An Alternating Automaton for First-Order Linear Temporal Logic: Extended version with proofs

Raphaël Khoury, Sylvain Hallé, and Yannick Lebrun

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

Abstract. In this paper we present an automata-based verification procedure for LTL-FO$^+$ properties. LTL-FO$^+$ is an extension of LTL that includes first-order quantification over bounded variables, thus greatly increasing the expressivity of the language. We show how to construct an automata representation of an LTL-FO$^+$ property which allows verification using a breadth-first search throughout the automata and we prove the correctness of the construction. Finally, we introduce Pelota, an automata-based monitor for LTL-FO$^+$ and show empirical results of its use on sample properties. Compared with existing solutions, Pelota exhibits superior time and space consumption, especially on memory intensive properties.

1 Introduction

LTL-FO$^+$ [6] is a formal language used for the specification of trace properties which distinguishes itself from other representations by its exceptional expressiveness. It allows users to state finer relationships between the different elements of several messages in a complex event trace. For example, Hallé [6] uses LTL-FO$^+$ to express properties related to XML message traces generated by web services. Such properties cannot be stated with a less expressive formalism such as LTL.

However, the on-the-fly verification algorithm is based on the decomposition and rewriting of formulæ. It runs in $O(n^2)$ for a trace of length n and is highly intensive in its space consumption, with multiple manipulations being performed even when it processes messages that have no bearing on the validity of the formula. Indeed, for some formulæ, the evaluation tree expands indefinitely, and equivalent subtrees, which could be pruned, are hard to identify. Furthermore, the elaborate syntax of LTL-FO$^+$ can make it difficult to state and read properties (see for example [8]).

In this paper, we present an automaton-based verification procedure for LTL-FO$^+$ properties. First, we show how to produce an automata-based representation of LTL-FO$^+$ formulæ. Then, we propose a verification procedure and prove the correctness of the approach. Finally, we introduce Pelota[1], an automata-based

[1] The name is created using the letters composing the name of the logic LTL as well as A($\forall$), E($\exists$) and P($+$). The code is available at https://github.com/RaphaelKhoury/Pelota

tool for monitoring LTL-FO$^+$ properties. We give experimental results and show that Pelota compares favorably to other solutions for LTL-FO$^+$ properties, in large part because its automata-based verification process allows it to reduce its memory footprint, and consequently, its total running time. The space and time complexity of the algorithm is $O(n * d)$, where n is the length of the input trace and d is the number of states simultaneously visited by the automaton at runtime. While a worst case in which $d = n$ is unavoidable for trace-length dependent properties, since the monitor may have to record every message indefinitely, we show through examples that in many cases, d can be substantially smaller than n, and that the approach under consideration exhibits a substantial improvement in execution time over existing solutions for LTL-FO$^+$ verification.

The automaton we propose is a variation of Vardi and Wolper's alternating automata [12], enriched with first-order quantifiers over a finite set of formula variables. This makes it possible to express intricate formulæ over complex events, where each event consists of an XML object with possibly multiple valuations for each variable in the same event. As is the case for the alternating automata, the proposed automata distinguish between *existential* and *universal* transitions. Existential transitions are analogous to non-deterministic transitions in regular Büchi automata. Upon encountering such a transition, the automata can be thought of as choosing between multiple destination states. Conversely, when encountering a universal transition, the automata continue their run in every target state simultaneously. A run over an alternating automaton generates a tree of states. A run is accepting if there *exists* at least one tree for which *every* branch visits an accepting state infinitely often. While alternating automata are equally expressive as non-deterministic Büchi automata, we show in this paper how the notion of existential and universal transitions can be used to model the quantifiers present in LTL-FO$^+$ formulæ. Additionally, our automata are enriched with a partial function that maps formula variables to their value. This function is manipulated by the automaton's transitions as the input sequence is read, and consulted to determine the truth value of elementary propositions.

Compared with existing automata models, the one presented in this paper distinguishes itself by its expressivity. The input language, which consists of sequences of unrestricted XML objects, is a strict superset of *data words* [10] (i.e pairs of the form $\langle \text{label} \times \text{value} \rangle$) used in several other models. Furthermore, even when restricted to data words, the proposed automata exhibit greater expressivity than other existing models, as we will illustrate in this paper using examples.

The remainder of this paper is organized as follows. Section 2 provides background information about LTL-FO$^+$. Section 3 surveys existing automata representations for other formal logics. In Section 4, we show how to construct an automaton $\mathcal{A}$ from an LTL-FO$^+$ property φ such that $\mathcal{A}$ accepts a sequence iff it satisfies the property φ. Section 5 gives the verification procedure for such an automaton. Section 6 gives general insights on the implementation and experimental results on sample traces. Concluding remarks are given in Section 7.

2 The First-Order Temporal Logic LTL-FO$^+$

LTL-FO$^+$ is a first-order extension of a well-known logic called Linear Temporal Logic (LTL), a logic first introduced to express properties about sequences of states in Kripke structures [3]. In the current case, the states under consideration are XML objects termed *messages*. A message is a set of pairs $p \in \Pi \times I$ where Π is a set of XPath expressions (in standard XPath 1.0 notation) and I is a set of values.

Definition 1 (Message) *Let Π is a set of XPath expressions and let I be a domain of values. A XML message m is a finite set of pairs $p \in \Pi \times I$*

We write M for the set of XML messages. A sequence $\overline{m}$ of messages $m_1, m_2 \ldots$, where $m_i \in M$ for every $i \geq 1$, is called a message trace. We write m_i to denote the i-th message of the trace $\overline{m}$, and $\overline{m}^i$ to denote the trace obtained from $\overline{m}$ by starting at the i-th message.

Definition 2 (Message trace) *Let M be a set of messages. A message trace $\overline{m}$ is a finite sequence $m_1, m_2 \ldots$, where $m_i \in M$ for every $i \geq 1$.*

A domain function $Dom_m(\pi) : M \times \Pi \to 2^I$ is used to fetch and compare values inside a message; it receives an argument a message $m \in M$ and a path $\pi \in \Pi$, and returns a subset $Dom_m(\pi)$ of I, representing the set of values appearing in message m at the end of the path π.

We write $\bigvee_{x \in S} f(x)$ (resp. $\bigwedge_{x \in S} f(x)$) for the boolean disjunction (resp. conjunction) of the expression $f(x)$ with each elements in the set S substituted in a disjunct (resp. conjunct.). Formally $\bigvee_{x \in Sf(x)} = f(x_1) \vee f(x_2) \vee f(x_3) ... \forall x_i \in S$ and $\bigwedge_{x \in Sf(x)} = f(x_1) \wedge f(x_2) \wedge f(x_3) ... \forall x_i \in S$.

LTL-FO$^+$'s syntax is based on classical propositional logic, using the connectives $\neg$ ("not"), $\vee$ ("or"), $\wedge$ ("and"), $\rightarrow$ ("implies"), to which five temporal operators have been added. An LTL-FO$^+$ formula is a well-formed combination of these operators and connectives, according to the usual construction rules:

Definition 3 (Syntax) *1. If x and y are variables or constants, then $x = y$ is an LTL-FO$^+$ formula;*
2. *If φ and ψ are LTL-FO$^+$ formulæ, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\mathbf{G}\,\varphi$, $\mathbf{F}\,\varphi$, $\mathbf{X}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi\mathbf{R}\psi$ are LTL-FO$^+$ formulæ;*
3. *If φ is an LTL-FO$^+$ formula, x is a free variable in φ, $p \in \Pi$ is an XPath value, then $\exists_p x : \varphi$ and $\forall_p x : \varphi$ are LTL-FO$^+$ formulæ.*

Note that the use of Boolean equality in the first rule easily generalizes to other connectives.

In the context of finite traces, Boolean connectives and LTL operators mostly carry their usual meaning. We say that a finite message trace $\overline{m} = m_1, m_2, \ldots, m_n$ *satisfies* (resp. *violates*) an LTL-FO$^+$ formula φ, written $[\overline{m} \models \varphi] = \top$ (resp.

$[\overline{m} \models \varphi] = \bot$), according to the rules of Definition 4. As usual, we define the semantics of the other connectors with the following identities: $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, $\mathbf{G}\ \varphi \equiv \neg(\mathbf{F}\ \neg\varphi)$, $\varphi\, \mathbf{R}\, \psi \equiv \neg(\neg\varphi\ \mathbf{U}\ \neg\psi)$, $\forall_p x : \varphi \equiv \neg(\exists_p x : \neg\varphi)$. The notation $\eta(x/v)$, in the rule $\exists$, indicates that in formula η, every instance of the variable x is bound to a value v read from the trace.

Any LTL-FO$^+$ formula also has an equivalent *negation normal form*. This form is always obtainable with the previous identities.

