Jeagle: a JAVA Runtime Verification tool

Marcelo D'Amorim[1]* ** and Klaus Havelund[2]

[1] Department of Computer Science, University of Illinois Urbana-Champaign, USA
[2] Kestrel Technology, NASA Ames Research Center, USA

Abstract. We introduce the temporal logic Jeagle and its supporting tool for runtime verification of Java programs. A monitor for an Jeagle formula checks if a finite trace of program events satisfies the formula. Jeagle is a programming oriented extension of the rule-based powerful Eagle logic 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, real-time and metric temporal logics, interval logics, forms of quantified temporal logics, and so on. Monitoring is achieved on a state-by-state basis avoiding any need to store the input trace. Jeagle extends Eagle with constructs for capturing parameterized program events such as method calls and method returns. Parameters can be the objects that methods are called upon, arguments to methods, and return values. Jeagle allows one to refer to these in formulas. The tool performs automated program instrumentation using AspectJ. We show the transformational semantics of Jeagle.

1 Introduction

Model Checking [6], Theorem Proving [14] and Static Analysis [16] are techniques aiming at static program verification. The first is concerned with checking if all possible traces derived from a program (or abstract model) satisfy a property of interest. The state-space explosion is known to be an issue when considering concurrency and unbounded types. Additional model abstraction, such as partial-order reduction, can reduce the model size considerably but *scalability* is still an issue when checking properties of programs in general. Theorem Proving relies on the language semantics and a proof system in order to come up with a proof that the system will behave correctly for all possible inputs. The proof system can be defined inductively on the syntax of the program. This technique requires user ingenuity to produce checkable predicates. For instance, there are valid properties which are non-inductive. That is, can not be proved valid using induction on the syntax. In practice, only reachable program states must satisfy stated properties. In order to cope with non-inductive properties, the user may start by specifying a high-level property and then providing more information to prune the set of possible transitions. This technique is thus not *fully mechanizable*. That is, it requires user intervention. As yet another technique, static analysis is concerned with analyzing the program offline and generating summarized information about its elements. The outcome of the analysis is usually imprecise but can be interpreted conservatively to produce program transformers that can, for example, optimize the code or simplify it with respect to some property that wants to be observed.

* CAPES grant# 15021917

** This author is grateful for the support received from MCT while participating in the Summer Student Research Program at the NASA Ames Research Center

In contrast to these techniques, this paper describes a logic and tool that employs dynamic analysis to detect bugs in software during its execution. Runtime Verification (RV) [1] is concerned with checking a single trace of the program against properties described in some logic. When a property is violated or validated the program can take actions to deal with it. The technique scales since just one model is considered. In addition, since the check is done during runtime only reachable states are touched. The technique can be used both for testing and monitoring. In the first case, one must come up with test cases [2] that might exercise a bug. In this setting RV is considered as an auxiliary tool to automate the creation of oracles that detect errors. An RV tool can also be used to monitor a program run so to take actions in response to the violation of a property. Under this perspective the RV tool may be used to define how the program reacts to bugs, possibly steering it to the correct behavior [8].

The paper describes a logic and its tools, named JEAGLE, for runtime verification of JAVA programs. By using a dialect of JAVA, the user can describe temporal properties relating different points in the program and their accessible objects, and verify the program against these properties during runtime. The logic is defined on top of EAGLE which is more expressive than several logics [3]. EAGLE not only allows one to state temporal, and interval properties but also to define new logics.

Instrumentation is acknowledged as an issue that runtime verification tools have to face in order to monitor programs [5, 13, 10]. Some tools provide no support for mechanical instrumentation, others use annotations in the source program to check against verification formulae. We understand that automated instrumentation is part of the problem we want to solve, and that a tight integration between the logic and the source language will not only simplify the task of writing and reasoning about properties but also give opportunity to mechanical instrumentation.

We claim that by augmenting the EAGLE language with a simple construct that allows one to bind data values from parameterized events of the program is a way of achieving this goal. This construct is the event expression and has been mainly influenced by aspect languages [12] and process algebras [15].

We present related work in the following section. In section 3 EAGLE is described. Section 4 presents JEAGLE as an extension to EAGLE. It first describes the tool; the language syntax and monitor examples are then given. Finally, a transformational semantics is defined and the implementation discussed. Section 5 concludes this work.

2 Related Work

EAGLE [3] is a language-independent runtime verification tool and logic. It requires the user to create a projection of the actual program state. User-defined formulae are evaluated with respect to this projected state. The EAGLE language essentially extends the μ-calculus with data parameterization. JEAGLE is defined on top of EAGLE and supports automated instrumentation and object reasoning in the expense of making the language specific to Java.

