Abstract—In this paper, a uniform calculus-based approach for synthesizing monitors checking correctness properties specified by a large variety of logics at runtime is provided, including future and past time logics, interval logics, state machine and parameterized temporal logics. We present a calculus mechanism to synthesize monitors from the logical specification for the incremental analysis of execution traces during test and real run. The monitor detects both good and bad prefix of a particular kind, namely those that are informative for the property under investigation. We elaborate the procedure of calculus as monitors.

Keywords—calculus, eagle logic, monitor synthesis, runtime verification

I. INTRODUCTION

RUNTIME verification (RV) is an emerging lightweight verification technique in which executions of systems under scrutiny are checked for satisfaction or violation of given correctness properties, while it complements verification techniques such as model checking and testing, it also paves the way for not-only detecting incorrect behavior of a software system but also for reacting and potentially healing the system when a correctness violation is encountered.

Typically, monitors are generated automatically from some high-level specification. Runtime verification, which has its roots in model checking, often employs some variant of linear temporal logic, such as Pnueli’s LTL [2]. Meanwhile some more other formal methods are required to describe different requirement, for example, future and past time logics, interval logics, state machine and parameterized temporal logics. Accordingly, a lot of monitor construction method has been studied. [3] presents a rewriting algorithm for efficiently testing future time line Temporal logic formulae. [4] makes use of the characterization that finite trace LTL can be defined recursively, both on the structure of the formulae and on the size of the executing trace, and presents that an efficient dynamic programming algorithms can be generated from any LTL formulae. The commercial tool Temporal Rover (TR) [5,6] supports a fixed future and past time LTL, with the possibility of specifying real-time and data constraints as annotations on the temporal operators. It implementation is based on alternating automata. Algorithms using alternating automaton to monitor LTL properties are also proposed in [7].

In [8], the approach consists of translating LTL formulae to finite-state automaton, the translation algorithms modifies standard LTL to Büchi automaton conversion techniques to generate automata that check finite program traces.

First, it’s clearly that above monitor construction methods orient to special monitor logic. Different monitor construction methods have been given for different specification logic. Second, the above methods are all based on finite trace semantics. The standard semantics of above specification logic are often based on infinite state sequences, but during the running, you can only see the finite prefix of the infinite run at any time. So it is rational for the monitor to give the verdict of the infinite run based on the finite prefix. But the above finite trace semantics are all inconsistencies. In other words, the monitor semantics (finite trace semantics) often are not consistent with the standard semantics. A logic L is called consistent if there exists not a model M and a formula \( \varphi = \bot \) such that \( M \models \varphi \) and \( M \models \neg \varphi \). For example, consider the LTL formula \( \varphi = Xp \) and \( \varphi = \neg Xp \). \( \varphi \in L \)

The remainder of this paper is organized as follows. Section 2 discusses two foundation works: informative prefix and Eagle logic. Section 3 introduces the calculus process for varieties of primitive operators, next-time operator \((\circ)\), previous-state operator \((\bigcirc)\), and rule definitions. Then shows the workings of the calculus algorithm through two examples, finally gives the concise proof of correctness. Section 4 closes the paper with discussion and conclusion.
II. PRELIMINARIES

A. Informative prefix

Various approaches to runtime verification and reasoning about systems based on truncated paths have been based upon a seminal paper by Kupferman and Vardi[1]. In [1], safety formulas are classified into three kinds, the intentionally safe, the accidentally safe and the pathologically safe, depending on the kinds of prefixes their properties possess. A prefix \( \sigma \) is called informative for a formula if it "tells the whole story"[1] of why the formula holds for every infinite state sequence of which \( \sigma \) is a prefix. Intentionally safe formulas are formulas of which every bad prefix is informative (e.g. \( \square p \)), an accidentally safe formula is a safety formula of which all state sequences that violate it, do have some informative bad prefix (e.g. \( \square (\neg p \land q \land \neg q) \), examples from [1])). Pathologically safe formulas are formulas that have computations that violate it without any informative bad prefix.

