# Rule-Based Runtime Verification

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 present a rule-based framework for defining and implementing finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, real-time logics, interval logics, forms of quantified temporal logics, and so on. Our logic, EAGLE, is implemented as a Java library and involves novel techniques for rule definition, manipulation and execution. Monitoring is done on a state-by-state basis, without storing the execution trace.

## 1 Introduction

Runtime verification, or runtime monitoring, comprises having a software module, an observer, monitor the execution of a program, and check its conformity with a requirement specification, often written in a temporal logic or as a state machine. 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, potentially steering the application back to a safety region if a property is violated. It is highly scalable. Several runtime verification systems have been developed, of which some were presented at three recent international workshops on runtime verification [1].

Linear temporal logic (LTL) [16] has been core to several of these attempts. The commercial tool Temporal Rover (TR) [4, 5] 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 [7], and a specialized LTL collecting statistics along the execution trace is described in [6]. The MAC logic [15] 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 [17] 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 large variety of logics prompted us to search for a small and general framework for defining monitoring logics, which would be powerful enough to capture essentially

* 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.

all of the above described logics, hence supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints, and statistics. The framework should support the definition of new logics in an easy manner and should support the monitoring of programs with their complex program states. The result of our search is the logic EAGLE presented in this paper. The EAGLE logic and its implementation for run-time monitoring has in particular been significantly influenced by earlier work of Barringer et al., see for example [3], on the executable temporal logic METATEM. A linear-time temporal formula can be separated [8] into a boolean combination of pure past, present and pure future time formulas - in particular, the combination can be written as a collection of "directly executable" global conditional rules of the form "**if** pure past-time **then** present-time and pure future-time". The present-time, or state, formulas determine how the state for the current moment in time is built and the pure future-time formulas yield obligations that need to be fulfilled at some time later. The separation result, rules and future obligations are central in our current work. However, the fundamental difference between METATEM and EAGLE is that the METATEM interpreter builds traces state by state, whereas EAGLE is used for checking given finite traces: costly implementation features, such as backtracking and loop-checking, are not required.

We recently discovered parallel work [14] using recursive equations to implement a real-time logic. However we had already developed the ideas further. We provide the language of recursive equations to the user, we support a mixture of future time and past time operators, we treat real-time as a special case of data values, and hence we allow a very general logic for reasoning about data, including the possibility of relating data values across the execution trace, both forwards and backwards.

The paper is structured as follows. Section 2 introduces our logic framework, then in section 3 we discuss the algorithm and calculus that underlies our implementation, which is then briefly described along with initial experimentation in section 4.

## 2 The Logic

In this section we introduce our temporal finite trace monitoring logic EAGLE. The logic 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 next-time and previous-time operators can be used for defining future time respectively past time temporal logics on top of EAGLE. The concatenation operator can be used to define interval logics and an extended regular expression language. Rules are parameterized to allow for reasoning about data values, including real-time. Atomic propositions are boolean expressions over a program state, Java states in the current implementation. The logic is first introduced informally through two examples whereafter its syntax and semantics is given. Finally, its relationship to some other important logics is outlined.

### 2.1 EAGLE by Example

**Fundamental Concepts** Assume we want to state a property about a program $P$, which contains the declaration of two integer variables $x$ and $y$. We want to state that whenever $x$ is positive then eventually $y$ becomes positive. The property can be written as follows in classical future time LTL: $\Box(x > 0 \rightarrow \Diamond y > 0)$. The formulas $\Box F$ (always $F$) and $\Diamond F$ (eventually $F$), for some property $F$, usually satisfy the following equivalences, where the temporal operator $\bigcirc F$ stands for *next* $F$ (meaning '*in next state* $F$'):

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

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 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, and the formula to be monitored ($M_1$):

$$\begin{array}{l}
\textbf{max}\ \texttt{Always}(\textbf{Form}\ F) = F \wedge \bigcirc\texttt{Always}(F) \\
\textbf{min}\ \texttt{Eventually}(\textbf{Form}\ F) = F \vee \bigcirc\texttt{Eventually}(F) \\
\textbf{mon}\ M_1 = \texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))
\end{array}$$

