International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:10, 2009

Calculus-based Runtime Verification

Xuan Qi, Changzhi Zhao

***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 descript 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 linear 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 $\varphi \in L$ such that $M \models \varphi$ and $M \models \neg\varphi$. For example, consider the LTL formulae $\varphi = Xp$ and $\varphi' = X\neg p$, $u \in \Sigma^*$ be a finite trace consisting of only a single element, such that u=a with a={p}. Theoretically, in a finite trace interpretation of LTL, we could have both $u|\neq\varphi$ and $u|\neq\neg\varphi$, since there is no successor action available to satisfy either case. But sometimes the consistency between finite trace semantics and standard semantics of the specification logic is vital important. We call the consistency of the monitor logic as impartiality.

How to design a unifying logic in which all these logics can be modeled and a unifying framework in which different monitor constructions are as the same way. And how to make the monitor satisfied the property of impartiality are two critical questions which will be solved in this paper. [9,10] present a rule-based framework, called EAGLE, that has been shown to be capable to defining and implementing a range of finite trace monitoring logics, including future and past time temporal logic, real-time and metric temporal logics, forms of quantified temporal logics, and so on. Our work is significantly influenced by idea and implementation of EAGLE. But the monitor semantics does obey impartiality and the corresponding monitor identifies the a particular kind of good and bad prefix, namely informative good/bad prefix for the property under investigation.

The remainder of this paper is organized as follows. Section 2 discuses two foundation works: informative prefix and Eagle logic. Section 3 introduces the calculus process for varieties of primitive operators, next-time operator (○), previous-state operator (□), 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..

Xuan Qi is with Beijing Institude of System Engineering, P.R.China (Beijing 9702 Box, 100101, P.R.China ; e-mail: qixuanhappy@126.com).

Changzhi Zhao is with School of Computer, National University of Defense Technology, P.R.China .

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 σ 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 σ is a prefix. Intentionally safe formulas are formulas of which every bad prefix is informative (e.g. □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. □(p∨(Oq∧O¬q, examples from [1]))) . Pathologically safe safety 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 σ= $a_0a_1\ldots a_n \in \sum^*$ be a finite state sequence, σ is informative for φ iff there exists a finite sequence $Tr \in (2^{LTL})^*$ of sets of formulas, for each n≤|Tr|, such that

$\varphi \in Tr(0)$

$Tr(n+1)$ *is empty,*

for all $0\langle i \leq n$ *and* $\psi \in Tr(i)$, *the following hold.*

$-$*if* ψ *is an atomic proposition , then* $\psi \in a_i$.

$-$*if* $\psi = \psi_1$ *op* ψ_2, *then* $\psi_1 \in Tr(i)$ *op* $\psi_2 \in Tr(i)$.

$-$*if* $\psi = \psi_1 \bigcup \psi_2$, *then* (1) $\psi_2 \in Tr(i)$ *or*

(2) ($\psi_1 \in Tr(i)$ *and* $\psi_1 \bigcup \psi_2 \in Tr(i+1)$).

$-$*if* $\psi = X\psi_1$, *then* $\psi_1 \in Tr(i+1)$.

We call such a sequence Tr an informative sequence. If such an informative sequence exists, it tells us why φ holds for any extension of the prefix σ. It indicates what formulas hold at what moment of the prefix and why. Since Tr(i) is at some point empty, this reasoning is complete and thus applies to any extension of the prefix. For example, if $\varphi_1 \wedge \varphi_2 \in Tr(i)$, according to the definition, $\varphi_1 \in Tr(i)$ and $\varphi_2 \in Tr(i)$, which tells us that $\varphi_1 \wedge \varphi_2$ 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 *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*. 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 (○F), a previous-state operator (□F) and a concatenation operator $F_1 \bullet F_2$.

