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 automatically evaluate 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) [19] has been core to several of these attempts. The commercial tool Temporal Rover (TR) [6, 7] 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 [9], and a specialized LTL collecting statistics along the execution trace is described in [8]. The MAC logic [18] 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 [20] has also been proposed and is argued to be more succinct for certain properties. The logic described in [16] 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 [14], using a rewriting system for monitoring future time logic [12, 13], or generating Büchi automata inspired algorithms adapted to finite trace LTL [11].

* 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 whilst at NASA Ames Research Center.

** 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.

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 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 runtime monitoring has been significantly influenced by earlier work on the executable, trace generating as well as trace checking, temporal logic METATEM [3]. In METATEM a linear-time temporal formula is separated [10] into a boolean combination of pure past, present and pure future time formulas. Conditioned by the past, 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 [17] 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.

This paper is structured as follows. Section 2 introduces our logic framework. 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. Further papers on EAGLE are available covering material that couldn't be covered in this paper. In [4], we illustrate that when EAGLE is specialized to a propositional LTL our monitoring algorithm is space efficient with an upper bound of $O(m^2 \log m 2^m)$, where m is the size of the monitored formula; in [5], we present full details of our current Java-based monitoring algorithm and its associated rewrite calculus.

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 can be parameterized with formulas, and with data to allow for the expression of data constraints, including real-time constraints. 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 for example show that $\Box F$ is a solution to the recursive equivalence $X \equiv F \wedge \bigcirc X$; in fact it is the maximal solution[4]. 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):

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

The `Always` operator is defined as having a maximal fix-point interpretation; the `Eventually` operator is defined as having a minimal interpretation. 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:

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

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

[4] Similarly, $\Diamond F$ is a *minimal* solution to the equivalence $X \equiv F \vee \bigcirc X$

However, it could have been defined recursively.

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

Data Parameters We have seen how rules can be parameterized with formulas. Let us modify the above example to include data parameters. Suppose we want to state the property: *"whenever at some point $x = k > 0$ for some k, then eventually $y = k$"*. This can be expressed as follows in quantified LTL: $\Box(x > 0 \rightarrow \exists k.(x = k \wedge \Diamond y = k))$. We use a parameterized rule to state this property, capturing the value of x when $x > 0$ as a rule parameter.

$$\underline{\text{min}}\ \texttt{R}(\underline{\text{int}}\ k) = \texttt{Eventually}(y = k) \qquad \underline{\text{mon}}\ M_2 = \texttt{Always}(x > 0 \rightarrow \texttt{R}(x))$$

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).

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

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.). Any recursive call on a rule must be strictly guarded by a temporal operator. The propositions of this logic are boolean expressions over an observer state. Formulas are composed using standard propositional logic operators 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 arguments must be type correct. That is, an argument of type Form can be any formula, with the restriction that if the argument is an expression, it must be of boolean type. An argument of a primitive type must be an expression of that type. Arguments can be referred to within the rule body (x_i).

In what follows, a rule N of the form

$$\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\underline{\text{Form}}\ f_1,\ldots,\underline{\text{Form}}\ f_m, T_1\ p_1,\ldots,T_n\ p_n) = B,$$

where $f_1,\ldots f_m$ are arguments of type $\underline{\text{Form}}$ and $p_1,\ldots p_n$ are arguments of primitive type, is written in short as

$$\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\overline{\underline{\text{Form}}}\ \overline{f}, \overline{T}\ \overline{p}) = B$$

where $\overline{f}$ and $\overline{p}$ represent tuples of type $\overline{\underline{\text{Form}}}$ and $\overline{T}$ respectively. Without loss of generality, in the above rule we assume that all the arguments of type $\underline{\text{Form}}$ appear first.

Semantics The semantics of the logic is defined in terms of a satisfaction relation $\models$ between execution traces and specifications. 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; if $i \geq j$ then $\sigma^{[i,j]}$ denotes the empty trace. In the implementation a state is a user defined Java object that is updated through a user provided *updateOnEvent* method for each new event generated by the program. Given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

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

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 $\models_D \subseteq (\mathit{Trace} \times \mathbf{nat}) \times \underline{\text{Form}}$, 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.