JAVA MAC [13] defines an event-based language to describe monitors. MAC is comprised of two specification languages, PEDL and MEDL. The first is tightly integrated to the programming language and defines events that might occur during the program execution. A MEDL specification, on the other hand, makes use of these events

in order to state high-level requirements. JEAGLE can define MAC conditions as rules and local variables can be cast as formal parameters in these rules. In contrast to MAC that allows user-defined high-level events to be described, JEAGLE event expressions only concern method calls and returns currently. However, this construct is designed to be extensible so to allow one to reason about other program events. In addition, JEAGLE supports data binding and object reasoning which we believe to be an essential feature of object-oriented program monitoring. To the best of our knowledge MAC does not support them.

JASS [4] is a JAVA tool providing a trace-assertion checker in addition to a language for describing pre and post conditions for methods, loop variants (used to assure loop termination) and invariants, and class invariants which are predicates about the state of objects of a particular class. These are defined in a similar fashion as in EIFFEL [19]. The language of trace-assertions is similar to CSP and interests us the most. Trace assertions are defined as class invariants in the form of annotations in the class file. The notation and semantics of the data-binding construct is similar to those used in modal logics for process-algebras like CCS and π-calculus [15]. These works influenced us severely on the integration of program and logic as well as on the notation and semantics of event expressions. We understand that the distinction between JASS and JEAGLE rely mainly on the expressiveness of their language. For example, JEAGLE do not provide built-in operators for parallel composition and hiding and this could make the set of possible traces easier to define.

TEMPORAL ROVER [7] allows the user specify LTL requirements to be checked during runtime. The programmer needs to manually instrument the program in order to emit events to the checker. Similarly to Temporal Rover, JEAGLE supports LTL with past and future combined. In contrast, JEAGLE provides automated program instrumentation, can capture data via events of different points in the program, and allows one to write specifications in different logics.

MOP [5] is a methodology and framework for building program monitors. In MOP the craft of a monitoring tool is divided into building a logic engine and a logic plugin. The first is concerned with generating a software artifact that will check the trace. The later is concerned with the integration of the target program and the logic engine. Instrumentation and IDE integration are supported by the engine. Several plugins have been created in this line already including those for ERE and LTL. We believe JEAGLE can be defined in MOP as well.

3 The Eagle Logic

In this section, the EAGLE finite-trace monitoring logic is introduced. This section as well as the Appendix A are modifications of part of [3] and serve to give background on EAGLE.

EAGLE offers a succinct but powerful set of primitives, essentially supporting recursive parameterized equations, with a minimal/maximal fix-point semantics together with three temporal operators: next-time, previous-time, and concatenation. The next-time and previous-time operators can be used for defining future and past time logics on top of EAGLE. The concatenation operator can be used to define interval logics and extended regular expressions. Rules can be parameterized with formulas and data, which

allows the definition of new combinators and contexts to be captured in different points in time.

Atomic propositions are boolean expressions over a *user-defined object* denoting the current state of the program. This design decision allows one to monitor programs written in different languages with reduced effort. That is, one needs to define such state object in JAVA, which is the EAGLE implementation language, and send events to it in order to keep it updated. The logic is first introduced informally by means of two examples. Syntax and semantics are given in Appendix A.

3.1 EAGLE by example

Assume we want to state a property about a program P, which contains the declaration of two integer variables x and y. We want to state that whenever x is positive then eventually y becomes positive. The property can be written as follows in classical future time LTL: $\Box(x > 0 \rightarrow \Diamond\, y > 0)$. The formulas $\Box F$ (meaning "*always F*") and $\Diamond F$ (meaning "*eventually F*"), for some property F, usually satisfy the following congruences [14], where the temporal operator $\bigcirc F$ stands for *next F* (meaning "*in next state F*"):

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

One can, for example, show that $\Box F$ is a solution to the recursive equation $X = F \wedge \bigcirc X$; in fact it is the maximal solution[3]. A fundamental idea in EAGLE is to support this kind of recursive definition, and to enable users to define their own temporal combinators using equations similar to those above. In this framework one can write the following definitions for the combinators `Always` and `Eventually`, and the formula to be monitored (M_1):

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

The `Always` operator is defined as having a maximal fix-point interpretation. That is, if by the end of the trace the property was not yet violated it is assumed to be validated. On the other hand, the `Eventually` operator is defined as having a minimal interpretation. If by the end of the trace the formula was not yet validated the eventuality is considered violated. Maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually happens). In EAGLE, the difference only becomes important when evaluating formulas at the boundaries of a trace. To understand how this works it suffices to say here that monitored rules evolve as new states appears in the trace. Assume that the end of the trace has been reached (we are beyond the last state) and a monitored formula F has evolved to F'. Then all rule applications in F' of maximal fix-point interpretation will evaluate to true, since they represent safety properties that apparently have been satisfied throughout the trace, while applications of minimal fix-point rules will evaluate to false, indicating that some event did not happen. Assume for example that we evaluate the formula M_1 in a state where $x > 0$ and $y \leq 0$, then as a liveness obligation for the