$$
\begin{aligned}
&S ::= DO\\
&D ::= R^*\\
&O ::= M\\
&R ::= \max \mid \min N(T_1 x_1, \cdots, T_n x_n) = F\\
&M ::= mon\ \ N = F\\
&T ::= Form \mid primitive\quad type\\
&F ::= \exp ression \mid true \mid false \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid\\
&\qquad F_1 \to F_2 \mid \circ F \mid \square F \mid F_1 \bullet F_2 \mid N(F_1, \cdots, F_n) \mid x_i
\end{aligned}
$$

Fig. 1 Syntax of EAGLE

Semantics The model of EAGLE logic are execution traces. An execution trace σ is a finite sequence of program states σ=$s_1s_2 \ldots s_n$, where |σ|=n is the length of the trace. The ith state s_i is denoted by σ(i). The term $\sigma^{[i,j]}$ denotes the sub-trace of σ 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 σ and a specification D O, satisfaction is defined as follows: σ|= D O iff ∀(mon N = F) ∈ O, σ, 1 $|=_D$F. 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 $|=_D \subseteq$(Trace×nat) ×Form, for a set of rule definitions D, is presented in Figure 2.

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

Fig. 2 the finite trace semantics of Eagle

From above definition, we can see that the low-level semantics of EAGLE logic are: for safety requirement, in the portion of the execution that we have observed, nothing bad happens, for eventualities, they are similarly required to be satisfied in the portion of the execution observed. Otherwise they will have 'not yet' been satisfied. It is clearly that the above definition doesn't follow the maxim of impartiality. So in this paper, we will modify the calculus process for EAGLE logic such that the monitor not only can give the precise result, i.e. if the monitor gives the positive result, the current running satisfied with the property, if the monitor gives the negative result, the current running didn't satisfy with the property. But also the calculus shows how and why the property is satisfied with or not. i.e. the calculus process identifies the good or bad informative prefix of the property under investigation.

III. Calculus Process

In this section, we outline the calculus process to determine that a given monitoring formula is satisfied, falsified or inconclusive (?) for some given finite input sequence of events. On the observer side a local state is maintained. The atomic propositions are specified with respect to the variables in this local state. Once an event is received, the observer modifies its local state; then evaluates the formula which has been evaluated on the prior states of 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, if the value of a formula is false, the formula is violated, otherwise, the value of the formula is inconclusive(?).

Our calculus process is inspired by monitoring algorithm used in EAGLE [9,10]. The calculus process is consisted of three steps. First, a monitor formula is transformed to other formula F' by applying rules recursively, until that the rule definition appears again. Second, the transformed formula is monitored against an execution trace by application of *eval*. The evaluation of a formula F on a state s=σ(i) in a trace σ results in a another formula F'=eval(F,s), F and F' satisfied the property that σ ,i|=F iff σ,i+1|=F'. The definition of the function eval: Form×State ->Form uses an auxiliary function *update*: Form ×State= Form. The role of the *update* function is to pre-evaluate a formula if it is guarded by the previous operator. Formally, update function has the property that σ,i|=OF iff σ,i+1 |=update(F, σ(i)). If the formula does not contain previous operator, the update function is not necessary. We can only use the identity: σ,i|=OF iff σ,i+1 |=F. At the end of the trace, a special function *fina-eval* : Form-> {true, false ,? } is applied. This is the key to determine which semantics the calculus process is followed. At the end of the observed finite trace, if the result formula is *true* formula, then the result of verification is true, if the result formula is *false* formula, then the result of verification is false, otherwise the result is inconclusive (?).

A. Calculus

The *transform, eval , update and final-eval* functions are defined a prior for all operators except for the rule application. The definitions of *transform, 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 *transform* and *update* are expressed as Form×Form×Form->Form and Form×State×Form×Form ->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 *transform and 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. If the *transform* is not yet in the context of a rule, its last two arguments are null. The definitions of *transform, eval, update* and *final-eval* on the different primitive operators are given in figure 3.

