EAGLE Monitors by Collecting Facts and Generating Obligations

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, called EAGLE, that has been shown to be capable of defining and implementing a range of finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, real-time and metric temporal logics, interval logics, forms of quantified temporal logics, and so on. A monitor for an EAGLE formula checks if a finite trace of states satisfies the given formula. We present, in details, an algorithm for the synthesis of monitors for EAGLE. The algorithm is implemented as a Java application and involves novel techniques for rule definition, manipulation and execution. Monitoring is achieved on a state-by-state basis avoiding any need to store the input trace of states. Our initial experiments have been successful as EAGLE detected a previously unknown bug while testing a planetary rover controller.

1 Introduction

Formal methods have been investigated for several decades in an effort to improve software quality. However, in the ambitious goal of proving correctness of the system/program, many of these technologies suffer from scalability problems. A complementary approach is to make incremental improvements to traditional and practical testing approaches. Runtime verification, or runtime monitoring as it is also called, is such a technique, where a program's execution is monitored by an oracle, which automatically determines whether the run is correct or not. The oracle takes as input a specification of what constitutes correct behavior, and checks that its second input, the execution trace, satisfies it. Runtime verification cannot only be used during testing, but can also be applied during operation to survey the health of an application, and actions associated with individual properties in the specification can be triggered when the properties get violated.

Several runtime verification systems have been developed, of which some were presented at three recent international workshops on runtime verification [1]. The different logics offer different capabilities, and the question obviously becomes: is there a unifying logic in which all these logics can be modeled, providing the union of these useful capabilities, and which still is theoretically simple and succinct? We propose a

* 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.

very powerful and succinct temporal finite trace monitoring logic, named EAGLE, as a solution. The logic was first presented in [5]. In this paper we detail the algorithm, its implementation, and give complexity results. We furthermore describe a case study where EAGLE has been applied to test one of NASA's planetary rover controllers.

An EAGLE specification consists of a set of rule definitions, defining new temporal operators. Rules are parameterized with formulas and data, and have maximal or minimal fix-point semantics depending on whether a safety property or a liveness property is being defined. Only three temporal primitives are provided: next-time, previous-time, and concatenation. EAGLE has in particular been significantly influenced by earlier work of Barringer et al., see for example [3], on the executable temporal logic METATEM. EAGLE is very expressive, and the following kinds of logics can for example easily be built on top of EAGLE, as illustrated in [5]: future and past time logics, extended LTL and the semantically equivalent fix-point temporal calculus, extended regular expressions, data constrained and statistical logics, real-time logics as a special case of data constraints, and context free languages. The logic also supports the mix of formulas with state machines, and probabilistic logics. In [4] we describe how pure propositional future and past time LTL is embedded in EAGLE, and the complexity of monitoring in that restricted case.

Linear temporal logic (LTL) [19] forms the basis of several runtime verification systems. The tool Temporal Rover [6,7] supports a fixed future and past time LTL, with the possibility of specifying real-time and data constraints as annotations on the temporal operators. The algorithm is based on alternating automata, as is the work in [9, 8], which supports a statistical logic. 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. In [20] is described a logic based on extended regular expressions. The logic described in [15] is a sophisticated interval logic. Our own previous work includes the development of several algorithms, such as generating dynamic programming algorithms for past time logic [12], using a rewriting system for monitoring future-time logic [11], or generating Büchi automata inspired algorithms adapted to finite trace LTL [10]. Parallel work [17] also uses recursive equations to implement a real-time logic. However, we go beyond by providing a language of recursive equations to the user, by supporting a mixture of future time and past time operators, and by providing parametrization to reason about data, of which we regard real-time as just a special case.

The paper is organized as follows. In Section 2, EAGLE is introduced through an example, and its syntax and semantics is presented. Section 3 presents the algorithm for monitoring EAGLE formulas on finite traces. Section 4 describes the implementation in Java, and a case study where the framework was applied to test a rover controller. Finally, Section 5 concludes the paper.

2 The EAGLE Logic

The EAGLE logic is designed to support finite trace monitoring, and contains a small set of powerful operators, which allow one to define new logics on top. EAGLE essentially supports recursive parameterized equations, with a minimal/maximal fix-point semantics, together with three temporal operators: next-time, previous-time, and concatenation. Rules can be parameterized with formulas, supporting the definition of new temporal operators, and they can be parameterized with values, thus supporting logics that can reason about data and, as a special case of data, real-time. As atomic propositions we assume boolean expressions over individual states comprising the finite trace. In the current implementation of the monitoring algorithm for EAGLE, states are Java objects, and propositions are boolean valued Java expressions.

The logical system 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 through the parameterization, and concatenation, we are strictly more expressive than, say, a linear temporal fixed point logic (over next and previous). We shall first, in Section 2.1, introduce the logic through an example. Then, in Section 2.2, we present its syntax and semantics.

