Program Monitoring with LTL in EAGLE

Howard Barringer*
University of Manchester, England

Allen Goldberg, Klaus Havelund
Kestrel Technology, NASA Ames Research Center, USA

Koushik Sen†
University of Illinois, Urbana Champaign, USA

Abstract

We briefly present a rule-based framework, called EAGLE*, 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 focus on a linear temporal logic (LTL) specialisation of* EAGLE*. 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. This bound is close to the lower bound* $O(2^{\sqrt{m}})$ *for future-time LTL presented in [18].* 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) [17] 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]. Also a wide variety of specialised logics, largely based on LTL, have been proposed, see for example, [6, 7, 9, 8, 16, 15, 12, 11, 10]. 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 [5], 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 practical effectiveness of a specialised version of the monitoring algorithm for the case of LTL containing a fixed number of past and future time temporal operators embedded as rules in EAGLE. We outline an effective 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 scalable in terms of space as we do not store the trace either explicitly or implicitly. Similar work was done in a rewriting framework for the case of future time LTL in [11]; 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 [12] addresses a monitoring algorithm for past time LTL only.

*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.

The paper is structured as follows. Section 2 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 in section 4 where complexity bounds for the implementation can also be found. Section 5 describes an experiment performed using EAGLE, and shows how cyclic deadlock potentials can be detected with EAGLE. Section 6 states conclusion and future work.

2 EAGLE and Linear Temporal Logic

EAGLE [5] 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. In Section 2.1 we motivate the fundamental concepts of EAGLE through some simple examples drawn from LTL before presenting its formal definition. Then, in Section 2.2 we present a full embedding of LTL in EAGLE and establish its correctness.

2.1 Introducing EAGLE

2.1.1 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 in such a fashion. In the current framework one can write the following definitions for the two combinators `Always` and `Sometime`:

$$\begin{array}{l}\underline{\text{max}}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc\texttt{Always}(F)\\ \underline{\text{min}}\ \texttt{Sometime}(\underline{\text{Form}}\ F) = F \vee \bigcirc\texttt{Sometime}(F)\end{array}$$

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 $\texttt{Always}(\texttt{Sometime}(x > 0))$. 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 `Sometime` operator is defined as a minimal, and $\texttt{Sometime}(F)$ 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.

EAGLE has been designed specifically as a general purpose kernel 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 as well as formulas as parameters and are able to encode the above as:

$$\begin{array}{l}\underline{\text{min}}\ R(\underline{\text{int}}\ k) = \texttt{Sometime}(y == k)\\ \underline{\text{mon}}\ M = \texttt{Always}(x > 0 \rightarrow R(x))\end{array}$$

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 [5].

2.1.2 EAGLE Syntax

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

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

A rule definition R is preceded by a keyword indicating whether the interpretation is maximal or minimal. 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*. 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.

2.1.3 EAGLE Semantics

The semantics of the logic is defined in terms of a satisfaction relation, $\models$, between execution traces and specifications. We assume that 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. Given a trace σ and a specification $D\ O$, we define:

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

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 $\models_D \subseteq (\mathit{Trace} \times \mathbf{nat}) \times \mathit{Form}$, for a set of rule definitions D, is presented below, where $0 \le i \le n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that 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.

$$\begin{array}{ll}
\sigma, i \models_D \mathit{exp} & \text{iff } 1 \le i \le |\sigma| \text{ and } \mathit{eval}(\mathit{exp})(\sigma(i)) \\
\sigma, i \models_D \underline{\text{true}} & \\
\sigma, i \not\models_D \underline{\text{false}} & \\
\sigma, i \models_D \neg F & \text{iff } \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff } \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff } \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff } \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff } i \le |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff } 1 \le i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff } \exists j \text{ s.t. } i \le j \le |\sigma|+1 \text{ and} \\
 & \qquad \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2
\end{array}$$

$$\sigma, i \models_D N(F_1, \ldots, F_m)$$
$$\text{iff} \left\{ \begin{array}{l}
\text{if } 1 \le i \le |\sigma| \text{ then:} \\
\quad \sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\
\quad \text{where } (N(T_1\ x_1, \ldots, T_m\ x_m) = F) \ \in D \\
\text{otherwise, if } i = 0 \text{ or } i = |\sigma|+1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D
\end{array} \right.$$

An atomic formula (exp) is evaluated in the current state, i, in case the position i is within the trace ($1 \le i \le n$); 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. The concatenation formula $F_1 \cdot F_2$ is true if the trace 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.