Definition 4 (Semantics of LTL-FO$^+$ (from [6]))

$$[\overline{m} \models x = y] = \begin{cases} \top & \textit{if } p(x) \textit{ is equal to } p(y) \\ \bot & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \neg\eta] = \begin{cases} \top & \textit{if } [\overline{m} \models \eta] = \bot \\ \bot & \textit{if } [\overline{m} \models \eta] = \top \\ ? & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \mu \vee \eta] = \begin{cases} \top & \textit{if } [\overline{m} \models \mu] = \top \textit{ or } [\overline{m} \models \eta] = \top \\ \bot & \textit{if } [\overline{m} \models \mu] = \bot \textit{ and } [\overline{m} \models \eta] = \bot \\ ? & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \mathbf{X}\ \eta] = \begin{cases} [\overline{m}^2 \models \eta] & \textit{if } n \geq 2 \\ ? & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \mathbf{F}\ \eta] = \begin{cases} \top & \textit{if } \exists\ 1 \leq j \leq n : [\overline{m}^j \models \eta] = \top \\ ? & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \mu\ \mathbf{U}\ \eta] = \begin{cases} \top & \textit{if } \exists\ 1 \leq j \leq n : (([\overline{m}^j \models \eta] = \top) \wedge (\forall\ 1 \leq k < j : [\overline{m}^k \models \mu] = \top)) \\ \bot & \textit{if } \exists\ 1 \leq j \leq n : (([\overline{m}^j \models \mu] = \bot) \wedge (\forall\ 1 \leq k \leq j : [\overline{m}^k \models \eta] = \bot)) \\ ? & \textit{otherwise;} \end{cases}$$

$$[\overline{m} \models \exists_\pi x : \eta] = \begin{cases} \top & \textit{if } \exists\ \overline{x} \in Dom_{m_1}(\pi) : [\overline{m} \models \eta(x/\overline{x})] = \top \\ \bot & \textit{if } \forall\ \overline{x} \in Dom_{m_1}(\pi) : [\overline{m} \models \eta(x/\overline{x})] = \bot \textit{ or } Dom_{m_1}(\pi) = \emptyset \\ ? & \textit{otherwise;} \end{cases}$$

Definition 5 *An LTL-FO$^+$ formula is in* negation normal form (NNF) *if it does not contain the connective $\rightarrow$, and if all negations $\neg$ are pushed inside until they precede equalities.*

We identify $\neg(x = y)$ with $x \neq y$ in order to eliminate the connective $\neg$ completely. In what follows, we consider only properties in NNF.

We denote by V the set of variables that occur in an LTL-FO$^+$ formula and by I the domain of values that may appear in a message. The state of all variables in V during a run of some automaton $\mathcal{A}$ can always be represented by a partial function $p : V \rightarrow I$. Some variables may not be assigned yet, but p is updated continuously as $\mathcal{A}$ reads the input trace, with $p(x)$ representing the valuation of a variable $x \in V$. Abusing the notation, we write $p(v) = v$ for any constant value $v \in I$ that occurs in a formula.

In previous work [6], LTL-FO$^+$ formulæ are monitored using a tableaux-like procedure. An automata representation could be substantially more efficient by allowing the evaluation of a property through a breadth-first traversal of the property automaton.

3 Related Representations

Let Σ be a finite alphabet of labels and D be an infinite set of data values. A *data word* is a finite sequence of pairs from $\Sigma \times D$. A *data language* is a set of data words. This representation, while similar, is less expressive than the streams of XML messages that form the input of LTL-FO$^+$ formulæ.

Segoufin [10] surveyed multiple automata-representations for data-languages. Amongst them we note the register automata and the pebble automata. The register automaton [7] is a finite state machine equipped with a finite number of registers which can be used to store values from D and compare them for equality with the value at the current position in the input sequence. The pebble automaton [9] possesses pebbles which it can drop or lift on the input sequence at chosen locations, and can compare the current value of the current position in the input sequence with those locations that are marked with pebbles.

Segoufin observes that the expressivity of both models are orthogonal. The language consisting of data words containing two positions labelled with 'a' and having the same data value (property *a is not a key*) is expressible using a register automaton but not a pebble automaton. Conversely, the language consisting of words for which each value labeled with 'a' is different is accepted by a pebble automaton but not by a register automaton. In this paper, we show how both of these properties can be stated with LTL-FO$^+$ and verified using Pelota. Indeed, Figures 1 and 2 show the automata representation of these two properties, as well as the associated LTL-FO$^+$ formulae. Observe that the latter property is written by simply negating the first operator and the first quantifier of the former, and that neither make explicit references to the register manipulations.

In addition, properties verified by the register automaton must explicitly describe register manipulations using the *freeze* operator. LTL-FO$^+$ internalizes register manipulations, allowing properties to be written using only LTL symbols and quantifiers. This enables users to state properties in a more intuitive way, separate from the mechanism used to enforce it.

QEAs [1] and DATEs [4] provide a highly expressive formalism for stating and verifying properties over streams of complex events. However, in both cases, the input is limited in that it requires that every event of its input trace must be of a specific format. LTL-FO$^+$ operates on unrestricted streams of XML messages, a strictly more expressive formalism.

Bauer et al. [2] propose a automaton representation for LTL^{FO}, and extension of LTL that subsumes LTL-FO$^+$. While more expressive than LTL-FO$^+$, properties expressed in LTL^{FO} are undecidable.

4 LTL-FO$^+$ to Alternating Automaton

4.1 LTL-FO$^+$Automaton Formalization

The automata representation of an LTL-FO$^+$ formula is a variant of Vardi and Wolper's alternating automata, enriched with a map linking the variables that occur in the formula to values observed in the trace. The automaton exhibits 4 distinct types of states, drawn from 4 disjoint sets: *regular states* (or *formula states*), which capture the input property or one of its subformulæ, *conditional states*, in which a value is added to the variable valuation map, *evaluation states* which consult the valuation map to determine the truth value of a predicate at a given point and *final states* (true and false).

Let X be a finite set. The set of positive boolean formulas over X, denoted by $\mathfrak{B}^+(X)$, contains $\top$ (true), $\bot$ (false), all elements from X, and all boolean combinations over X built using $\wedge$ and $\vee$. Let $X' \in \mathfrak{B}^+(X)$ We write $f(X')$ to indicate the result of applying function f to every operand of X'. For instance $f(x \vee (y \wedge z)) = (f(x) \vee (f(y) \wedge f(z))$.

The automaton's formal definition rests upon the notion of dynamic states. Let $M : \Pi \times I$ be a set of messages and let Q be a set of automata states, as defined below. A dynamic state $s \in ((\Pi \to I) \times Q)$ is a pair comprising a automaton state and a map that associates labels from Π with values from I. Let p be a map. We write $p(x)$ for the image y of x in p. We write $p\dagger[\pi \mapsto v]$ to indicate p to which the pair $[\pi \mapsto v]$ is added. The automaton's execution is initialized with the dynamic state $s_0 = ([], q_0)$ where [] is the empty map. From that point on, each message will make the automaton transition into a new boolean configuration of states, as well as possibly modify the label-value mapping associated with each state.

Definition 6 (Automaton for LTL-FO$^+$) *An automaton $\mathcal{A}$ is a tuple $\langle Q, \Pi, I, q_o, \delta, \lambda \rangle$ where :*

- *$Q = Q_r \cup Q_c \cup Q_e \cup Q_f$ a finite set of states. Q_r, Q_c, Q_e and Q_f are disjoint subsets of states defined as follow:*
  - *Q_r is a finite set of formula states;*
  - *Q_c is a finite set of conditional states. Conditional states aggregate the values read in the message trace into the automaton's state in concordance with the existential and universal quantifiers present in the formula;*
  - *Q_e is a finite set of evaluation states. Evaluation states consults the values stored in the dynamic states to ascertain the respect of a boolean formula;*
  - *$Q_f = \{q_\top, q_\bot\}$ is a set of final states, with $q_\top$ being the unique accepting state.*
- *$q_0 \in Q_r$ is the initial state;*
- *Π and I are XML paths and values, respectively;*
- *$\delta : (\Pi \to I) \times Q \to (\mathfrak{B}^+((\Pi \to I) \times Q))$ and $\lambda : ((\Pi \to I) \times Q) \to ((\Pi \to I) \times Q)$ are two transition functions, whose operation is detailed below.*

Aside from the label-value map, the automaton is distinctive in that processing a single message may cause the verification procedure automaton to follow multiple

consecutive transitions. In fact, starting from the current state, transitions are followed until a regular state or a final state is encountered on each path. The conditional states and evaluation states capture the complex multiple steps that the automaton must perform when processing a single event. These steps are conceptually split into multiple automaton states for ease of understanding and motivate the use of two distinct transition function δ and λ. The two final states each possess a single outgoing identity transition for every message.

The transition function δ returns a boolean formula of successor states of its input state, and leaves the associated map unchanged. The λ function is then called on each output dynamic state.

The λ transition function queries the current message and updates the map present in a dynamic state with any relevant value encountered in the trace. It also consults this map as needed to determine the validity of the property. As described above, the Q_c states capture the semantics of existential and universal quantifiers. Let $q_c \in Q_c$ be a conditional state. Each conditional state carries an indication of whether it models an existential or universal quantifier, denoted by $q_c.type \in \{\exists, \forall\}$, and of the label that is being quantified $q_c.\pi \in \Pi$.