2.1 An Example

We shall illustrate the logic through an example. First, we define some general temporal operators, using EAGLE's parameterized rules and the next-time operator. Then we present some more definitions, illustrating concatenation and how data parameters can be used in past-time as well as in future-time formulas.

Defining Basic Temporal Operators Assume that we are interested in monitoring the behavior of a program, and assume that the result of executing the program is a sequence of observable states. Each state is time stamped by the program, and the time stamp of the current state can be obtained by calling the method *currentTime*(). We shall refer to this sequence as the execution trace, or just trace for short. Our goal is to state properties about these traces. Say we want to observe the following relationship between two state predicates P and Q along the trace: "*Whenever P occurs then Q must occur within 10 seconds*". The property can be written as follows in metric future time LTL [16]: $\Box(P \rightarrow \Diamond_{<10}Q)$. The formula contains the two temporal operators $\Box$ (always) and $\Diamond_{<t}$ (eventually within t time units). If we consider the formula $\Box F$, for some sub-formula F, then normally (in temporal logic literature) it satisfies the following equivalence, where the temporal operator $\bigcirc F$ stands for *next F* (meaning '*in next state F*'):

$$\Box F \equiv F \wedge \bigcirc(\Box F)$$

It can in fact be shown that $\Box F$ is the maximal solution to the recursive equation $X = F \wedge \bigcirc X$. A fundamental idea in our logic is to support this kind of recursive definitions. In EAGLE, the two just discussed temporal operators, $\Box F$ and $\Diamond_{<t}F$, and the property to be monitored can be defined as in Figure 1. The $\Box F$ operator is modelled by the `Always` operator, the definition of which directly follows the equation above. The relative-time temporal operator $\Diamond_{<t}$ is represented by the `EventuallyRel` operator, which itself is defined in terms of the absolute-time `EventuallyAbs` operator. That is, the `EventuallyAbs` operator takes as argument the absolute time within which the formula given as second argument must be satisfied. It therefore checks that the current time has not exceeded the absolute time argument. If the property F is not true, the obligation is "pushed forward" to hold in the next state. The `EventuallyRel` operator just adds the current time to its relative time argument and calls `EventuallyAbs`. We

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

$\underline{\text{min}}\ \texttt{EventuallyAbs}(\underline{\text{float}}\ t, \underline{\text{Form}}\ F) =$
$\quad currentTime() \leq t \wedge ((\neg F) \rightarrow \bigcirc\texttt{EventuallyAbs}(t, F))$

$\underline{\text{min}}\ \texttt{EventuallyRel}(\underline{\text{float}}\ t, \underline{\text{Form}}\ F) =$
$\quad \texttt{EventuallyAbs}(currentTime() + t, F)$

$\underline{\text{mon}}\ M = \texttt{Always}(P \rightarrow \texttt{EventuallyRel}(10, Q))$

Fig. 1. EAGLE definitions

see that real-time is modeled as a floating point number, which is made as parameter to the rules. This illustrates the general capability of the logic to handle general data values, real-time being a specific example.

The `Always` operator is defined as a maximal fix-point operator, while the operators `EventuallyAbs` and `EventuallyRel` are defined as minimal operators. Maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually must happen). The difference becomes important only at the end of the trace analysis (beyond the last state): a maximal rule just evaluates to true, while a minimal rule evaluates to false. This reflects the intuition that a safety property is true if it is true on all proper states in the trace, but a liveness property is true only if the expected events occur, and beyond the end of the trace no such event can be expected to occur.

Concatenation and More about Data We shall now use these defined operators and extend our example a bit. Assume the following requirement for some concept of task execution:

> "If a start command is given, the task should begin executing within 10 seconds. It begins by issuing an observable begin event and ends by issuing an observable end event. In between the begin and end events the task should report errors correctly. That is, if an error occurs, then an error report should be emitted on that error, identifying the error code. An error report is emitted precisely once on each kind of occurring error."

We assume that the states in the trace can be observed through the following boolean valued functions: $start()$ – task is being requested to begin, $begin()$ – task begins, $end()$ – task ends, and the following integer valued functions: $error()$ – non-zero identification of an occurred error, and $report()$ – non-zero identification of an error reported. Then the specification can be written as in Figure 2, extending the definitions in Figure 1.

