Post}_{\mathcal{A}}(\iota, vw)$, thus $\mathsf{Acc}_{\mathcal{A}}(vw)$ is satisfiable as well, and there is no need for u, since the non-emptiness of $\mathcal{A}$ can be proved using v alone. However, even with this optimization, the enumeration of sequences from Σ^* diverges in many real cases, because infinite antichains exist in many interpretations, e.g. $q \wedge x \approx 0,\ q \wedge x \approx 1, \ldots$ for $\mathsf{Data}^{\mathcal{I}} = \mathbb{N}$.

A *safety invariant* for $\mathcal{A}$ is a function $\mathsf{I} : (Q \mapsto \mathbb{B}) \to 2^{\mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}}}$ such that, for every Boolean valuation $\beta : Q \to \mathbb{B}$, every valuation $\nu : \mathbf{x} \mapsto \mathsf{Data}^{\mathcal{I}}$ of the data variables and every finite sequence $u \in \Sigma^*$ of input events, the following hold:

1. $\mathcal{I}, \beta \cup \nu \models \mathsf{Post}_{\mathcal{A}}(\iota, u) \Rightarrow \nu \in \mathsf{I}(\beta)$, and
2. $\nu \in \mathsf{I}(\beta) \Rightarrow \mathcal{I}, \beta \cup \nu \not\models \mathsf{Acc}_{\mathcal{A}}(u)$.

If I satisfies only the first point above, we call it an *invariant*. Intuitively, a safety invariant maps every Boolean valuation into a set of data valuations, that contains the initial configuration $\iota \equiv \mathsf{Post}_{\mathcal{A}}(\iota, \varepsilon)$, whose data variables are unconstrained, over-approximates the set of reachable valuations (point 1) and excludes the valuations satisfying the acceptance condition (point 2). A formula $\phi(Q, \mathbf{x})$ is said to *define* I iff for all $\beta : Q \to \mathbb{B}$ and $\nu : \mathbf{x} \to \mathsf{Data}^{\mathcal{I}}$, we have $\mathcal{I}, \beta \cup \nu \models \phi$ iff $\nu \in \mathsf{I}(\beta)$.

Lemma 2. *For any automaton $\mathcal{A}$, we have $L(\mathcal{A}) = \emptyset$ if and only if $\mathcal{A}$ has a safety invariant.*

Turning back to the issue of divergence of language emptiness semi-algorithms in the case $L(\mathcal{A}) = \emptyset$, we can observe that an enumeration of input sequences $u_1, u_2, \ldots \in \Sigma^*$ can stop at step k as soon as $\bigvee_{i=1}^{k} \mathsf{Post}_{\mathcal{A}}(\iota, u_i)$ defines a safety invariant for $\mathcal{A}$. Although this condition can be effectively checked using a decision procedure for the theory $\mathbb{T}(\mathsf{S}, \mathcal{I})$, there is no guarantee that this check will ever succeed.

The solution we adopt in the sequel is abstraction to ensure the termination of invariant computations. However, it is worth pointing out from the start that abstraction alone will only allow us to build invariants that are not necessarily

[2] Since each state occurs positively in $\mathsf{Acc}_{\mathcal{A}}(u)$, this formula has a model iff it has a model with every $q \in F$ set to true.

[3] Given a partial order $(D, \preceq)$ an antichain is a set $A \subseteq D$ such that $a \not\preceq b$ for any $a, b \in A$.

safety invariants. To meet the latter condition, we resort to counterexample guided abstraction refinement (CEGAR).

Formally, we fix a set of formulae $\Pi \subseteq \mathsf{Form}(Q, \mathbf{x})$, such that $\bot \in \Pi$ and refer to these formulae as *predicates*. Given a formula ϕ, we denote by $\phi^\sharp \equiv \bigwedge \{\pi \in \Pi \mid \phi \models \pi\}$ the abstraction of ϕ w.r.t. the predicates in Π. The abstract versions of the post-image and acceptance condition are defined as follows:

$$\mathsf{Post}^\sharp_{\mathcal{A}}(\phi, \varepsilon) \equiv \phi \;\; \mathsf{Post}^\sharp_{\mathcal{A}}(\phi, ua) \equiv \left(\mathsf{Post}_{\mathcal{A}}(\mathsf{Post}^\sharp_{\mathcal{A}}(\phi, u), a)\right)^\sharp$$
$$\mathsf{Acc}^\sharp_{\mathcal{A}}(u) \equiv \mathsf{Post}^\sharp_{\mathcal{A}}(\iota, u) \wedge \textstyle\bigwedge_{q \in Q \setminus F}(q \rightarrow \bot), \text{ for any } u \in \Sigma^*$$

