EAGLE does Space Efficient LTL Monitoring

Howard Barringer*[1], Allen Goldberg[2], Klaus Havelund[2] and Koushik Sen**[3]

[1] University of Manchester, England
[2] Kestrel Technology, NASA Ames Research Center, USA
[3] University of Illinois, Urbana Champaign, USA

Abstract. We briefly present a rule-based framework, called EAGLE, that has been shown to be capable of defining and implementing finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, real-time and metric temporal logics (MTL), interval logics, forms of quantified temporal logics, and so on. In this paper we show how EAGLE can perform linear temporal logic (LTL) monitoring in an efficient way. For an initial formula of size m, we establish upper bounds of $O(m^2 2^m \log m)$ and $O(m^4 2^{2m} \log^2 m)$ for the space and time complexity, respectively, of single step evaluation over an input trace. EAGLE has been successfully used, in both LTL and metric LTL forms, to test a real-time controller of an experimental NASA planetary rover.

1 Introduction

Linear temporal logic (LTL) [15] is now widely used for expressing properties of concurrent and reactive systems. Associated, production quality, verification tools have been developed, most notably based on model-checking technology, and have enjoyed much success when applied to relatively small-scale models. Tremendous advances have been made in combating the combinatoric state space explosion inherent with data and concurrency in model checking, however, there remain serious limitations for its application to full-scale models and to software. This has encouraged a shift in the way model checking techniques are being applied, from full state space coverage to bounded use for sophisticated testing, or debugging, and from static application to dynamic, or runtime, application. Our work on EAGLE concerns this latter direction.

In runtime verification a software component, an observer, monitors the execution of a program and checks its conformity with a requirement specification. Runtime verification can be applied to evaluate automatically test runs, either on-line or off-line, analyzing stored execution traces; or it can be used on-line during operation. Several runtime verification systems have been developed, of which some were presented at three recent international workshops on runtime verification [1]. The commercial tool Temporal Rover (TR) [5, 6] supports a fixed future and past time LTL, with the possibility of specifying real-time and data constraints (time-series) as annotations on the temporal operators; its implementation is based on alternating automata. Algorithms using alternating automata to monitor LTL properties are also proposed in [8], and a specialized LTL collecting statistics along the execution trace is described in [7]. The

* This author is most grateful to RIACS/USRA and to the UK's EPSRC under grant GR/S40435/01 for the partial support provided to conduct this research.
** This author is grateful for the support received from RIACS to undertake this research while participating in the Summer Student Research Program at the NASA Ames Research Center.

MAC logic [14] is a form of past-time LTL with operators inspired by interval logics and which models real-time via explicit clock variables. A logic based on extended regular expressions [16] has also been proposed and is argued to be more succinct for certain properties. The logic described in [13] is a sophisticated interval logic, argued to be more user-friendly than plain LTL. Our own previous work includes the development of several algorithms, such as generating dynamic programming algorithms for past time logic [11], using a rewriting system for monitoring future-time logic [10], or generating Büchi automata inspired algorithms adapted to finite trace LTL [9].

This wide variety of logics caused us to search for a compact but general framework for defining monitoring logics, which would be powerful enough to capture essentially all of the above described logics, and more. Much influenced by our earlier work on executable temporal logic METATEM, see for example [3], the logic EAGLE was the result. In [4], we showed the richness and expressivity of EAGLE, described an algorithm to synthesize monitors for EAGLE and commented on an implementation of the framework in Java and some initial experiments. However, we found that the efficiency and complexity analysis of the general EAGLE monitoring algorithm is difficult and can be shown to be dependent on both the length of the trace and the size of the initial formula in the worst case. In this paper, we thus investigate the complexity and efficiency of the monitoring algorithm for the special case of LTL containing a fixed number of past and future time temporal operators embedded as rules in EAGLE. We outline a space efficient implementation of the monitoring algorithm and prove that its space and time complexity is exponential in the size of the formula and which is independent of the length of the trace for single step evaluation. This makes it very efficient in terms of space as we do not store the trace either explicitly or implicitly. To our knowledge this is the first work on general LTL monitoring, with fully mixed past and future operators, that avoids the use of an automaton. Similar work was done in a rewriting framework for the case of future time LTL in [10]; however, there the complexity of the monitoring was not clear as it was dependent on the strategy used by the rewrite engine for rewriting. The work in [11] addresses a monitoring algorithm for past time LTL only.

The paper is structured as follows. Section 2 gives a formal definition of LTL on finite traces, introduces our logic framework EAGLE and then specializes it to LTL. In section 3 we discuss the monitoring algorithm and calculus with an illustrative example. This underlies our implementation for the special case of LTL, which is briefly described along with complexity and initial experimentation in section 4.

2 Linear Temporal Logic and the logic EAGLE

In this section, we first define linear temporal logic (LTL) for finite trace monitoring and then introduce our general purpose finite trace monitoring logic, EAGLE [4]. EAGLE offers a succinct but powerful set of primitives, essentially supporting recursive parameterized equations, with a minimal/maximal fix-point semantics together with three temporal operators: next-time, previous-time, and concatenation. The parametrization of rules supports reasoning about data values as well as the embedding of real-time, metric and statistical temporal logics; for examples of such, see [4]. In Section 2.2 we motivate the fundamental concepts of EAGLE through some simple examples drawn from LTL before presenting its formal definition. Finally, in Section 2.3 we present a full embedding of LTL in EAGLE and establish its correctness.

2.1 Propositional Linear Temporal Logic

In specification and verification contexts, LTL is usually defined over infinite sequences of states, each sequence corresponding to a "complete" infinite, or non-terminating, computation. For run-time verification purposes, properties are checked on finite traces. Thus our definition of LTL differs from the usual for the boundary cases. We assume the usual collection of temporal operators for reasoning over the past, present and future. Furthermore, we place no restrictions on the nesting or intermixing of past and future operators. The syntax of LTL is as follows.

$$\begin{array}{lll} F ::= a \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \neg F \mid F \wedge F \mid F \vee F \mid F \rightarrow F & & \text{propositional} \\ \quad \bigcirc F \mid \Box F \mid \Diamond F \mid F\ \mathcal{U}\ F \mid F\ \mathcal{W}\ F & & \text{future time} \\ \quad \odot F \mid \boxdot F \mid \Diamond\!\!\!\!-\ F \mid F\ \mathcal{S}\ F \mid F\ \mathcal{Z}\ F & & \text{past time} \end{array}$$