First, note that the use of the concatenation operator $F_1 \cdot F_2$ in the definition of `Execute`. It states that the trace can be divided into two parts, one satisfying F_1 and one satisfying F_2. Because of this operator, the `Always` operator within the `ReportErrors` rule does not extend beyond the $end()$. Note that the $begin()$ proposition cannot be composed with the `ReportErrors()` with concatenation since the concatenation accepts any split of the trace where $begin()$ satisfies the first sub-trace and where

$$\underline{\text{max}}\ \texttt{Task}() = \\ \quad \texttt{Always}(\mathit{start}() \rightarrow \texttt{EventuallyRel}(10, \texttt{Execute}()))$$

$$\underline{\text{max}}\ \texttt{Execute}() = \\ \quad (\mathit{begin}() \wedge \bigcirc\texttt{ReportErrors}()) \cdot \mathit{end}()$$

$$\underline{\text{max}}\ \texttt{ReportErrors}() = \\ \quad \texttt{Always}((\mathit{error}() \neq 0 \wedge \neg\texttt{Reported}(\mathit{error}())) \rightarrow \texttt{Report}(\mathit{error}())) \wedge \\ \quad \texttt{Always}((\mathit{report}() \neq 0) \rightarrow (\neg\texttt{Reported}(\mathit{report}()) \wedge \texttt{ErrorOccurred}(\mathit{report}())))$$

$$\underline{\text{min}}\ \texttt{Report}(\underline{\text{int}}\ \mathit{errorCode}) = \\ \quad (\mathit{report}() = \mathit{errorCode}) \vee \bigcirc\texttt{Report}(\mathit{errorCode})$$

$$\underline{\text{min}}\ \texttt{Reported}(\underline{\text{int}}\ \mathit{errorCode}) = \\ \quad \odot(\mathit{report}() = \mathit{errorCode}) \vee \odot\,\texttt{Reported}(\mathit{errorCode})$$

$$\underline{\text{min}}\ \texttt{ErrorOccurred}(\underline{\text{int}}\ \mathit{errorCode}) = \\ \quad (\mathit{error}() = \mathit{errorCode}) \vee \odot\,\texttt{ErrorOccurred}(\mathit{errorCode})$$

$$\underline{\text{mon}}\ M = \texttt{Task}()$$

Fig. 2. EAGLE definitions continued

ReportErrors() satisfies the other sub-trace, hence missing to enforce error reports during the first sub-trace.

The ReportErrors rule itself illustrates the previous operator, $\odot$, which is the mirror of the next operator, $\bigcirc$. The formula says: "if there is an error, and it has not been reported before, then it should be reported in the future. Furthermore, if an error is reported, it must not have been reported in the past, and it must have occurred". The rules Report, Reported and ErrorOccurred each takes as argument an error code, and checks whether this error code is being reported in the future or in the past, and whether the error has occurred. For example, Report checks whether there is a report of the error now, and if not, if there is such a report in the future (starting from the next state). The sub-formula $\texttt{Report}(\mathit{error}())$, for example, represents the following quantified LTL formula: $\exists k.(k = \mathit{error}() \wedge \Diamond(\mathit{report}() = k))$.

2.2 Syntax and Semantics

Syntax The syntax of EAGLE is shown in Figure 3. A specification S consists of a declaration part D and an observer part O. D comprises zero or more rule definitions R, and O comprises zero or more monitor definitions M, which specify what is to be monitored. Rules and monitors are named (N). Each rule definition R is preceded by one of the keywords max or min, indicating whether the interpretation is maximal or minimal. A parameter type can either be Form, representing formulas, or a primitive type 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

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

Fig. 3. Syntax of EAGLE

propositional logic operators together with a next-state operator ($\bigcirc F$), a previous-state operator ($\odot F$), and a concatenation-operator ($F_1 \cdot F_2$). 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).

Semantics The models of our logic are execution traces. 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 ith state s_i of a trace σ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of σ from position i to position j, both positions included. The semantics of the logic is defined in terms of a satisfaction relation between execution traces and specifications. That is, given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

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

A trace satisfies a specification if the trace, observed from position 1 (the first state), satisfies each monitored formula. The definition of the satisfaction relation $\models_D \subseteq (\textit{Trace} \times \textbf{nat}) \times \underline{\text{Form}}$, for a set of rule definitions D, is presented in Figure 4. 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. Note that the first formula F_1 is not checked on the second trace σ_2, and, similarly, the second formula F_2 is not checked on the first trace σ_1. Note also that either σ_1 or σ_2 may be an empty sequence. 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.

3 Monitoring Algorithm

In this section, we outline the monitoring algorithm used to determine whether a given monitoring formula holds for some given input sequence of events. The algorithm is followed by an example illustrating how it works. 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 its local state; then *evaluates* the monitored formulas on that 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. In what follows, we will assume `null` as a special formula that is not equal to any other formula.