Let (p, q_c)be the current state with $q_c \in Q_c$, and let m be the current message, and let $Dom_m(q_c.\pi) = S$. λ is defined as :

$$\lambda(f, q_c) = \begin{cases} ([], q_\bot), & \text{if } S = \emptyset \wedge q_c.type = \exists; \\ ([], q_\top), & \text{if } S = \emptyset \wedge q_c.type = \forall; \\ \bigvee\limits_{v \in S} \delta(p \dagger [q_c.\pi \mapsto v], q_c), & \text{if } S \neq \emptyset \wedge q_c.type = \exists; \\ \bigwedge\limits_{v \in S} \delta(p \dagger [q_c.\pi \mapsto v], q_c), & \text{otherwise.} \end{cases}$$

Informally, when processing an existential (resp. universal) conditional state, the λ function either moves to the rejecting (resp. accepting) final state if the quantification domain is empty, discarding the now irrelevant valuation map, or else disjunctively (resp. conjunctively) moves to the conditional state's other successor, updating the valuation map with the new value(s) encountered in the current message.

Let $s = (p, q_e) \in \mathcal{D}$ be a dynamic state, with $q \in Q_e$. The evaluation state contains a operator $q.op$, a left operand $q.leftOp$ and aright operand $q.rightOp$. We write $eval(c_1, q.op, c_2)$ for the result of comparing the values c_1 and c_2 with operator op. The conditional state may consult the current dynamic state's map p to obtain the values of $q.leftOp$ and $q.rightOp$, or these can be present within the state itself if they are constants.

Let (p, q_e)be the current state with $q_e \in Q_e$, λ is defined as :

$$\lambda(p, q_e) = \begin{cases} ([], q_\top), \text{ if } eval(q_e.leftOp, q_e.op, q_e.rightOp) = \textbf{true} \ ; \\ ([], q_\bot), & \text{otherwise.} \end{cases}$$

Finally, when applied to a state from $Q_r \cup Q_f$, λ operates as an identity function: $\lambda(p, q) = (p, q)$ iff $q \in Q_r \cup Q_f$. Observe that whenever the automaton reaches a final state, the valuation map is discarded and replaced with [] in the dynamic state, since it can no longer has any relevance at that point.

Since the automaton contains both existential and universal transitions, a traversal of the automaton generates a tree of dynamic states. We omit a formal

definition of a run here since other than for the distinctions mentioned above, it is identical that used by other alternating automaton. The notions of slice and sliceSets will be useful in the next section as part or the verification algorithm, as well as to prove the correctness of the approach. Let $n \geq 0$ be an integer, and let T be a run tree of some alternating automaton $\mathcal{A}$ on a trace $\overline{m}$. We define the ith *slice* of T to be the collection of all nodes of T at distance i from the root.

Definition 7 (slice) *Let $\mathcal{A}$ be an automaton, and let $i < \mathbb{N}$. $slice_i = \{s \in \mathcal{D} | \text{there exists a run } T \text{ of } \mathcal{A} \text{ s.t.} (s, i) \text{ is in } T\}$*

Definition 8 (sliceSet) *Let $\mathcal{A}$ be an automaton, and let $i < \mathbb{N}$. $sliceSet_i$ is the set of all slices of length i on $\mathcal{A}$ on a given input.*

We can now state the acceptance condition for the automaton. Let $\mathcal{A}$ be an automaton and let $\overline{m}$ be a message trace of length n. $\mathcal{A}$ accepts $\overline{m}$ iff in there exists a slice of the sliceSet $sliceSet_i$ generated by $\mathcal{A}$ on input $\overline{m}$, for which every node is $([], \top)$. Conversely, $\mathcal{A}$ rejects the $\overline{m}$ iff every slice in the sliceSet $sliceSet_i$ contains a node $([], \bot)$.

Definition 9 (Acceptance Condition) *Let $\mathcal{A}$ be an automaton, and let $i < \mathbb{N}$ and let $\overline{m}$ be a finite message trace of length i. $\mathcal{A}$ accepts $\overline{m}$ iff there exists a slice slice in the $sliceSet_i$ generated by $\mathcal{A}$ on the input $\overline{m}$, $\forall s \in slice : s = ([], \top)$. $\mathcal{A}$ rejects $\overline{m}$ iff every slice slice in the $sliceSet_i$ generated by $\mathcal{A}$ on the input $\overline{m}$, $\exists s \in slice : s = ([], \bot)$.*

Figures 1 and 2 give examples using the two properties mentioned above. Formula states are depicted with circles, final states are depicted by double circles with $\top$ being the unique accepting state. Conditional and evaluation states are depicted by rectangles. In these examples, each automaton contains a single evaluation state, labeled with a boolean formula, and succeeded by the final states. Each conditional state is graphically represented by a pair of rectangles in the figure, simply to make more evident the transitions that are taken from that state in each possible case, as illustrated below. Note that the smaller nodes marked with $\vee$ denote disjunctive transition, rather than actual states. The top-level XML separator between each message (in this case, *message*) is omitted from the automata for concision.

As an example, consider the property in Figure 1. The automaton is initialized in state $s_0 = ([], q_0)$. Upon consuming an input message m, the automaton disjunctively reaches both the initial state as well as a conditional state. From the latter state, if m does not contains the path $message/x$, the execution transitions to state $\bot$. The automaton is then in states $(s_0 \vee \bot)$, which is trivially simplified to (s_0). Otherwise, if $message/x$ is present in m, the automaton will transition (disjunctively) into a new state for each value v labeled with $message/x$ in m and assigns these values to a. For each value v, the automaton moves both into the second formula state $([a = v], q_1)$, which contains the inner **F** where it halts, as well as into a second series of conditional states that allow the automaton to look for a second values to assign to b. The final state is an evaluation state.

Upon encountering it, the automaton fetches the values of a and b from its map, performs the comparison and moves into the corresponding final states. Note that final states can be either accepting or non-accepting. Thus if the initial message of a trace is : $<message >< x > 3 < \backslash x >< \backslash message>$, upon having consumed this message, the automaton finds itself in state : $(s_0 \vee ([a = v], q_1) \vee \bot)$.

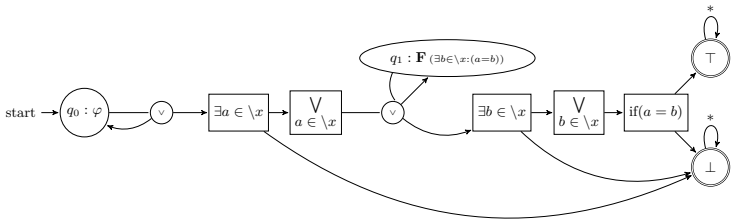

Fig. 1: Property 1: 'a' is not a key
$\varphi = \mathbf{F}\ (\exists a \in message/x : (\mathbf{F}\ (\exists b \in message/x : (a = b))))$

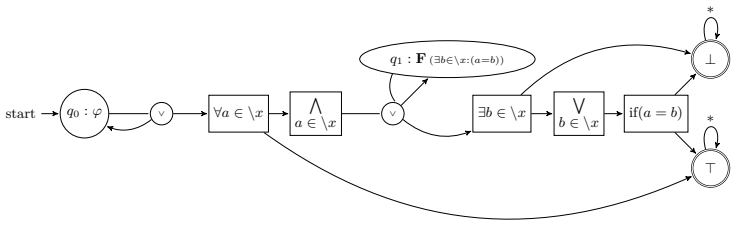

Fig. 2: Property 2: 'a' is a key
$\mathbf{G}\ (\forall a \in /message/x : (\mathbf{F}\ (\exists y \in /message/x : (a = y))))$

4.2 LTL-FO$^+$ Automaton Construction

The function `BuildAutomaton` takes as input an LTL-FO$^+$ formula ψ and return the automaton A_ψ (through a handle to its initial state s_0^ψ), which accepts a sequence iff ψ is satisfiable. The recursive subfunction `BuildTransition` takes an LTL-FO$^+$ formula ψ as input and returns the transition formula related to ψ.

The automaton has finite size regardless of the domain of quantified variables, and the number of states is linear in the size of the property. The boolean nature of dynamic states lends itself to multiple optimizations based on basic identities such as $(s \wedge \top) \equiv \top$ or $(s \wedge s) \equiv s$, which reduce their size, and as a consequence, the evaluation time of the formula. In fact these apparently simple identities are key to bounding the number of states visited by during a run of the automaton, and consequently to bounding the time and memory complexity of the problem.