Lemma 3. *For any bijection $\mu : \mathbb{N} \rightarrow \Sigma^*$, there exists $k > 0$ such that $\bigvee_{i=0}^{k} \mathsf{Post}^\sharp_{\mathcal{A}}(\iota, \mu(i))$ defines an invariant $\mathsf{I}^\sharp$ for $\mathcal{A}$.*

We are left with fulfilling point (2) from the definition of a safety invariant. To this end, suppose that, for a given set Π of predicates, the invariant $\mathsf{I}^\sharp$, defined by the previous lemma, meets point (1) but not point (2), where $\mathsf{Post}_{\mathcal{A}}$ and $\mathsf{Acc}_{\mathcal{A}}$ replace $\mathsf{Post}^\sharp_{\mathcal{A}}$ and $\mathsf{Acc}^\sharp_{\mathcal{A}}$, respectively. In other words, there exists a finite sequence $u \in \Sigma^*$ such that $\nu \in \mathsf{I}^\sharp(\beta)$ and $\mathcal{I}, \beta \cup \nu \models \mathsf{Acc}^\sharp_{\mathcal{A}}(u)$, for some Boolean $\beta : Q \rightarrow \mathbb{B}$ and data $\nu : \mathbf{x} \rightarrow \mathsf{Data}^{\mathcal{I}}$ valuations. Such a $u \in \Sigma^*$ is called a *counterexample*.

Once a counterexample u is discovered, there are two possibilities. Either (i) $\mathsf{Acc}_{\mathcal{A}}(u)$ is satisfiable, in which case u is *feasible* and $L(\mathcal{A}) \neq \emptyset$, or (ii) $\mathsf{Acc}_{\mathcal{A}}(u)$ is unsatisfiable, in which case u is *spurious*. In the first case, our semi-algorithm stops and returns a witness for non-emptiness, obtained from the satisfying valuation of $\mathsf{Acc}_{\mathcal{A}}(u)$ and in the second case, we must strenghten the invariant by excluding from $\mathsf{I}^\sharp$ all pairs (β, ν) such that $\mathcal{I}, \beta \cup \nu \models \mathsf{Acc}^\sharp_{\mathcal{A}}(u)$. This strengthening is carried out by adding to Π several predicates that are sufficient to exclude the spurious counterexample.

Given an unsatisfiable conjunction of formulae $\psi_1 \wedge \ldots \wedge \psi_n$, an *interpolant* is a tuple of formulae $\langle I_1, \ldots, I_{n-1}, I_n \rangle$ such that $I_n \equiv \bot$, $I_i \wedge \psi_i \models_{\mathcal{T}} I_{i+1}$ and I_i contains only variables and function symbols that are common to ψ_i and ψ_{i+1}, for all $i \in [n-1]$. Moreover, by Lyndon's Interpolation Theorem [16], we can assume without loss of generality that every Boolean variable with at least one positive (negative) occurrence in I_i has at least one positive (negative) occurrence in both ψ_i and ψ_{i+1}. In the following, we shall assume the existence of an interpolating decision procedure for $\mathbb{T}(\mathsf{S}, \mathcal{I})$ that meets the requirements of Lyndon's Interpolation Theorem.

A classical method for abstraction refinement is to add the elements of the interpolant obtained from a proof of spuriousness to the set of predicates. This guarantees progress, meaning that the particular spurious counterexample, from which the interpolant was generated, will never be revisited in the future. Though not always, in many practical test cases this progress property eventually yields a safety invariant.

Given a non-empty spurious counterexample $u = a_1 \dots a_n$, where $n > 0$, we consider the following interpolation problem:

$$\Theta(u) \equiv \theta_0(Q_0) \wedge \theta_1(Q_0 \cup Q_1, \mathbf{x}_0 \cup \mathbf{x}_1) \wedge \dots \\ \wedge\ \theta_n(Q_{n-1} \cup Q_n, \mathbf{x}_{n-1} \cup \mathbf{x}_n) \wedge \theta_{n+1}(Q_n) \tag{1}$$

where $Q_k = \{q_k \mid q \in Q\}$, $k \in [0, n]$ are time-stamped sets of Boolean variables corresponding to the set Q of states of $\mathcal{A}$. The first conjunct $\theta_0(Q_0) \equiv \iota[Q_0/Q]$ is the initial configuration of $\mathcal{A}$, with every $q \in \mathrm{FV}^{\mathsf{Bool}}(\iota)$ replaced by q_0. The definition of θ_k, for all $k \in [1, n]$, uses *replacement sets* $R_\ell \subseteq Q_\ell$, $\ell \in [0, n]$, which are defined inductively below:

- $R_0 = \mathrm{FV}^{\mathsf{Bool}}(\theta_0)$,
- $\theta_\ell \equiv \bigwedge_{q_{\ell-1} \in R_{\ell-1}} (q_{\ell-1} \rightarrow \Delta(q, a_\ell)[Q_\ell/Q, \mathbf{x}_{\ell-1}/\overline{\mathbf{x}}, \mathbf{x}_\ell/\mathbf{x}])$ and $R_\ell = \mathrm{FV}^{\mathsf{Bool}}(\theta_\ell) \cap Q_\ell$, for each $\ell \in [1, n]$.
- $\theta_{n+1}(Q_n) \equiv \bigwedge_{q \in Q \setminus F} (q_n \rightarrow \bot)$.