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

Fig. 4. Definition of $\sigma, i \models_D F$ for $0 \le i \le |\sigma|+1$ for some trace $\sigma = s_1 s_2 \ldots s_{|\sigma|}$

First, a monitor formula F is transformed to the formula $\mathit{init}\langle\!\langle F, \texttt{null}, \texttt{null}\rangle\!\rangle$[4] by applying the function $\mathit{init} : \underline{\text{Form}} \times \underline{\text{Form}} \times \underline{\text{Form}} \rightarrow \underline{\text{Form}}$. *init*'s second argument is used to determine termination for a recursive application of *init* on a rule - it is the head formula of a recursive rule application; its third argument denotes the recursion variable that will replace any embedded recursive call on the head formula. These two arguments are `null` for the initial application of *init* as it is not yet in the context of a rule. Next, the transformed formula is monitored against an execution trace by application of *eval*. The evaluation of a formula F on a state $s = \sigma(i)$ in a trace σ results in an another formula $\mathit{eval}\langle\!\langle F, s\rangle\!\rangle$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models \mathit{eval}\langle\!\langle F, s\rangle\!\rangle$. The definition of the function $\mathit{eval} : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ uses an auxiliary function *update* with signature $\mathit{update} : \underline{\text{Form}} \times \underline{\text{State}} \times \underline{\text{Form}} \times \underline{\text{Form}} \rightarrow \underline{\text{Form}}$. *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 $\sigma, i+1 \models \mathit{update}\langle\!\langle F, s, \texttt{null}, \texttt{null}\rangle\!\rangle$. 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. The last two arguments of *update* have a similar role to those of *init*. At the end (or at the beginning) of a trace, the function $\mathit{value} : \mathit{Form} \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$ 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 $\mathit{value}\langle\!\langle \mathit{eval}\langle\!\langle \ldots \mathit{eval}\langle\!\langle \mathit{eval}\langle\!\langle \mathit{init}\langle\!\langle F, \texttt{null}, \texttt{null}\rangle\!\rangle, s_1\rangle\!\rangle, s_2\rangle\!\rangle \ldots s_n\rangle\!\rangle\rangle\!\rangle$ is $\underline{\text{true}}$. The functions *init*, *eval*, *update* and *value* are the basis of the calculus for our rule-based framework.

3.1 Calculus

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

[4] We use braces $\langle\!\langle \ldots \rangle\!\rangle$ in lieu of $(\ldots)$ to help the reader parse deeply nested formulas.

eval, *update* and *value* on the different primitive operators are given below.

$$\begin{aligned}
init\langle\!\langle \underline{\text{true}}, Z, b' \rangle\!\rangle &= \underline{\text{true}}\\
init\langle\!\langle \underline{\text{false}}, Z, b' \rangle\!\rangle &= \underline{\text{false}}\\
init\langle\!\langle jexp, Z, b' \rangle\!\rangle &= jexp\\
init\langle\!\langle F_1 \ op\ F_2, Z, b' \rangle\!\rangle &= init\langle\!\langle F_1, Z, b' \rangle\!\rangle \ op\ init\langle\!\langle F_2, Z, b' \rangle\!\rangle\\
init\langle\!\langle \neg F, Z, b' \rangle\!\rangle &= \neg init\langle\!\langle F, Z, b' \rangle\!\rangle\\
init\langle\!\langle F_1 \cdot F_2, Z, b' \rangle\!\rangle &= init\langle\!\langle F_1, Z, b' \rangle\!\rangle \cdot init\langle\!\langle F_2, Z, b' \rangle\!\rangle
\end{aligned}$$

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

$$\begin{aligned}
value\langle\!\langle \underline{\text{true}} \rangle\!\rangle &= \underline{\text{true}}\\
value\langle\!\langle \underline{\text{false}} \rangle\!\rangle &= \underline{\text{false}}\\
value\langle\!\langle jexp \rangle\!\rangle &= \underline{\text{false}}\\
value\langle\!\langle F_1 \ op\ F_2 \rangle\!\rangle &= value\langle\!\langle F_1 \rangle\!\rangle \ op\ value\langle\!\langle F_2 \rangle\!\rangle\\
value\langle\!\langle \neg F \rangle\!\rangle &= \neg value\langle\!\langle F \rangle\!\rangle\\
value\langle\!\langle F_1 \cdot F_2 \rangle\!\rangle &= value\langle\!\langle F_1 \rangle\!\rangle \wedge value\langle\!\langle F_2 \rangle\!\rangle
\end{aligned}$$

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

In the above definitions, *op* can be $\wedge, \vee, \rightarrow$. Observe that we never used the last two arguments of *init* and *update*. 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. In that case 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 and they satisfy F_1 and F_2; hence we get conjunction in $value\langle\!\langle F_1 \cdot F_2 \rangle\!\rangle$.