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

An 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 $i = n+1$) a proposition evaluates 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. 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; if an argument is of primitive type its evaluation in the current state is substituted for the associated formal parameter of the rule, thereby capturing a desired freeze variable semantics. 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, $\odot$, 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`.

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

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)[5]. 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 is increased sometime later, possibly after the moment when x exceeds 0.

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

[5] See [4] for the correctness argument for such an embedding of propositional LTL in EAGLE.

Extended LTL and μTL: 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:

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

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

$$\underline{\text{max}}\ \texttt{X}() = p \wedge \bigcirc\bigcirc\texttt{X}() \wedge \texttt{Y}() \qquad \underline{\text{min}}\ \texttt{Y}() = (q \wedge \bigcirc\texttt{X}()) \vee \odot\texttt{Y}()$$

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 runtime monitoring. EREs 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 $\underline{\text{max}}\ \texttt{Empty}() = \neg\bigcirc\underline{\text{true}}$ which is true only when evaluated on an empty (suffix) sequence. **Tr** is inductively defined as follows.

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

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 $\diamond^{[t_1,t_2]}F$ has the formula true if and only if F holds at some time in the future when the real-time clock has a value within the interval $[t_1,t_2]$. For our purposes, we assume that the states being monitored are time-stamped and that the variable *clock* holds the value of the real-time clock for the associated state. The rule

$$\begin{array}{l}
\underline{\text{min}}\ \texttt{EventAbs}(\underline{\text{Form}}\ F, \underline{\text{float}}\ t_1, \underline{\text{float}}\ t_2) = \\
\quad (F \wedge t_1 \leq clock \wedge clock \leq t_2) \vee \\
\quad ((clock < t_1 \vee (\neg F \wedge clock \leq t_2)) \wedge \bigcirc\texttt{EventAbs}(F,t_1,t_2))
\end{array}$$

defines the operator $\diamond^{[t_1,t_2]}$ for absolute values of the clock. The rule will succeed when the formula F evaluates to true and *clock* is within the specified interval $[t_1,t_2]$. If either the formula F doesn't hold and the time-stamp is within the upper time bound, or the lower bound hasn't been reached, then the rule is applied to the next input state. Note that the rule will fail as soon as either the time-stamp is beyond the given interval, i.e. $clock > t_2$, and the formula F has not been satisfied, or the end of the input trace has been passed. A relativized version of the modality can then be defined as:

$$\underline{\text{min}}\ \texttt{EventRel}(\underline{\text{Form}}\ F, \underline{\text{float}}\ t_1, \underline{\text{float}}\ t_2) = \texttt{EventAbs}(F, clock+t_1, clock+t_2)$$

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 F 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 $\Box_p F$ defined by:

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

An encoding within our logic can then be given as:

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

The auxiliary rule `A` counts the number of failures of F in its argument f and the number of events monitored in argument t. Thus, at the end of monitoring, the first line of `A`'s body determines whether F has held with the desired probability, i.e. $\geq p$. `AtLeast` therefore calls `A` with arguments f and t initialized to 0 and 1 respectively.

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, characterized, respectively, by the formulas *login* and *logout*. We can define a rule $\texttt{Match}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2)$ and monitor with $\texttt{Match}(\textit{login}, \textit{logout})$ where:

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

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 briefly outline the computation mechanism used to determine whether a given monitoring formula holds for some given input sequence of events. For details on the algorithm, the interested readers can refer to [5]. In the algorithm, we assume that a local state is maintained on the observer side. The *expressions* or *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 the new state, and generates a new set of monitored formulas. At the end of the trace the values of the monitored formulas are determined. If the value of a formula is true, the formula is satisfied, otherwise the formula is violated.