The Always operator is defined as a maximal fix-point operator; the Eventually operator is defined as a minimal fix-point operator. Maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually happens). For us, the 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. Assume for example that we evaluate the formula $M_1$ in a state where $x > 0$ and $y \leq 0$, then as a liveness obligation for the future we will have the expression:

$$\bigcirc\texttt{Eventually}(y > 0) \wedge \bigcirc\texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))$$

Assume that we at this point detect the end of the trace; that is: we are beyond the last state. The outstanding liveness obligation Eventually$(y > 0)$ has not yet been fulfilled, which is an error. This is captured by the evaluation of the minimal fix-point combinator Eventually to false at this point. The remaining other obligation from the $\wedge$-formula, namely, Always$(x > 0 \rightarrow$ Eventually$(y > 0))$, is a safety property and evaluates to true.

For completeness we provide remaining definitions of the future time LTL operators $\mathcal{U}$ (until) and $\mathcal{W}$ (unless) below. Note how $\mathcal{W}$ is defined in terms of other operators. However, it could have been defined recursively.

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

**Data Parameters** We have seen how rules can be parameterized with formulas. Let's complicate the example with data parameters. Suppose we want to state the property:

*"whenever at some point $k = x > 0$ for some k, then eventually $y == k$"*. This can be stated as follows in quantified LTL: $\Box(x > 0 \rightarrow \exists k.(k = x \wedge \Diamond y = k))$. We use parameterized rules to state this property, capturing the value of $x$ when $x > 0$ as a rule parameter.

$$\textbf{min}\, R(\textbf{int}\ k) = \texttt{Eventually}(s.y == k) \qquad \textbf{mon}\, M_2 = \texttt{Always}(s.x > 0 \rightarrow R(s.x))$$

Rule $R$ is parameterized with an integer $k$, and is instantiated in $M_2$ when $x > 0$, hence capturing the value of $x$ at that moment. Rule $R$ replaces the existential quantifier. The logic also provides a previous-time operator, which allows us to define past time operators; the data parametrization works uniformly for rules over past as well as future, which is non-trivial to achieve since the implementation does not store the trace, see Section 4. Data parametrization is also used to elegantly model real-time logics.

## 2.2 Syntax and Semantics

**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 to be monitored. Rules and monitors are named ($N$).

$$\begin{array}{lll}
S & ::= & \textbf{dec}\ D\ \textbf{obs}\ O \\
D & ::= & R^* \\
O & ::= & M^* \\
R & ::= & \{\textbf{max}\ \mid \textbf{min}\ \}\ N(T_1\ x_1, \ldots, T_n\ x_n) = F \\
M & ::= & N = F \\
T & ::= & \textbf{Form}\ \mid \textit{java primitive type} \\
F & ::= & \textit{java expression} \mid \textbf{True} \mid \textbf{False} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
 & & \bigcirc F \mid \quad F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n)
\end{array}$$

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 Java type, such as **int**, **long**, **float**, etc.. The body of a rule/monitor is a formula of the syntactic category *Form* (with meta-variables $F$, etc.). The propositions of this logic are Java expressions over an observer state. These can be arbitrary Java expressions using all of Java's expression language constructs, recommended not to have no side effects. Formulas are composed using standard propositional logic operators together with a next-state operator ($\bigcirc F$), a previous-state operator ( $F$), 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 exception that if an argument is a java expression, it must be of boolean type.

**Semantics** The semantics of the logic is defined in terms of a satisfaction relation $\models \subseteq \textit{Trace} \times \textit{Form}$ between execution traces and specifications. An execution trace $\sigma$ 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 $\sigma$ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of $\sigma$ from position $i$ to position $j$, both positions included. In the implementation a state is a user defined Java object that is updated through a user provided *update* method for each new event generated by the program. Given a trace $\sigma$ and a specification **dec** D **obs** O, satisfaction is defined as follows:

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

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 $\models_D \subseteq (\mathit{Trace} \times \textbf{nat}) \times \mathit{Form}$, for a set of rule definitions $D$, is presented below, where $0 \leq i \leq n+1$ 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.