The functions *init*, *eval*, *update*, and *value* are defined in a special way for the operators $\bigcirc$ and $\odot$. For the operator $\bigcirc$ we introduce the operator $\underline{\text{Next}} : \underline{\text{Form}} \rightarrow \underline{\text{Form}}$. Then we define *init*, *eval*, *update*, and *value* as follows:

$$\begin{aligned}
&init\langle\!\langle \bigcirc F, Z, b' \rangle\!\rangle = \underline{\text{Next}}(init\langle\!\langle F, Z, b' \rangle\!\rangle)\\
&eval\langle\!\langle \underline{\text{Next}}(F), s \rangle\!\rangle = update\langle\!\langle F, s, \texttt{null}, \texttt{null} \rangle\!\rangle\\
&update\langle\!\langle \underline{\text{Next}}(F), s, Z, b' \rangle\!\rangle = \underline{\text{Next}}(update\langle\!\langle F, s, Z, b' \rangle\!\rangle)\\
&value\langle\!\langle \underline{\text{Next}}(F) \rangle\!\rangle = \begin{cases} F & \text{if at the beginning of trace}\\ \underline{\text{false}} & \text{if at the end of trace} \end{cases}
\end{aligned}$$

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*.

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 $\underline{\text{Previous}} : \underline{\text{Form}} \times \underline{\text{Form}} \rightarrow \underline{\text{Form}}$. The second argument for this operator acts as the placeholder. We define *init*, *eval*, *update*, and *value* for $\odot$ as follows:

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

Here, in *eval*, the subformula F guarded by the previous operator is 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 $value\langle\!\langle Y \rangle\!\rangle$ that appears in the definition of *init* is the value of Y 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

For every rule R we introduce an operator $\underline{\text{R}}$ to replace R in a formula during the application of *init*. For example, consider a rule of the form

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

where $f_1, \ldots f_m$ are arguments of type $\underline{\text{Form}}$ and $p_1, \ldots p_n$ are arguments of primitive type. Without loss of generality, in the above rule we assume that all the arguments of type $\underline{\text{Form}}$ appear first. Such a rule can be written in short as

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

where $\overline{f}$ and $\overline{p}$ represents tuples of type $\overline{\underline{\text{Form}}}$ and $\overline{T}$ respectively. For such a rule we introduce an operator $\underline{\text{R}} : \underline{\text{Form}} \times \overline{T} \rightarrow \underline{\text{Form}}$. Informally, the first argument of $\underline{\text{R}}$ represents the transformed right hand side of the rule. 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 points back to the root node of the graph. Our implementation uses this structural solution.

For the rule $\{\underline{\text{max}}|\underline{\text{min}}\}\ \text{R}(\overline{\underline{\text{Form}}}\ \overline{f}, T_p\ \overline{p}) = B$ we *synthesize* the definitions of *init*, *eval*, *update*, and *value* as follows:

$$\begin{array}{l}
init\langle\!\langle \text{R}(\overline{F}, \overline{P}), \text{R}(\overline{F}, \overline{P'}), b' \rangle\!\rangle = \underline{\text{R}}(b', \overline{P}) \\
init\langle\!\langle \text{R}(\overline{F}, \overline{P}), Z, b' \rangle\!\rangle = \underline{\text{R}}(\rho b.init\langle\!\langle B[\overline{f} \mapsto \overline{Y}], \text{R}(\overline{F}, \overline{P}), b \rangle\!\rangle, \overline{P}) \\
\quad \text{where } \overline{Y} = init\langle\!\langle \overline{F}, Z, b' \rangle\!\rangle \text{ and } Z \text{ does not match } \text{R}(\overline{F}, ?)
\end{array}$$

In the first equation for *init*, the name of the rule and its arguments of type $\underline{\text{Form}}$ are the same for both the first and second arguments passed to *init*. This occurs when *init* has been applied to a recursive call of the rule R. So *init* is terminated and $\underline{\text{R}}(b', \overline{P})$

is returned. Otherwise, the formal variables $\overline{f}$ in the formula B (representing the right hand side of a rule) are substituted by the initialized version of the actual arguments $\overline{F}$ to get $B[\overline{f} \mapsto \overline{Y}]$. Then this B, with proper substitutions, is initialized. In the *init* function $\mathrm{R}(\overline{F},\overline{P})$ and b are passed as the second and third argument, respectively, to make sure that if B contains any recursive call to $\mathrm{R}(\overline{F},\overline{P})$ then the first definitional equation of *init* applies. We then obtain the recursive structure $\rho b.init\langle\!\langle B[\overline{f} \mapsto \overline{Y}], \mathrm{R}(\overline{F},\overline{P}), b\rangle\!\rangle$. This structure represents the right hand side or the body of the rule R possibly with recursive call to R. Note that here the variable b should be a fresh name to avoid possible variable capturing. Moreover, observe that we do not substitute the formal variables of primitive type as their values will be available at the time of monitoring. Lines 1 to 6 of the example in Section 3.3 show this transformation process.