[3] Similarly, $\Diamond F$ is a *minimal* solution to the recursive equation $X = F \vee \bigcirc X$

future we will have the expression:

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

Assume that, at this point, we detect the end of the trace. That is, we are beyond the last state. The outstanding liveness obligation $\texttt{Eventually}(y > 0)$ has not yet been fulfi lled, which is an error. This is captured by the evaluation of the minimal fi x-point combinator `Eventually` being false at this point. The obligation corresponding to the right-hand side of the $\wedge$, namely, $\texttt{Always}(x{>}0 \rightarrow \texttt{Eventually}(y{>}0))$, is a safety property and evaluates to true.

For completeness we provide remaining defi nitions of the future time LTL operators $\mathcal{U}$ (until) and $\mathcal{W}$ (unless) below, and also the past-time operator $\mathcal{S}$ (since) used in an example later on. Note how `Unless` is defi ned in terms of other operators. However, it could have been defi ned recursively.

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

Data Parameters

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

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

Rule R is parameterized with an integer k, and is instantiated in M_2 when $x > 0$, hence capturing the value of x at that moment. Rule `R` replaces the existential quantifi er. The logic also provides a previous-time operator, which allows us to defi ne past time operators. Data parameterization is also used to elegantly model real-time logics. The syntax and semantics of EAGLE is defi ned in Appendix A. See [3] for more details on EAGLE and how to encode LTL, MTL in the language. The textual notations for $\bigcirc$ and $\odot$ in EAGLE are respectively @ and #.

3.2 The EAGLE tool

EAGLE monitors and rules are specifi ed in a text fi le. In order to verify the program against the stated properties, the programmer must instrument the application in points affecting any formula in the specifi cation. In the example above, at any place where x and y are updated. In these points, the EAGLE state must be updated and then the formula verifi ed as fi gure 1 shows. Straight lines denote events sent from the instrumented program.

When an instrumentation point is hit, the EAGLE state is updated (1). Then, the observer corresponding to the specifi ed properties (spec) is notifi ed (2). In response, the observer evaluates the formulae in the current state (3) and derive new obligations for the future which are stored in its internal state.

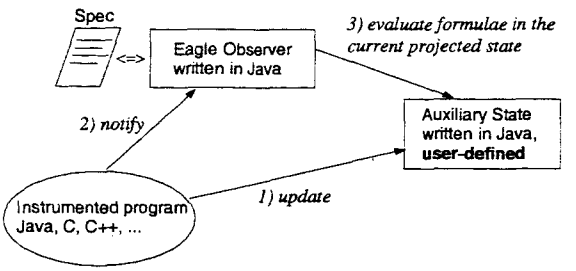

Fig. 1. Eagle architecture

4 JEAGLE

JEAGLE is a logic and tool for runtime verifi cation of JAVA programs. It is built on top of EAGLE. That is, monitors defi ned in this language are translated to EAGLE monitors. Because of this, we claim that JEAGLE is not more expressive than EAGLE. On the one hand, the logic JEAGLE aims at providing a more concise syntax for writing object-oriented temporal specifi cations. On the other hand, the tool automates program instrumentation and the generation of EAGLE monitors therefore reducing errors and programming effort.

In the previous examples x and y are fi elds declared in the EAGLE state. In EAGLE, this state is all the observer class can access (see fi gure 1). In EAGLE it is not possible to write, directly in the specifi cation fi le, properties about program objects and how they interact. In order to achieve this, one needs to insert (or update) such objects in the EAGLE state whenever an event of interest takes place; and create rules denoting the scope of these objects, as R in $M2$. These tasks are time-consuming and error-prone.

The JEAGLE language allows the user to reason about program objects rather than this artifi cial state. That is, the user is able to refer to the program state directly not through a possibly inaccurate projection of it. This is realized through events. In practice, the user is able to declare interest in program events and these can carry references to objects that are subject to reasoning.

JEAGLE extends the EAGLE language with JAVA expressions and with a construct hereafter called *event expression*. This construct has been mainly inspired by modal logic and aspect languages.

JEAGLE in runtime

During program execution the state contains information about the most recent event emitted which is also declared in the specifi cation. We create AspectJ aspects [12] to track events that occur in the formulae and update the state.