$$\begin{array}{ll}
\sigma, i \models_D \mathit{jexp} & \text{iff } 1 \leq i \leq |\sigma| \text{ and } \mathit{evaluate(jexp)}(\sigma(i)) == \mathit{true} \\
\sigma, i \models_D \textbf{True} & \\
\sigma, i \not\models_D \textbf{False} & \\
\sigma, i \models_D \neg F & \text{iff } \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff } \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff } \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff } \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff } i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \quad F & \text{iff } 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff } \exists j \geq i \text{ s.t. } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(F_1, \ldots, F_m) & \text{iff } \left\{ \begin{array}{l}
\text{if } 1 \leq i \leq |\sigma| \text{ then:} \\
\quad \sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_n \mapsto F_n] \\
\quad \text{where } (N(T_1\, x_1, \ldots, T_n\, x_n) = F) \in D \\
\text{otherwise, if } i = 0 \text{ or } i = |\sigma| + 1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as } \textbf{max} \text{ in } D
\end{array} \right.
\end{array}$$

A Java expression (a proposition) is evaluated in the current state in case the position $i$ is within the trace ($1 \leq i \leq n$). In the boundary cases ($i = 0$ and $i = n+1$) Java expressions evaluate to false. Propositional operators have their standard 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 $\sigma$ can be split into two sub-traces $\sigma = \sigma_1 \sigma_2$, such that $F_1$ is true on $\sigma_1$, observed from the current position $i$, and $F_2$ is true on $\sigma_2$ (ignoring $\sigma_1$, 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 Relationship to Other Logics

The logical system defined above is expressively rich; indeed, any linear-time temporal logic, whose temporal modalities can be recursively defined over the next, past or concatenation modalities, can be embedded within it. Furthermore, since in effect we have a limited form of quantification over possibly infinite data sets, and concatenation, we are strictly more expressive than, say, a linear temporal fixed point logic (over next and previous). A formal characterization of the logic is beyond the scope of this paper, however, we demonstrate the logic's utility and expressiveness through examples.

**Past Time LTL:** A past time linear temporal logic, i.e. one whose temporal modalities only look to the past, could be defined in the mirror way to the future time logic exemplified in the introduction by using the built-in previous modality, , in place of the future next time modality, $\bigcirc$. Here, however, we present the definitions in a more hierarchic (and logical) fashion. Note that the `Zince` rule defines the past-time correspondent to the future time unless, or weak until, modality, i.e. it is a weak version of `Since`.

$$\begin{aligned}
&\textbf{min}\ \mathtt{Since}(\textbf{Form}\ F_1, \textbf{Form}\ F_2) = F_2 \vee (F_1 \wedge {}^{J}\ \mathtt{Since}(F_1, F_2))\\
&\textbf{min}\ \mathtt{EventuallyInPast}(\textbf{Form}\ F) = \mathtt{Since}(\ \textbf{True}\ , F)\\
&\textbf{max}\ \mathtt{AlwaysInPast}(\textbf{Form}\ F) = \neg\mathtt{EventuallyInPast}(\neg F)\\
&\textbf{max}\ \mathtt{Zince}(\textbf{Form}\ F_1, \textbf{Form}\ F_2) = \mathtt{Since}(F_1, F_2) \vee \mathtt{AlwaysInPast}(F_1)
\end{aligned}$$

**Combined Future and Past Time LTL:** By 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). We are thus able to express constraints such as if ever the variable $x$ exceeds 0, there was an earlier moment when the variable $y$ was 4 and then remains with that value until it gets increased sometime later, possibly after the moment when $x$ exceeds 0.

$$\textbf{mon}\ M_2 = \mathtt{Always}(x > 0 \rightarrow \mathtt{EventuallyInPast}(y == 4 \wedge \mathtt{Until}(y == 4, y > 4)))$$

**Extended LTL and $\mu$TL:** The ability to define temporal modalities recursively provides the ability to define Wolper's ETL or the semantically equivalent fixpoint temporal calculus. Such expressiveness is required to capture regular properties such as temporal formula $F$ is required to be true on every even moment of time:

$$\textbf{max}\ \mathtt{Even}(\textbf{Form}\ F) = F \wedge \bigcirc\bigcirc\mathtt{Even}(F)$$