First, a monitor formula F is transformed to another formula F'. This transformation addresses the semantics of EAGLE at the beginning of a trace. Next, the transformed formula is monitored against an execution trace by repeated application of *eval*. The evaluation of a formula F on a state $s = \sigma(i)$ in a trace σ results in an another formula $eval\langle\langle F,s\rangle\rangle$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models eval\langle\langle F,s\rangle\rangle$. The definition of the function $eval : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ uses an auxiliary function *update* with signature $update : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$. *update*'s role is to pre-evaluate a formula if it is guarded by the previous operator $\odot$. Formally, *update* has the property that $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models update\langle\langle F,s\rangle\rangle$. Had there been no past time modality in EAGLE *update* would be unnecessary and the identity $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models F$ could have been used. At the end (or at the beginning) of a trace, the function $value : Form \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$ when applied on F returns $\underline{\text{true}}$ iff $\sigma, |\sigma|+1 \models F$ (or $\sigma, 0 \models F$) and returns $\underline{\text{false}}$ otherwise. Thus given a sequence of states $s_1 s_2 \ldots s_n$, an EAGLE formula F is said to be satisfied by the sequence of states if and only if $value\langle\langle eval\langle\langle \ldots eval\langle\langle eval\langle\langle F', s_1\rangle\rangle, s_2\rangle\rangle \ldots, s_n\rangle\rangle\rangle\rangle$ is $\underline{\text{true}}$. The functions *eval*, *update* and *value* are the basis of the calculus for our rule-based framework.

3.1 Calculus

The *eval*, *update* and *value* functions are defined a priori for all operators except for the rule application. The definitions of *eval*, *update* and *value* for rules get generated based on the definition of rules in the specification. The definitions of *eval*, *update* and *value* on the different primitive operators are given below.

$$
\begin{aligned}
eval\langle\langle \underline{\text{true}}, s\rangle\rangle &= \underline{\text{true}} \\
eval\langle\langle \underline{\text{false}}, s\rangle\rangle &= \underline{\text{false}} \\
eval\langle\langle jexp, s\rangle\rangle &= \text{ value of } jexp \text{ in } s \\
eval\langle\langle F_1\ op\ F_2, s\rangle\rangle &= eval\langle\langle F_1, s\rangle\rangle\ op\ eval\langle\langle F_2, s\rangle\rangle \\
eval\langle\langle \neg F, s\rangle\rangle &= \neg eval\langle\langle F, s\rangle\rangle \\
eval\langle\langle \bigcirc F, s\rangle\rangle &= update\langle\langle F, s\rangle\rangle \\
eval\langle\langle F_1 \cdot F_2, s\rangle\rangle &= \\
&= \begin{cases} eval\langle\langle F_1, s\rangle\rangle \cdot F_2 \text{ if } value\langle\langle F_1\rangle\rangle = \underline{\text{false}} \\ (eval\langle\langle F_1, s\rangle\rangle \cdot F_2) \vee eval\langle\langle F_2, s\rangle\rangle \text{ otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
value\langle\langle \underline{\text{true}}\rangle\rangle &= \underline{\text{true}} \\
value\langle\langle \underline{\text{false}}\rangle\rangle &= \underline{\text{false}} \\
value\langle\langle jexp\rangle\rangle &= \underline{\text{false}} \\
value\langle\langle F_1\ op\ F_2\rangle\rangle &= value\langle\langle F_1\rangle\rangle\ op\ value\langle\langle F_2\rangle\rangle \\
value\langle\langle \neg F\rangle\rangle &= \neg value\langle\langle F\rangle\rangle \\
value\langle\langle \bigcirc F\rangle\rangle & \\
&= \begin{cases} F & \text{if at the beginning of trace} \\ \underline{\text{false}} & \text{if at the end of trace} \end{cases} \\
value\langle\langle F_1 \cdot F_2\rangle\rangle &= value\langle\langle F_1\rangle\rangle \wedge value\langle\langle F_2\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
update\langle\langle \underline{\text{true}}, s\rangle\rangle &= \underline{\text{true}} \\
update\langle\langle \underline{\text{false}}, s\rangle\rangle &= \underline{\text{false}} \\
update\langle\langle jexp, s\rangle\rangle &= jexp \\
update\langle\langle F_1\ op\ F_2, s\rangle\rangle &= update\langle\langle F_1, s\rangle\rangle\ op\ update\langle\langle F_2, s\rangle\rangle \\
update\langle\langle \neg F, s\rangle\rangle &= \neg update\langle\langle F, s\rangle\rangle \\
update\langle\langle F_1 \cdot F_2, s\rangle\rangle &= update\langle\langle F_1, s\rangle\rangle \cdot F_2
\end{aligned}
$$