5 Verification

As mentioned above, the transitions of an alternating automaton built by the `BuildAutomaton` algorithm combine universal and existential choices. The former are represented by the $\wedge$ transitions and make the automaton *simultaneously*

```
Function BuildTransition(LTL-FO+ ψ)
    if ψ instanceOf(x = y) or ψ instanceOf(x ≠ y) then
        return if(ψ){NodeSet.getNode(⊤)}else{NodeSet.getNode(⊥)}
    if ψ instanceOf(μ ∨ η) then
        return BuildTransition(μ) ⩒ BuildTransition(η)
    if ψ instanceOf(μ ∧ η) then
        return BuildTransition(μ) ⩓ BuildTransition(η)
    if ψ instanceOf(∃π x : η) then
        return ⨒_{x∈Dom(π)} BuildTransition(η)
    if ψ instanceOf(∀π x : η) then
        return ⨓_{x∈Dom(π)} BuildTransition(η)
    if ψ instanceOf(X η) then
        nodeηRef ← NodeSet.addNode(η);
        nodeηRef.setTransition(BuildTransition(η));
        return nodeηRef
    else
        nodeψRef ← NodeSet.addNode(ψ);
        if ψ instanceOf(F η) then
            transitionψ ← BuildTransition(η) ⩒ nodeψRef;
        if ψ instanceOf(G η) then
            transitionψ ← BuildTransition(η) ⩓ nodeψRef;
        if (ψ instanceOf(μ U η) then
            transitionψ ← BuildTransition(η) ⩒ (BuildTransition(μ) ⩓
              nodeψRef);
        if ψ instanceOf(μ R η) then
            transitionψ ← BuildTransition(η) ⩓ (BuildTransition(μ) ⩒
              nodeψRef);
        nodeψRef.setTransition(transitionψ);
        return transitionψ
```

consider every option. The latter are represented by the $\veebar$ transitions and make the automaton *non-deterministically* consider every option. Hence, a run of this automaton takes the form of a tree and multiple run trees exist for a given input sequence. As usual with alternation, the output is $\top$ iff every branch of at least one run tree ends on the accepting state $\top$. Otherwise, iff every run tree has a branch that ends on the rejecting state $\bot$. In any other case, the output is '?'.

Following [5], the verification algorithm proposed in this paper simulates a breadth first traversal of the run tree, examining the trace only once, and keeping track of all possible run trees. A crucial observation to make is that a verification algorithm for $\mathcal{A}$ does not need to remember every slice of T as $\overline{m}$ is traversed. Indeed, since $\mathcal{A}$ depends only on the current message and states for its next move, the information held in the nth slice of T suffices to compute its $(n+1)$th slice along with any other candidates that arise from non-determinism. As a consequence the computations are performed on *slicesets*. The nodes inside a slice consist in dynamic states,i.e. pairs $(p, state)$ where $p\colon V \to D$ is a partial function that assigns values to variables.

Function `BuildAutomaton(`*LTL-FO*$^+$ ψ`)`

```
NodeSet ← [];
node⊤Ref ←
  NodeSet.addNode(⊤);
node⊤Ref.setTransition(node⊤Ref)
node⊥Ref ←
  NodeSet.addNode(⊥);
node⊥Ref.setTransition(node⊥Ref)
nodeψRef ←
  NodeSet.addNode(ψ);
nodeψRef.setTransition(
  BuildTransition(ψ));
return nodeψRef
```

Function `BreadthFirst(`*Automaton* s_0, *Trace* $\overline{m}$`)`

```
SliceSet ← {{([], s0)}};
for i = 1 to |m̄| do
    SliceSet←
        ⋁_{slice ∈ SliceSet} ⋀_{(p,s) ∈ slice} NSS(p, s.trans, m_i)
end
if (SliceSet = true) then
    return ⊤
if (SliceSet = false) then
    return ⊥
else
    return ?
```

The sliceset is initialized with the pair $([], q_0)$. Then, for each message m of the input trace, we update SliceSet with in two-step procedure. First, for every pair $(p, state)$ of every slice that composes SliceSet, we compute the pairs succeeding $(p, state)$. This first step is performed in the call `NextSliceSet(`p, *state.trans*, m`)`. Since the transition formula of *state* may use existential ($\veebar$) and universal ($\barwedge$) choices, several non-deterministic possibilities with multiple branches may arise. Therefore, the pairs succeeding each pair $(p, state)$ may form a sliceset. In this case, a second step consisting in combining slicesets so that the output consists in a single sliceset is performed. In this respect, it is sufficient to note that a boolean expression can always be restated in a disjunction of conjunction.

The algorithm consists in a breadth first search (function `BreadthFirst`), function `NextSliceSet` (written as `NSS` for short) which generates the next sliceSet, from a single current state and input message, and function `CondIsVerified` that checks if an atomic comparison holds given the current valuation map (the latter has been omitted from this paper out of space considerations). Function `BreadthFirst` takes as input the initial state of the automaton, as returned by the algorithm `BuildAutomaton` above, and an input trace. Note that if SliceSet, at some iteration, is equal to **true** (resp. **false**), its value at the next iteration, and all subsequent iterations will also be **true** (resp. **false**).

The proof correction has been omitted out of space consideration. The interested reader is referred to our companion tech report.

6 Pelota : Implementation and Experimental Results

Pelota takes as input an LTL-FO$^+$ property in text format, and an XML trace. The software begins by constructing the automaton in the manner described in the previous section. Pelota then performs a breadth-first search over the automaton. At each step the current run of dynamic states is evaluated and a three-valued truth value is assigned to current run. Pelota then performs manipulations to simplify the current run by applying logical identities. Previous work with LTL-FO$^+$ [8], showed that memory consumption was the limiting

```
Function NextSliceSet(AssignMap p, Transition tr, Message m)
    switch tr do
        case if(cond){tr1}else{tr2} do
            if CondIsVerified(p, cond, m) then
                return NextSliceSet(p, tr1, m)
            return NextSliceSet(p, tr2, m)
        end
        case tr1 ⍒ tr2 do
            return NextSliceSet(p, tr1, m) ∨ NextSliceSet(p, tr2, m)
        end
        case tr1 ⍓ tr2 do
            return NextSliceSet(p, tr1, m) ∧ NextSliceSet(p, tr2, m)
        end
        case ⨬_{x∈Dom(π)} tr do
            return ⋁_{x̄ ∈ Dom_m(π)} NextSliceSet(p.add(x, x̄), tr, m)
        end
        case ⨁_{x∈Dom(π)} tr do
            return ⋀_{x̄ ∈ Dom_m(π)} NextSliceSet(p.add(x, x̄), tr, m)
        end
        case nodeRef do
            if (nodeRef.label = ⊤) then
                return true
            if (nodeRef.label = ⊥) then
                return false
            else
                return {{(p, nodeRef)}}
        end
    end
```

factors in trace verification. The favorable results displayed in this section are partly attributable to Pelota's aggressive memory management.

6.1 Experimental Results

We tested Pelota using 5 properties. and compared it's execution time to that of BeepBeep [6], another monitor for LTL-FO$^+$ traces in each case.

Property 1 is a variant the property '*a is not a key*' that cannot be enforced using a register automaton, and verifies that for every value labelled with $'x'$, there exists subsequently in the trace a label $'y'$ with the same associated value. Conversely property 2, is a variant of the'*a is a key*' property which cannot be verified by pebble automaton, and checks that at least one value labelled by $'x'$ is also subsequently labelled by $'y'$.

Figure 4 shows the evaluation of Property 1 over a trace of length 500 000, containing random values between 0 and 10 labeled by $'x'$ and $'y'$, 2 per event,

as shown in Figure 3. As can be shown, Pelota's ability to maintain a narrow set of states, combined with its ability to process many events efficiently allow for a substantial speed-up in execution over BeepBeep.

```
<message>
  <id>0</id>
  <x>4</x>
  <y>8</y>
</message>
<message>
  <id>1</id>
  <x>5</x>
  <y>3</y>
</message>
```

Fig. 3: A fragment of a random number trace

As discussed above, Pelota runs in time $O(n * d)$, where n is the length of the trace and d is the number of states simultaneously visited by the automaton at runtime, while the previously used algorithm runs in $O(n^2)$. Property 1 gives a illustrative example of the manner by which Pelota's automata-based algorithm can improve the time and space complexity of the verification process. Each new value a labelled with $'x'$ that is encountered in the trace creates a new dynamic state, recording that this value has been seen. The automaton moves into this state, while simultaneously (universally) remaining in its original state, which allows it to proceed examining the input trace and create further states whenever new values of $'x'$ are seen. If a $'y'$ is later encountered in the trace with associated value a, this state will become equivalent to $\top$ and suppressed by Pelota. It follows that at runtime, Pelota maintains an automaton whose size is equal to the number of values of $'x'$ that have been seen so far, and for which no corresponding value of $'y'$ has yet been seen (plus one for the initial state). Observe that the size of the runtime environment only increases when a new value of $'x'$ is seen. The occurrence of a value which has previously been seen, but for which the corresponding $'y'$ value has not yet been observed, results in the in the creation of a duplicate state, which is immediately pruned by Pelota.

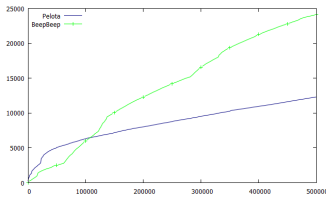

Fig. 4: Prop. 1

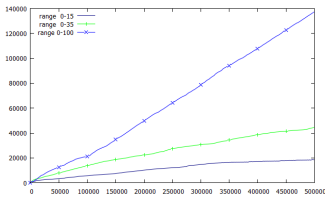

Fig. 5: Prop. 1 over 3 traces

As a consequence, the execution time is sensitive to the range of possible values seen in the trace. Figure 5 shows the execution time of Pelota on property 1, on three different traces of random numbers with values in the in the 0-15, 0-26 and 0-35 interval.As can be seen, execution time linearly increases with the number of possible values for $'x'$. This results from fact that if more different values of $'x'$ occur in the trace, then at runtime the automaton will find itself in a larger set of states, all of which must be evaluated for each event.