$$\begin{array}{l}
update\langle\!\langle \underline{\mathrm{R}}(\rho b.H(b),\overline{P}), s, \underline{\mathrm{R}}(\rho b.H(b),\overline{P'}), b'\rangle\!\rangle = \underline{\mathrm{R}}(b',\overline{P}) \\
update\langle\!\langle \underline{\mathrm{R}}(\rho b.H(b),\overline{P}), s, Z, b''\rangle\!\rangle = \\
\qquad \underline{\mathrm{R}}(\rho b'.update\langle\!\langle H(\rho b.H(b)), s, \underline{\mathrm{R}}(\rho b.H(b),\overline{P}), b'\rangle\!\rangle, \overline{P}) \\
\qquad\qquad\qquad \text{where } Z \text{ does not match } \underline{\mathrm{R}}(\rho b.H(b), ?)
\end{array}$$

The first equation for *update* detects when $\underline{\mathrm{R}}$ and $\rho b.H(b)$ are the same for both the first and third arguments passed to *update*; it terminates the application of *update* in a similar way to *init* and $\underline{\mathrm{R}}(b',\overline{P})$ is returned. Otherwise, as in the second equation, $\rho b.H(b)$ is expanded to $H(\rho b.H(b))$. *update* is applied to $H(\rho b.H(b))$ with $\underline{\mathrm{R}}(\rho b.H(b),\overline{P})$ and b' as the last two arguments; this makes sure that if $H(\rho b.H(b))$ contains the subformula $\underline{\mathrm{R}}(\rho b.H(b), ?)$, the first equation for *update* applies and *update* terminates. This process is exemplified in lines 9 to 13 of the example in Section 3.3.

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

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. Steps 1 and 2 of the second example in Section 3.3 illustrate the partial evaluation.

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

The *value* of a <u>max</u> rule is <u>true</u> and that of a <u>min</u> rule is <u>false</u>.

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

$$\sigma, 1 \models_D F \text{ iff } value\langle\!\langle eval\langle\!\langle \ldots eval\langle\!\langle eval\langle\!\langle init\langle\!\langle F, \mathtt{null}, \mathtt{null}\rangle\!\rangle F, s_1\rangle\!\rangle, s_2\rangle\!\rangle \ldots s_n\rangle\!\rangle\rangle\!\rangle.$$

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

3.3 Examples

We provide two examples to show the workings of the monitor synthesis algorithm. For the first, we consider the initial transformation of, and then a single application of evaluation to the transformed formula for, the temporal monitor specified by:

$$\begin{array}{l}
\underline{\text{min}}\ \mathrm{Ep}(\underline{\text{Form}}\ f) = f \vee \odot\, \mathrm{Ep}(f) \\
\underline{\text{mon}}\ M = \bigcirc \mathrm{Ep}(q)
\end{array}$$

$$\begin{array}{ll}
1. & \mathit{init}\langle\!\langle \bigcirc\texttt{Ep}(q),\texttt{null},\texttt{null}\rangle\!\rangle \\
2. & \quad = \underline{\mathrm{Next}}(\mathit{init}\langle\!\langle \texttt{Ep}(q),\texttt{null},\texttt{null}\rangle\!\rangle) \\
3. & \quad = \underline{\mathrm{Next}}(\underline{\mathrm{Ep}}(\rho b.\mathit{init}\langle\!\langle (q \vee \odot\,\texttt{Ep}(q)),\texttt{Ep}(q),b\rangle\!\rangle)) \\
4. & \quad = \underline{\mathrm{Next}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \mathit{init}\langle\!\langle \odot\,\texttt{Ep}(q),\texttt{Ep}(q),b\rangle\!\rangle))) \\
5. & \quad = \underline{\mathrm{Next}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\mathit{init}\langle\!\langle \texttt{Ep}(q),\texttt{Ep}(q),b\rangle\!\rangle,\mathit{value}\langle\!\langle \mathit{init}\langle\!\langle \texttt{Ep}(q),\texttt{Ep}(q),b\rangle\!\rangle\rangle\!\rangle)))) \\
6. & \quad = \underline{\mathrm{Next}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))))
\end{array}$$

init is first applied to the monitor formula $\bigcirc\texttt{Ep}(q)$ together with two `null` arguments. It then converts the primitive $\bigcirc$ operator to the operator $\underline{\mathrm{Next}}$ – line 2. This results in *init* being applied to the Ep rule application which yields the new rule form $\underline{\mathrm{Ep}}$ being applied to the yet to be transformed recursively defined rule body – line 3. By line 5, *init* has transformed the primitive $\odot$ operator to the $\underline{\mathrm{Previous}}$ rule with its two arguments, respectively the transformation of the immediate subformula of $\odot$, i.e. *init* applied to $\texttt{E}(q)$, and then the initial boundary value of that particular transformed subformula. Note that in both cases the last two arguments of `init` are instantiated with the initial recursive call and with the "pointer" to the body. The transformation is completed by line 6 through `init` terminating via its first definitional clause. In line 7 below, *eval* is now applied to the *init* formula of line 1, i.e. *eval* is applied to the resulting transformation together with a state s (in which we assume that q is $\underline{\mathrm{true}}$) – line 8.