The intuition is that $R_0, \dots, R_n$ are the sets of states replaced, $\theta_0, \dots, \theta_n$ are the sets of transition rules fired on the run of $\mathcal{A}$ over u and θ_{n+1} is the acceptance condition, which forces the last remaining non-final states to be false. We recall that a run of $\mathcal{A}$ over u is a sequence:

$$\phi_0(Q) \Rightarrow \phi_1(Q, \mathbf{x}_0 \cup \mathbf{x}_1) \Rightarrow \dots \Rightarrow \phi_n(Q, \mathbf{x}_0 \cup \dots \cup \mathbf{x}_n)$$

where ϕ_0 is the initial configuration ι and for each $k > 0$, ϕ_k is obtained from ϕ_{k-1} by replacing each state $q \in \mathrm{FV}^{\mathsf{Bool}}(\phi_{k-1})$ by the formula $\Delta(q, a_k)[\mathbf{x}_{k-1}/\overline{\mathbf{x}}, \mathbf{x}_k/\mathbf{x}]$, given by the transition function of $\mathcal{A}$. Observe that, because the states are replaced with transition formulae when moving one step in a run, these formulae lose track of the control history and are not suitable for producing interpolants that relate states and data.

The main idea behind the above definition of the interpolation problem is that we would like to obtain an interpolant $\langle \top, I_0(Q), I_1(Q, \mathbf{x}), \dots, I_n(Q, \mathbf{x}), \bot \rangle$ whose formulae *combine states with the data constraints that must hold locally*, whenever the control reaches a certain Boolean configuration. This association of states with data valuations is tantamount to defining efficient semi-algorithms, based on lazy abstraction [8]. Furthermore, the abstraction defined by the interpolants generated in this way can also *over-approximate the control structure* of an automaton, in addition to the sets of data values encountered throughout its runs.

The correctness of this interpolation-based abstraction refinement setup is captured by the progress property below, which guarantees that adding the formulae of an interpolant for $\Theta(u)$ to the set Π of predicates suffices to exclude the spurious counterexample u from future searches.

Lemma 4. *For any sequence $u = a_1 \dots a_n \in \Sigma^*$, if $\mathsf{Acc}_{\mathcal{A}}(u)$ is unsatisfiable, the following hold:*

1. *$\Theta(u)$ is unsatisfiable, and*
2. *if $\langle \top, I_0, \dots, I_n, \bot \rangle$ is an interpolant for $\Theta(u)$ such that $\{I_i \mid i \in [0, n]\} \subseteq \Pi$ then $\mathsf{Acc}^\sharp_{\mathcal{A}}(u)$ is unsatisfiable.*

5 Lazy Predicate Abstraction for ADA Emptiness

We have now all the ingredients to describe the first emptiness checking semi-algorithm for alternating data automata. Algorithm[4] 1 builds an *abstract reachability tree* (ART) whose nodes are labeled with formulae over-approximating the concrete sets of configurations, and a covering relation between nodes in order to ensure that the set of formulae labeling the nodes in the ART forms an antichain. Any spurious counterexample is eliminated by computing an interpolant and adding its formulae to the set of predicates (cf. Lemma 4). Formally, an ART is tuple $\mathcal{T} = \langle N, E, \mathtt{r}, \Lambda, R, T, \lhd \rangle$, where:

- N is a set of nodes,
- $E \subseteq N \times \Sigma \times N$ is a set of edges,
- $\mathtt{r} \in N$ is the root of the directed tree (N, E),
- $\Lambda : N \to \mathsf{Form}(Q, \mathbf{x})$ is a labeling of the nodes with formulae, such that $\Lambda(\mathtt{r}) = \iota$,
- $R : N \to 2^Q$ is a labeling of nodes with replacement sets, such that $R(\mathtt{r}) = \mathrm{FV}^{\mathsf{Bool}}(\iota)$,
- $T : E \to \bigcup_{i=0}^{\infty} \mathsf{Form}^+(Q_i, \mathbf{x}_i, Q_{i+1}, \mathbf{x}_{i+1})$ is a labeling of edges with time-stamped formulae, and
- $\lhd \subseteq N \times N$ is a set of *covering edges*.