A second aspect of the input trace can contribute to Pelota's performance. When evaluating a trace, it frequently happens that the current event has no

bearing the evaluation of the property, because it does not contain any XPath whose label occurs in the formula. Such an event triggers the default case of the quantifiers, and can be processed quite efficiently by Pelota. This optimization is not possible with BeepBeep because it's algorithm pushes quantifiers to the bottom of the leaves of its evaluation tree, requiring a complete tree traversal on each event. A comparison of Pelota and BeepBeep's execution time for Property 1 over such a sparse trace is given in Figure 7. The trace comprised 500 000 events, each of contains a single value between 1 and 15, labelled with any one of 26 possible labels (including $'x'$ and $'y'$) with equal probability.

Figure 6 gives the result of testing property 2 on a similar trace. These results warrant some detailing. Early in the trace, a matching value is necessarily encountered. When this happens, Pelota's automata structure simplifies itself to a single state, the singleton $\top$. From then on, every event can be trivially processed since a good prefix has already been encountered. With a single state, event processing becomes linear in the size of the trace (which the monitor still reads). BeepBeep exhibits a similar behavior: once it reaches a verdict, the remainder of the trace is read and processed in time linear to its length. As a result, both monitors exhibit an execution time for this trace that is both linear, and much lower than for the proceeding one.

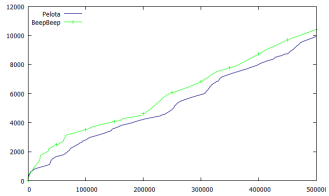

Fig. 6: Prop. 2

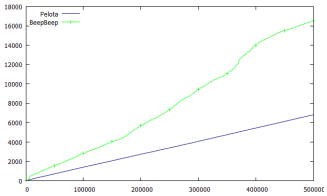

Fig. 7: Prop. 1 over a sparse trace

As a third example, we tested the property: $\mathbf{F}(\forall a \in message/x : ((a = 0) \wedge \mathbf{G}(\exists b \in message/x : (\mathbf{G}(b > 5)))))$. This property examines the trace to determine if it contains a monotonically increasing sequence of values labelled with $'x'$, disregarding all other-labelled values. It serves as model for any LTL property that looks for a specific pattern in the trace. Since it can never be satisfied, it avoids the behavior exhibited by Property 2 and serves as informative test on the behavior of the monitors. Figure 8 shows the result of verifying this property on a trace of 1 million events, each consisting of a single labelled value between 1-15, with 26 possible labels occurring with equal probability. As these results show, Pelota exhibits a substantial speed up over BeepBeep, and that this speed-up is especially marked as the size of the input trace increases. This is because BeepBeep's execution time is largely constrained by its memory consumption, which increases monotonically regardless of the content of the trace. On the other hand, Pelota's efficiency is proportional to the number of states the automaton finds itself in during a run, and this number is reset to 1 every time the pattern described by the property does not hold.

Finally, we tested the monitors on two real-life properties, taken from previous research [8] on the detection of behavior in assembly traces. The trace captures all assembly instruction performed during the execution of a program, with each line of these files corresponds to a single assembly-level instruction and detailing the registers, memory locations and values it manipulates.

Property 4 is a program comprehension property (property 3 from [8], *return address protection*) [2]. The property states that the return address on the stack is not modified before the function returns (no buffer overflow exploit). Property 5 is a security property, (property 4 from [8], *pointer subterfuge detection*). It states that the values written to memory through certain kinds of buffer to buffer copy cannot subsequently be used as pointers for memory accesses, thus aiding in the detection of a potentially malicious behavior. Experimental results for these two properties, on traces of length 500 000 are given in Figure 9. As can be shown, Pelota outperforms BeepBeep in both cases.

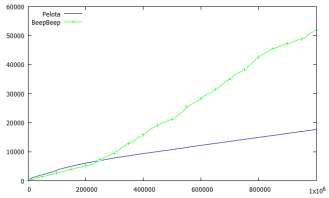

Fig. 8: Prop. 3

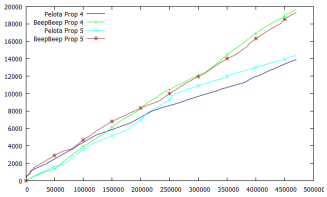

Fig. 9: Prop. 4-5

In summary, while BeepBeep's execution time is proportional to the size of the size of the property under consideration, and especially to the number of quantifiers contained in the latter, Pelota's execution time is largely dependant on the size of the state-space visited by the automaton at runtime, a value that depends both on the property as well as on the underlying execution trace. Since the number of states occurring at runtime is at least partly predictable through a reasoned analysis of the property and the trace, an astute user of LTL-FO$^+$ can make an informed conjecture about which monitor, between Pelota and BeepBeep, will likely exhibit optimal performance in his particular case.

7 Conclusion

In this paper, we presented an automata representation for LTL-FO$^+$ properties and gave algorithms for both automata construction and verification. Finally, we presented Pelota, a new monitor for LTL-FO$^+$ properties. Pelota exhibits a highly promising results due to its effective memory management, which translates into lower execution time. The expressive power of LTL-FO$^+$ permits users to state fine-grained properties, drawing on the specific values present in the trace. Furthermore, the Pelota monitor is sufficiently powerful to verify these properties on real traces in tractable time, and with a light memory footprint.

[2] Slight semantic-preserving modifications were performed on the properties since Pelota does not admit the $\rightarrow$ amd $\neg$ operators, while BeepBeep disallows $\neq$.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012)
2. Bauer, A., Küster, J.C., Vegliach, G.: From Propositional to First-Order Monitoring, pp. 59–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), https://doi.org/10.1007/978-3-642-40787-1_4
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
4. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs. In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM). pp. 33–37. IEEE Computer Society (Nov 2009)
5. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata (2003)
6. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012), http://dx.doi.org/10.1109/TSC.2011.10
7. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329 – 363 (1994), http://www.sciencedirect.com/science/article/pii/0304397594902429
8. Khoury, R., Hallé, S., Waldmann, O.: Execution trace analysis using LTL-FO+. In: 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (IsoLa 16), Corfu, Greece (2016)
9. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic 5(3), 403–435 (Jul 2004), http://doi.acm.org/10.1145/1013560.1013562
10. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4207, pp. 41–57. Springer (2006), http://dx.doi.org/10.1007/11874683_3
11. Syropoulos, A.: Mathematics of Multisets, pp. 347–358. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
12. Vardi, M., Wolper, P.: Reasoning about infinite computations. Inf. Comput. 115(1), 1–37 (Nov 1994), http://dx.doi.org/10.1006/inco.1994.1092

A Appendix

We include in this appendix complete version of proofs and theorems that have been elided from the paper out of space considerations as a courtesy for the reviewers. While the paper itself is self-contained and complete, the materiel in this appendix may provide precisions.

We begin in Section A.1with some preliminary notions related to multiset theory, which will be of use in the proof. The main theorem of this paper is also stated in this initial section. Section A.2 provides a supporting lemma related to multiset manipulation. Section A.3 provides the proof of correction of the Verification algorithm stated in Section 5. Section A.4 includes pseudocode for an auxiliary algorithm referred to in section Section 5, which was elided from the main paper out of space considerations.

A.1 Peliminaries

In what follows, we represent a slice, which is a nonempty collection of pairs, as a multiset of pairs. Analogously, we represent a sliceset, which is a nonempty collection of slices, as a multiset of slices. A *multiset* is a generalized set that allows multiple occurrences of any element (see [11] for reference). Multisets naturally support duplicate run trees (i.e. repeated slices in a sliceset) and duplicate branches (i.e. repeated pairs in a slice). Thus, they accurately describe the execution of the alternating automata. Although it is obvious that duplicates can be ignored for memory optimization (as was the case in [5] where sets are favored over multisets or the software Pelota presented in section 6), it is easier, for our correctness proof to keep them.

We mathematically define $\vee$ and $\wedge$ over sliceSets w.r.t. operator $\uplus$ that denotes the *sum* of two multisets. For any multisets A and B, the sum $A \uplus B$ is a multiset that contains every occurrence of every element of A and B. In other words, the number of occurrences of any element in $A \uplus B$ is equal to the sum of number of occurrences of this element in A and in B. This definition of $\vee$ and $\wedge$ is consistent with our intuitive understanding of sliceset manipulations: for instance sliceset $A \vee B$ equals $\top$ iff A *or* B does, and that it equals $\bot$ iff A *and* B do. Likewise, the semantics of $\wedge$, ensure that a sliceset $A \wedge B$ induces $\top$ iff A *and* B do, and it induces $\bot$ iff A *or* B does.

Definition 10 *Let* $A = \{A_1, \ldots, A_m\}$ *and* $B = \{B_1, \ldots, B_n\}$ *be slicesets:*

- $A \vee B = A \uplus B = \{A_1, \ldots, A_m, B_1, \ldots, B_n\}$;
- $A \wedge B = \{A_i \uplus B_j \mid 1 \leq i \leq m, 1 \leq j \leq n\}$.