2.2 Linear Temporal Logic in EAGLE

We have briefly seen 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. Figure 1 gives the semantic definition of the since and until LTL temporal operators over finite traces; the definitions of $\bigcirc$ and $\odot$, and the propositional connectives, are as for EAGLE. We assume the usual collection of future and past linear-time temporal operators.

$$\begin{array}{l}
\sigma, i \models F_1\ \mathcal{U}\ F_2 \quad \text{iff} \\
\quad 1 \le i \le |\sigma| \text{ and } \exists i_2 :\ i \le i_2 \le |\sigma| \text{ and } \sigma, i_2 \models F_2 \text{ and} \\
\qquad\qquad \forall\ i_1 :\ i \le i_1 < i_2 \text{ implies } \sigma, i_1 \models F_1 \\
\sigma, i \models F_1\ \mathcal{S}\ F_2 \quad \text{iff} \\
\quad 1 \le i \le |\sigma| \text{ and } \exists i_2 :\ 1 \le i_2 \le i \text{ and } \sigma, i_2 \models F_2 \text{ and} \\
\qquad\qquad \forall\ i_1 :\ i_2 < i_1 \le i \text{ implies } \sigma, i_1 \models F_1
\end{array}$$

with definitions

$$\begin{array}{ll}
\Diamond F = \underline{\text{true}}\ \mathcal{U}\ F & \Diamond F = \underline{\text{true}}\ \mathcal{S}\ F \\
\Box F = \neg \Diamond \neg F & \boxdot F = \neg \Diamond \neg F \\
F_1\ \mathcal{W}\ F_2 = F_1\ \mathcal{U}\ F_2 \vee \Box F_1 & F_1\ \mathcal{Z}\ F_2 = F_1\ \mathcal{S}\ F_2 \vee \boxdot F_1
\end{array}$$

Figure 1. Semantic definitions for LTL

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:

$\underline{\text{min}}$ `Next`($\underline{\text{Form}}$ F) $= \bigcirc F$
$\underline{\text{max}}$ `Always`($\underline{\text{Form}}$ F) $= F \wedge \bigcirc$`Always`(F)
$\underline{\text{min}}$ `Sometime`($\underline{\text{Form}}$ F) $= F \vee \bigcirc$`Sometime`(F)
$\underline{\text{min}}$ `Until`($\underline{\text{Form}}$ F_1, $\underline{\text{Form}}$ F_2) $= F_2 \vee (F_1 \wedge \bigcirc$`Until`($F_1, F_2$))
$\underline{\text{max}}$ `Unless`($\underline{\text{Form}}$ F_1, $\underline{\text{Form}}$ F_2) $= F_2 \vee (F_1 \wedge \bigcirc$`Unless`($F_1, F_2$))

The past time operators of LTL give rise to the following rules.

$\underline{\text{min}}$ `Previous`($\underline{\text{Form}}$ F) $= \odot F$
$\underline{\text{max}}$ `AlwaysPast`($\underline{\text{Form}}$ F) $= F \wedge \odot$`AlwaysPast`(F))
$\underline{\text{min}}$ `SometimePast`($\underline{\text{Form}}$ F) $= F \vee \odot$`SometimePast`(F))
$\underline{\text{min}}$ `Since`($\underline{\text{Form}}$ F_1, $\underline{\text{Form}}$ F_2) $= F_2 \vee (F_1 \wedge \odot$`Since`($F_1, F_2$))
$\underline{\text{max}}$ `Zince`($\underline{\text{Form}}$ F_1, $\underline{\text{Form}}$ F_2) $= F_2 \vee (F_1 \wedge \odot$`Zince`($F_1, F_2$))

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 EAGLE 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.

Correctness of Embedding:

To justify the above EAGLE definitions of LTL temporal operators, we can define an embedding function *Embed* :

$LTL \rightarrow$ EAGLE that maps $\bigcirc F$ to $\texttt{Next}(\mathit{Embed}(F))$, $\Box F$ to $\texttt{Always}(\mathit{Embed}(F))$, etc., and then formally establish that $\sigma, i \models_{LTL} F$ iff $\sigma, i \models_{\text{EAGLE}} \mathit{Embed}(F)$ for all traces σ and indices i. The proof follows by induction over the structure of the formula F; insufficient space allows for its inclusion, but see [4].

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. 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 $\mathit{value(eval(\ldots eval(eval(}F, s_1), s_2) \ldots s_n))$ is $\underline{\text{true}}$. The definition of the functions *eval*, *update* and *value* forms the calculus of the LTL subset of EAGLE.