The semantics of the logic is defined in terms of a satisfaction relation, $\models_{LTL}$, between an execution trace, an index and an LTL formula. An execution trace σ is a finite sequence of program states $\sigma = s_1 s_2 \ldots s_n$, where $|\sigma| = n$ is the length of the trace. The i'th state s_i of a trace σ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of σ from position i to position j, both positions included. The notion that a finite trace σ *satisfies* a formula F is then denoted by $\sigma, 1 \models_{LTL} F$. It is defined inductively by the definitions given below, where $0 \leq i \leq n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that the index i for a trace can become 0 (before the first state) when going backwards, and can become $n+1$ (after the last state) when going forwards. Since the context is clear, we omit below the index LTL from $\models_{LTL}$.

$$\begin{array}{ll} \sigma, i \models a & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \mathit{evaluate}(a)(\sigma(i)) == \mathit{true} \\ \sigma, i \models \underline{\text{true}} & \\ \sigma, i \not\models \underline{\text{false}} & \\ \sigma, i \models \neg F & \text{iff } \sigma, i \not\models F \\ \sigma, i \models F_1 \wedge F_2 & \text{iff } \sigma, i \models F_1 \text{ and } \sigma, i \models F_2 \\ \sigma, i \models F_1 \vee F_2 & \text{iff } \sigma, i \models F_1 \text{ or } \sigma, i \models F_2 \\ \sigma, i \models F_1 \rightarrow F_2 & \text{iff } \sigma, i \models F_1 \text{ implies } \sigma, i \models F_2 \\ \sigma, i \models \bigcirc F & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \sigma, i+1 \models F \\ \sigma, i \models \odot F & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \sigma, i-1 \models F \\ \sigma, i \models \Box F & \text{iff if } 1 \leq i \leq |\sigma| \text{ then } \forall j:\ i \leq j \leq |\sigma| \text{ implies } \sigma, j \models F \\ \sigma, i \models \boxdot F & \text{iff if } 1 \leq i \leq |\sigma| \text{ then } \forall j:\ 1 \leq j \leq i \text{ implies } \sigma, j \models F \\ \sigma, i \models \Diamond F & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \exists j:\ i \leq j \leq |\sigma| \text{ and } \sigma, j \models F \\ \sigma, i \models \Diamond\!\!\!\!-\ F & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \exists j:\ 1 \leq j \leq i \text{ and } \sigma, j \models F \\ \sigma, i \models F_1\ \mathcal{U}\ F_2 & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \exists i_2:\ i \leq i_2 \leq |\sigma| \text{ and } \sigma, i_2 \models F_2 \text{ and} \\ & \qquad \forall\, i_1:\ i \leq i_1 < i_2 \text{ implies } \sigma, i_1 \models F_1 \\ \sigma, i \models F_1\ \mathcal{S}\ F_2 & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \exists i_2:\ 1 \leq i_2 \leq i \text{ and } \sigma, i_2 \models F_2 \text{ and} \\ & \qquad \forall\, i_1:\ i_2 < i_1 \leq i \text{ implies } \sigma, i_1 \models F_1 \\ \sigma, i \models F_1\ \mathcal{W}\ F_2 & \text{iff } \sigma, i \models F_1\ \mathcal{U}\ F_2 \text{ or } \sigma, i \models \Box F_1 \\ \sigma, i \models F_1\ \mathcal{Z}\ F_2 & \text{iff } \sigma, i \models F_1\ \mathcal{S}\ F_2 \text{ or } \sigma, i \models \boxdot F_1 \end{array}$$

Note that for the boundary cases, $\sigma, 0 \models \boxdot F$ and $\sigma, |\sigma|+1 \models \Box F$ hold. The usual dualities are maintained, e.g. $\Box F \equiv \neg\Diamond\neg F$, and hence $\sigma, |\sigma|+1 \models \Diamond F$ doesn't hold. Further, note that the next operator, $\bigcirc$, is defined to be strong (or existential), hence $\sigma, |\sigma|+1 \models \bigcirc F$ doesn't hold. Thus the formula $\neg \bigcirc \underline{\text{true}}$ only evaluates to true beyond the final state, i.e. on an empty future trace.

2.2 Introducing EAGLE

Fundamental Concepts In most temporal logics, the formulas $\Box F$ and $\Diamond F$ satisfy the following equivalences:

$$\Box F \equiv F \wedge \bigcirc(\Box F) \qquad \Diamond F \equiv F \vee \bigcirc(\Diamond F)$$

One can show that $\Box F$ is a solution to the recursive equation $X = F \wedge \bigcirc X$; in fact it is the maximal solution. A fundamental idea in our logic, EAGLE, is to support this kind of recursive definition, and to enable users define their own temporal combinators using equations similar to those above. In the current framework one can write the following definitions for the two combinators `Always` and `Eventually`:

$$\underline{\max}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \texttt{Always}(F)$$
$$\underline{\min}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \bigcirc \texttt{Eventually}(F)$$

First note that these rules are parameterized by an EAGLE formula (of type Form). Thus, assuming an atomic formula, say $x < 0$, then, in the context of these two definitions, we will be able to write EAGLE formulas such as, $\texttt{Always}(x > 0)$, or $\texttt{Always}(\texttt{Eventually}(x > 0))$. Secondly, note that the `Always` operator is defined as maximal; when applied to a formula F it denotes the maximal solution to the equation $X = F \wedge \bigcirc X$. On the other hand, the `Eventually` operator is defined as a minimal, and $\texttt{Eventually}(F)$ represents the minimal solution to the equation $X = F \vee \bigcirc X$. In EAGLE, this difference only becomes important when evaluating formulas at the boundaries of a trace. To understand how this works it suffices to say here that monitored rules evolve as new states are appearing. Assume that the end of the trace has been reached (we are beyond the last state) and a monitored formula F has evolved to F'. Then all applications in F' of maximal fix-point rules will evaluate to true, since they represent safety properties that apparently have been satisfied throughout the trace, while applications of minimal fix-point rules will evaluate to false, indicating that some event did not happen.

EAGLE has been designed specifically as a general purpose, but low-level, temporal logic for run-time monitoring. So to complete this very brief introduction to EAGLE suppose one wished to monitor the following property of a Java program state containing two variables *x* and *y*: "*whenever we reach a state where* $x = k > 0$ *for some value k, then eventually we will reach a state at which* $y == k$". In a linear temporal logic augmented with first order quantification, we would write: $\Box(x > 0 \rightarrow \exists k.(k = x \wedge \Diamond y = k))$. The parametrization mechanism of EAGLE allows data parameters as well as formula ones and we are then able to encode the above as:

$$\underline{\min}\ R(\underline{\text{int}}\ k) = \texttt{Eventually}(y == k) \qquad \underline{\text{mon}}\ M = \texttt{Always}(x > 0 \rightarrow R(x))$$

The definition starting with keyword mon specifies the EAGLE formula to be monitored. The rule *R* is parameterized with an integer *k*; it is instantiated in the monitor *M* when $x > 0$ and hence captures the value of *x* at that moment. Rule *R* replaces the existential quantifier. EAGLE also provides a previous-time operator, which allows us to define past time operators, and a concatenation operator, which allows users to define interval based logics, and more. Data parametrization works uniformly for rules over past as well as future; this is non-trivial to achieve as the implementation doesn't store the trace, see [4]. Data parametrization is also used to elegantly model real-time, metric and statistical logics.