In the above definitions, *op* can be $\wedge, \vee, \rightarrow$. In most of the definitions we simply propagate the function to the subformulas. However, the concatenation operator is handled in a special way. The *eval* of a formula $F_1 \cdot F_2$ on a state s first checks if $value\langle\langle F_1\rangle\rangle$ is true or not. If the value is true then one can *non-deterministically* split the trace just before the state s. Hence, the evaluation becomes $(eval\langle\langle F_1, s\rangle\rangle \cdot F_2) \vee eval\langle\langle F_2, s\rangle\rangle$ where $\vee$ expresses the non-determinism. Otherwise, if the trace cannot be split, the evaluation becomes simply $eval\langle\langle F_1, s\rangle\rangle \cdot F_2$. The function *update* on the formula $F_1 \cdot F_2$ simply updates the formula F_1, as F_2 is not effected by the trace that effects F_1. At the end of a trace, that $F_1 \cdot F_2$ is satisfied means that the remaining empty trace can be split into two empty traces satisfying respectively F_1 and F_2; hence the conjunction in $value\langle\!\langle F_1 \cdot F_2 \rangle\!\rangle$. Since the semantics of $\bigcirc$ is different at the beginning and at the end of a trace, we have to consider the two cases in the definition of *value* for $\bigcirc F$.

The operator $\odot$ requires special attention. If a formula F is guarded by a previous operator 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 for a $\odot$ operator, we introduce the operator $\underline{\text{Previous}} : \underline{\text{Form}} \times \underline{\text{Form}} \to \underline{\text{Form}}$. The second argument for this operator acts as the placeholder. We transform a formula $\odot F$ at the beginning of monitoring as follows:

$$\odot F \to \underline{\text{Previous}}(F', value\langle\!\langle F' \rangle\!\rangle) \text{ where } F' \text{ is the transformed version of } F$$

We define *eval*, *update*, and *value* for $\underline{\text{Previous}}$ as follows:

$$\begin{array}{l}
eval\langle\!\langle \underline{\text{Previous}}(F, past), s \rangle\!\rangle = eval\langle\!\langle past, s \rangle\!\rangle \\
update\langle\!\langle \underline{\text{Previous}}(F, past), s \rangle\!\rangle = \underline{\text{Previous}}(update\langle\!\langle F, s \rangle\!\rangle, eval\langle\!\langle F, s \rangle\!\rangle) \\
value\langle\!\langle \underline{\text{Previous}}(F, past) \rangle\!\rangle = \begin{cases} \underline{\text{false}} & \text{if at the beginning of trace} \\ value\langle\!\langle past \rangle\!\rangle & \text{if at the end of trace} \end{cases}
\end{array}$$

Here, *eval* of $\underline{\text{Previous}}(F, past)$ returns the *eval* of the second argument of $\underline{\text{Previous}}$, *past*, that contains the evaluation of F in the previous state. In *update* we not only update the first argument F but also evaluate F and pass it as the second argument of $\underline{\text{Previous}}$. Thus in the next state the second argument of $\underline{\text{Previous}}$, *past*, is bound to $\odot F$. The $value\langle\!\langle F' \rangle\!\rangle$ that appears in the transformation of $\odot F$ is the value of F' at the beginning of the trace. This takes care of the semantics of EAGLE at the beginning of a trace.

3.2 Monitor Synthesis for Rules

In what follows, $\rho b.H(b)$ denotes a recursive structure where free occurrences of b in H point back to $\rho b.H(b)$. Formally, $\rho b.H(b)$ is a closed form term that denotes a fix-point solution to the equation $x = H(x)$ and hence $\rho b.H(b) = H(\rho b.H(b))$. The open form $H(b)$ denotes a formula with free recursion variable b. In structural terms, a solution to $x = H(x)$ can be represented as a graph structure where the leaves, denoted by `x`, point back to the root node of the graph. Our implementation uses this structural solution.