Formally, the definition of informative prefix can be defined as follow:

Definition 1 Let \( \sigma = a_0 a_1 \ldots a_n \in \Sigma^* \) be a finite state sequence, \( \sigma \) is informative for \( \varphi \) iff there exists a finite sequence \( T \in (2^{\Sigma^*})^* \) of sets of formulas, for each \( n \subseteq |T_r| \), such that \( \varphi \in T_r(0) \).

\( T_r(n+1) \) is empty,

for all \( 0 \leq i < n \) and \( \varphi \in T(i) \), the following hold.

- if \( \varphi \) is an atomic proposition, then \( \varphi \in a_i \),
- if \( \varphi = \psi_1 \lor \psi_2 \), then \( \psi_1 \in T_r(i) \lor \psi_2 \in T_r(i) \).
- if \( \varphi = \neg \psi_1 \), then \( \psi_1 \in T_r(i+1) \).

We call such a sequence \( T_r \) an informative sequence. If such an informative sequence exists, it tells us why \( \varphi \) holds for any extension of the prefix \( \sigma \). It indicates what formulas hold at what moment of the prefix and why. Since \( T(i) \) is at some point empty, this reasoning is complete and thus applies to any extension of the prefix. For example, if \( \varphi_1 \land \varphi_2 \in T_r(i) \), according to the definition, \( \varphi_1 \in T_r(i) \) and \( \varphi_2 \in T_r(i) \), which tells us that \( \varphi_1 \land \varphi_2 \) holds for any extension of the prefix. The informative bad prefixes can be considered as the only proper counterexamples, since they demonstrate why the formula does not hold or hold. So it is helpful to fault diagnosis and fault localization. To find the informative prefix of the property under investigation is one of the main goal for runtime verification. [11] elaborates on the construction of the monitor for temporal logic properties in which the automaton forms the basis of a monitor that detects both good and bad informative prefix for the property under investigation. In this paper, we will give a monitor construction method based on calculus which support much more formal property specification, including future and past time logics, interval logics, state machine and parameterized temporal logics.

B. Eagle

The Eagle logic is designed to support finite trace monitoring, and contains a small set of powerful operators, which allow on 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. The equations are also called as rules. 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. Here we assume boolean expressions over individual states as automatic propositions which comprise the finite trace. The expressiveness of the logic system is rich. Actually, any linear-time temporal logic, whose temporal modalities can be recursively defined over the next, past or concatenation modalities, can be embedded within it. Meanwhile the logic has supported a limited form of quantification. Interesting reader can refer to [9, 10] for details. We present the syntax and semantics below:

Syntax the syntax of EAGLE is shown in figure 1. A specification is 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 both named (\( N \)). Each rule definition is preceded by one of the keywords \( min \) or \( max \), indicating at the end of the trace how to interpret the semantic of the rules. A parameter type can either be for, 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 \( Form \). Any recursive call on a rule must be strictly guarded by a temporal operator. The propositions of the logic are Boolean expressions over an observer state. Formulas are composed using standard proposition logic operators together with a next-time operator (\( \circ \), a previous-state operator (\( \odot \) and a concatenation operator \( ( \circ F \), a previous-state operator \( ( \odot F \) and a concatenation operator \( F_1 \circ F_2 \)).

\[
S := DO
\]

\[
D := R^*
\]

\[
O := M
\]

\[
R ::= \max \min N(T_1 x_1, \ldots, T_n x_n) = F
\]

\[
M ::= \text{mon } N = F
\]

\[
T ::= \text{Form} | \text{primitive} | \text{type}
\]

\[
F ::= \exp \text{expression} \text{true} | \text{false} \mid \neg F \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid F_1 \circ F_2 \mid F_1 \odot F_2 \mid N(F_1, \ldots, F_n) | x_j
\]