Next, we introduce the function `BreadthFirst(`p, s_0^φ, $\overline{m}$`)`, a straight copy of `BreadthFirst(`s_0^φ, $\overline{m}$`)` with the only difference that SliceSet is initially assigned to $\{\{(p, s_0^\varphi)\}\}$ instead of $\{\{([], s_0^\varphi)\}\}$, where $p\colon V \to D$ is a partial function that assigns the free variables in φ and s_0^φ) is the initial state, representing the entire property (φ). This notation will allow us to reason about the behavior of the

algorithm during intermediate phases in which it manipulates subformulas of φ containing free variables. We say that (p, φ) is a well-formed formula iff every free variable occurring in φ is assigned a value in p. Following the semantics in Section 2, for any nonempty finite trace $\overline{m}$ and any well-formed formula (p, φ), $[\overline{m} \models (p, \varphi)]$ is equal to $[\overline{m} \models \varphi(p)]$. In this context, the correctness of the algorithm presented in section 5 can be derived from the following theorem, when $p = []$ and φ is devoid of free variables:

Theorem 1 *Let (p, φ) be a well-formed formula, and let $\overline{m} = m_1, \ldots, m_n$ be a finite message trace. The output of* `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` *is equal to $\top$ (resp.$\bot$) iff $[\overline{m} \models (p, \varphi)] = \top$ (resp. $[\overline{m} \models (p, \varphi)] = \bot$).*

Observe that if $[\overline{m} \models (p, \varphi)]$ evaluates to '?', the breath first algorithm returns a mustiest of multisets of values, which it require for the evaluation of the formula to proceed to the next message of the sequence. However, if the final message of the input sequence has been reached, any value returned by `BreadthFirst()` different from $\top$ or $\bot$ can be trivially converted to '?', allowing the verdict of the verification algorithm to correspond to the semantics of LTL-FO$^+$ in all cases. The remainder of this section is devoted to the proof of Theorem 1. We proceed by strong induction on the size of formulas. Recall that the size of a formula φ, denoted by $|\varphi|$, is the height of its tree decomposition. The recursive nature of the algorithms favors this decision: the size of the input of `BuildAutomaton` is reduced by 1 size unit with each recursive call.

A.2 Lemma on SliceSets

In what follows,We omit the subscripted index of the slice and sliceSet when it is clear from context. The following notation identifies specific SliceSets (or *terms*) computed by `BreadthFirst`:

Definition 11 *Let (p, φ) be a well-formed formula, and let $\overline{m} = m_1, \ldots, m_n$ ($n \geq 0$) be a finite message trace. The sequence $(SliceSet_i(p, \varphi, \overline{m}))_{0 \leq i \leq n}$ of slicesets and truth values (***true***, **false***) is recursively defined as follows:*

$$SliceSet_0(p, \varphi, \overline{m}) = \{\{(p, s_0^\varphi)\}\};$$

$$SliceSet_i(p, \varphi, \overline{m}) = \bigvee_{\substack{slice \in \\ SliceSet_{i-1}(p,\varphi,\overline{m})}} \bigwedge_{\substack{(q,state) \in \\ slice}} \texttt{NSS(}q\text{, }state.tr\text{, }m_i\texttt{)}.$$

The following lemma describes more formally the relationship between SliceSets and the `BreadthFirst` algorithm.

Lemma 1 *1 Let μ and η be formulas. Let $p\colon V \setminus \{x\} \to D$ be a partial function, and for all $\overline{x} \in D$, denote $p \cup \{(x, \overline{x})\}$ by the shorthand $p_{\overline{x}}$. If the expressions $(p_{\overline{x}}, \eta)$, (p, η) and (p, μ) are well-formed, then for any path π, any finite message trace $\overline{m} = m_1, \ldots, m_n$ $(n \geq 1)$, and for all $1 \leq i \leq n$:*

$$(\exists)\quad SliceSet_i(p, \exists_\pi x : \eta, \overline{m}) = \bigvee\nolimits_{\overline{x} \in Dom_{m_1}(\pi)} SliceSet_i(p_{\overline{x}}, \eta(x/\overline{x}), \overline{m})$$

$$(\forall)\quad SliceSet_i(p, \forall_\pi x : \eta, \overline{m}) = \bigwedge\nolimits_{\overline{x} \in Dom_{m_1}(\pi)} SliceSet_i(p_{\overline{x}}, \eta(x/\overline{x}), \overline{m}).$$

$$(\vee)\quad SliceSet_i(p, \mu \vee \eta, \overline{m}) = SliceSet_i(p, \mu, \overline{m}) \vee SliceSet_i(p, \eta, \overline{m})$$

$$(\wedge)\quad SliceSet_i(p, \mu \wedge \eta, \overline{m}) = SliceSet_i(p, \mu, \overline{m}) \wedge SliceSet_i(p, \eta, \overline{m})$$

$$(\mathbf{X})\quad SliceSet_i(p, \mathbf{X}\ \eta, \overline{m}) = SliceSet_{i-1}(p, \eta, \overline{m}^2)$$

$$(\mathbf{F})\quad SliceSet_i(p, \mathbf{F}\ \eta, \overline{m}) = \bigvee_{j=1}^{i} SliceSet_{i-j+1}(p, \eta, \overline{m}^j) \vee \{\{(p, s_0^{\mathbf{F}\,\eta})\}\}$$

$$(\mathbf{G})\quad SliceSet_i(p, \mathbf{G}\ \eta, \overline{m}) = \bigwedge_{j=1}^{i} SliceSet_{i-j+1}(p, \eta, \overline{m}^j) \wedge \{\{(p, s_0^{\mathbf{G}\,\eta})\}\}$$

$$(\mathbf{U})\quad SliceSet_i(p, \mu\ \mathbf{U}\ \eta, \overline{m}) =$$
$$\bigvee_{j=1}^{i} \Big(\bigwedge_{k=1}^{j-1} SliceSet_{i-k+1}(p, \mu, \overline{m}^k) \wedge SliceSet_{i-j+1}(p, \eta, \overline{m}^j) \Big)$$
$$\vee \Big(\bigwedge_{k=1}^{i} SliceSet_{i-k+1}(p, \mu, \overline{m}^k) \wedge \{\{(p, s_0^{\mu\,\mathbf{U}\,\eta})\}\} \Big)$$

$$(\mathbf{R})\quad SliceSet_i(p, \mu\,\mathbf{R}\,\eta, \overline{m}) =$$
$$\bigvee_{j=1}^{i} \Big(\bigwedge_{k=1}^{j} SliceSet_{i-k+1}(p, \eta, \overline{m}^k) \wedge SliceSet_{i-j+1}(p, \mu, \overline{m}^j) \Big)$$
$$\vee \Big(\bigwedge_{k=1}^{i} SliceSet_{i-k+1}(p, \eta, \overline{m}^k) \wedge \{\{(p, s_0^{\mu\mathbf{R}\eta})\}\} \Big).$$

A.3 Proof of correctness the main Theorem

In this section, we give a more complete and detailed proof of the main theorem of this paper.

We proceed by induction on the depth of the property.

Base case: Theorem 1 holds for any nonempty finite trace $\overline{m}$ and any well-formed couple (p, φ) when $|\varphi| = 0$.

Let $\overline{m} = m_0, \ldots, m_n$ be some finite trace, and let (p, φ) represent a well-formed formula where $|\varphi| = 0$. By the definition of $|\cdot|$, φ is an equality $x = y$, inequality $x \neq y$, or some other boolean comparison where x and y are constants

or (free) variables. The couple (p, φ) is well-formed, so if x (or y) is a variable p assigns it a value.

The transition formula of the state s_0^{φ} is given by `BuildTransition(`φ`)` and is equal to if(φ){$node\top$}else{$node\bot$}. Therefore:

$$\begin{aligned}
SliceSet_0(p, \varphi, \overline{m}) &= \texttt{NSS(}p\texttt{, }s_0^{\varphi}\texttt{.}tr\texttt{, }m_1\texttt{)} \\
&= \texttt{NSS(}p\texttt{, if(}\varphi\texttt{)\{}node\top\texttt{\}else\{}node\bot\texttt{\}, }m_1\texttt{)} \\
&= \begin{cases} \texttt{NSS(}p\texttt{, }node\top\texttt{, }m_1\texttt{)} & \text{if } \texttt{CondIsVerified(}p\texttt{, }\varphi\texttt{, }m_1\texttt{)} \\ \texttt{NSS(}p\texttt{, }node\bot\texttt{, }m_1\texttt{)} & \text{otherwise;} \end{cases} \\
&= \begin{cases} \textbf{true} & \text{if } \texttt{CondIsVerified(}p\texttt{, }\varphi\texttt{, }m_1\texttt{)} \\ \textbf{false} & \text{otherwise.} \end{cases}
\end{aligned}$$

The truth values **true** and **false** are final once they are obtained, so this equation also holds for $SliceSet_n(p, \varphi, \overline{m})$ and leads to the following output:

$$\texttt{BreadthFirst(}p\texttt{, }s_0^{\varphi}\texttt{, }\overline{m}\texttt{)} = \begin{cases} \top & \text{if } \texttt{CondIsVerified(}p\texttt{, }\varphi\texttt{, }m_1\texttt{)} \\ \bot & \text{otherwise.} \end{cases}$$

If φ is an equality (resp. inequality), the condition `CondIsVerified(`p`, `φ`, `m_1`)` evaluates to $p(x) = p(y)$ (resp. $p(x) \neq p(y)$), which is equal to $\varphi(p)$. Hence, the final output is equal to $[\overline{m} \models \varphi(p)] = [\overline{m} \models (p, \varphi)]$.