EAGLE Syntax A specification S consists of a declaration part D and an observer part O. D consists of zero or more rule definitions R, and O consists of zero or more monitor definitions M, which specify what is to be monitored. Rules and monitors are named (N).

$$
\begin{array}{rcl}
S & ::= & D\ O \\
D & ::= & R^* \\
O & ::= & M^* \\
R & ::= & \{\underline{\text{max}} \mid \underline{\text{min}}\}\ N(T_1\ x_1, \ldots, T_n\ x_n) = F \\
M & ::= & \underline{\text{mon}}\ N = F \\
T & ::= & \underline{\text{Form}} \mid \textit{primitive type} \\
F & ::= & \textit{expression} \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
 & & \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n) \mid x_i
\end{array}
$$

A rule definition R is preceded by a keyword indicating whether the interpretation is maximal or minimal (which we recall determines the value of a rule application at the boundaries of the trace). Parameters are typed, and can either be a formula of type Form, or of a primitive type, such as int, long, float, etc.. The body of a rule/monitor is a boolean-valued formula of the syntactic category *Form* (with meta-variables F, etc.). However, a monitor cannot have a recursive definition, that is, a monitor defined as mon $N = F$ cannot use N in F. For rules we do not place such restrictions. The propositions of this logic are boolean expressions over an observer state. Formulas are composed using standard propositional connectives together with a next-state operator ($\bigcirc F$), a previous-state operator ($\odot F$), and a concatenation-operator ($F_1 \cdot F_2$). Finally, rules can be applied and their parameters must be type correct; formula arguments can be any formula, with the restriction that if an argument is an expression, it must be of boolean type.

EAGLE Semantics The semantics of the logic is defined in terms of a satisfaction relation, $\models$, between execution traces and specifications. As with the LTL semantics, we assume an execution trace σ is a finite sequence of program states $\sigma = s_1 s_2 \ldots s_n$, where $|\sigma| = n$ is the length of the trace, and adopt the same notation for indexing and slicing. Given a trace σ and a specification $D\ O$, we define:

$$\sigma \models D\ O \ \text{ iff } \ \forall\, (\underline{\text{mon}}\ N = F) \in O\ .\ \sigma, 1 \models_D F$$

That is, a trace satisfies a specification if the trace, observed from position 1 (the first state), satisfies each monitored formula. The definition of the satisfaction relation $\models_D \subseteq (\textit{Trace} \times \textbf{nat}) \times \textit{Form}$, for a set of rule definitions D, is presented in Figure 1 below, where $0 \leq i \leq n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that, as with the LTL definitions, the position of a trace can become 0 (before the first state) when going backwards, and can become $n+1$ (after the last state) when going forwards both cases causing rule applications to evaluate to either true if maximal or false if minimal, without considering the body of the rules at that point. An atomic formula (*atom*) is evaluated in the current state, i, in case the position i is within the trace ($1 \leq i \leq n$); for the boundary cases ($i = 0$ and $i = n+1$) it evaluates to false. Propositional connectives have their usual semantics in all positions. A next-time formula $\bigcirc F$ evaluates to true if the current position is not beyond the last state and F holds in the next position. Dually for the previous-time formula. This means that these formulas always evaluate to false in the boundary positions (0 and $n+1$). The concatenation formula $F_1 \cdot F_2$ is true if the trace

$$
\begin{array}{lll}
\sigma, i \models_D \mathit{atom} & \text{iff} & 1 \le i \le |\sigma| \text{ and } \mathit{evaluate}(\mathit{atom})(\sigma(i)) == \mathit{true} \\
\sigma, i \models_D \underline{\text{true}} & & \\
\sigma, i \not\models_D \underline{\text{false}} & & \\
\sigma, i \models_D \neg F & \text{iff} & \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff} & i \le |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff} & 1 \le i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff} & \exists j \text{ s.t. } i \le j \le |\sigma|+1 \text{ and } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(F_1, \ldots, F_m) & \text{iff} & \left\{ \begin{array}{l} \text{if } 1 \le i \le |\sigma| \text{ then:} \\ \quad \sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\ \quad \text{where } (N(T_1\ x_1, \ldots, T_m\ x_m) = F) \in D \\ \text{otherwise, if } i = 0 \text{ or } i = |\sigma|+1 \text{ then:} \\ \quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D \end{array} \right.
\end{array}
$$

Fig. 1. Definition of $\sigma, i \models_D F$ for $0 \le i \le |\sigma|+1$ for some trace $\sigma = s_1 s_2 \ldots s_{|\sigma|}$

σ can be split into two sub-traces $\sigma = \sigma_1\sigma_2$, such that F_1 is true on σ_1, observed from the current position i, and F_2 is true on σ_2 (ignoring σ_1, and thereby limiting the scope of past time operators). Applying a rule within the trace (positions $1 \ldots n$) consists of replacing the call with the right-hand side of the definition, substituting arguments for formal parameters. At the boundaries (0 and $n+1$) a rule application evaluates to true if and only if it is maximal.

2.3 Linear Temporal Logic in EAGLE

We have briefly shown how in EAGLE one can define rules for the $\Box$ and $\Diamond$ temporal operators for LTL. Here we complete an embedding of propositional LTL in EAGLE and prove its semantic correspondence. For each temporal operator, future and past, we define a corresponding EAGLE rule. The embedding is straightforward and requires little explanation. The future time operators give rise to the following set of rules:

$$
\begin{array}{l}
\underline{\text{min}}\ \texttt{Next}(\underline{\text{Form}}\ F) = \bigcirc F \\
\underline{\text{max}}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \texttt{Always}(F)) \\
\underline{\text{min}}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \bigcirc \texttt{Eventually}(F)) \\
\underline{\text{min}}\ \texttt{Until}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc \texttt{Until}(F_1, F_2)) \\
\underline{\text{max}}\ \texttt{Unless}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc \texttt{Unless}(F_1, F_2))
\end{array}
$$

Note that the unless modality is defined as maximal since we require that $\texttt{Unless}(F_1, F_2)$ evaluates to true on the empty sequence, unlike $\texttt{Until}(F_1, F_2)$ that must evaluate to false on the empty sequence. The past time operators of LTL give rise to the following rules.

$$
\begin{array}{l}
\underline{\text{min}}\ \texttt{Previous}(\underline{\text{Form}}\ F) = \odot F \\
\underline{\text{max}}\ \texttt{AlwaysInPast}(\underline{\text{Form}}\ F) = F \wedge \odot \texttt{AlwaysInPast}(F)) \\
\underline{\text{min}}\ \texttt{EventuallyInPast}(\underline{\text{Form}}\ F) = F \vee \odot \texttt{EventuallyInPast}(F)) \\
\underline{\text{min}}\ \texttt{Since}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \odot \texttt{Since}(F_1, F_2)) \\
\underline{\text{max}}\ \texttt{Zince}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \odot \texttt{Zince}(F_1, F_2))
\end{array}
$$

An EAGLE context containing all of the above rules then enables any propositional LTL monitoring formula to be expressed as a monitoring formula in EAGLE by mapping the LTL operators to the EAGLELTL counterparts. Note that through simply combining the definitions for the future and past time LTLs defined above, we obtain a temporal logic over the future, present and past, in which one can freely intermix the future and past time modalities (to any depth).