$$\begin{array}{ll}
7. & \mathit{eval}\langle\!\langle \mathit{init}\langle\!\langle \bigcirc\texttt{Ep}(q),\texttt{null},\texttt{null}\rangle\!\rangle,s\rangle\!\rangle \\
8. & \quad = \mathit{eval}\langle\!\langle \underline{\mathrm{Next}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}})))),s\rangle\!\rangle \\
9. & \quad = \mathit{update}\langle\!\langle \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),s,\texttt{null},\texttt{null}\rangle\!\rangle \\
10. & \quad = \underline{\mathrm{Ep}}(\rho b'.\mathit{update}\langle\!\langle q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),\underline{\mathrm{false}}),s, \\
 & \qquad\qquad \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),b'\rangle\!\rangle) \\
11. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \mathit{update}\langle\!\langle \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),\underline{\mathrm{false}}),s, \\
 & \qquad\qquad \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),b'\rangle\!\rangle)) \\
12. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \underline{\mathrm{Previous}}(\mathit{update}\langle\!\langle \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),s, \\
 & \qquad\qquad \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),b'\rangle\!\rangle, \\
 & \qquad\qquad \mathit{eval}\langle\!\langle \underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),s\rangle\!\rangle))) \\
13. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b'), \\
 & \qquad\qquad \mathit{eval}\langle\!\langle q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),\underline{\mathrm{false}}),s\rangle\!\rangle))) \\
14. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b'),\mathit{eval}\langle\!\langle q,s\rangle\!\rangle \vee \\
 & \qquad\qquad \mathit{eval}\langle\!\langle \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(\rho b.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b),\underline{\mathrm{false}}))),\underline{\mathrm{false}}),s\rangle\!\rangle))) \\
15. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b'),(\underline{\mathrm{true}} \vee \mathit{eval}\langle\!\langle \underline{\mathrm{false}},s\rangle\!\rangle)))) \\
16. & \quad = \underline{\mathrm{Ep}}(\rho b'.(q \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b'),\underline{\mathrm{true}})))
\end{array}$$

The evaluation of a next time formula in a state reduces, in effect, to its immediate subformula where any subsequent past time subformulas have been appropriately updated. Lines 9 to 13 above show how the function *update* works over the given subformula. Lines 14 to 16 complete the sub-evaluation which results with the $\underline{\mathrm{Previous}}$ subformula containing $\underline{\mathrm{true}}$ as the value of $Ep(q)$ for the next application of *eval*.

Another example Having now seen some detailed working of *init* and *update*, we now illustrate the evaluation of a temporal monitor containing both future and past temporal operators, together with data values, i.e. the first order linear-time temporal logic formula, $\Box(x > 0 \to \exists k((k = x) \wedge \Diamond(z > 0 \wedge y = k)))$. A specification for this monitor can be presented in EAGLE as follows:

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

At the beginning of monitoring the function *init* is applied to M as follows:

$$\begin{aligned}
F &= init\langle\!\langle M, \mathtt{null}, \mathtt{null}\rangle\!\rangle\\
&= init\langle\!\langle \mathtt{A}(x > 0 \rightarrow \mathtt{Ev}(x)), \mathtt{null}, \mathtt{null}\rangle\!\rangle\\
&= \underline{\text{A}}(\rho b_1.((x > 0) \rightarrow\\
&\qquad \underline{\text{Ev}}(\rho b_2.\underline{\text{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \underline{\text{Previous}}(\underline{\text{Ep}}(b_3), \underline{\text{false}}))), x)) \wedge \underline{\text{Next}}(\underline{\text{A}}(b_1)))
\end{aligned}$$

Given the state 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 above formula on this sequence takes place as follows:
Step 1: $s = \{x = 0, y = 3, z = 1\}$

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

Observe that in the above step the second argument of <u>Previous</u> is partially evaluated as the value of k is not available.
Step 2: $s = \{x = 0, y = 5, z = 2\}$

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

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, Complexity and Experiments

In this section we describe an implementation of the monitoring framework, discuss its complexity, and describe briefly an experimentation on a NASA software.

4.1 Implementation

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 and represents the datatype for that 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. Currently the implementation 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.