Induction step: For some integer $d \geq 0$, if Theorem 1 holds for any nonempty finite trace $\overline{m}$ and any well-formed couple (p, φ) when $|\varphi| \leq d$, then it also holds when $|\varphi| = d + 1$.

Let $\overline{m}$ be some finite non-empty trace, and let (p, φ) be a well-formed formula. Suppose $|\varphi| = d + 1$. Since $d \geq 0$, φ cannot be an equality nor an inequality; rather, it must take one of nine possible forms. If φ is of the form $\exists_\pi x : \eta$, $\forall_\pi x : \eta$, $\mathbf{X}\ \eta$, $\mathbf{F}\ \eta$ or $\mathbf{G}\ \eta$, then the induction hypothesis covers η because $|\eta| = d$. If φ is of the form $\mu \vee \eta$, $\mu \wedge \eta$, $\mu\ \mathbf{U}\ \eta$ or $\mu\, \mathbf{R}\, \eta$, then it covers both μ and η because $\max\{|\mu|, |\eta|\} = d$.

For every form of φ, there exists an equation from Lemma 1 that, for all $0 \leq i \leq n$, links $SliceSet_i(p, \varphi, \overline{m})$ to slicesets related to η (and μ). These equations, combined with the induction hypothesis, allow us to prove that `BreadthFirst(`p`, `s_0^{φ}`, `$\overline{m}$`)` returns $[\overline{m} \models (p, \varphi)]$. The details depend on φ, but it is always sufficient to show that `BreadthFirst(`p`, `s_0^{φ}`, `$\overline{m}$`)` returns $\top$ (resp. $\bot$) iff $[\overline{m} \models (p, \varphi)] = \top$ (resp. $\bot$) since these two equivalences imply that `BreadthFirst(`p`, `s_0^{φ}`, `$\overline{m}$`)` also returns ? iff $[\overline{m} \models (p, \varphi)] = ?$.

Case $\boldsymbol{\varphi = \mu \vee \eta}$ Lemma 1 holds for any $1 \leq i \leq n$, so if $i = n$, we have

$$SliceSet_n(p, \varphi, \overline{m}) = SliceSet_n(p, \mu, \overline{m}) \vee SliceSet_n(p, \eta, \overline{m}).$$

We previously observed that $|\mu| \leq d$ and $|\eta| \leq d$. Since (p, φ) is well-formed (p, μ) and (p, η) are also well-formed and are covered by the induction hypothesis. It follows from this hypothesis, the equation above, and the Definition of LTL-FO$^+$ given in Definition 4, that

$$\begin{aligned}
&SliceSet_n(p, \varphi, \overline{m}) = \textbf{true} \\
&\Leftrightarrow SliceSet_n(p, \mu, \overline{m}) = \textbf{true} \texttt{ or } SliceSet_n(p, \eta, \overline{m}) = \textbf{true} \\
&\Leftrightarrow [\overline{m} \models (p, \mu)] = \top \texttt{ or } [\overline{m} \models (p, \eta)] = \top && \text{(Induction)} \\
&\Leftrightarrow [\overline{m} \models (p, \varphi)] = \top. && \text{(Definition 4)}
\end{aligned}$$

As a result, `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` returns $\top$ iff $[\overline{m} \models (p, \varphi)] = \top$. A similar reasoning shows that `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` returns $\bot$ iff $[\overline{m} \models (p, \varphi)] = \bot$. Hence, `BreadthFirst(`$p$, s_0^φ, $\overline{m}$`)` returns the correct value.

Case $\boldsymbol{\varphi = \mu \wedge \eta}$ This case is analogous to the previous one. By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = SliceSet_n(p, \mu, \overline{m}) \wedge SliceSet_n(p, \eta, \overline{m}).$$

A straightforward adaptation of the previous case shows that `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` returns $\top$ (resp. $\bot$) iff $[\overline{m} \models (p, \varphi)] = \top$ (resp. $\bot$).

Case $\boldsymbol{\varphi = \exists_{\pi} x : \eta}$ By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = \bigvee_{\overline{x} \in Dom_{m_1}(\pi)} SliceSet_n(p_{\overline{x}}, \eta(x/\overline{x}), \overline{m}).$$

We already established that $|\eta| \leq d$, and that the couple $(p_{\overline{x}}, \eta(x/\overline{x}))$ is well-formed. Therefore, the induction hypothesis covers $(p_{\overline{x}}, \eta(x/\overline{x}))$.

If $Dom_{m_1}(\pi)$ is empty, the disjunction in the equation above is conventionally set to **false**, so $SliceSet_n(p, \varphi, \overline{m}) = \textbf{false}$. Otherwise, if $Dom_{m_1}(\pi)$ is not empty,

$$\begin{aligned}
&SliceSet_n(p, \varphi, \overline{m}) = \textbf{false} \\
&\Leftrightarrow \forall\ \overline{x} \in Dom_{m_1}(\pi) : SliceSet_n(p_{\overline{x}}, \eta(x/\overline{x}), \overline{m}) = \textbf{false} \\
&\Leftrightarrow \forall\ \overline{x} \in Dom_{m_1}(\pi) : [\overline{m} \models (p_{\overline{x}}, \eta(x/\overline{x}))] = \bot. && \text{(Induction)}
\end{aligned}$$

When we unify both scenarios, we get that $SliceSet_n(p, \varphi, \overline{m}) = \textbf{false}$ iff $[\overline{m} \models (p, \varphi)] = \bot$ (Definition 4). In other words, `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` returns $\bot$ iff $[\overline{m} \models (p, \varphi)] = \bot$.

Similarly, $SliceSet_n(p, \varphi, \overline{m}) = \textbf{true}$ iff there exists some $\overline{x} \in Dom_{m_1}(\pi)$ for which $SliceSet_n(p_{\overline{x}}, \eta(\pi/x), \overline{m}) = \textbf{true}$ ($Dom_{m_1}(\pi) \neq \emptyset$ is implied). It follows from the induction hypothesis and Definition 4 that `BreadthFirst(`p, s_0^φ, $\overline{m}$`)` returns $\top$ iff $[\overline{m} \models (p, \varphi)] = \top$. Hence, `BreadthFirst(`$p$, s_0^φ, $\overline{m}$`)` returns the correct value.

Case $\boldsymbol{\varphi = \forall_{\pi} x : \eta}$ This case is analogous to the previous one. By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = \bigwedge\nolimits_{\overline{x} \in Dom_{m_1}(\pi)} SliceSet_n(p_{\overline{x}}, \eta(x/\overline{x}), \overline{m}).$$

If $Dom_{m_1}(\pi)$ is empty, the conjunction in the equation is conventionally set to **true**, so $SliceSet_n(p, \varphi, \overline{m}) = \mathbf{true}$ in this instance. As expected, we can show, by adapting the argument of the previous case, that `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` returns $\top$ (resp. $\bot$) iff $[\overline{m} \models (p, \varphi)] = \top$ (resp. $\bot$). Hence, `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` returns the correct value.

Case $\boldsymbol{\varphi = \mathbf{X}\ \eta}$ By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = SliceSet_{n-1}(p, \eta, \overline{m}^2).$$

If $n = 1$, the last sliceset computed by `BreadthFirst(`p, φ, $\overline{m}$`)` is $\{\{(p, s_{\eta}^0)\}\}$. Since a verdict could not be assigned at that point, it evaluates to ?, which is also the value of $[\overline{m} \models (p, \varphi)]$ (Definition 4). Otherwise, since (p, η) is well-formed and $|\eta| \leq d$, we can use the induction hypothesis on $SliceSet_{n-1}(p, \eta, \overline{m}^2)$. Doing so results in the desired equivalences:

$$SliceSet_n(p, \varphi, \overline{m}) = SliceSet_{n-1}(p, \eta, \overline{m}^2) = \mathbf{true} \Leftrightarrow [\overline{m}^2 \models (p, \eta)] = \top;$$

$$SliceSet_n(p, \varphi, \overline{m}) = SliceSet_{n-1}(p, \eta, \overline{m}^2) = \mathbf{false} \Leftrightarrow [\overline{m}^2 \models (p, \eta)] = \bot.$$

Again, the output of `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` is $[\overline{m} \models (p, \varphi)]$.