Correctness of Embedding: To justify the above EAGLE definitions of LTL temporal operators, we can define an embedding function $Embed : LTL \rightarrow \text{EAGLE}$ that maps $\bigcirc F$ to $\texttt{Next}(Embed(F))$, $\Box F$ to $\texttt{Always}(Embed(F))$, etc., and then formally establish that $\sigma, i \models_{LTL} F$ iff $\sigma, i \models_{\text{EAGLE}} Embed(F)$ for all traces σ and indices i. Note that $\models_{LTL}$ refers to the semantic definition of LTL in Section 2.1 and $\models_{\text{EAGLE}}$ refers to the semantic definition of EAGLE as in Section 2.2 together with the rule definitions for `Next`, `Always`, etc.. The proof is by induction over the structure of the formula F. For brevity, we only show one step of the inductive proof, that relating to the $\mathcal{U}$ operator. The base case is when the formula is atomic and trivially holds. The main result is then used as the inductive hypothesis, namely: $\sigma, i \models_{LTL} F$ iff $\sigma, i \models_{\text{EAGLE}} Embed(F)$ for all traces σ and indices i. For the $\mathcal{U}$ operator, we must then show $\sigma, i \models_{LTL} F_1 \; \mathcal{U} \; F_2$ iff $\sigma, i \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$ for arbitrary σ and i.

First, assume $\sigma, i \models_{LTL} F_1 \; \mathcal{U} \; F_2$ for arbitrary σ and i. By the definition of $\mathcal{U}$, there exists an i_2 such that $i \leq i_2 \leq |\sigma|$ and $\sigma, i_2 \models_{LTL} F_2$ and for all i_1, $i \leq i_1 < i_2$, $\sigma, i_1 \models_{LTL} F_1$. Hence by the inductive hypothesis, $\sigma, i_2 \models_{\text{EAGLE}} Embed(F_2)$ and for all i_1, $i \leq i_1 < i_2$, $\sigma, i_1 \models_{\text{EAGLE}} Embed(F_1)$. We will show by a downward iterative argument on i_2 that $\sigma, i \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$. For the basis when $i_2 = i$, it is clear by the definition of `Until` that $\sigma, i \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$. For the case that $|\sigma| \geq i_2 > i$ again by the definition of `Until`, we can derive that $\sigma, i_2 \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$, and hence by the definition of $\bigcirc$ that $\sigma, i_2 - 1 \models_{\text{EAGLE}} \bigcirc\texttt{Until}(Embed(F_1), Embed(F_2))$. But note we also have that $\sigma, i_1 \models_{LTL} F_1$ for $i \leq i_1 < i_2$, hence for $i_1 = i_2 - 1$ we have $\sigma, i_2 - 1 \models_{LTL} F_1$. Therefore by the inductive hypothesis we have $\sigma, i_2 - 1 \models_{\text{EAGLE}} Embed(F_1)$.

Thus: $\sigma, i_2 - 1 \models_{\text{EAGLE}} Embed(F_1) \wedge \bigcirc\texttt{Until}(Embed(F_1), Embed(F_2))$
hence: $\sigma, i_2 - 1 \models_{\text{EAGLE}} Embed(F_2) \vee Embed(F_1) \wedge \bigcirc\texttt{Until}(Embed(F_1), Embed(F_2))$
i.e.: $\sigma, i_2 - 1 \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$

From which we can infer $\sigma, i \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$.

Second, assume $\sigma, i \not\models_{LTL} F_1 \; \mathcal{U} \; F_2$. Then, by the semantics of $\mathcal{U}$: either (i) for i_1, $i \leq i_1 \leq |\sigma|$, $\sigma, i_1 \not\models_{LTL} F_2$; or (ii) there is some i_2, $i \leq i_2 \leq |\sigma|$ s.t. $\sigma, i_2 \not\models_{LTL} F_1$ and for all i_1, $i \leq i_1 \leq i_2$, $\sigma, i_1 \not\models_{LTL} F_2$. For both these cases, it can be shown that $\sigma, i \not\models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$. For (i) the evaluation of $\sigma, i \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$ will reduce to the evaluation of $\sigma, |\sigma| + 1 \models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$, which is false since the `Until` rule is defined as minimal. For (ii), by the semantics of `Until` and the inductive hypothesis, $\sigma, i_2 \not\models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$. Furthermore, as $\sigma, i_1 \not\models_{LTL} F_2$ for all i_1, $i \leq i_1 \leq i_2$, it follows that $\sigma, i_1 \not\models \text{EAGLE}\texttt{Until}(Embed(F_1), Embed(F_2))$, and hence $\sigma, i \not\models_{\text{EAGLE}} \texttt{Until}(Embed(F_1), Embed(F_2))$.

The cases for the other temporal operators are as straightforward.

3 Algorithm

In this section, we now outline the computation mechanism used to determine whether a monitoring formula given in LTL holds for some given input sequence of events. We assume that a user of the system specifies a set of (EAGLE embedded) LTL formulas that needs to be monitored; no rule definitions are given in the specification. The monitoring system, or the *observer*, maintains a local state. The *atomic propositions* are specified with respect to the variables in this local state. At every event the observer modifies the local state based on that event and then evaluates the monitored formulas on that state and generates a new set of monitored formulas. At the end of the trace the value of the monitored formulas are determined. If the value is true for a formula we say that the formula is satisfied, otherwise we say that the formula is violated.