Fig. 1 Syntax of EAGLE

Semantics The model of EAGLE logic are execution traces. An execution trace \( \sigma \) is a finite sequence of program states \( \sigma = s_0 s_1 \ldots s_n \) where \( |\sigma| = n \) is the length of the trace. The ith state \( s_i \) is denoted by \( \sigma(i) \). The term \( \sigma(i,j) \) denotes the sub-trace of \( \sigma \) from position \( i \) to position \( j \), both position is included. The semantics of the logic is defined in terms of a satisfaction relation between execution traces and specifications. That is, given a trace \( \sigma \) and a specification \( D O \), satisfaction is defined as follows: \( \sigma \models D O \iff \forall (\text{mon } N = F) \in O, \sigma, 1 \models D O \). That is to say, if the trace, observed from position 1 (the first state) satisfied each monitored formula in a specification, the trace
satisfied the specification. The definition of the satisfaction relation \( \vdash_p \subseteq (\text{Trace} \times \text{nat}) \times \text{Form} \), for a set of rule definitions \( D \), is presented in Figure 2.

The evaluation of a formula \( F \) on a state \( s = \sigma \), \( i \models F \) iff \( \text{eval}(\sigma(i)) = \text{true} \) and evaluate(\( \text{eval}(\sigma(i)) \)) = \text{true}.

1. \( i \models F \) is true
2. \( i \not\models F \)
3. \( i \not\models \neg F \)
4. \( i \models F \land F \)
5. \( i \not\models F \lor F \)
6. \( i \models F \implies F \)
7. \( i \models F \land \neg F \)
8. \( i \not\models F \lor \neg F \)
9. \( i \models F \land \neg F \)
10. \( i \not\models F \lor \neg F \)

Formally, \( \langle \text{Form}, \text{exp}, \text{update}, \text{final-eval} \rangle \) is used to represent the semantics of \( \text{EAGLE} \).

A. Calculus

The \text{transform}, \text{eval}, update and final-eval functions are defined a priori for all operators except for the rule application.

The definitions of \text{transform}, \text{eval}, update and final-eval about rule application get generated based on the definition of rules in the specification.

For the sake of expression, function \text{transform}, \text{eval} and \text{update} are expressed as \( \text{Form} \times \text{Form} \rightarrow \text{Form} \) and \( \text{State} \times \text{Form} \times \text{Form} \rightarrow \text{Form} \) respectively. In other words, we give the two functions two more parameters respectively. The first parameter represents the formula which is before rule application. It is used to determine termination for a recursive rule application of \text{transform} and \text{update} on a rule, it is the head formula of a recursive rule application. The second parameter denotes the recursive variable that will replace any embedded recursive call on the head formula.

The \text{transform} is not yet in the context of a rule, its last two arguments are null. The definitions of \text{transform}, \text{eval}, update and \text{final-eval} on the different primitive operators are given in figure 3.

\text{transform}(true, Z, b) = true
\text{transform}(false, Z, b) = false
\text{transform}(\text{exp}, Z, b) = \exp
\text{transform}(F_1 \oplus F_2, Z, b) = \text{transform}(F_1, Z, b) \oplus \text{transform}(F_2, Z, b)
\text{transform}(-F, Z, b) = -\text{transform}(F, Z, b)
\text{eval}(true, s) = true
\text{eval}(false, s) = false
\text{eval}(\text{exp}, s) = \begin{cases} \text{value of exp in s, if s isn't "virtual" state} & \text{if exp is "virtual" state} \\ \text{exp, if s is "virtual" state} & \text{if s is "virtual" state} \\ \end{cases}
\text{eval}(F_1, s) = \text{eval}(F_1, s) \oplus \text{eval}(F_2, s)
\text{eval}(-F, s) = -\text{eval}(F, s)
\text{update}(true, s, Z, b) = true
\text{update}(false, s, Z, b) = false
\text{update}(\text{exp}, s, Z, b) = \exp
\text{update}(F_1, s) = \text{update}(F_1, s) \oplus \text{update}(F_2, s, s, Z, b)
\text{update}(-F, s, Z, b) = -\text{update}(F, s, Z, b)
\text{final} - \text{eval}(true) = true
\text{final} - \text{eval}(false) = false
\text{final} - \text{eval}(\text{exp}) = ?
\text{final} - \text{eval}(F_1, op F_2) = ?
\text{final} - \text{eval}(-F) = ?