The $\mu TL$ formula $\nu x.p \wedge \bigcirc\bigcirc x \wedge \mu y.q \wedge \bigcirc x \vee {}^{J}\ y$, where $p$ and $q$ are atomic formulas, would be denoted by the formula, $X()$, where rules $X$ and $Y$ are:

$$\textbf{max}\ \mathtt{X}() = p \wedge \bigcirc\bigcirc\mathtt{X}() \wedge \mathtt{Y}() \qquad \textbf{min}\ \mathtt{Y}() = q \wedge \bigcirc\mathtt{X}() \vee {}^{J}\ \mathtt{Y}()$$

**Extended Regular Expressions:** The language of Extended Regular Expressions (ERE), i.e. adding complementation to regular expressions, has been proposed as a powerful formalism for run-time monitoring. ERE can straightforwardly be embedded within our rule-based system. Given, $E ::= \emptyset|\varepsilon|a|E \cdot E|E + E|E \cap E|\neg E|E^*$, let $\mathbf{Tr}(E)$ denote the ERE $E$'s corresponding EAGLE formula. For convenience, we define the rule $\textbf{max}\ \mathtt{Empty}() = \neg\bigcirc\ \textbf{True}$ which is true only when evaluated on an empty (suffix) sequence. **Tr** is inductively defined as follows.

$$\begin{array}{llll}
\mathbf{Tr}(\emptyset) & = \mathbf{False} & \mathbf{Tr}(\varepsilon) & = \texttt{Empty()} \\
\mathbf{Tr}(a) & = a \wedge \bigcirc\texttt{Empty()} & \mathbf{Tr}(E_1 \cdot E_2) & = \mathbf{Tr}(E_1) \cdot \mathbf{Tr}(E_2) \\
\mathbf{Tr}(E_1 + E_2) & = \mathbf{Tr}(E_1) \vee \mathbf{Tr}(E_2) & \mathbf{Tr}(E_1 \cap E_2) & = \mathbf{Tr}(E_1) \wedge \mathbf{Tr}(E_2) \\
\mathbf{Tr}(\neg E) & = \neg\mathbf{Tr}(E) & & \\
\mathbf{Tr}(E^*) & \multicolumn{3}{l}{= X() \text{ where } \mathbf{max}\, X() = \texttt{Empty()} \mid (\mathbf{Tr}(E) \cdot X())}
\end{array}$$

**Real Time as a Special Case of Data Binding:** Metric temporal logics, in which temporal modalities are parameterized by some underlying real-time clock(s), can be straightforwardly embedded into our system through rule parameterization. For example, consider the metric temporal modality, $\diamond^{[t_1,t_2]}$ in a system with just one global clock. An absolute interpretation of $\diamond^{[t_1,t_2]}\phi$ has the formula true if and only if $\phi$ holds at some time in the future when the real-time clock has value within the interval $[t_1,t_2]$. For our context, we assume that the finite sequence of states being monitored contains a variable *clock* giving the real-time value of the clock for the associated state. The rule

$$\begin{array}{l}
\mathbf{min}\ \texttt{EventAbs}(\mathbf{Form}\ F, \mathbf{long}\ t_1, \mathbf{long}\ t_2) = \\
\quad clock <= t_2 \wedge (F \rightarrow t_1 <= clock) \wedge (\neg F \rightarrow \bigcirc\texttt{EventAbs}(F,t_1,t_2))
\end{array}$$

defines the operator $\diamond^{[t_1,t_2]}$ for absolute values of the clock. A relativized version of the modality can then be defined as:

$$\mathbf{min}\ \texttt{EventRel}(\mathbf{Form}\ F, \mathbf{long}\ t_1, \mathbf{long}\ t_2) = \texttt{EventAbs}(F, clock + t_1, clock + t_2))$$

**Counting and Statistical Calculations:** In a monitoring context, one may wish to gather statistics on the truth of some property, for example whether a particular state property $\phi$ holds with at least some probability $p$ over a given sequence, i.e. it doesn't fail with probability greater than $(1-p)$. Consider the operator $\Box_p\phi$ defined by:

$$\sigma, i \models \Box_p\phi \ \mathbf{iff}\ \exists S \subseteq \{i..|\sigma|\} \text{ s.t. } \frac{|S|}{|\sigma| - i} \geq p \wedge \forall j \in S \,.\, \sigma, j \models \phi$$