The evaluation of a formula F on a state $s = \sigma(i)$ in a trace σ results in another formula $\mathit{eval}(F,s)$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models \mathit{eval}(F,s)$. The definition of the function $\mathit{eval} : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ uses another auxiliary function $\mathit{update} : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$. The role of the function *update* is to pre-evaluate a formula if it is guarded by a previous operator. Formally, *update* has the property that $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models \mathit{update}(F,s)$. Had there been no past time modality in EAGLE we could have ignored *update* and simply written $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models F$. The value of a formula F at the end of a trace is given by $\mathit{value}(F)$. The function $\mathit{value} : \underline{\text{Form}} \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$ when applied on F returns $\underline{\text{true}}$ if F is satisfied at the end of the trace or in other words iff $\sigma, |\sigma| + 1 \models F$ and returns $\underline{\text{false}}$ otherwise. Thus given a sequence of states $s_1 s_2 \ldots s_n$, an LTL formula F written in EAGLE is said to be satisfied by the sequence of states if and only if $\mathit{value}(\mathit{eval}(\ldots \mathit{eval}(\mathit{eval}(F, s_1), s_2) \ldots s_n))$ is $\underline{\text{true}}$. The definition of the functions *eval*, *update* and *value* forms the calculus of the LTL subset of EAGLE. We define this calculus next.

3.1 Calculus

The *eval*, *update* and *value* functions are defined a priori for all operators. Note that, unlike in general EAGLE where new temporal operators in the form of rules can be defined, in LTL the operators are fixed. So instead of giving a general algorithm to synthesize the definitions of *eval*, *update* and *value* for the rules [4], we can provide these definitions for the fixed operators of LTL before hand and make them part of our calculus. We do not define the functions on the previous operator $\bigodot$, since this operator is eliminated in the calculus that we present next. The definition of *eval*, *update* and *value* on the different primitive EAGLE operators is given below.

$$
\begin{array}{rl}
\mathit{eval}(\underline{\text{true}}, s) = & \underline{\text{true}} \\
\mathit{eval}(\underline{\text{false}}, s) = & \underline{\text{false}} \\
\mathit{eval}(\mathit{jexp}, s) = & \text{value of jexp in } s \\
\mathit{eval}(F_1 \mathit{\ op\ } F_2, s) = & \mathit{eval}(F_1, s) \mathit{\ op\ } \mathit{eval}(F_2, s) \\
\mathit{eval}(\neg F, s) = & \neg \mathit{eval}(F, s) \\
\mathit{eval}(\bigcirc F, s) = & \mathit{update}(F, s)
\end{array}
\qquad
\begin{array}{rl}
\mathit{value}(\underline{\text{true}}) = & \underline{\text{true}} \\
\mathit{value}(\underline{\text{false}}) = & \underline{\text{false}} \\
\mathit{value}(\mathit{jexp}) = & \underline{\text{false}} \\
\mathit{value}(F_1 \mathit{\ op\ } F_2) = & \mathit{value}(F_1) \mathit{\ op\ } \mathit{value}(F_2) \\
\mathit{value}(\neg F) = & \neg \mathit{value}(F) \\
\mathit{value}(\bigcirc F) = & \underline{\text{false}}
\end{array}
$$

$$
\begin{aligned}
\mathit{update}(\underline{\text{true}}, s) &= \underline{\text{true}} \\
\mathit{update}(\underline{\text{false}}, s) &= \underline{\text{false}} \\
\mathit{update}(\mathit{jexp}, s) &= \mathit{jexp} \\
\mathit{update}(F_1 \mathit{\ op\ } F_2, s) &= \mathit{update}(F_1, s) \mathit{\ op\ } \mathit{update}(F_2, s) \\
\mathit{update}(\neg F, s) &= \neg \mathit{update}(F, s) \\
\mathit{update}(\bigcirc F, s) &= \bigcirc \mathit{update}(F, s)
\end{aligned}
$$

In the above definitions, *op* can be $\wedge, \vee, \rightarrow$. Note that *eval* of a formula of the form $\bigcirc F$ on a state s reduces to the *update* of F on state s. This ensures that if F contains any past time operators then *update* of F updates them properly. Moreover, $\mathit{value}(\bigcirc F)$ is $\underline{\text{false}}$ as the operator $\bigcirc$ has a strong interpretation in EAGLE. The *value* of a $\underline{\text{max}}$ rule is $\underline{\text{true}}$ and that of a $\underline{\text{min}}$ rule is $\underline{\text{false}}$.

$$
\begin{aligned}
\mathit{value}(\mathtt{R}(F_1, \ldots, F_n)) &= \underline{\text{true}} \text{ if } \mathtt{R} \text{ is } \underline{\text{max}} \\
\mathit{value}(\mathtt{R}(F_1, \ldots, F_n)) &= \underline{\text{false}} \text{ if } \mathtt{R} \text{ is } \underline{\text{min}}
\end{aligned}
$$

The definition of the *eval* and *update* functions for the rules are not generic for all LTL operators. However, as we have a fixed number of rules for a fixed number of LTL operators we can define these functions for each rule and make them part of the calculus.

Future Time Operators Consider the `Always` operator:

$$
\underline{\text{max}}\ \mathtt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \mathtt{Always}(F)
$$

For this rule *eval* and *update* are defined as follows.

$$
\begin{aligned}
\mathit{eval}(\mathtt{Always}(F), s) &= \mathit{eval}(F \wedge \bigcirc \mathtt{Always}(F), s) \\
\mathit{update}(\mathtt{Always}(F), s) &= \mathtt{Always}(\mathit{update}(F, s))
\end{aligned}
$$

Note that if we define the *update* function in a way similar to *eval*:

$$
\mathit{update}(\mathtt{Always}(F), s) = \mathit{update}(F \wedge \bigcirc \mathtt{Always}(F), s)
$$

then definition of *update* results in infinite recursion. To stop the recursion we note that the rule `Always` does not contain any previous operator, although the argument F may contain some. So we simply propagate the *update* to the argument F. Similarly we can give the calculus for the other future time LTL operators as follows:

$$
\begin{aligned}
\mathit{eval}(\mathtt{Next}(F), s) &= \mathit{eval}(\bigcirc F, s) \\
\mathit{update}(\mathtt{Next}(F), s) &= \mathtt{Next}(\mathit{update}(F, s))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{eval}(\mathtt{Eventually}(F), s) &= \mathit{eval}(F \vee \bigcirc \mathtt{Eventually}(F), s) \\
\mathit{update}(\mathtt{Eventually}(F), s) &= \mathtt{Eventually}(\mathit{update}(F, s))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{eval}(\mathtt{Until}(F_1, F_2), s) &= \mathit{eval}(F_2 \vee (F_1 \wedge \bigcirc \mathtt{Until}(F_1, F_2)), s) \\
\mathit{update}(\mathtt{Until}(F), s) &= \mathtt{Until}(\mathit{update}(F_1, s), \mathit{update}(F_2, s))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{eval}(\mathtt{Unless}(F_1, F_2), s) &= \mathit{eval}(F_2 \vee (F_1 \wedge \bigcirc \mathtt{Unless}(F_1, F_2)), s) \\
\mathit{update}(\mathtt{Unless}(F), s) &= \mathtt{Unless}(\mathit{update}(F_1, s), \mathit{update}(F_2, s))
\end{aligned}
$$