3.1 Calculus

The *eval*, *update* and *value* functions are defined a priori for all operators, which is not possible for fully general EAGLE [5]. We do not define the functions on the previous operator $\odot$, since this operator is eliminated in the calculus. The definition of *eval*, *update* and *value* on the different primitive EAGLE operators is given in Figure 2. In the given 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 $\underline{\text{false}}$.

$$\mathit{value}(\texttt{R}(F_1, \ldots, F_n)) = \underline{\text{true}} \text{ if } \texttt{R} \text{ is } \underline{\text{max}}$$
$$\mathit{value}(\texttt{R}(F_1, \ldots, F_n)) = \underline{\text{false}} \text{ if } \texttt{R} \text{ is } \underline{\text{min}}$$

Future Time Operators

Consider the `Always` operator:

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

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

Figure 2. *eval*, *value* **and** *update* **definitions**

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

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

Similarly we can give the calculus for the other future time LTL operators as follows:

$$\mathit{eval}(\texttt{Next}(F), s) = \mathit{eval}(\bigcirc F, s)$$
$$\mathit{update}(\texttt{Next}(F), s) = \texttt{Next}(\mathit{update}(F, s))$$

$$\mathit{eval}(\texttt{Sometime}(F), s) = \mathit{eval}(F \vee \bigcirc\texttt{Sometime}(F), s)$$
$$\mathit{update}(\texttt{Sometime}(F), s) = \texttt{Sometime}(\mathit{update}(F, s))$$

$$\mathit{eval}(\texttt{Until}(F_1, F_2), s) = \mathit{eval}(F_2 \vee (F_1 \wedge \bigcirc\texttt{Until}(F_1, F_2)), s)$$
$$\mathit{update}(\texttt{Until}(F_1, F_2), s) = \texttt{Until}(\mathit{update}(F_1, s), \mathit{update}(F_2, s))$$

$$\mathit{eval}(\texttt{Unless}(F_1, F_2), s) = \mathit{eval}(F_2 \vee (F_1 \wedge \bigcirc\texttt{Unless}(F_1, F_2)), s)$$
$$\mathit{update}(\texttt{Unless}(F_1, F_2), s) = \texttt{Unless}(\mathit{update}(F_1, s), \mathit{update}(F_2, s))$$

Past Time 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 must 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.

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

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

$$\begin{array}{l}\texttt{AlwaysPast}(F) = \texttt{AlwaysPast}'(F,\underline{\text{true}})\\ \textit{eval}(\texttt{AlwaysPast}'(F,\textit{past}_1),s) = \textit{eval}(F \wedge \textit{past}_1,s)\\ \textit{update}(\texttt{AlwaysPast}'(F,\textit{past}_1),s) =\\ \quad \texttt{AlwaysPast}'(\textit{update}(F,s),\textit{eval}(\texttt{AlwaysPast}'(F,\textit{past}_1),s))\end{array}$$

Here, in *eval*, the subformula $\odot\texttt{AlwaysPast}(F)$ guarded by the previous operator is replaced by the argument $\textit{past}_1$ 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 $\texttt{AlwaysPast}'(F,\textit{past}_1)$ and pass it as second argument of $\texttt{AlwaysPast}'$. Thus in the next state $\textit{past}_1$ is bound to $\odot\texttt{AlwaysPast}'(F,\textit{past}_1)$. Note that in the definition of $\texttt{AlwaysPast}'$ we pass $\underline{\text{true}}$ as the second argument. This is because, $\texttt{AlwaysPast}$ 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:

$$\begin{array}{l}\texttt{Previous}(F) = \texttt{Previous}'(F,\underline{\text{false}})\\ \textit{eval}(\texttt{Previous}'(F,\textit{past}_1),s) = \textit{eval}(\textit{past}_1,s)\\ \textit{update}(\texttt{Previous}'(F,\textit{past}_1),s) =\\ \quad \texttt{Previous}'(\textit{update}(F,s),\textit{eval}(F,s))\end{array}$$

$$\begin{array}{l}\texttt{SometimePast}(F) = \texttt{SometimePast}'(F,\underline{\text{false}})\\ \textit{eval}(\texttt{SometimePast}'(F,\textit{past}_1),s) = \textit{eval}(F \vee \textit{past}_1,s)\\ \textit{update}(\texttt{SometimePast}'(F,\textit{past}_1),s) =\\ \quad \texttt{SometimePast}'(\textit{update}(F,s),\textit{eval}(\texttt{SometimePast}'(F,\textit{past}_1),s))\end{array}$$