Each node $n \in N$ corresponds to a unique path from the root to n, labeled by a sequence $\lambda(n) \in \Sigma^*$ of input events. The *least infeasible suffix* of $\lambda(n)$ is the smallest sequence $v = a_1 \ldots a_k$, such that $\lambda(n) = wv$, for some $w \in \Sigma^*$ and the following formula is unsatisfiable:

$$\Psi(v) \equiv \Lambda(p)[Q_0/Q] \wedge \theta_1(Q_0 \cup Q_1, \mathbf{x}_0 \cup \mathbf{x}_1) \wedge \ldots \wedge \theta_{k+1}(Q_k) \tag{2}$$

where $\theta_1, \ldots, \theta_{k+1}$ are defined as in (1) and $\theta_0 \equiv \Lambda(p)[Q_0/Q]$. The *pivot* of n is the node p corresponding to the start of the least infeasible suffix. We assume the existence of two functions $\textsc{FindPivot}(u, \mathcal{T})$ and $\textsc{LeastInfeasibleSuffix}(u, \mathcal{T})$ that return the pivot and least infeasible suffix of a sequence $u \in \Sigma^*$ in an ART $\mathcal{T}$, without detailing their implementation.

With these considerations, Algorithm 1 uses a worklist iteration to build an ART. We keep newly expanded nodes of $\mathcal{T}$ in a queue `WorkList`, thus implementing a breadth-first exploration strategy, which guarantees that the shortest counterexamples are explored first. When the search encounters a counterexample candidate u, it is checked for spuriousness. If the counterexample is feasible, the procedure returns a data word $w \in L(\mathcal{A})$, which interleaves the input events of u with the data valuations from the model of $\mathsf{Acc}_{\mathcal{A}}(u)$ (since u is feasible, clearly $\mathsf{Acc}_{\mathcal{A}}(u)$ is satisfiable). Otherwise, u is spurious and we compute its pivot p (line 12), add the interpolants for the least unfeasible suffix of u to Π, remove and recompute the subtree of $\mathcal{T}$ rooted at p.

Termination of Algorithm 1 depends on the ability of a given interpolating decision procedure for the combined Boolean and data theory $\mathbb{T}(\mathsf{S}, \mathcal{I})$ to provide

[4] Though termination is not guaranteed, we call it algorithm for conciseness.

Algorithm 1. Lazy Predicate Abstraction for ADA Emptiness

input: an ADA $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$ over the alphabet Σ of input events
output: `true` if $L(\mathcal{A}) = \emptyset$ and a data word $w \in L(\mathcal{A})$ otherwise

let $\mathcal{T} = \langle N, E, \mathtt{r}, \Lambda, \lhd \rangle$ be an ART
initially $N = E = \lhd = \emptyset$, $\Lambda = \{(\mathtt{r}, \iota)\}$, $\Pi = \{\bot\}$, `WorkList` $= \langle \mathtt{r} \rangle$,
while `WorkList` $\neq \emptyset$ **do**
  dequeue n from `WorkList`
  $N \leftarrow N \cup \{n\}$
  let $\lambda(n) = a_1 \dots a_k$ be the label of the path from `r` to n
  if $\mathsf{Post}^\sharp_{\mathcal{A}}(\lambda(n))$ is satisfiable **then** ▷ counterexample candidate
    if $\mathsf{Acc}_{\mathcal{A}}(u)$ is satisfiable **then** ▷ feasible counterexample
      get model $(\beta, \nu_1, \dots, \nu_k)$ of $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$
      return $w = (a_1, \nu_1) \dots (a_k, \nu_k)$ ▷ $w \in L(\mathcal{A})$ by construction
    else ▷ spurious counterexample
      $p \leftarrow \textsc{FindPivot}(\lambda(n), \mathcal{T})$
      $v \leftarrow \textsc{LeastInfeasibleSuffix}(\lambda(n), \mathcal{T})$
      $\Pi \leftarrow \Pi \cup \{I_0, \dots, I_\ell\}$, where $\langle \top, I_0, \dots, I_\ell, \bot \rangle$ is an interpolant for $\Psi(v)$
      let $\mathcal{S} = \langle N', E', p, \Lambda', \lhd' \rangle$ be the subtree of $\mathcal{T}$ rooted at p
      for $(m, q) \in \lhd$ such that $q \in N'$ **do**
        remove m from N and enqueue m into `WorkList`
        remove $\mathcal{S}$ from $\mathcal{T}$
        enqueue p into `WorkList` ▷ recompute the subtree rooted at p
  else
    for $a \in \Sigma$ **do** ▷ expand n
      $\phi \leftarrow \mathsf{Post}^\sharp_{\mathcal{A}}(\Lambda(n), a)$
      if exist $m \in N$ such that $\phi \models \Lambda(m)$ **then**
        $\lhd \leftarrow \lhd \cup \{(n, m)\}$ ▷ m covers n
      else
        let s be a fresh node
        $E \leftarrow E \cup \{(n, a, s)\}$
        $\Lambda \leftarrow \Lambda \cup \{(s, \phi)\}$
        $R \leftarrow \{m \in \mathtt{WorkList} \mid \Lambda(m) \models \phi\}$ ▷ worklist nodes covered by s
        for $r \in R$ **do**
          for $m \in N$ such that $(m, b, r) \in E$, $b \in \Sigma$ **do**
            $\lhd \leftarrow \lhd \cup \{(m, s)\}$ ▷ redirect covered children from R into s
          for $(m, r) \in \lhd$ **do**
            $\lhd \leftarrow \lhd \cup \{(m, s)\}$ ▷ redirect covered nodes from R into s
          remove R from $\mathcal{T}$
          enqueue s into `WorkList`
return `true`