Past Time Operators However, the definitions are different for past time LTL operators. The past time LTL operators are defined in the form of rules containing a $\odot$ operator. In general, if a rule contains a formula F guarded by a previous operator on its right hand side then we evaluate F at every event and use the result of this evaluation in the next state. Thus, the result of evaluating F is required to be stored in some temporary placeholder so that it can be used in the next state. To allocate a placeholder, we introduce, for every formula guarded by a previous operator, an argument in the rule and use these arguments in the definition of *eval* and *update* for this rule. Let us illustrate this as follows.

$$\underline{\max}\ \texttt{AlwaysInPast}(\underline{\text{Form}}\ F) = F \wedge \odot \texttt{AlwaysInPast}(F)$$

For this rule we introduce another auxiliary rule $\texttt{AlwaysInPast}'$ that contains an extra argument corresponding to the formula $\odot(\texttt{AlwaysInPast}(F))$. In any LTL formula, we use this prime version of the rule instead of the original rule.

$$\begin{aligned}
&\texttt{AlwaysInPast}(F) = \texttt{AlwaysInPast}'(F, \underline{\text{true}})\\
&\mathit{eval}(\texttt{AlwaysInPast}'(F, \mathit{past}_1), s) = \mathit{eval}(F \wedge \mathit{past}_1, s)\\
&\mathit{update}(\texttt{AlwaysInPast}'(F, \mathit{past}_1), s) =\\
&\qquad \texttt{AlwaysInPast}'(\mathit{update}(F, s), \mathit{eval}(\texttt{AlwaysInPast}'(F, \mathit{past}_1), s))
\end{aligned}$$

Here, in *eval*, the subformula $\odot(\texttt{AlwaysInPast}(F))$ guarded by the previous operator is replaced by the argument past_1 that contains the evaluation of the subformula in the previous state. In *update* we not only update the argument F but also evaluate the subformula $\texttt{AlwaysInPast}'(F, \mathit{past}_1)$ and pass it as second argument of $\texttt{AlwaysInPast}'$. Thus in the next state past_1 is bound to $\odot(\texttt{AlwaysInPast}'(F, \mathit{past}_1))$. Note that in the definition of $\texttt{AlwaysInPast}'$ we pass $\underline{\text{true}}$ as the second argument. This is because, $\texttt{AlwaysInPast}$ being defined a maximal operator, its previous value at the beginning of the trace is $\underline{\text{true}}$. Similarly, we can give the calculus for the other past time LTL operators as follows:

$$\begin{aligned}
&\texttt{Previous}(F) = \texttt{Previous}'(F, \underline{\text{false}})\\
&\mathit{eval}(\texttt{Previous}'(F, \mathit{past}_1), s) = \mathit{eval}(\mathit{past}_1, s)\\
&\mathit{update}(\texttt{Previous}'(F, \mathit{past}_1), s) = \texttt{Previous}'(\mathit{update}(F, s), \mathit{eval}(F, s))
\end{aligned}$$

$$\begin{aligned}
&\texttt{EventuallyInPast}(F) = \texttt{EventuallyInPast}'(F, \underline{\text{false}})\\
&\mathit{eval}(\texttt{EventuallyInPast}'(F, \mathit{past}_1), s) = \mathit{eval}(F \vee \mathit{past}_1, s)\\
&\mathit{update}(\texttt{EventuallyInPast}'(F, \mathit{past}_1), s) =\\
&\qquad \texttt{EventuallyInPast}'(\mathit{update}(F, s), \mathit{eval}(\texttt{EventuallyInPast}'(F, \mathit{past}_1), s))
\end{aligned}$$

$$\begin{aligned}
&\texttt{Since}(F_1, F_2) = \texttt{Since}'(F_1, F_2, \underline{\text{false}})\\
&\mathit{eval}(\texttt{Since}'(F_1, F_2, \mathit{past}_1), s) = \mathit{eval}(F_2 \vee (F_1 \wedge \mathit{past}_1), s)\\
&\mathit{update}(\texttt{Since}'(F_1, F_2, \mathit{past}_1), s) =\\
&\qquad \texttt{Since}'(\mathit{update}(F_1, s), \mathit{update}(F_1, s), \mathit{eval}(\texttt{Since}'(F_1, F_2, \mathit{past}_1), s))
\end{aligned}$$

$$\begin{aligned}
&\texttt{Zince}(F_1, F_2) = \texttt{Zince}'(F_1, F_2, \underline{\text{true}})\\
&\mathit{eval}(\texttt{Zince}'(F_1, F_2, \mathit{past}_1), s) = \mathit{eval}(F_2 \vee (F_1 \wedge \mathit{past}_1), s)\\
&\mathit{update}(\texttt{Zince}'(F_1, F_2, \mathit{past}_1), s) =\\
&\qquad \texttt{Zince}'(\mathit{update}(F_1, s), \mathit{update}(F_1, s), \mathit{eval}(\texttt{Zince}'(F_1, F_2, \mathit{past}_1), s))
\end{aligned}$$

For the sake of completeness of the calculus we explicitly define *value* on the above LTL operators as follows:

$$\begin{array}{ll} \mathit{value}(\texttt{Always}(F)) = \underline{\text{true}} & \mathit{value}(\texttt{Eventually}(F)) = \underline{\text{false}} \\ \mathit{value}(\texttt{Until}(F_1,F_2)) = \underline{\text{false}} & \mathit{value}(\texttt{Unless}(F_1,F_2)) = \underline{\text{true}} \\ \mathit{value}(\texttt{AlwaysInPast}'(F,\mathit{past}_1)) = \underline{\text{true}} & \mathit{value}(\texttt{EventuallyInPast}'(F,\mathit{past}_1)) = \underline{\text{false}} \\ \mathit{value}(\texttt{Since}'(F_1,F_2,\mathit{past}_1)) = \underline{\text{false}} & \mathit{value}(\texttt{Zince}'(F_1,F_2,\mathit{past}_1)) = \underline{\text{true}} \end{array}$$

Note that in the above calculus we have eliminated the previous operator by introducing an auxiliary argument or placeholder for every formula guarded by the $\odot$ operator. Hence, we cannot use the operator $\odot$ while writing an LTL formula. Instead we use the rule `Previous` as defined above.

Correctness of Evaluation Given a set of definitions of *eval*, *update* and *value* functions for the different operators of LTL, as detailed above, we claim that for a given sequence $\sigma = s_1 s_2 \ldots s_n$ and an EAGLE embedded LTL formula F

$$\sigma, 1 \models_{\text{EAGLE}} F \text{ iff } \mathit{value}(\mathit{eval}(\ldots \mathit{eval}(\mathit{eval}(F, s_1), s_2) \ldots s_n)).$$

Insufficient space prohibits inclusion of the proof, or part thereof. However, we illustrate the evaluation calculus with a small example.