$$
\begin{array}{l}
transfrom(true, Z, b) = true \\
transfrom(\,false, Z, b) = \,false \\
transfrom(\exp, Z, b) = \exp \\
transfrom(F_1\ op\ F_2, Z, b) = transfrom(F_1, Z, b)\ op\ transfrom(F_2, Z, b) \\
transfrom(\neg F, Z, b) = \neg tranfrom(F, Z, b) \\
eval(true, s) = true \\
eval(\,false, s) = \,false \\
eval(\exp, s) = \begin{Bmatrix} value\ of\ \exp\ in\ s, if\ \ s\ isn't\ "virtual"\ state \\ \exp, if\ \ s\ is\ "virtual"\ state \end{Bmatrix} \\
eval(F_1\ op\ F_2, s) = eval(F_1, s)\ op\ eval(F_2, s) \\
eval(\neg F, s) = \neg eval(F, s) \\
update(true, s, Z, b) = true \\
update(\,false, s, Z, b) = \,false \\
update(\exp, s, Z, b) = \exp \\
update(F_1\ op\ F_2, s, Z, b) = update(F_1, s, Z, b)\ op\ update(F_2, s, Z, b) \\
update(\neg F, s, Z, b) = \neg update(F, s, Z, b) \\
final - eval(true) = true \\
final - eval(\,false) = \,false \\
final - eval(\exp) = ? \\
final - eval(F_1\ op\ F_2) = ? \\
final - eval(\neg F) = ?
\end{array}
$$

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

In the above definition, op can be ∧,∨,→. Observe that for the definitions on primitive operator, we never use the last two arguments of *transform* and *update*. In most of the definitions we simply propagate the arguments to the subformula. The only difference from the counterparts 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 ○and □ **·** For the operator ○we introduce the operator Next: Form ->Form. Then we define *transform, eval, update* as follows:

$$transform(\circ F, Z, b) = Next(transform(F, Z, b))$$

$$eval(Next(F), s) = update(F, s, null, null)$$

$$update(Next(F), s, Z, b) = Next(update(F, s, Z, b))$$

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 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 □ operator, we introduce the operator Previous: Form×Form->Form. We define *transform*, *eval*, *update* for □ as follows:

$$transform(\Box F, Z, b) = \Pr evious(Y, eval(Y))$$
$$where\ Y = transform(F, Z, b)$$

$$eval(\Pr evious(F, past), s) = eval(past, s)$$

$$update(\Pr evious(F, past), s, Z, b) = \Pr evious(update(F, s, Z, b),$$
$$eval(F, s))$$

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 *final-eval* 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 ○ and □ any 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 {max/min} R (Form f_1, … , Form f_m, T_1 p_1, …, T_n p_n)=B where f_1, … , f_m are arguments of type Form and p_1, …, 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 *final-eval* function is not dependent on the rule types.

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

$$R(\overline{Form}\ \ \overline{f}, \overline{T}\ \ \overline{p}) = B$$

Where $\overline{f}$ and $\overline{p}$ represent tuples of type $\overline{Form}$ and $\overline{T}$ respectively. For such a rule we intro- duce an operator $\overline{R} : Form \times \overline{T} \to Form$. Informally, the first argument of $\overline{R}$ represents the transformed right hand side of the rule.

For the rule $R(\overline{Form}\ \ \overline{f}, \overline{T}\ \ \overline{p}) = B$, the definitions of *transform*, *eval*, *update* are synthesized as follows:

$$transform(R(\overline{F},\overline{P}), R(\overline{F}, \overline{P'}), b) = \overline{R}(b, \overline{P})$$

$$transform(R(\overline{F},\overline{P}), Z, b) = \overline{R}(\rho b'.transform(F[\overline{f} \mapsto \overline{Y}], R(\overline{F}, \overline{P}), b'), \overline{P})$$
$$where\ \overline{Y} = transform(\overline{F}, Z, b)\ and\ Z\ not\ match\ R(\overline{F}, ?)$$

$$update(\overline{R}(\rho b.H(b), \overline{P}), s, \overline{R}(\rho b.H(b), \overline{P'}), b') = \overline{R}(b', \overline{P})$$

$$update(\overline{R}(\rho b.H(b), \overline{P}), s, Z, b') = \overline{R}(\rho b''.update(H(\rho b.H(b)), s, \overline{R}(\rho b.H(b), \overline{P})), \overline{P})$$
$$where\ Z\ \ not\ match\ \overline{R}(\rho b.(H(b), ?)$$

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

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.