interpolants that yield a safety invariant, whenever $L(\mathcal{A}) = \emptyset$. In this case, we use the covering relation $\lhd$ to ensure that, when a newly generated node is covered by a node already in N, it is not added to the worklist, thus cutting the current branch of the search.

Formally, for any two nodes $n, m \in N$, we have $n \lhd m$ iff $\mathsf{Post}^\sharp_{\mathcal{A}}(\Lambda(n), a) \models \Lambda(m)$ for some $a \in \Sigma$, in other words, if n has a successor whose label entails the label of m.

Example. Consider the automaton given in Fig. 1. First, Algorithm 1 fires the sequence a, and since there are no other formulae than $\bot$ in Π, the successor of $\iota \equiv q_0$ is $\top$, in Fig. 2(a). The spuriousness check for a yields the root of the ART as pivot and the interpolant $\langle q_0, q_1 \rangle$, which is added to the set Π. Then the $\top$ node is removed and the next time a is fired, it creates a node labeled q_1. The second sequence aa creates a successor node q_1, which is covered by the first, depicted with a dashed arrow, in Fig. 2(b). The third sequence is ab, which results in a new uncovered node $\top$ and triggers a spuriousness check. The new

predicate obtained from this check is $x \leq 0 \wedge q_2 \wedge y \geq 0$ and the pivot is again the root. Then the entire ART is rebuilt with the new predicates and the fourth sequence *aab* yields an uncovered node $\top$, in Fig. 2(c). The new pivot is the endpoint of a and the newly added predicates are $q_1 \wedge q_2$ and $y > x - 1 \wedge q_2$. Finally, the ART is rebuilt from the pivot node and finally all nodes are covered, thus proving the emptiness of the automaton, in Fig. 2(d). □

The correctness of Algorithm 1 is proved below:

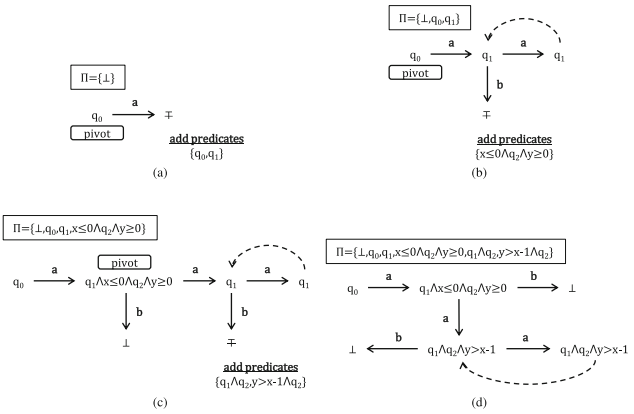


Fig. 2. Proving emptiness of the automaton from Fig. 1 by Algorithm 1

Theorem 1. *Given an automaton $\mathcal{A}$, such that $L(\mathcal{A}) \neq \emptyset$, Algorithm 1 terminates and returns a word $w \in L(\mathcal{A})$. If Algorithm 1 terminates reporting* `true`*, then $L(\mathcal{A}) = \emptyset$.*

6 Checking ADA Emptiness with IMPACT

As pointed out by a number of authors, the bottleneck of predicate abstraction is the high cost of reconstructing parts of the ART, subsequent to the refinement of the set of predicates. The main idea of the IMPACT procedure [17] is that this can be avoided and the refinement (strengthening of the node labels of the ART) can be performed in-place. This refinement step requires an update of the covering relation, because a node that used to cover another node might not cover it after the strengthening of its label.

We consider a total alphabetical order $\prec$ on Σ and lift it to the total lexicographical order $\prec^*$ on Σ^*. A node $n \in N$ is *covered* if $(n, p) \in \lhd$ or it has an ancestor m such that $(m, p) \in \lhd$, for some $p \in N$. A node n is *closed* if it is covered, or $\Lambda(n) \not\models \Lambda(m)$ for all $m \in N$ such that $\lambda(m) \prec^* \lambda(n)$. Observe that we use the coverage relation $\lhd$ here with a different meaning than in Algorithm 1.