To make the implementation efficient, we use the propositional logic decision procedure of Hsiang [13]. The procedure reduces a tautological formula to the constant true, a false formula to the constant false, and all other formulas to canonical forms, which are an 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. In particular the equations $\phi \wedge \phi = \phi$ and $\phi \oplus \phi =$ false below ensures that the size of a formula remains small during monitoring.

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

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 an `update` method corresponding to the *init*, *value*, *eval* and *update* functions in the calculus. The arguments of a transformed rule corresponds to the fields in the class and they are initialized through the constructor. The choice of generating Java classes for each rule is for efficiency. In our implementation EAGLE expressions can be any Java expression. 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 the other different Java expression objects corresponding to the different variables (formula 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`.

When all the Java classes have been generated, the engine compiles them all, creates a list of monitors (which are also formulas) and starts their evaluation. 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 becomes the new monitors for the next state. If at any point a monitor 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.

4.2 Complexity in Special Case

In our work in [4] we showed how EAGLE can perform linear temporal logic (LTL) monitoring in an efficient way. For an initial formula of size m, we established upper bounds of $O(m^2 2^m \log m)$ and $O(m^4 2^{2m} \log^2 m)$ for the space and time complexity, respectively, of single step evaluation over the input trace. This shows that our implementation's space and time complexity is exponential in the size of the formula when we restrict ourselves to LTL with both past and future time temporal operators. This is independent of the length of the trace for single step evaluation. This makes it very efficient in terms of space as we do not store the trace either explicitly or implicitly. However, we found that the efficiency and complexity analysis of the general EAGLE monitoring algorithm is difficult and can be shown to be dependent on both the length of the trace and the size of initial formula in the worst case. In particular, it can be shown that in the general case with data-values the size of a formula depends both on the length of the trace and the size of the initial formula.

4.3 Experiment

The EAGLE logic has been applied in the testing of a planetary rover controller, as part of an ongoing collaborative effort with other colleagues (see [2]) to create a fully automated test-case generation and execution environment for this application. The controller operates a rover, named K9, which essentially is a small car/robot on wheels. K9 itself is a prototype, and serves to form the basis of experiments with rover missions on Mars. The controller consists in its current state of 35,000 lines of C++ code and executes plans given as input. A plan is a tree-like structure of actions and sub-actions. The leaf-actions control the rover hardware components. Each action has associated with it time constraints indicating when it should start and when it should terminate.

The testing environment, named X9 (explorer of K9), contains a test-case generator, that automatically generates input plans for the controller from a grammar describing the structure of plans. A model checker extended with symbolic execution is used to generate the plans [14]. Additionally, for each input plan a set of temporal formulas is generated, that the execution of that plan should satisfy. The controller is executed on each generated plan, and the implementation of EAGLE is used to monitor that the generated execution trace satisfies the formulas generated for that particular plan. The controller has been hand-instrumented in a few places to generate this trace. As an example, consider that a plan contains an action *moveCamera*, and that it should execute for no longer than 10 seconds. Then the following real-time temporal property can be generated, and monitored during execution:

$$\begin{array}{l}\underline{\text{max}}\ \texttt{CheckCameraMovement()} = \\ \quad \texttt{Always}(\mathit{start}(\text{“}\mathit{moveCamera}\text{”}) \rightarrow \texttt{EventuallyRel}(10, \mathit{end}(\text{“}\mathit{moveCamera}\text{”})))\ .\end{array}$$

During the very first test of the controller using EAGLE, approximately 300 test-cases were generated, and a previously unknown error was detected, demonstrating that a certain task did not recognize the too early termination of some other task. In earlier experiments, see [2], just propositional temporal logic without the real-time constraints was used. For example, the formula that would be monitored would be: $\Box(\mathit{start}(\text{“}\mathit{moveCamera}\text{”}) \rightarrow \Diamond \mathit{end}(\text{“}\mathit{moveCamera}\text{”}))$. The above error was not caught during the earlier experiments, although others were.

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 described an elegant monitoring algorithm for EAGLE that avoids the storage of trace. 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. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle does Space Efficient LTL Monitoring. Pre-Print CSPP-25, University of Manchester, Department of Computer Science, October 2003. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp25.pdf.
5. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04) (To appear in LNCS)*, January 2004. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp24.pdf.
6. D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
7. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.
8. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of Runtime Verification (RV'02)* [1], pages 36–55.
9. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of Runtime Verification (RV'01)* [1], pages 44–60.
10. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. ENTCS, 2001. Coronado Island, California.
11. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
12. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
13. Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
14. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of TACAS 2003. Warsaw, Poland.*, April 2003.
15. D. Kortenkamp, T. Milam, R. Simmons, and J. Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proceedings of RV'01* [1], pages 133–151.
16. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
17. 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.
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 Workshop on* Runtime Verification (RV'03) [1], pages 162–181.