An encoding within our logic can then be given as:

$$\begin{array}{l}
\mathbf{min}\ \texttt{A}(\mathbf{Form}\ \phi, \mathbf{float}\ p, \mathbf{int}\ f, \mathbf{int}\ t) = \\
\quad (\bigcirc\texttt{Empty()} \wedge ((\phi \wedge (1 - \frac{f}{t}) >= p) \vee (\neg\phi \wedge (1 - \frac{f+1}{t}) >= p))) \vee \\
\quad (\neg\texttt{Empty()} \wedge ((\phi \rightarrow \bigcirc\texttt{A}(\phi,p,f,t+1)) \wedge (\neg\phi \rightarrow \bigcirc\texttt{A}(\phi,p,f+1,t+1)))) \\
\mathbf{min}\ \texttt{AtLeast}(\mathbf{Form}\ \phi, \mathbf{float}\ p) = \texttt{A}(\phi,p,0,1)
\end{array}$$

**Towards Context Free:** Above we showed that EAGLE could encode logics such as ETL, which extend LTL with regular grammars (when restricted to finite traces), or even extended regular expressions. In fact, we can go beyond regularity into the world of context-free languages, necessary, for example, to express properties such as every login is matched by a logout and at no point are there more logouts than logins. Indeed, such a property can be expressed in several ways in EAGLE. Assume we are monitoring a sequence of *login* and *logout* events. We can define a rule $\texttt{Match}(\mathbf{Form}\ F_1, \mathbf{Form}\ F_2)$ and monitor with $\texttt{Match}(login, logout)$ where:

$$\mathbf{min}\ \texttt{Match}(\mathbf{Form}\ F_1, \mathbf{Form}\ F_2) = F_1 \cdot \texttt{Match}(F_1,F_2) \cdot F_2 \cdot \texttt{Match}(F_1,F_2)\ \vee\ \texttt{Empty()}$$

Less elegantly, and which we leave as an exercise, one could use the rule parametrization mechanism to count the numbers of logins and logouts.

## 3 Algorithm

In this section, we now outline the computation mechanism used to determine whether a given monitoring formula holds for some given input sequence of events. On the observer side a local state is maintained. The *atomic propositions* are specified with respect to the variables in this local state. At every event the observer modifies the local state of the observer 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. The evaluation of a formula $F$ on a state $s = \sigma(i)$ in a trace $\sigma$ results in an another formula $eval(F, s)$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models eval(F, s)$. The definition of the operator $eval : Form \times State \longrightarrow Form$ uses another auxiliary operator $update : Form \times State \longrightarrow Form$. The intuition behind using the operator *update* is to update a formula properly in presence of previous operators. The value of a formula $F$ at the end of a trace is given by $value(F)$. The operator $value : Form \longrightarrow \{\textbf{True}, \textbf{False}\}$ returns **True** if the formula is satisfied by an empty trace and returns **False** otherwise. The definition of the operators *eval*, *update* and *value* forms the calculus of the recursive rule-based framework. We define this calculus next.

### 3.1 Calculus

The *eval*, *update* and *value* operators are defined a priori for all operators except for the previous operator and rule application. The definitions of *eval*, *update* and *value* for the rule application get generated based on the definition of rules in the specification. We do not define the functions on the previous operator, since this operator is eliminated in the translations we perform before we apply the rules. The definition of *eval*, *update* and *value* on the different operators is given below.

$$\begin{aligned}
eval(jexp, s) &= \text{ value of jexp in } s \\
eval(F_1 \; op \; F_2, s) &= eval(F_1, s) \; op \; eval(F_2, s) \text{ where op } \in \{\wedge, \vee, \rightarrow\} \\
eval(\neg F, s) &= \neg eval(F, s) \\
eval(\bigcirc F, s) &= update(F, s)
\end{aligned}$$

$$\begin{aligned}
update(jexp, s) &= jexp \\
update(F_1 \; op \; F_2, s) &= update(F_1, s) \; op \; update(F_2, s) \text{ where op } \in \{\wedge, \vee, \rightarrow\} \\
update(\neg F, s) &= \neg update(F, s) \\
update(\bigcirc F, s) &= \bigcirc update(F, s)
\end{aligned}$$

$$\begin{aligned}
value(jexp, s) &= \textbf{ False} \\
value(F_1 \; op \; F_2, s) &= value(F_1, s) \; op \; value(F_2, s) \text{ where op } \in \{\wedge, \vee, \rightarrow\} \\
value(\neg F, s) &= \neg value(F, s) \\
value(\bigcirc F, s) &= \textbf{ False}
\end{aligned}$$