We replace every rule `R` by an operator $\underline{\text{R}}$ during transformation. For a subformula $\mathtt{R}(\overline{F}, \overline{P})$, where the rule `R` is defined as $\{\underline{\text{max}}|\underline{\text{min}}\}\ \mathtt{R}(\overline{\underline{\text{Form}}\ f}, \overline{T\ p}) = B$, we transform the subformula as follows:

$$\begin{array}{l}
\mathtt{R}(\overline{F}, \overline{P}) \to \underline{\text{R}}(\rho b.B'[\overline{f} \mapsto \overline{F'}], \overline{P}) \\
\qquad\qquad\qquad \text{where } B' \text{ and } F' \text{ are transformed versions of } B \text{ and } F \text{ respectively} \\
\mathtt{R}(\overline{F}, \overline{P'}) \to \underline{\text{R}}(b, \overline{P'}) \text{ where } \mathtt{R}(\overline{F}, \overline{P'}) \text{ is a subformula of } B
\end{array}$$

The second equation is invoked if a recursion is detected [6]; that is while transforming B if $\mathtt{R}(\overline{F}, \overline{P'})$ is encountered as a subformula of B. Note that here the variable b should

[6] A formal description of recursion detection mechanism can be found in [5].

be a fresh name to avoid possible variable capturing. For example consider the formula $\Box(x > 0 \rightarrow \exists k(k = x \wedge \Diamond\!\!\!\cdot\,(z > 0 \wedge y = k)))$. A specification for this monitor can be presented in EAGLE as follows:

$$\begin{array}{l}
\underline{\text{max}}\ \texttt{A}(\underline{\text{Form}}\ f) = f \wedge \bigcirc \texttt{A}(f) \\
\underline{\text{min}}\ \texttt{Ep}(\underline{\text{Form}}\ f) = f \vee \odot\, \texttt{Ep}(f) \\
\underline{\text{min}}\ \texttt{Ev}(\underline{\text{int}}\ k) = \texttt{Ep}(z > 0 \wedge y = k) \\
\underline{\text{mon}}\ M = \texttt{A}(x > 0 \rightarrow \texttt{Ev}(x))
\end{array}$$

The transformed version of M is as follows:

$$\begin{array}{l}
M' = \underline{\text{A}}(\rho b_1.((x > 0) \rightarrow \\
\qquad \underline{\text{Ev}}(\rho b_2.\underline{\text{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \underline{\text{Previous}}(\underline{\text{Ep}}(b_3), \underline{\text{false}}))), x)) \wedge \underline{\text{Next}}(\underline{\text{A}}(b_1)))
\end{array}$$

The definitions of *update*, *eval* and *value* for $\underline{\text{R}}$ are as follows:

$$\begin{array}{l}
update\langle\!\langle \underline{\text{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = \underline{\text{R}}(\rho b'.update\langle\!\langle H(\rho b.H(b)), s\rangle\!\rangle, \overline{P}) \\
update\langle\!\langle \underline{\text{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = \underline{\text{R}}(b', \overline{P}) \\
\qquad\qquad \text{if } \underline{\text{R}}(\rho b.H(b), \overline{P}) \text{ is a subformula of } H(\rho b.H(b))
\end{array}$$

Here, $\rho b.H(b)$ is first expanded to $H(\rho b.H(b))$ and then *update* is applied on it; that is, the body of the rule is updated. The second equation detects a recursion, that is, update of $H(\rho b.H(b))$ encounters $\underline{\text{R}}(\rho b.H(b), \overline{P})$ as a subformula of $H(\rho b.H(b))$. In that case $\underline{\text{R}}(b', \overline{F}, \overline{P})$ is returned terminating the recursion.

$$eval\langle\!\langle \underline{\text{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = eval\langle\!\langle H(\rho b.H(b))[\overline{p} \mapsto eval\langle\!\langle \overline{P}, s\rangle\!\rangle], s\rangle\!\rangle$$

Here, $\rho b.H(b)$ is first expanded to $H(\rho b.H(b))$ and then any arguments of primitive type are evaluated and substituted in the expansion. The function *eval* is then applied on the expansion. Note that the result of $eval\langle\!\langle P, s\rangle\!\rangle$, where P is an expression, may be a partially evaluated expression if expressions referred to by some of the variables in P are partially evaluated. The expression gets fully evaluated once all the variables referred to by the expressions are fully evaluated.