$$\begin{array}{l}\texttt{Since}(F_1,F_2) = \texttt{Since}'(F_1,F_2,\underline{\text{false}})\\ \textit{eval}(\texttt{Since}'(F_1,F_2,\textit{past}_1),s) = \textit{eval}(F_2 \vee (F_1 \wedge \textit{past}_1),s)\\ \textit{update}(\texttt{Since}'(F_1,F_2,\textit{past}_1),s) =\\ \quad \texttt{Since}'(\textit{update}(F_1,s),\textit{update}(F_1,s),\textit{eval}(\texttt{Since}'(F_1,F_2,\textit{past}_1),s))\end{array}$$

$$\begin{array}{l}\texttt{Zince}(F_1,F_2) = \texttt{Zince}'(F_1,F_2,\underline{\text{true}})\\ \textit{eval}(\texttt{Zince}'(F_1,F_2,\textit{past}_1),s) = \textit{eval}(F_2 \vee (F_1 \wedge \textit{past}_1),s)\\ \textit{update}(\texttt{Zince}'(F_1,F_2,\textit{past}_1),s) =\\ \quad \texttt{Zince}'(\textit{update}(F_1,s),\textit{update}(F_1,s),\textit{eval}(\texttt{Zince}'(F_1,F_2,\textit{past}_1),s))\end{array}$$

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

$$\begin{array}{l}\textit{value}(\texttt{Always}(F)) = \textit{value}(\texttt{AlwaysPast}'(F,\textit{past}_1))\\ = \textit{value}(\texttt{Unless}(F_1,F_2)) = \textit{value}(\texttt{Zince}'(F_1,F_2,\textit{past}_1)) = \underline{\text{true}}\end{array}$$

$$\begin{array}{l}\textit{value}(\texttt{Sometime}(F)) = \textit{value}(\texttt{SometimePast}'(F,\textit{past}_1))\\ = \textit{value}(\texttt{Until}(F_1,F_2)) = \textit{value}(\texttt{Since}'(F_1,F_2,\textit{past}_1)) = \underline{\text{false}}\end{array}$$

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. So, we can't use the operator $\odot$ when writing an LTL formula; instead we use the rule $\texttt{Previous}$ 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:

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

Insufficient space prohibits inclusion of the proof.

4 Implementation and Complexity

We have implemented in Java the EAGLE monitoring framework. In order to make the implementation efficient we use the decision procedure of Hsiang [13]. 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.

<u>`simplify:`</u>

$$\begin{array}{ll}\text{true} \wedge \phi = \phi & \text{false} \wedge \phi = \text{false}\\ \phi \wedge \phi = \phi & \phi_1 \vee \phi_2 = (\phi_1 \wedge \phi_2) \oplus \phi_1 \oplus \phi_2\\ \text{false} \oplus \phi = \phi & \phi_1 \rightarrow \phi_2 = \text{true} \oplus \phi_1 \oplus (\phi_1 \wedge \phi_2)\\ \phi \oplus \phi = \text{false} & \phi_1 \equiv \phi_2 = \text{true} \oplus \phi_1 \oplus \phi_2\\ \neg\phi = \text{true} \oplus \phi & \phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3)\end{array}$$

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 set of 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.Moreover, after a series of applications of *eval* on the states $s_1, s_2, ..., s_n$, the conjuncts in the normal form $\textit{eval}(\ldots\textit{eval}(\textit{eval}(\phi,s_1),s_2)\ldots,s_n)$ 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.[1] Therefore, one

[1] 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.

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. 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`. Note that the application of `simplify` on the conjunction of two formulas 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 $O(m^2 2^m \log m) \cdot O(m^2 2^m \log m)$ 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.*

5 Examples and Experiments

This section illustrates the use of Eagle on two concurrency-related applications – detection of deadlock potentials and testing of a real-time concurrent system.

5.1 Using Eagle for Deadlock Detection

We present an example that illustrates the use of EAGLE to detect a simple class of cyclic deadlocks. Specifically EAGLE monitors an event stream of lock acquisitions and releases, and reports any cyclic lock dependencies. If there are two threads t_1 and t_2 such that t_1 takes lock l_1, and then prior to releasing l_1, takes lock l_2, and furthermore if t_2 takes lock l_2, and then prior to releasing l_2, takes lock l_1, then there is a *cyclic lock dependency* that indicates the possibility of deadlock. This is a simplification of the general dining philosopher problem, restricted to cycles of length two.