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, $value(\bigcirc F)$ is **False** as the operator $\bigcirc$ is assumed to have strong interpretation in the logic. The *value* of a **max** rule is **True** and that of a **min** rule is **False**.

$$value(\mathtt{R}(F_1,\ldots,F_n),s) = \textbf{True} \text{ if } \mathtt{R} \text{ is } \textbf{max}$$
$$value(\mathtt{R}(F_1,\ldots,F_n),s) = \textbf{False} \text{ if } \mathtt{R} \text{ is } \textbf{min}$$

However, the definition of the *eval* and *update* operators for the rules are not generic for all rules. They are synthesized according to the definition of the rules in the specification. We call this algorithm the monitor synthesis algorithm. We describe the algorithm informally through examples. Consider the following rule.

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

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

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

However, the definition of *update* results in infinite recursion. To break 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 $F$. Thus the new definition of *update* becomes:

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

If the 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 with the following example.

$$\textbf{max}\ \mathtt{R}(\textbf{Form}\ F_1, \textbf{Form}\ F_2) = F_1 \wedge {}^{J}\ (F_2 \vee \mathtt{R}(F_1,F_2))$$

For this rule we introduce another auxiliary rule $\mathtt{R}'$ which contains an extra argument corresponding to the formula ${}^{J}\ (F_2 \vee \mathtt{R}(F_1,F_2))$.

$\mathtt{R}(\textbf{Form}\ F_1,\textbf{Form}\ F_2) = \mathtt{R}'(F_1,F_2,value(F_2 \vee \mathtt{R}(F_1,F_2)))$
$eval(\mathtt{R}'(F_1,F_2,past_1),s) = eval(F_1 \wedge past_1,s)$
$update(\mathtt{R}'(F_1,F_2,past_1),s) = \mathtt{R}'(update(F_1,s),update(F_2,s),eval(F_2 \vee \mathtt{R}'(F_1,F_2,past_1),s))$

Here, in *eval*, the subformula ${}^{J}\ (F_2 \vee \mathtt{R}(F_1,F_2))$ 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 arguments $F_1$ and $F_2$ but also evaluate the subformula $F_2 \vee \mathtt{R}'(F_1,F_2,past_1)$ and pass it as third argument of $\mathtt{R}'$. Thus in the next state $past_1$ is bound to ${}^{J}\ (F_2 \vee \mathtt{R}'(F_1,F_2,past_1))$. The translation, however, does not work correctly in case of *update* for a rule whose right hand side contains other rules. For example let us consider the rule:

$$\textbf{max}\ \mathtt{R}_2(\textbf{Form}\ F) = \mathtt{R}(F,F) \vee \bigcirc\mathtt{R}_2(F)$$

This rule contains the rule $\mathrm{R}$ on its right hand side. Now if we simply define the *update* operator for this rule as follows

$$update(\mathrm{R}_2(F), s) = \mathrm{R}_2(update(F, s))$$

then this definition is not entirely correct as it does not update the rule $\mathrm{R}(F, F)$ that contains past time operators. The ideal definition of *update* should be:

$$update(\mathrm{R}_2(F), s) = update(\mathrm{R}(F, F) \vee \bigcirc \mathrm{R}_2(F), s)$$

But, as before, this definition results in infinite recursion. This is resolved by introducing additional arguments as in the case of previous operators. For every rule that appears on the right hand side we introduce an argument and bind the rule to that argument. For example for $\mathrm{R}_2$ we define the *eval* and *update* as follows.

$$\begin{aligned}
&\mathrm{R}_2(F) = \mathrm{R}'_2(F, \mathrm{R}(F, F)) \\
&eval(\mathrm{R}'_2(F, inner_1), s) = eval(inner_1 \vee \bigcirc \mathrm{R}'_2(F, inner_1), s) \\
&update(\mathrm{R}'_2(F, inner_1), s) = \mathrm{R}'_2(update(F, s), update(inner_1, s))
\end{aligned}$$

In case the type of arguments passed to a rule are different from **Form** the definition of the operator *eval* changes for that rule. Before doing the eval on the right hand side of the rule we first evaluate, may be partially, the arguments whose types are not **Form**. Let us consider the following example.