Example Let us consider the LTL formula $\Box(p \rightarrow \Diamond q)$ which is written as $\texttt{Always}(p \rightarrow \texttt{EventuallyInPast}(q))$ in EAGLE. This formula, at the beginning of monitoring, gets transformed into the formula

$$\texttt{Always}(\neg p \vee \texttt{EventuallyInPast}'(q, \underline{\text{false}}))$$

To save space we will use $\texttt{A}$ and $\texttt{Ep}'$ as shorthand for `Always` and $\texttt{EventuallyInPast}'$, respectively. Thus the formula will be written as $\texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}}))$.

If we have a sequence of states $\{\neg p, q\}, \{\neg p, \neg q\}, \{p, \neg q\}$, then the step by step monitoring of the above formula on this sequence takes place as follows:

Step 1: $s = \{\neg p, q\}$

$$\begin{array}{l} \mathit{eval}(\ \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}})), s) \\ \quad = \mathit{eval}((\neg p \vee \texttt{Ep}'(q, \underline{\text{false}})) \wedge \bigcirc \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}})), s) \\ \quad = (\mathit{eval}(\neg p, s) \vee \mathit{eval}(\texttt{Ep}'(q, \underline{\text{false}}), s)) \wedge \mathit{eval}(\bigcirc \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}})), s) \\ \quad = (\underline{\text{true}} \vee \mathit{eval}(\texttt{Ep}'(q, \underline{\text{false}}), s)) \wedge \mathit{update}(\texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}})), s) \\ \quad = \underline{\text{true}} \wedge \texttt{A}(\mathit{update}(\neg p \vee \texttt{Ep}'(q, \underline{\text{false}}), s)) \\ \quad = \texttt{A}(\mathit{update}(\neg p, s) \vee \mathit{update}(\texttt{Ep}'(q, \underline{\text{false}}), s)) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(\mathit{update}(q, s), \mathit{eval}(\texttt{Ep}'(q, \underline{\text{false}}), s))) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(q, \mathit{eval}(q \vee \underline{\text{false}}, s))) \quad = \quad \texttt{A}(\neg p \vee \texttt{Ep}'(q, \mathit{eval}(q, s) \vee \mathit{eval}(\underline{\text{false}}, s))) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}} \vee \underline{\text{false}})) \quad = \quad \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}})) \end{array}$$

Note in the above step, q being true in the state, the placeholder in the formula $\texttt{Ep}'$ becomes $\underline{\text{true}}$. We now describe the other steps in short:

Step 2: $s = \{\neg p, \neg q\}$

$$\begin{array}{l} \mathit{eval}(\ \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}})), s) \\ \quad = \underline{\text{true}} \wedge \mathit{eval}(\bigcirc \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}}), s)) \\ \quad = \texttt{A}(\mathit{update}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}}), s)) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(\mathit{update}(q, s), \mathit{eval}(\texttt{Ep}'(q, \underline{\text{true}}), s))) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(q, \mathit{eval}(q \vee \underline{\text{true}}, s))) \\ \quad = \texttt{A}(\neg p \vee \texttt{Ep}'(q, \underline{\text{true}})) \end{array}$$

Step 3: $s = \{p, \neg q\}$

$$\begin{aligned}
&eval(\ \mathsf{A}(\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}})), s) \\
&\quad = eval((\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}})), s) \wedge eval(\bigcirc \mathsf{A}(\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}}), s)) \\
&\quad = (\underline{\text{false}} \vee \underline{\text{true}}) \wedge \mathsf{A}(update(\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}}), s)) \\
&\quad = \mathsf{A}(\neg p \vee \mathsf{Ep}'(update(q, s), eval(\mathsf{Ep}'(q, \underline{\text{true}}), s))) \\
&\quad = \mathsf{A}(\neg p \vee \mathsf{Ep}'(q, eval(q \vee \underline{\text{true}}, s))) \\
&\quad = \mathsf{A}(\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}}))
\end{aligned}$$

Final Value:

$$value(\mathsf{A}(\neg p \vee \mathsf{Ep}'(q, \underline{\text{true}}))) = \underline{\text{true}}$$

Thus the formula is satisfied on the trace.

4 Implementation and Complexity

We have an implementation for the monitoring framework for EAGLE in Java. The implemented system works in two phases. First, it compiles the specification file to *synthesize* a set of Java classes; a class is generated for each rule. Second, the Java class files are compiled into Java bytecode and then the monitoring engine dynamically loads the Java classes for rules at monitoring time and monitors a trace. However, for the purpose of LTL monitoring we do not have to synthesize the Java classes as the set of rules are fixed. Rather, we hardwire the whole algorithm in the implementation.

In order to make the implementation efficient we use the decision procedure of Hsiang [12]. The procedure reduces a tautological formula to the constant true, a false formula to the constant false, and all other formulas to canonical forms, each as an exclusive disjunction ($\oplus$) of conjunctions. The procedure is given below using equations that are shown to be Church-Rosser and terminating modulo associativity and commutativity.

simplify:

$$\begin{array}{lll}
\text{true} \wedge \phi = \phi & \text{false} \wedge \phi = \text{false} & \phi_1 \vee \phi_2 = (\phi_1 \wedge \phi_2) \oplus \phi_1 \oplus \phi_2 \\
\phi \wedge \phi = \phi & \text{false} \oplus \phi = \phi & \phi_1 \rightarrow \phi_2 = \text{true} \oplus \phi_1 \oplus (\phi_1 \wedge \phi_2) \\
\phi \oplus \phi = \text{false} & \neg\phi = \text{true} \oplus \phi & \phi_1 \equiv \phi_2 = \text{true} \oplus \phi_1 \oplus \phi_2
\end{array}$$

$$\phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3)$$

In particular the equations $\phi \wedge \phi = \phi$ and $\phi \oplus \phi = \text{false}$ ensures that, at the time of monitoring, we do not expand the formula beyond bound. The bound is given by the following theorem:

Theorem 1. *The size of the formula at any stage of monitoring is bounded by* $O(m^2 2^m \log m)$*, where m is the size of the initial LTL formula* ϕ *for which we started monitoring.*