We present two implementations. One illustrates how EAGLE integrates with Java, allowing one to intermix algorithms written in a general programming language with EAGLE monitors. The other is a "pure" solution that just uses EAGLE rules. Each solution utilizes the ability of EAGLE to parameterize rules with data values as well as formulas.

For both implementations the state observed by EAGLE contains three integer variables that get updated each time a new lock or release event is sent to the observer. Let `s` be the object representing the observer state. The variable `s.type` is set to 1 if the event is a lock event and 2 if it is a release event. `s.thread` is an integer which uniquely identifies the thread and `s.lock` uniquely identifies the lock. For clarity we define predicates `s.lock()` and `s.release()` that test whether `s.type` is set to 1 or 2, respectively. We first present the pure solution.

$$
\begin{array}{l}
\underline{\text{min}}\ \texttt{Conflict}(\underline{\text{int}}\ t, \underline{\text{int}}\ l_1, \underline{\text{int}}\ l_2) = \\
\quad \texttt{Until}(\neg(s.release() \wedge s.thread = t \wedge s.lock = l_2), \\
\qquad s.lock() \wedge s.thread = t \wedge s.lock = l_1) \\
\underline{\text{min}}\ \texttt{ConflictLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l_1, \underline{\text{int}}\ l_2) = s.lock() \wedge \\
\quad s.thread \neq t \wedge s.lock = l_2 \wedge \texttt{Conflict}(s.thread, l_1, l_2) \\
\underline{\text{min}}\ \texttt{NestedLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = \\
\quad \texttt{Until}(\neg(s.release() \wedge s.thread = t \wedge s.lock = l), \\
\qquad s.lock() \wedge s.thread = t \wedge s.lock \neq l \wedge \\
\qquad (\bigcirc \texttt{Sometime}(\texttt{ConflictLock}(t, l, s.lock)) \vee \\
\qquad \odot\, \texttt{SometimePast}(\texttt{ConflictLock}(t, l, s.lock)))) \\
\underline{\text{mon}}\ M = \neg \texttt{Sometime}(s.lock() \wedge \\
\qquad\qquad \texttt{NestedLock}(s.thread, s.lock))
\end{array}
$$

The intuition is that the Sometime in monitor M is satisfied in a state where a lock is taken that is the "first" of the four locks in the pattern described above. The thread and the lock value of that lock are passed as data parameters to *NestedLock* which "searches" for a subsequent lock taken by that thread prior to the release of the first lock. If such a second lock is found, it binds the data value of the second lock to a data parameter and searches both forward and backward through the trace with *ConflictLock* for a second thread that takes the two locks in reverse order.

The second implementation uses a set data structure within the observer state that holds triples of values of the form $[t, l_1, l_2]$ recording that thread t took nested locks l_1 and then l_2. The predicate *addTriple* inserts such a triple into the set and evaluates to true if there is no conflicting triple in the set. A conflicting triple is one of the form $[t_2, l_2, l_1]$ for $t_2 \neq t$.

$$
\begin{aligned}
&\underline{\text{max}}\ \texttt{DiffLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = s.lock() \wedge s.thread = t \wedge s.lock \neq l \\
&\underline{\text{max}}\ \texttt{CheckLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = s.lock() \wedge s.thread = t \wedge \\
&\qquad\qquad s.lock \neq l \wedge s.addTriple(t, l, s.lock) \\
&\underline{\text{max}}\ \texttt{Release}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = s.release() \wedge s.thread = t \wedge s.lock = l \\
&\underline{\text{min}}\ \texttt{NestedDiffLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = \\
&\qquad\qquad \texttt{Until}(\neg\texttt{Release}(t,l), \texttt{DiffLock}(t,l)) \\
&\underline{\text{min}}\ \texttt{NestedCheckLock}(\underline{\text{int}}\ t, \underline{\text{int}}\ l) = \\
&\qquad\qquad \texttt{Until}(\neg\texttt{Release}(t,l), \texttt{CheckLock}(t,l)) \\
&\underline{\text{mon}}\ M = \texttt{Always}((s.lock() \wedge \texttt{NestedDiffLock}(s.thread, s.lock)) \\
&\qquad\qquad \rightarrow \texttt{NestedCheckLock}(s.thread, s.lock)
\end{aligned}
$$

The monitor identifies a first lock and the rule *NestedDiffLock* returns true if a second, nested, lock is taken. If so, *NestedCheckLock* adds the triple to the set and returns false if a conflict exists.