$$\mathbf{max}\ \mathrm{R}_3(\mathbf{int}\ k) = (s.x == k) \vee \bigcirc \mathrm{R}_3(s.y + k)$$

The *eval* and *update* for the above rule are defined as follows.

$$\begin{aligned}
&eval(\mathrm{R}_3(arg_1), s) = \mathbf{let}\ k = eval(arg_1, s)\ \mathbf{in}\ eval((s.x == k) \vee \bigcirc \mathrm{R}_3(s.y + k), s) \\
&update(\mathrm{R}_3(arg_1), s) = \mathrm{R}_3(arg_1)
\end{aligned}$$

Here, the result of $eval(arg_1, s)$, where $arg_1$ is a java expression, may be a partially evaluated java expression if java expressions referred by some of the variables in $arg_1$ are partially evaluated. The java expression gets fully evaluated once the java expressions referred by all the variables are fully evaluated. Thus, we translate the rules in the specification to a set of definition of *eval* and *update* operators. Once we have this translation we can easily execute, or in other words, evaluate all the monitors at each state in a trace of a running program.

## 4 Implementation and Experiments

We have implemented this monitoring framework in Java. Currently it does not allow mutually recursive rules, however, this will be supported for the case where all the rules in the specification are purely future time. The implemented system works in two phases. First, it compiles the specification file to generate 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 runs on a trace; the engine dynamically loads the Java classes for rules at monitoring time.

Our implementation of propositional logic uses 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 which are exclusive or ($\oplus$) of conjunctions. The procedure is given below using equations that are shown to be Church-Rosser and terminating modulo associativity and commutativity.

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

The above ensures that the size of a formula is small. In the translational phase, a Java class is generated for each rule in the specification. The Java class contains a constructor, a `value` method, an `eval` method, and a `update` method corresponding to the *value*, *eval* and *update* operators in the calculus. The arguments are made fields in the class and they are initialized through the constructor. The choice of generating Java classes for each rule is for getting an efficient implementation. To handle partial evaluation we wrap every java expression in a Java class. Each of those classes contains a method `isAvailable()` that returns `true` whenever the java expression representing that class is fully evaluated and returns `false` otherwise. The class also stores, as fields, the different other java expression objects corresponding to the different variables (formula variables and state variables) that it uses in its java expression. Once all those java expressions are fully evaluated, the object for the java expression evaluates itself and any subsequent call of `isAvailable()` on this object returns `true`.

Once all the Java classes have been generated, the engine compiles all the generated Java classes, creates a list of monitors (which are also formulas) and starts monitoring all of them. During monitoring the engine takes the states from the trace, one by one, and evaluates the list of monitors on each to generate another list of formulas that become the new monitors for the next state. If at any point a monitor (a formula) becomes false an error message is generated and that monitor is removed from the list. At the end of a trace the value of each monitor is calculated and if false, a warning message for the particular monitor is generated. The details of the implementation are beyond the scope of the paper. However, interested readers can get the tool from the authors.

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, consists of a test-case generator, automatically generating input plans for the controller. Additionally, for each input plan a set of temporal formulas is generated that the plan execution should satisfy. The controller is executed on the generated plans and the implementation of EAGLE is used to monitor that execution traces satisfy the formulas. An unknown error was detected in the first run, demonstrating that a certain task did not recognize the too early termination of some other task.

## 5 Conclusion and Future Work

We have presented the succinct and powerful logic EAGLE, based on recursive parameterized rule definitions over three primitive temporal operators. We have indicated its power by expressing some other sophisticated logics in it. Initial experiments have been successful. Future work includes: optimizing the current implementation; supporting user-defined surface syntax; associating actions with formulas; and incorporating automated program instrumentation.

## 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. 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.
5. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.
6. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of Runtime Verification (RV'02)* [1], pages 36–55.
7. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of Runtime Verification (RV'01)* [1], pages 44–60.
8. D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification, Altrincham, April 1987*, volume 398 of *LNCS*, pages 409–448, 1989.
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. Jieh 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. K. Jelling Kristoffersen, C. Pedersen, and H. R. Andersen. Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems. In *Proceedings of Runtime Verification (RV'03)* [1], pages 146–161.
15. 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.
16. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
17. 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.