Example the property under investigation which is in modified Eagle form is:

$$Ep(Form\quad f) = f \vee \Box\, Ep(f)$$

$$mon\quad M = \circ\, Ep(q)$$

The finite trace is σ={q}{}.

First, *transform* function is applied:

$$transform(\circ\, Ep(q), null, null) ==$$
$$Next(\overline{Ep}(\rho b.(q \vee \Pr evious(\overline{Ep}(b),\ false))))$$

Second, *eval* function is applied on state $\sigma_1 = \{q\}$:

$$eval(transform(\circ\, Ep(q), null, null), \sigma_1) ==$$
$$\overline{Ep}(\rho b'.(q \vee \Pr evious(\overline{Ep}(b'), true)))))$$

Third, eval function is applied again on state $\sigma_2 = \{\}$:

$$eval(\overline{Ep}(\rho b'(q \vee \Pr evious(\overline{Ep}(b', true)))), \sigma_2)$$

1. $= eval(q \vee \Pr evious(\overline{Ep}(\rho b'.(q \vee$
   $\Pr evious(\overline{Ep}(b'), true))), true), \sigma_2)$
2. $= eval(q, \sigma_2) \vee eval(\Pr evious(\overline{Ep}(\rho b'.(q \vee$
   $\Pr evious(\overline{Ep}(b'), true))), true), \sigma_2)$
3. $= q \vee eval(true, \sigma_2)$
4. $= q \vee true = true$

Finally, at the end of the trace, the *final-eval* function is applied on the result formula:

$$final - eval(true) = true$$

It is easy to see that σ={q}{} is the informative good prefix of the property and the calculus process identifies the informative prefix and gives the verdict result: true. If σ={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:

$$\min(Form\ f) = f \vee \Box\, Ep(f)$$

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

[1] Kupferman, O. and M. Y. Vardi, “Model checking of safety properties,” in: N. Halbwachs and D. Peled, editors, Computer Aided Verification: 11th International Conference Proceedings, CAV’99, Trento, Italy, July 6-10, 1999 (LNCS 1633) (1999), pp.172-183.

[2] Pnueli, A. “The temporal logic of programs.” In: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS), 1977, pp.46-57.

[3] Havelund, K., Rosu, G. “Monitoring Programs using rewriting.” In proceedings of Inter- national Conference on Automated Software Engineering (ASE’01), 2001, pp.135-143.

[4] Havelun, K., Rosu, G. “Synthesizing monitors for Safety Properties.” In Tools and Algorithms for Construction and Analysis of Systems (TACAS’02), 2002, pp.342-356.

[5] Drusinsky, D. “The Temporal Rover and the ATG Rover.” In SPIN Model Checking and Software Verification, volume 1885 of LNCS, 2000, pp.323-330.

[6] Drusinsky, D. “Monitoring Temporal Rules Combined with Time Series.” In CAV’03, volume 2725 of LNCS, 2003, pp.114-118.

[7] Finkbeiner, B., Sipma, H. “Checking Finite Traces using Alternating Automata.” In Proceedings of the 1st International Workshop on Runtime Verification(RV’01), 2001, pp.44-60.

[8] Giannakopoulou, D., Havelund, K. “Automata-Based Verification of Temporal Properties on Running Programs.” In Proceedings of International Conference on Automated Software Engineering(ASE’01), 2001, pp. 412-416.

[9] Barringer, H., Goldberg, A., Havelund, K., and Sen, K. “Eagle Monitors by Collecting Facts and Generating Obligations.” Pre-Print CSPP-26, University of Manchester, Department of Computer Science, 2003.

[10] Barringer, H., Goldberg, A., Havelund, K., and Sen, K. “Rule-Based Runtime Verification.” In Proceedings of Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI’04), 2004.

[11] Geilen, M. “On the construction of monitors for temporal logic properties.” In Electronic Notes in Theoretical Computer Science, 55, 2001.