The execution of Algorithm 2 consists of three phases[5]: *close*, *refine* and *expand*. Let n be a node removed from the worklist at line 4. If $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$

[5] Corresponding to the CLOSE, REFINE and EXPAND in [17].

Algorithm 2. IMPACT for ADA Emptiness

input: an ADA $\mathcal{A} = \langle \mathbf{x}, Q, \iota, F, \Delta \rangle$ over the alphabet Σ of input events
output: `true` if $L(\mathcal{A}) = \emptyset$ and a data word $w \in L(\mathcal{A})$ otherwise

1: let $\mathcal{T} = \langle N, E, \mathtt{r}, \Lambda, R, T, \lhd \rangle$ be an ART
2: initially $N = E = T = \lhd = \emptyset$, $\Lambda = \{(\mathtt{r}, \iota)\}$, $R = \mathrm{FV}^{\mathsf{Bool}}(\iota[Q_0/Q])$, `WorkList` $= \{\mathtt{r}\}$
3: **while** `WorkList` $\neq \emptyset$ **do**
4: dequeue n from `WorkList`
5: $N \leftarrow N \cup \{n\}$
6: let $(\mathtt{r}, a_1, n_1), (n_1, a_2, n_2), \ldots, (n_{k-1}, a_k, n)$ be the path from `r` to n
7: **if** $\mathsf{Acc}_{\mathcal{A}}(a_1 \ldots a_k)$ is satisfiable **then** ▷ counterexample is feasible
8: get model $(\beta, \nu_1, \ldots, \nu_k)$ of $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$
9: **return** $w = (a_1, \nu_1) \ldots (a_k, \nu_k)$ ▷ $w \in L(\mathcal{A})$ by construction
10: **else** ▷ spurious counterexample
11: let $\langle \top, I_0, \ldots, I_k, \bot \rangle$ be an interpolant for $\Theta(a_1 \ldots a_k)$
12: $b \leftarrow$ `false`
13: **for** $i = 0, \ldots, k$ **do**
14: **if** $\Lambda(n_i) \not\models I_i$ **then**
15: $\lhd \leftarrow \lhd \setminus \{(m, n_i) \in \lhd \mid m \in N\}$
16: $\Lambda(n_i) \leftarrow \Lambda(n_i) \wedge I_i$ ▷ strenghten the label of n_i
17: **if** $\neg b$ **then**
18: $b \leftarrow$ CLOSE(n_i)
19: **if** n is not covered **then**
20: **for** $a \in \Sigma$ **do** ▷ expand n
21: let s be a fresh node and $e = (n, a, s)$ be a new edge
22: $E \leftarrow E \cup \{e\}$
23: $\Lambda \leftarrow \Lambda \cup \{(s, \top)\}$
24: $T \leftarrow T \cup \{(e, \theta_k)\}$
25: $R \leftarrow R \cup \{(s, \bigcup_{q \in R(n)} \mathrm{FV}^{\mathsf{Bool}}(\Delta(q, a)))\}$
26: enqueue s into `WorkList`
27: **return** `true`

1: **function** CLOSE(x) **returns** Bool
2: **for** $y \in N$ such that $\lambda(y) \prec^* \lambda(x)$ **do**
3: **if** $\Lambda(x) \models \Lambda(y)$ **then**
4: $\lhd \leftarrow (\lhd \setminus \{(p, q) \in \lhd \mid q \text{ is } x \text{ or a successor of } x\}) \cup \{(x, y)\}$
5: **return** `true`
6: **return** `false`

is satisfiable, the counterexample $\lambda(n)$ is feasible, in which case a model of $\mathsf{Acc}_{\mathcal{A}}(\lambda(n))$ is obtained and a word $w \in L(\mathcal{A})$ is returned. Otherwise, $\lambda(n)$ is a spurious counterexample and the procedure enters the refinement phase (lines 11–18). The interpolant for $\Theta(\lambda(n))$ (cf. formula 1) is used to strenghten the labels of all the ancestors of n, by conjoining the formulae of the interpolant to the existing labels.

In this process, the nodes on the path between `r` and n, including n, might become eligible for coverage, therefore we attempt to close each ancestor of n that is impacted by the refinement (line 18). Observe that, in this case the call to CLOSE must uncover each node which is covered by a successor of n (line 4 of the CLOSE function). This is required because, due to the over-approximation of the sets of reachable configurations, the covering relation is not transitive, as explained in [17]. If CLOSE adds a covering edge (n_i, m) to $\lhd$, it does not have to be called for the successors of n_i on this path, which is handled via the Boolean flag b.

Finally, if n is still uncovered (it has not been previously covered during the refinement phase) we expand n (lines 20–26) by creating a new node for each successor s via the input event $a \in \Sigma$ and inserting it into the worklist.