Fig. 3 the definitions of \text{transform}, \text{eval}, update and final-value on primitive operator.

In the above definition, \( op \) can be \( \land, \lor, \rightarrow \). Observe that for the definitions on primitive operator, we never use the last two arguments of \text{transform} and \text{update}. In most of the definitions
we simply propagate the arguments to the subformula. The only difference from the counterexamples in the traditional EAGLE logic [9][10] is the final-value definition on formulas except true and false. At the end of finite trace, if the result formula is true, it shows that the finite trace is the informative good sequence for the property under investigation, if the result formula is false, it shows that the finite trace is the informative bad sequence for the property. According to the definition 1, the calculus process identifies the informative prefix for the property under investigation. Otherwise it shows that the property has no informative prefix or the current finite trace is only the proper prefix of its informative prefix. So we can not given the monitoring result only based on the current observed finite trace.

The functions transform, eval, update are defined in a special way for operators \( \oplus \) and \( \otimes \). For the operator \( \odot \) we introduce the operator Next: Form \( \rightarrow \) Form. Then we define transform, eval, update as follows:

\[
\text{transform}(F, Z, b) = \text{Next}(\text{transform}(F, Z, b))
\]

\[
\text{eval}(\text{Next}(F, s)) = \text{update}(F, s, null, null)
\]

\[
\text{update}(\text{Next}(F), s, Z, b) = \text{Next}(\text{update}(F, s, Z, b))
\]

The operator 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 the evaluation of this 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 Previous: Form \( \rightarrow \) Form. We define transform, eval, update for as follows:

\[
\text{transform}(F, Z, b) = \text{Previous}(\text{eval}(Y, Y))
\]

\[
\text{where } Y = \text{transform}(F, Z, b)
\]

\[
\text{eval}(\text{Previous}(F, past), s) = \text{eval}(past, s)
\]

\[
\text{update}(\text{Previous}(F, past), s, Z, b) = \text{Previous}(\text{update}(F, s, Z, b), \text{eval}(F, s))
\]

Here, the definitions of the three functions on operator are as same as the two-valued EAGLE logic. In update function we not only update the first argument F but also evaluate F and pass is as the second argument of Previous. Note that the \( \odot \) function only is used at the end of the finite trace, it only concerns whether the result formula is true/false or not. So it does not need to be defined on \( \oplus \) and \( \otimes \) more. The reason is same for rule application below.

B. Monitor Synthesis for Rules

In this paper, we will give different forms of rule definitions. In traditional Eagle logic [9][10], without loss of generality, the standard form of a rule is \( R(\text{Form} f_1, \ldots, \text{Form} f_m, T_1 p_1, \ldots, T_n p_n) = B \) where \( f_1, \ldots, f_m \) are arguments of type Form and \( p_1, \ldots, p_n \) are arguments of primitive type. There the rule definition is divided into two styles: max rules and min rules. But in this paper, it is not needed any more, because \( \odot \) function is not dependent on the rule types.

Without loss of generality, in this paper, the standard form of a rule is \( R(\text{Form} f_1, \ldots, \text{Form} f_m, T_1 p_1, \ldots, T_n p_n) = B \) where \( f_1, \ldots, f_m \) are arguments of type Form and \( p_1, \ldots, p_n \) are arguments of primitive type. Such a rule can be written in short as:

\[
R(\text{Form} \ f_1, \ T_1 \ p_1) = B
\]

Where \( f_1 \) and \( p_1 \) represent tuples of type \( \text{Form} \) and \( \text{Term} \) respectively. For such a rule we introduce an operator \( \overline{R} : \text{Form} \times \text{Term} \rightarrow \text{Form} \). Informally, the first argument of \( \overline{R} \) represents the transformed right hand side of the rule.