$$value\langle\!\langle \underline{\text{R}}(B, \overline{P})\rangle\!\rangle = \underline{\text{false}} \text{ if } \texttt{R} \text{ is minimal} \qquad value\langle\!\langle \underline{\text{R}}(B, \overline{P})\rangle\!\rangle = \underline{\text{true}} \text{ if } \texttt{R} \text{ is maximal}$$

The *value* of a $\underline{\text{max}}$ rule is $\underline{\text{true}}$ and that of a $\underline{\text{min}}$ rule is $\underline{\text{false}}$.

For example, for a sequence of states sequence $\{x = 0, y = 3, z = 1\}, \{x = 0, y = 5, z = 2\}, \{x = 2, y = 2, z = 0\}$, step-by-step monitoring of the formula M' on this sequence takes place as follows:

Step 1: $s = \{x = 0, y = 3, z = 1\}$

$$\begin{array}{l}
F_1 = eval\langle\!\langle F, s\rangle\!\rangle \\
\quad = \underline{\text{A}}(\rho b_1.((x > 0) \rightarrow \\
\qquad \underline{\text{Ev}}(\rho b_2.\underline{\text{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \underline{\text{Previous}}(\underline{\text{Ep}}(b_3), (3 = k)))), x)) \wedge \underline{\text{Next}}(\underline{\text{A}}(b_1)))
\end{array}$$

Observe that in the above step the second argument of $\underline{\text{Previous}}$ is partially evaluated as the value of k is not available. The value of k becomes available when we apply *eval* on $\underline{\text{Ev}}$. At that time *eval* also replaces the free variable k appearing in the first argument of $\underline{\text{Ev}}$ by the actual value. This replacement can easily be seen if the definition of *eval* of

$\underline{\text{Ev}}$ is written down.

Step 2: $s = \{x = 0, y = 5, z = 2\}$

$$
\begin{aligned}
F_2 &= eval\langle\!\langle F_1, s\rangle\!\rangle \\
&= \underline{\text{A}}(\rho b_1.((x > 0) \rightarrow \underline{\text{Ev}}(\rho b_2.\underline{\text{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \\
&\qquad\qquad \underline{\text{Previous}}(\underline{\text{Ep}}(b_3), (3 = k) \vee (5 = k))))), x)) \wedge \underline{\text{Next}}(\underline{\text{A}}(b_1)))
\end{aligned}
$$

Step 3: $s = \{x = 2, y = 2, z = 0\}$

$$F_3 = eval\langle\!\langle F_2, s\rangle\!\rangle = \underline{\text{false}}$$

Thus the formula is violated on the third state of the trace.

4 Implementation and Experiments

We have implemented this monitoring framework in Java. 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 [15]. 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.

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

The above equations ensure 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 the rules was made in order to achieve 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. The automated testing system found a missing feature that had been overlooked by the developers: the lower bounds on action execution duration were not checked by the implementation, causing some executions to succeed while they in fact should fail due to the too early termination of some action execution. The temporal formulas, however, correctly predicted failure in these cases. This error showed up later during actual rover operation before it was corrected.

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)*, 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. Eagle Monitors by Collecting Facts and Generating Obligations. Pre-Print CSPP-26, University of Manchester, Department of Computer Science, October 2003. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp26.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 the 2nd International Workshop on Runtime Verification (RV'02)* [1], pages 36–55.
9. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 44–60.
10. 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.
11. 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.
12. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 97–114. Extended version to appear in the journal: Formal Methods in System Design, Kluwer, 2004.
13. 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.
14. 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 *LNCS*, pages 342–356. Springer, 2002. Extended version to appear in the journal: Software Tools for Technology Transfer, Springer, 2004.
15. Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
16. D. Kortenkamp, T. Milam, R. Simmons, and J. Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 133–151.
17. K. Jelling Kristoffersen, C. Pedersen, and H. R. Andersen. Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1], pages 146–161.
18. 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.
19. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
20. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1], pages 162–181.