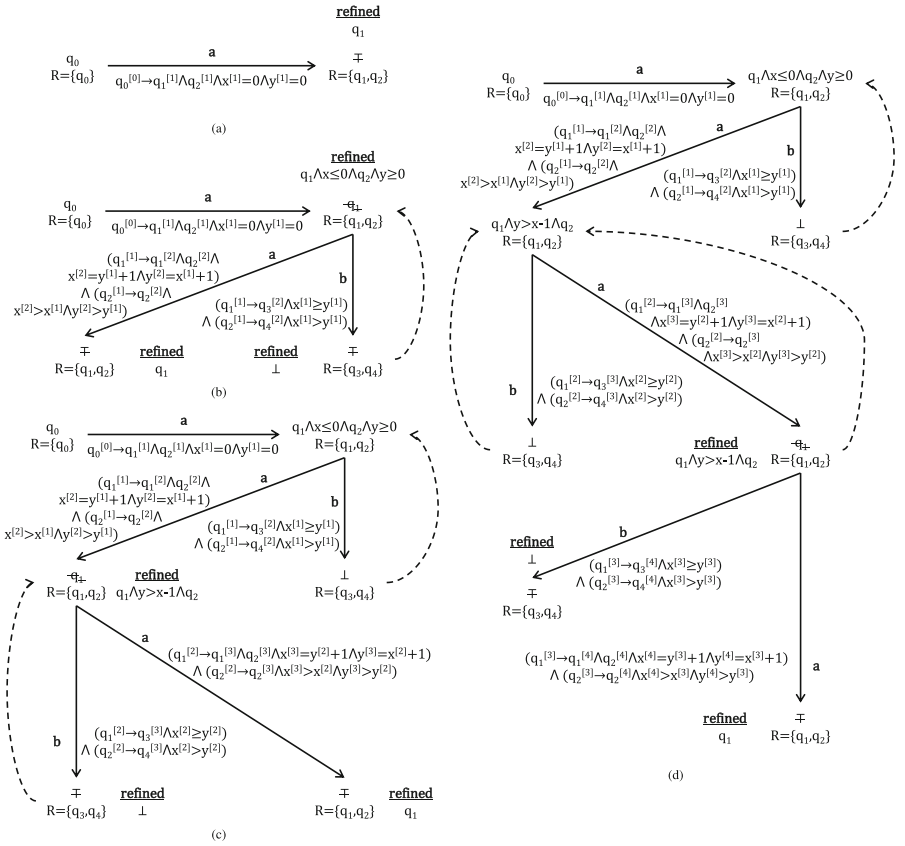


Fig. 3. Proving emptiness of the automaton from Fig. 1 by Algorithm 2

Example. We show the execution of Algorithm 2 on the automaton from Fig. 1. Initially, the procedure fires the sequence a, whose endpoint is labeled with $\top$, in Fig. 3(a). Since this node is uncovered, we check the spuriousness of the counterexample a and refine the label of the node to q_1. Since the node is still uncovered, two successors, labeled with $\top$ are computed, corresponding to the sequences aa and ab, in Fig. 3(b). The spuriousness check for aa yields the interpolant $\langle q_0, x \leq 0 \wedge q_2 \wedge y \geq 0 \rangle$ which strengthens the label of the endpoint of a from q_1 to $q_1 \wedge x \leq 0 \wedge q_2 \wedge y \geq 0$. The sequence ab is also found to be spurious, which changes the label of its endpoint from $\top$ to $\bot$, and also covers it (depicted with a dashed edge). Since the endpoint of aa is not covered, it is expanded to aaa and aab, in Fig. 3(c). Both sequences aaa and aab are found to be spurious, and the enpoint of aab, whose label has changed from $\top$ to $\bot$, is now covered. In the process, the label of aa has also changed from q_1 to $q_1 \wedge y > x - 1 \wedge q_2$, due to the sstrengthening with the interpolant from aab. Finally, the only uncovered node aaa is expanded to $aaaa$ and $aaab$, both found to be spurious, in

Fig. 3(d). The refinement of $aaab$ causes the label of aaa to change from q_1 to $q_1 \wedge y > x - 1 \wedge q_2$ and this node is now covered by aa. Since its successors are also covered, there are no uncovered nodes and the procedure returns `true`. □

The correctness of Algorithm 2 is coined by the theorem below:

Theorem 2. *Given an automaton $\mathcal{A}$, such that $L(\mathcal{A}) \neq \emptyset$, Algorithm 2 terminates and returns a word $w \in L(\mathcal{A})$. If Algorithm 2 terminates reporting* `true`*, then $L(\mathcal{A}) = \emptyset$.*

7 Experimental Evaluation

We have implemented both Algorithms 1 and 2 in a prototype tool[6] that uses the MathSAT5 SMT solver[7] via the Java SMT interface[8] for the satisfiability queries and interpolant generation, in the theory of linear integer arithmetic with uninterpreted Boolean functions (UFLIA). We compared both algorithms with a previous implementation of a trace inclusion procedure, called INCLUDER[9], that uses on-the-fly determinisation and lazy predicate abstraction with interpolant-based refinement [11] in the LIA theory. The datasets generated during and/or analysed during the current study are available in the figshare repository: https://doi.org/10.6084/m9.figshare.5925472.v1 [12].