Proof. The above equations, when regarded as simplification rules, keeps any LTL formula in a canonical form, which is an exclusive disjunction of conjunctions, where the conjuncts are either propositions or subformulas having temporal operators at top (see Fig 2). Moreover, after a series of applications of *eval* on the states $s_1, s_2, ..., s_n$, the conjuncts in the normal form $eval(\ldots eval(eval(\phi, s_1), s_2) \ldots, s_n)$ are propositions or

subformulas of the initial formula ϕ, each having a temporal operator at its top. Since there are at most m such subformulas, it follows that there are at most 2^m possibilities to combine them in a conjunction. The space requirement for a conjunction is $O(m \log m)$, assuming that in the conjunction, instead of keeping the actual conjuncts, we keep a pointer to the conjuncts and assuming that each pointer takes $O(\log m)$ bits.[4] Therefore, one needs space $O(m2^m \log m)$ to store the structure of any exclusive disjunction of such conjunctions. Now, we need to consider the storage requirements for each of the conjuncts that appears in the conjunction. Note that, if a conjunct contains a nested past time operator, the $past_1$ argument of that operator can be a formula. However, instead of storing the actual formula at the argument $past_1$ we can have a pointer to the formula. Thus, each conjunct can take space up to $O(m \log m)$. Hence space required by all the conjuncts is $O(m^2 \log m)$. Now for each past operator we have a formula that is pointed to by the $past_1$ argument and all those formulas by the above reasoning can take up space $O(m^2 2^m \log m)$. Hence the total space requirement is $O(m \log m 2^m + m^2 \log m + m^2 2^m \log m)$, which is $O(m^2 2^m \log m)$. □

The implementation contains a strategy for the application of these equations that ensures that the time complexity of each step in monitoring is bounded. We next describe the strategy briefly. Since, our LTL formulas are exclusive disjunction of conjunctions we can treat them as a tree of depth two: the root node at depth 0 representing the $\oplus$ operator, the children of the root at depth 1 representing the $\wedge$ operators, and the leaf nodes at depth 2 representing propositions and subformulas having temporal operators at the top. For example, Fig. 2 shows the tree representation of the formula $p \rightarrow \Diamond(q \ \mathcal{U} \ r)$, whose canonical form is $\text{true} \oplus p \oplus (p \wedge \Diamond(q \ \mathcal{U} \ r))$.

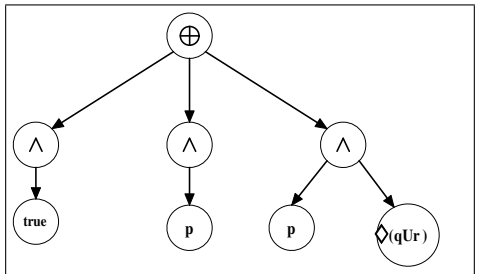

Fig. 2. Tree representation of $p \rightarrow \Diamond(q \ \mathcal{U} \ r)$

The application of the *eval* function on a formula is done in depth-first fashion on this tree and we build up the resultant formula in a bottom-up fashion. At the leaves the application of *eval* results either in the evaluation of a proposition or the evaluation of a rule. The evaluation of a proposition returns either true or false. We assume that this evaluation takes unit time. On the other-hand, the evaluation of a rule may result in another formula in canonical form. The formula at any internal node (i.e a $\wedge$ node or a $\oplus$ node) is then evaluated by taking the conjunction (or exclusive disjunction) of the formulas of the children nodes as they get evaluated and then simplifying them using the set of equations `simplify`. The following gives the pseudocode for the strategy:

[4] Every unique subformula having a temporal operator at the top in the original formula can give rise to several copies in the process of monitoring. For example, if we consider $F_1 = \Box\Diamond q$ after some steps, it may get converted to $F_2 = \Diamond q \wedge \Box\Diamond q$. In F_2 the two subformulas $\Diamond q$ are essentially copies of $\Diamond q$ in F_1. It is easy to see all such copies at any stage of monitoring will be same. So we can keep a single copy of them and in the formula we use a pointer to point to that copy.

```
1:Form eval(F,s)
2: begin
3:    Form F';
4:    if F is conjunction of subformulas then
5:       F' = true;
6:       for each subformula Fsub of F do
7:            F' = simplify(F' ∧ eval(Fsub,s));
8:       endfor
9:    else if F is exclusive disjunction of subformulas then
10:       F' = false;
11:       for each subformula Fsub of F do
12:            F' = simplify(F' ⊕ eval(Fsub,s));
13:       endfor
14:   else if F is a rule or proposition then
15:       ...
16:   endif
17:   return F';
18: endsub
```

Note that the application of `simplify` on the conjunction of two formulas (see line no. 7 in the pseudocode) requires the application of the distributive equation $\phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3)$ and possibly other equations.

At any stage of this algorithm there are three formulas that are active: the original formula F on which *eval* is applied, the formula F', and the result of the evaluation of the subformula *Fsub*. So, by theorem 1 we can say that the space complexity of this algorithm is $O(m^2 2^m \log m)$. Moreover, as the algorithm traverses the formula once at each node it can possibly spend $O(m^2 2^m \log m)$ time to do the conjunction and exclusive disjunction. Hence the time complexity of the algorithm is $O(m^2 2^m \log m).O(m^2 2^m \log m)$ or $O(m^4 2^{2m} \log^2 m)$. These two bounds are given as the following theorem.

Theorem 2. *At any stage of monitoring the space and time complexity of the evaluation of the monitored LTL formula on the current state is* $O(m^2 2^m \log m)$ *and* $O(m^4 2^{2m} \log^2 m)$ *respectively.*

Experiments: EAGLE has been applied to test a planetary rover controller in a collaborative effort with other colleagues, see [2] for an earlier similar experiment using a simpler logic. The rover controller, written in 35,000 lines of C++, executes action plans. The testing environment contains a test-case generator that automatically generates input plans for the controller. Additionally, for each input plan a set of temporal properties is generated that the plan execution should satisfy. The properties are expressed using either un-timed or metric, i.e real-time, LTL EAGLE formulas. The controller is executed on the generated plans and the implementation of EAGLE is used to monitor that execution traces satisfy the formulas. A previously unknown real-time error was detected in the first run, demonstrating that a certain task did not recognize the premature termination of some other task.

5 Conclusion and Future Work

We have presented a representation of linear temporal logic with both past and future temporal operators in EAGLE. We have shown how the generalized monitoring algorithm for EAGLE becomes simple and elegant for this particular case. We have bounded the space and time complexity of this specialized algorithm and thus showed that general LTL monitoring is space efficient if we use the EAGLE framework. Initial experiments have been successful. Future work includes: optimizing the current implementation and investigating other efficient subsets of EAGLE.

References

1. *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *ENTCS*. Elsevier Science: 2001, 2002, 2003.
2. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, LNCS, pages 87–107. Springer, March 2003.
3. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: An Introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International VMCAI conference (VMCAI'04) (To appear in LNCS)*, January 2004. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp24.pdf.
5. D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
6. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.
7. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of Runtime Verification (RV'02)* [1], pages 36–55.
8. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of Runtime Verification (RV'01)* [1], pages 44–60.
9. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. ENTCS, 2001. Coronado Island, California.
10. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
11. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
12. J. Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
13. D. Kortenkamp, T. Milam, R. Simmons, and J. Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proceedings of RV'01* [1], pages 133–151.
14. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
15. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
16. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of the 3rd Workshop on* Runtime Verification (RV'03) [1], pages 162–181.