For the rule \( R(\text{Form} \ f_1, \ T_1 \ p_1) = B \), the definitions of transform, eval, update are synthesized as follows:

- \( \overline{\text{transform}}(R(F, P_i, b), \text{eval}(F, P_i)) = \overline{R}(b, P_i) \)
- \( \overline{\text{transform}}(R(F, P_i, b), \text{eval}(F, P_i)) = \overline{R}(b, P_i) \)
- \( \overline{\text{update}}(\text{eval}(R(b, P_i)), \overline{R}(b, P_i), \text{eval}(P_i, b)) = \overline{R}(b, P_i) \)
- \( \overline{\text{update}}(\text{eval}(R(b, P_i)), \overline{R}(b, P_i), \text{eval}(P_i, b)) = \overline{R}(b, P_i) \)
- \( \overline{\text{update}}(\text{eval}(R(b, P_i)), \overline{R}(b, P_i), \text{eval}(P_i, b)) = \overline{R}(b, P_i) \)
- \( \overline{\text{update}}(\text{eval}(R(b, P_i)), \overline{R}(b, P_i), \text{eval}(P_i, b)) = \overline{R}(b, P_i) \)

Note that the result of eval(P, s), where P is an expression, may be a partially evaluated expression if some of the variables referred to by the expressions are partially evaluated. The expression gets fully evaluated once all the variables referred to by the expressions are fully evaluated. The reader can refer to [9][10] for the detail.

C. Examples

We provide one example to show the workings of above calculus process which identifies the informative prefix for the property under investigation. In order to compare with traditional EAGLE logic, we use the example in [9], but different result will give.

\[
\text{Example}
\]

The property under investigation which is in modified Eagle form is:

\[
\text{Ep}(\text{Form } f) = f \lor \text{ Ep}(f)
\]

\[
\text{mon } M = \text{ eval}(q)
\]

The finite trace is \( \sigma \{ q \} \). For the \( \odot \) function, the first argument of \( \oplus \) function is applied on state \( \sigma_1 \) and \( \sigma_2 \) respectively. For such a rule we introduce an operator \( \overline{R} : \text{Form} \times \text{Term} \rightarrow \text{Form} \). Informally, the first argument of \( \overline{R} \) represents the transformed right hand side of the rule.

First, transform function is applied:

\[
\text{transform}(\text{eval}(P_i, s), \text{null, null}) \Rightarrow
\]

\[
\text{Next}(\text{eval}(P_i, s), \text{eval}(P_i, s), \text{false)))
\]

Second, eval function is applied on state \( \sigma_1 = \{ q \} \):

\[
\text{eval}(\text{transform}(\text{eval}(P_i, s), \text{null, null}), \sigma_1) \Rightarrow
\]

\[
\text{eval}(P_i, s), \text{true, true, true}, \sigma_2
\]

Third, eval function is applied again on state \( \sigma_2 = \{ q \} \):

\[
\text{eval}(P_i, s), \text{true, true, true, true}, \sigma_2
\]

Finally, at the end of the trace, the \( \odot \) function is applied on the result formula:

\[
\text{final} = \text{eval}(true) = true
\]

It is easy to see that \( \sigma = \{ q \} \) is the informative good prefix of the property and the calculus process identifies the
informative prefix and gives the verdict result: true. If $\sigma = \{q\}$, in traditional Eagle logic, the result of calculus will be false, because in [9], at the end of the trace, if the result formula is min rule, the verdict is false, if the result formula is max rule, the verdict is true. There the definition of Ep rule will be:

$$\min(\text{Form } f) = f \lor \ Ep(f)$$

Which is min rule, So at the end of the trace (second step), the calculus process of traditional Eagle will give the verdict: false. But based on our calculus, the calculus process will give verdict: ?. it shows that the current trace $\{q\}$ is only the proper prefix of informative prefix $\{p\}$.

IV. CONCLUSION

In this paper, a calculus-based approach for synthesizing monitors checking correctness properties specified in multiple kinds of logic which can be represented by modified Eagle logic. Different from the traditional Eagle logic, the rule definition in modified Eagle logic does not distinguish with the max and min rule. So at the end of the trace, the verdict result is not dependent on the rule type, but only concerns with whether the result formula is true, false or otherwise. Indeed the calculus process provides a informative sequence for the property under investigation. So the calculus process identifies the informative good or bad prefix for the property under investigation.

REFERENCES