Some methods declared in the state check if an event has occurred. These methods are called from the observer to decide if the state satisfi es the (event) guard of an event expression. In practice, whenever a program point of interest is hit, the EAGLE state gets updated and the formulae are checked by the observer.

4.1 Syntax and Informal Semantics

The syntax of JEAGLE is defined in figure 2. This is a simplification of a grammar defined in a JLEX/JAVA CUP [11] specification.

Event expressions take the form: `[event]assertion` and `⟨event⟩assertion`, and extend the language of boolean propositions. So far events correspond to a method being called or returning from and may bind variables in the scope of an JEAGLE `assertion`. Question marks are used within the event description for this purpose. In addition, the construct `[...]...` has an implicative semantics, and `⟨...⟩...` a conjunctive semantics. Therefore, `[e]false` means that e must not occur while `⟨e⟩true` means that e must occur.

A transformation system is formally defined in the next section. The semantics of JEAGLE can be inferred by interpreting the rules of this system. However, the semantics is very detailed. Therefore, we understand it is still worth showing the informal semantics of the event expression. We use a functional pseudo-language in this attempt.

Informal semantics of an event expression

In what follows we show the semantics of an event expression whose event is associated to a method call. Note that other events are possible to be defined:

$[\![$ [*eaglepp_method_expression*: ev] *eaglepp_expr*: epp $]\!] \equiv [\![$

```
if ev_ then
  let {(x,y) | y? ∈ FV(ev) ∧ x ∉ FV(epp) } in
    epp [x⃗ / y⃗]
else true
```

$]\!]$

ev_ is a predicate that is valid when an event associated to the method call *eaglepp_method_expression* occurs, and $[\vec{x} / \vec{y}]$ denotes the sequence of substitutions $[x_1/y_1] \ldots [x_n/y_n]$ over the pairs (x_i, y_i) in the binding set. Note that y is an identifier occurring free in ev and labeled with a question mark, and x does not occur free in epp. That is, it is a fresh name.

Expressions defined inside brackets are **not** evaluated. The program will be instrumented to track the events associated to them. When the event triggers, the expression that follows the bracket (epp) is evaluated in the extended environment. The other forms of event expression have similar semantics with identifiers labeled with question marks possibly binding the calling thread or the result of a method.