Table 1.

Example	$\lvert\mathcal{A}\rvert$ (bytes)	$L(\mathcal{A}) = \emptyset$?	Algorithm 1 (sec)	Algorithm 2 (sec)	INCLUDER (sec)
simple1	309	No	0.774	0.064	0.076
simple2	504	Yes	0.867	0.070	0.070
simple3	214	Yes	0.899	0.095	0.095
array_shift	874	Yes	2.889	0.126	0.078
array_simple	3440	Yes	Timeout	9.998	7.154
array_rotation1	1834	Yes	7.227	0.331	0.229
array_rotation2	15182	Yes	Timeout	Timeout	31.632
abp	6909	No	9.492	0.631	2.288
train	1823	Yes	19.237	0.763	0.678
hw1	322	Yes	1.861	0.163	0.172
hw2	674	Yes	24.111	0.308	0.473

The results of the experiments are given in Table 1. We applied the tool first to several array logic entailments, which occur as verification conditions for imperative programs with arrays [2] (array_shift, array_simple, array_rotation1+2)

[6] The implementation is available at https://github.com/cathiec/JAltImpact.
[7] http://mathsat.fbk.eu/.
[8] https://github.com/sosy-lab/java-smt.
[9] http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/.

available online [19]. Next, we applied it on proving safety properties of hardware circuits (hw1+2) [22]. Finally, we considered two timed communication protocols, consisting of systems that are asynchronous compositions of timed automata, whom correctness specifications are given by timed automata monitors: a timed version of the Alternating Bit Protocol (abp) [25] and a controller of a railroad crossing (train) [9]. All results were obtained on x86_64 Linux Ubuntu virtual machine with 8 GB of RAM running on an Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00 GHz. The automata sizes are given in bytes needed to store their ASCII description on file and the execution times are in seconds.

As in the case of non-alternating nondeterministic integer programs [17], the alternating version of Impact (Algorithm 2) outperforms lazy predicate abstraction for checking emptiness by at least one order of magnitude. Moreover, Impact is comparable, on average, to the previous implementation of Includer, which uses also MathSAT5 via the C API. We believe the reason for which Includer outperforms Impact on some examples is the hardness of the UFLIA entailment checks used in Algorithm 2 (lines 14 and 3 in the function Close) as opposed to the pure LIA entailment checks used in Includer. According to our statistics, Algorithm 2 spends more than 50% of the time waiting for the SMT solver to finish answering entailment queries.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
2. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_15
3. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1), 114–133 (1981)
4. D'Antoni, L., Kincaid, Z., Wang, F.: A symbolic decision procedure for symbolic alternating finite automata. CoRR, abs/1610.01722 (2016)
5. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: alternative algorithms for LTL satisfiability and model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_6
6. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. SIGPLAN Not. **50**(1), 407–420 (2015)
7. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. SIGPLAN Not. **47**(6), 405–416 (2012)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Not. **37**(1), 58–70 (2002)
9. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. **111**, 394–406 (1992)
10. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13

11. Iosif, R., Rogalewicz, A., Vojnar, T.: Abstraction refinement and antichains for trace inclusion of infinite state systems. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 71–89. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_5
12. Iosif, R., Xu, X.: Artifact related to abstraction refinement for emptiness checking of alternating data automata. In: TACAS 2018 (2018). https://doi.org/10.6084/m9.figshare.5925472.v1
13. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994)
14. Lasota, S., Walukiewicz, I.: Alternating timed automata. In: Sassone, V. (ed.) FoSSaCS 2005. LNCS, vol. 3441, pp. 250–265. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31982-5_16
15. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. Ann. Pure Appl. Logic **56**(1), 239–311 (1992)
16. Lyndon, R.C.: An interpolation theorem in the predicate calculus. Pacific J. Math. **9**(1), 129–142 (1959)
17. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_14
18. McMillan, K.L.: Lazy annotation revisited. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 243–259. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_16
19. Numerical Transition Systems Repository (2012). http://nts.imag.fr/index.php/Flata
20. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: closing a decidability gap. In: Proceedings of LICS 2004, pp. 54–63 (2004)
21. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS 1977, pp. 46–57. IEEE (1977)
22. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: HVC 2007, pp. 51–68 (2007)
23. Vardi, M., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
24. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: algorithms and applications. In: Proceedings of POPL 2012. ACM (2012)
25. Zbrzezny, A., Polrola, A.: Sat-based reachability checking for timed automata with discrete data. Fundamenta Informaticae **79**, 1–15 (2007)