Case $\boldsymbol{\varphi = \mathbf{F}\ \eta}$ By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = \bigvee_{j=1}^{n} SliceSet_{n-j+1}(p, \eta, \overline{m}^j) \ \vee\ \{\{(p, s_0^{\varphi})\}\}.$$

The sliceset $\{\{(p, s_0^{\varphi})\}\}$ evaluates to ?. As a result, the disjunction $SliceSet_n(p, \varphi, \overline{m})$ can never evaluate to **false**,and `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` will never return $\bot$. However, it returns $\top$ iff $SliceSet_{n-j+1}(p, \eta, \overline{m}^j) = \mathbf{true}$ for some $1 \leq j \leq n$.

As with the previous case, the couple (p, η) is well-formed and $|\eta| \leq d$. Also, for any $1 \leq j \leq n$, $SliceSet_{n-j+1}(p, \eta, \overline{m}^j)$ is the last term computed by `BreadthFirst(`p, s_{η}^0, $\overline{m}^j$`)`. Therefore, we can use the induction hypothesis on this term and conclude that:

$$\begin{aligned}
&SliceSet_n(p, \varphi, \overline{m}) = \mathbf{true} \\
&\Leftrightarrow \exists\ 1 \leq j \leq n : SliceSet_{n-j+1}(p, \eta, \overline{m}^j) = \mathbf{true} \\
&\Leftrightarrow \exists\ 1 \leq j \leq n : [\overline{m}^j \models (p, \eta)] = \top && \text{(Induction)} \\
&\Leftrightarrow [\overline{m} \models (p, \varphi)] = \top. && \text{(Definition 4)}
\end{aligned}$$

In summary, `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` never returns $\bot$ just like $[\overline{m} \models (p, \varphi)]$ is never equal to $\bot$, and it returns $\top$ iff $[\overline{m} \models (p, \varphi)] = \top$.

Case $\boldsymbol{\varphi = \mathbf{G}\ \eta}$ This case is analogous to the previous one. By Lemma 1, if $i = n$, then

$$SliceSet_n(p, \varphi, \overline{m}) = \bigwedge_{j=1}^{n} SliceSet_{n-j+1}(p, \eta, \overline{m}^j) \wedge \{\{(p, s_0^{\varphi})\}\}.$$

Since $\{\{(p, s_0^{\varphi})\}\}$ evaluates to ?, the conjunction $SliceSet_n(p, s_0^{\varphi}, \overline{m})$ can never be equal to **true**. However, it is equal to **false** if, for some $1 \leq j \leq n$, $SliceSet_{n-j+1}(p, \eta, \overline{m}^j) = \textbf{false}$. The remainder of the argument follows exactly as that of the previous case. `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` never returns $\top$, but $\bot$ is returned iff $[\overline{m} \models (p, \varphi)] = \bot$.

Case $\boldsymbol{\varphi = \mu\ \mathbf{U}\ \eta}$ By Lemma 1, if $i = n$, then $SliceSet_n(p, \varphi, \overline{m})$ is equal to

$$\bigvee_{j=1}^{n} \Big(\bigwedge_{k=1}^{j-1} SliceSet_{n-k+1}(p, \mu, \overline{m}^k) \wedge SliceSet_{n-j+1}(p, \eta, \overline{m}^j) \Big)$$
$$\vee \Big(\bigwedge_{k=1}^{n} SliceSet_{n-k+1}(p, \mu, \overline{m}^k) \wedge \quad \{\{(p, s_0^{\varphi})\}\} \quad \Big).$$

The couples (p, μ) and (p, η), just like in the case $\varphi = \mu \vee \eta$, are well-formed and the size $|\cdot|$ of their formulas is less than or equal to d. Since the equation is a disjunction of conjunctions, $SliceSet_n(p, \varphi, \overline{m})$ is **true** iff one conjunction is **true**. The sliceset $\{\{(p, s_0^{\varphi})\}\}$ causes the last one to fail, so the results depends on the first n conjunctions:

$$SliceSet_n(p, \varphi, \overline{m}) = \textbf{true}$$
$$\Leftrightarrow \begin{cases} \exists\ 1 \leq j \leq n : \bigwedge_{k=1}^{j-1} SliceSet_{n-k+1}(p, \mu, \overline{m}^k) \\ \qquad\qquad \wedge\ SliceSet_{n-j+1}(p, \eta, \overline{m}^j) \end{cases} = \textbf{true}$$
$$\Leftrightarrow \begin{cases} \exists\ 1 \leq j \leq n : SliceSet_{n-j+1}(p, \eta, \overline{m}^j) = \textbf{true} \\ \forall\ 1 \leq k < j : SliceSet_{n-k+1}(p, \mu, \overline{m}^k) = \textbf{true} \end{cases}$$
$$\Leftrightarrow \begin{cases} \exists\ 1 \leq j \leq n : [\overline{m}^j \models (p, \eta)] = \top \\ \forall\ 1 \leq k < j : [\overline{m}^k \models (p, \mu)] = \top \end{cases} \qquad \text{(Induction)}$$
$$\Leftrightarrow [\overline{m} \models (p, \varphi)] = \top. \qquad \text{(Definition 4)}$$

Thus `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` returns $\top$ iff $[\overline{m} \models (p, \varphi)] = \top$.

The following analogous reasoning reasoning can be used to show that `BreadthFirst(`p, s_0^{φ}, $\overline{m}$`)` returns $\bot$ iff $[\overline{m} \models (p, \varphi)] = \bot$. We omit it here out of space considerations:

A disjunction like $SliceSet_n(p, \varphi, \overline{m})$ is equal to **false** iff all its conjunctions are **false**. In particular must admit a **false** term. Since $\{\{(p, s_0^{\varphi})\}\}$ evaluates to ?, $SliceSet_n(p, \varphi, \overline{m})$ can only evaluate to **false** i $SliceSet_{n-k+1}(p, \mu, \overline{m}^k)$ fails for some $1 \leq k \leq n$. Let k^* denote the smallest such value.

Let us now inspect every conjunction indexed by j where $j \leq k^*$. Because $k \leq j - 1 < k^*$, any $SliceSet_{n-k+1}(p, \mu, \overline{m}^k)$ composing them is not **false** due to the minimality of k^*. However, the conjunctions themselves still fail, so $SliceSet_{n-j+1}(p, \eta, \overline{m}^j)$ must be **false** for all $j \leq k^*$. In summary:

$$
\begin{aligned}
& SliceSet_n(p, \varphi, \overline{m}) = \textbf{false} \\
\Leftrightarrow & \begin{cases} \exists\, 1 \leq k^* \leq n : SliceSet_{n-k+1}(p, \mu, \overline{m}^k) = \textbf{false} \\ \forall\, 1 \leq j \leq k^* : SliceSet_{n-j+1}(p, \eta, \overline{m}^j) = \textbf{false} \end{cases} \\
\Leftrightarrow & \begin{cases} \exists\, 1 \leq k^* \leq n : [\overline{m}^k \models (p, \mu)] = \bot \\ \forall\, 1 \leq j \leq k^* : [\overline{m}^j \models (p, \eta)] = \bot \end{cases} && \text{(Induction)} \\
\Leftrightarrow & [\overline{m} \models (p, \varphi)] = \bot. && \text{(Definition 4)}
\end{aligned}
$$

We showed that `BreadthFirst(`p`,` s_0^φ`,` $\overline{m}$`)` returns $\bot$ iff $[\overline{m} \models (p, \varphi)] = \bot$. Hence, `BreadthFirst(`$p$`,` s_0^φ`,` $\overline{m}$`)` returns the correct value.

Case $\boldsymbol{\varphi = \mu\, \mathbf{R}\, \eta}$ This final case is analogous to the previous one. By Lemma 1, if $i = n$, then $SliceSet_n(p, \varphi, \overline{m})$ is equal to

$$
\begin{aligned}
& \bigvee_{j=1}^{n} \Big(\bigwedge_{k=1}^{j} SliceSet_{n-k+1}(p, \eta, \overline{m}^k) \wedge SliceSet_{n-j+1}(p, \mu, \overline{m}^j) \Big) \\
& \vee \Big(\bigwedge_{k=1}^{n} SliceSet_{n-k+1}(p, \eta, \overline{m}^k) \wedge \qquad \{\{(p, s_0^\varphi)\}\} \qquad \Big).
\end{aligned}
$$

This equation is almost identical to the one for $\varphi = \mu\ \mathbf{U}\ \eta$, and the supporting argument requires little adaptation. `BreadthFirst(`p`,` s_0^φ`,` $\overline{m}$`)` can be shown to return $\top$ (resp. $\bot$) iff $[\overline{m} \models (p, \varphi)] = \top$ (resp. $\bot$).

A.4 Function CondIsVerified

Function `CondIsVerified` consults the valuation map and\or the current message and determines if a conditional expression holds. Boolean equality can easily be generalized to other connectives. Pelota currently supports equality and inequality between integers and strings as well as $'<'$ comparison between integers. We omitted the pseudocode of this function from the main paper out of space considerations, and due to its simplicity, but include it here for completeness.

```
Function CondIsVerified(AssignMap p, Condition c, Message m)
    switch c do
        case x = y do
            return p(x) = p(y)
        end
        case x ≠ y do
            return p(x) ≠ p(y)
        end
    end
```