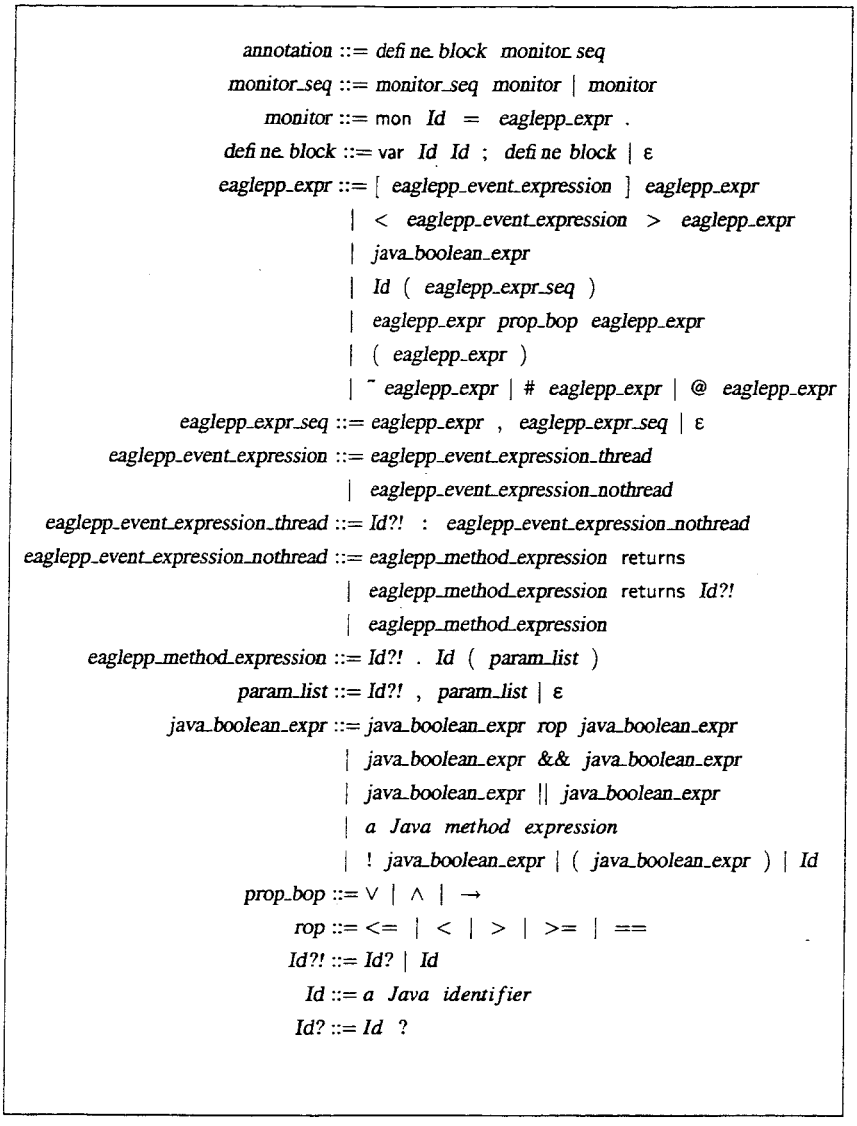

Fig. 2. Subset of the JEAGLE grammar

4.2 JEAGLE by Example

Temporal Buffer Requirements

```
observer BufferMonitor {
  ...
  var Buffer b ;
  var Object o ;

  mon M0 =
    Always ( [b?.put(o?)] Eventually( <b.get() returns k?> k == o ) ) .

  mon M1 =
    Always ( [b?.put(o?)] @ ( Always ( [b.put(o)]false ) ) ) .
}
```

Monitor `M0` states a property that all buffers must be empty by the end of the trace, while monitor `M1` states that an object can not be added to any buffer more than once. The eventuality of `M0` can also be expressed as: `<b.get() returns o>true`.
Note that we assume events to be disjoint. That is, two events do not occur simultaneously. In other words, events have an interleaving semantics. In addition, recall that formulae are interpreted as data unless they have to be evaluated in the current state.

Strict Alternation in acquire and release of Locks

The monitors `M2` and `M3` below state that there should not be an acquire of a lock without a future release, and there should not be a release without a past acquire. The term "`t? :`" qualifies the event description with the thread from which the event was sent.

```
observer FileSystemMonitor {
  ...
  var Thread t ;
  var FileSystem fs ;
  var int l ;

  mon M2 =
    Always ([t? : fs?.acquireLock(l?)] @ (
      Until([t: fs.acquireLock(l)]false, <t: fs.releaseLock(l)>true) ) ) .

  mon M3 =
    Always ( [t? : fs?.releaseLock(l?)] # (
      Since( [t: fs.releaseLock(l)]false , <t: fs.acquireLock(l)>true) ) ) .
}
```

The first requirement detects a missing release that could lead to starvation and deadlock since other threads could depend on this lock. The second detects the release of a lock whose current thread does not own.

4.3 Transformation Rules

JEAGLE specifications are translated into EAGLE monitors and instrumentation artifacts. Three components need to be produced out of an JEAGLE specification: (1) An aspect in the ASPECTJ [12] language that will instrument the program to emit events associated to event expressions, (2) the EAGLE state (as described in section 3.2), and (3) standard EAGLE monitors and rules corresponding to the JEAGLE requirements.

We now describe a subset of the transformation semantics for JEAGLE as a set of axioms and rules over the relation $\rhd \subseteq Config \times Config$, where $Config = \Gamma \times \mathcal{N} \times \mathcal{B} \times \mathcal{R} \times \mathcal{A} \times \mathcal{S} \times Term$, defined as follows:

- Γ is the type environment that is carried over in order to type methods that will be generated.
- $\mathcal{N}$ is the set of natural numbers. This item is used to generate fresh identifiers.
- $\mathcal{B}$ denotes an environment qualifying names that occur in event expressions with attributes that we associate with some runtime abstraction, e.g. "the first argument of method m". $\mathcal{B}$ denotes a function $Id \rightarrow (\ keywords \cup Id\ )$. Keywords represent data values captured in the last event notified by the program run. For example, the event `b?.put(o?)` adds the pairs [b $\mapsto$ caller] and [o $\mapsto$ arg1] to the map. Note that the mappings above are only valid until the next event arrives. The mappings of `b` and `o` are replaced by the identity when a rule binding these names is created.
- $\mathcal{R}$ denotes the set of rules to be added to the resulting EAGLE specification. Each rule has the form: $R_n \mapsto (\overrightarrow{p}, t)$, where R_n is a name identifying the rule, $\overrightarrow{p}$ is a list of (name $\times$ type) denoting the formals, and t is the rule body.
- $\mathcal{A}$ is the set of aspect pointcuts and advices [12] for program instrumentation.
- $\mathcal{S}$ is the set of methods that need to be defined in the EAGLE `State` class. This is the only way standard EAGLE can access data values from the formula.
- $Term = \bigcup \mathcal{L}(k)$, where k is a syntactic category in the grammar defined. *Term* corresponds to the union of the languages defined by each JEAGLE connected component.