5.2 Testing a Planetary Rover

The EAGLE logic has been applied in the testing of a planetary rover controller, as part of an ongoing collaborative effort with other colleagues (see [2]) to create a fully automated test-case generation and execution environment for this application. The controller consists of 35,000 lines of C++ code and is implemented as a multi-threaded system, where synchronization between threads is performed through shared variables, mutexes and condition variables. The controller operates a rover, named K9, which essentially is a small car/robot on wheels. K9 itself is a prototype, and serves to form the basis of experiments with rover missions on Mars. The controller executes plans given as input. A plan is a tree-like structure of actions and sub-actions. The leaf-actions control the rover hardware components. Each action is optionally associated with time constraints indicating when it should start and when it should terminate. Figure 3 presents an example input plan. The plan is named P and consists of two sub-tasks T1 and T2, which

```
(block
  :id P
  :node-list (
     (task
       :id T1
       :start-temporal-conditions (( P start (1 5)))
       :end-temporal-conditions (( T1 start (1 30)))
     )
     (task
       :id T2
       :start-temporal-conditions (( T1 end (10 20)))
     )
  )
)
```

Figure 3. Example plan

are supposed to be executed sequentially in the given order. The plan specifies that T1 should start 1-5 seconds after P starts and should end 1-30 seconds after T1 starts. Task T2 should start 10-20 seconds after T1 ends. The controller has been hand-instrumented in a few places to generate an execution trace when executed. An example execution trace of the plan in Figure 3 is presented below:

```
start P 397
start T1 1407
success T1 2440
start T2 14070
success T2 15200
success P 15360
```

In addition to information about start and (successful or failing) termination, each event in the trace is associated with a time-stamp in milliseconds since the start if the application. The testing environment, named X9 (explorer of K9), contains a test-case generator, that automatically generates input plans for the controller from a grammar describing the structure of plans. A model checker extended with symbolic execution is used to generate the plans [14]. Additionally, for each input plan a set of temporal formulas is generated, that the execution trace obtained by executing that plan should satisfy. The controller is executed on each generated plan, and the implementation of EAGLE is used to monitor that the generated execution trace satisfies the formulas generated for that particular plan. The properties generated for the plan in Figure 3 are presented in Figure 4, and should be self-explainable.

X9 was evaluated by seeding errors in the rover controller. One error had to do with the closeness in time between termination of one task and the start of the successor. If a task T_1 ended in a particular time range (after the start time of the successor T_2), then task T_2 would wrongly fail rather than execute. Running X9 detected this problem immediately. Note that the property violated was binary/propositional in nature: a task failed that should have succeeded.

```
mon M0 = Sometime({s.start("P")}) .
mon M1 = Always({s.start("P")} ->
           (Sometime({s.success("P")}
            \/ {s.fail("P")}))) .
mon M2 = Always({s.start("P")} ->
          Sometime({s.start("T1")})) .
mon M3 = Always({s.success("T2")} ->
          Sometime({s.success("P")}))) .
mon M4 = Always({s.start("T1")} ->
            (Sometime({s.success("T1")})
             \/ Sometime({s.fail("T1)})))) .
mon M5 = Always({s.fail("T1")} ->
          not Sometime({s.start("T2")}))) .
mon M6 = Always({s.success("T1")} ->
          Sometime({s.start("T2")})) .
mon M7 = Always({s.start("T2")} ->
           (Sometime({s.success("T2")}
            \/ {s.fail("T2")}))) .
```

Figure 4. Generated properties

EAGLE allows for the formulation of real-time properties that take the time stamps into account. Such an experiment is mentioned in [5]. In that experiment a real unknown bug was located. It was discovered that the application did not check lower bounds on durations, whereas it should. That is, if a task finished before it was supposed to, the task should fail, but it wrongly succeeded. The bug was not immediately corrected, and later showed up during a field test of the rover.

6 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)*, volume 2589 of *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. EAGLE does Space Efficient LTL Monitoring. Pre-Print CSPP-25, University of Manchester, Department of Computer Science, October 2003. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp25.pdf.

[5] 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.

[6] 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.

[7] D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.

[8] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of Runtime Verification (RV'02)* [1], pages 36–55.

[9] B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of Runtime Verification (RV'01)* [1], pages 44–60.

[10] 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.

[11] 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.

[12] 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.

[13] J. Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.

[14] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of TACAS 2003. Warsaw, Poland.*, April 2003.

[15] 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.

[16] 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.

[17] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[18] K. Sen, G. Roşu, and G. Agha. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIAN'03) (To appear in LNCS)*, December 2003.