Assume the following variables: ev denotes an *eaglepp_event_expression*, epp an *eaglepp_expression*, eppseq an *eaglepp_expr_seq*. t denotes a *Term*, tseq a *Term Sequence*, γ a type environment, n a natural number, b a name environment, r a set of rules, a a set of aspects. s denotes a set of method declarations, and id an *Id*. These names denote different variables when appearing primed or with a number suffix.

[Event expression]:

$$\frac{(\gamma, n, b, r, a, s, ev) \quad \rhd \quad (\gamma, n, b', r, a', s', t) \qquad (\gamma, n, b'', r, a', s', epp) \rhd (\gamma, n', b, r', a'', s'', t')}{(\gamma, n, b, r, a, s, [ev]epp) \rhd (\gamma, n', b, r'[R_n \mapsto (\overrightarrow{p}, t')], a'', s'', t \rightarrow R_n(\overrightarrow{x}))}$$

The following applies to the variables in the rule: $n' = n + 1$, $\overrightarrow{x}$ is a sequence of the form <..., `getValue(b'(k))`,...>, and $\overrightarrow{p}$ is a sequence of the form <...,(k,γ(k)),...> where k $\in FV$(ev) $\cap$ Id?. In other words, k is a free identifier labeled with a question mark, declared in ev, and possibly used in epp. The rule call $R_n(\overrightarrow{x})$ defines a new scope in

which these identifiers are bound. The binding map b' gives the keywords to each k, and γ their types. `getValue` is the name of a method in the JEAGLE state that will be used to access the objects denoting these keywords. Note that a different map (b") is carried over to transform epp. This happens because the identifiers are already bound in the rule R_n. So b" equals to b with $[k \mapsto k]$ for all $k \in FV(ev) \cap Id?$. b' and b" are discarded after the transformation.

The event expression $\langle ... \rangle ...$ has similar semantics. They differ in the final term produced. Instead of $t \rightarrow R_n(\overrightarrow{x})$ this construct produces the term $t \wedge R_n(\overrightarrow{x})$.

[Rule Application]:

$$\frac{(\gamma, n, b, r, a, s, eppseq) \rhd (\gamma, n, b, r', a', s', tseq)}{(\gamma, n, b, r, a, s, id0(eppseq)) \rhd (\gamma, n, b, r', a', s', id0(tseq))}$$

meaning that the transformation of a rule application depends only on its actual parameters. The rule for JEAGLE expression sequence follows:

$$\frac{\begin{array}{c}(\gamma, n0, b, r, a, s, epp) \quad \rhd \quad (\gamma, n1, b, r', a', s', t) \\ (\gamma, n1, b, r', a', s', epp\_seq) \rhd (\gamma, n2, b, r'', a'', s'', t')\end{array}}{(\gamma, n0, b, r, a, s, \ \ epp \text{ ";" } epp\ seq) \rhd (\gamma, n2, b, r' \cup r'', a' \cup a'', s' \cup s'', \ \ t \text{ ";" } ts)}$$

[Event binding all variables (example)]:

One can define events of many forms and each may bind variables differently. We decided to fix the format of an event in order to make clear the use and definition of bindings. The next two transformations are instantiations of the "Event" transformation axiom (not showed).

We here define the translation of events denoting "method returns" in which the issuing thread, arguments, result, and target object are all passed as parameters to the rule that defines the continuing obligation. That is, all identifiers in the event are binding (appear labeled by a question mark).

$$\frac{}{(\gamma, n, b, r, a, s, \ id0? \ : \ id1?.id2.(plist) \ \text{returns} \ id3?) \rhd (\gamma, n', b', r, a', s', id_n())}$$

In this transformation n' = n + 1, id0 denotes the thread name, id1 the calling object, id2 the method name, and id3 the name of the returned value. This event is translated to a boolean expression ($id_n()$) retuning true when the declared event is the current event notified in the EAGLE state. In addition to this, the name environment must be updated as well as the aspect and set of state methods:

b' = b \ $\{[p_i \mapsto \text{arg}_i] \mid p_i = \text{head}(\text{tail}^i(\text{plist}))$ and $p_i \in Id?$ and $0 \leq i < |\text{plist}|\}$ \
{ [id1 $\mapsto$ caller], [id3 $\mapsto$ return], [id0 $\mapsto$ issuingThread] }. The symbol \ denotes right overriding of bindings as usual.

a' = a $\cup$ "pointcut to track returns of method id2 passing the calling thread, call target, return value, and parameters as arguments to a corresponding advice".

s' = s $\cup$ { " `public boolean` id_n `( ){ ... }` " }, where id_n is a fresh identifier. This method denotes an event that will be used in the EAGLE formula corresponding to

the event expression associated to this event. The event denoted by "`t: a.m(i?,j) returns o?`" drives the generation of a method that will return true when the caller is `a`, the method called is `m` with parameters as defined by $\gamma(i)$ and $\gamma(j)$, the second argument of the call is `j`, and the calling thread is `t`.

[JEAGLE Event using bound variables (example)]:

The previous rule and this differ essentially in how they build the aspect and the method denoting the event. These constructions are described by a' and s', n' = n + 1:

$$\frac{}{(\gamma, n, b, r, a, s,\ \mathsf{id0}\ :\ \mathsf{id1.id2.(plist)\ returns\ id3}) \rhd (\gamma, n', b, r, a', s', \mathsf{id}_n(\overrightarrow{x}))}$$

a' = a ∪ "pointcut and advice to track returns of method id2.

s' = s ∪ { " `public boolean` id_n $(\overrightarrow{p})$`{ ... }` " }, where id_n is a fresh identifier. This method returns true when the last event tracked with an aspect advice has id0 as the issuing thread, id1 as the calling object, and so forth. These identified variables are passed as parameters $(\overrightarrow{p})$ to $\mathsf{id}_n()$ while the last event is part of the EAGLE global state.

$\overrightarrow{x}$ is a sequence of the form[4]: $<\ldots, k, \ldots>$, where $k \in FV$("id0 : id1 . id2 . (plist) returns id3") $\cap$ (b $\lhd$ *Id*). Since all names appear without question marks there must be an enclosing rule in which these names bind formal parameters. This is checked by intersecting the set of names with (b $\lhd$ *Id*).

[JAVA Boolean Expression]:

$$\frac{}{(\gamma, n, b, r, a, s, \mathsf{bexp}) \rhd (\gamma, n', b, r, a, s', \mathsf{id}_n(\overrightarrow{x}))}$$

s' = s ∪ { " `public boolean` id_n `(` $\overrightarrow{p}$ `){ bexp }` " }. JAVA boolean expressions are translated into boolean methods that can be used as predicates in the generated EAGLE formula. Note that no transformation is applied within the JAVA boolean expression since the syntax of these are closed in the JAVA language (see section 4.1).

The method call construction is as defined in the previous rule and thus omitted. In addition, n' = n + 1.

[Monitor]:

$$\frac{(\gamma, 0, b, r, a, s, \mathsf{epp}) \rhd (\gamma, n, b, r', a', s', t)}{(\gamma, 0, \emptyset, \emptyset, \emptyset, \emptyset, \mathsf{mon\ \ id0\ \ =\ \ epp}) \rhd (\gamma, n, \emptyset, r', a', s', t)}$$

We omitt the rules for the define block since it is only associating names to type. Note that no new term is generated particularly in this rule. The EAGLE expression can be output as the concatenation of all rules collected (r') and a monitor of the form:

$$\mathtt{mon}\ \ \mathsf{id}_n = t\ .$$

Similarly, the aspects can be generated from a' and the state from s'.

[4] In the general "Event" transformation $\overrightarrow{x} = \overrightarrow{x}_1 \cdot \overrightarrow{x}_2$ where the first sequence is in charge of passing the values bound by names with question marks, and second sequence takes the form defined here.

4.4 Buffer Example revisited

Figure 3 depicts the format of a logic observer specification in JEAGLE for a Buffer monitor defined in the beginning of this section. The transformation of this specification produces the following EAGLE monitor and rules: `max R2(Object o, Object k) = compare_references(o,k) .`

```
max R1(Object b, Object o)=Eventually(get_(b)∧R2(o,getValue(ht,''return'')) .
mon M1 = Always(put_() → R1(getValue(ht,''caller'' ),getValue(ht,''arg1'')) .
```

We changed the name of the methods (to `get_()`, `put_()`, `compare_references()`) to improve understanding. These are declared in the EAGLE state.

```
...Eagle specification file with additional rules and monitors

observer BufferObserver {

 classPath = C:/downloads/src
 targetPath = C:/downloads/src
 terminationMethod = bufferexample.Barrier.end()

 var Buffer b ;
 var Object o ;
 var Object k ;

 mon M1 = Always( [b?.put(o?)]
             Eventually ( <b.get() returns k?> (o == k) ) ) .

}
```

Fig. 3. JEAGLE observer

`ht` is a hash table stored in the state and carried over to access the parameters of the last event. This is necessary because methods declared in the EAGLE state and called from the observer (corresponding to the specification above) can only access actual parameters. So we have to pass a table that maps keywords to their associated objects in the latest event.

Note that rules have `Object` as formals. This is not relevant because an EAGLE observer makes reflective calls to a state method assuming it has the formal types equal to the actual types of the arguments. We regard these as EAGLE implementation issues so they have been omitted thus far.

4.5 Implementation

A parser for JEAGLE was built using JLEX and JAVA CUP. Each transformation rule or axiom in the previous section corresponds to a `visit` method in a Visitor class [9] in charge of transforming the expression. The tool has 6500 lines of Java source code. The compiler works by transforming JEAGLE phrases written in a specification file into equivalent EAGLE phrases. A new file is produced as the result.

The state and aspect class associated to this specification are generated under the directory `targetPath` in a package named `monitors`. The user must inform where the classes mentioned in the specification are located. The `classPath` will be appended to the `ajc` (AspectJ compiler) `classpath` directive which will be called from inside the compiler. In addition, the user must inform the name of a method - `terminationMethod` - he will call when the program terminates. We track this call and inform EAGLE to finish observation. This is necessary to check eventualities.

5 Conclusion

We have described a tool that generates observers to monitor temporal properties of JAVA program. The tool and language is named JEAGLE and uses EAGLE as its base logic. The contribution of this work is twofold. First, instrumentation is automated. Recall that EAGLE does not support instrumentation since it is language-independent. Second, one can reason about program objects directly in the formula. This makes easier to writte object-oriented specifications and also guides program instrumentation.

Further work includes: (1) enhancing the tool to support additional events and wildcards, (2) developing a visualization tool for the remaining obligation of the formulae, and (3) pass vector-clocks as parameters of events in order to tame true concurrency. The first allows one collecting data-values from other events in the program and also declaring wildcards as actual parameters for the kinds of events described here. This should make the definition more concise and clear since only the essential variables are declared. The second is a tool to support temporal debugging, which has been reported to be a hard task in temporal specification [18]. Finally, we should consider a true concurrent model where events might happen simultaneously in different processors. We believe that by tagging events with vector clocks, as in [17], one can build the partial-order from the trace an that could be used as the model to observe.

References

1. *1st, 2nd, and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *ENTCS*. Elsevier Science: 2001, 2002, 2003.
2. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, volume 2589 of *LNCS*, pages 87–107. Springer, March 2003.
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of the 5th International Conference on Verification, Model Chacking, and Abstract Interpretation (VMCAI'04)*, volume 55(2), 70(4), 89(2) of *LNCS*, Venice, Italy, Jan 2004. Springer.
4. D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, Paris, France, 2001. Elsevier Science.
5. F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 106–125. Elsevier Science, 2003.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
7. 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.
8. D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification, Altrincham, April 1987*, volume 398 of *LNCS*, pages 409–448, 1989.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
10. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 97–114. Extended version to appear in the journal: Formal Methods in System Design, Kluwer, 2004.
11. S. E. Hudson, F. Flannery, C. S. Ananian, D. Wang, and A. W. Appel. CUP Parser Generator for Java. http://www.cs.princeton.edu/ appel/modern/java/CUP/.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th ECOOP*, Lecture Notes in Computer Science, pages 327–353. Springer-Verlag, 2001.
13. M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
14. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
15. R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, New York, 1992.
16. F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
17. K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *ESEC/FSE'03*. ACM, 2003. 1 - 5 September 2003, Helsinki, Finland.
18. M. H. Smith, G. J. Holzmann, and K. Etessami. Events and Constraints: A Graphical Editor for Capturing L ogic Requirements of Programs. In *Proceedings of the 5th IEEE Intl. Symposium on Requirements Engineering*, pages 14–22. IEEE Computer Society, Washington DC USA, 2001.
19. Eiffel Software. Eiffel language. http://www.eiffel.com/.

Appendix A: EAGLE Syntax and Formal Semantics

This appendix is a fragment of [3].

Syntax

A specification S consists of a declaration part D and an observer part O. D consists of zero or more rule definitions R, and O consists of zero or more monitor definitions M, which specify what to be monitored. Rules and monitors are named (N).

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

A rule definition R is preceded by a keyword indicating whether the interpretation is maximal or minimal (which determines the value of a rule application at the boundaries of the trace). Parameters are typed, and can either be a formula of type Form, or of a primitive type, such as int, long, float, etc.. The body of a rule/monitor is a boolean valued formula of the syntactic category *Form* (with meta-variables F, etc.). Any recursive call on a rule must be strictly guarded by a temporal operator. The propositions of this logic are boolean expressions over an observer state. Formulas are composed using standard propositional logic operators together with a next-state operator ($\bigcirc F$), a previous-state operator ($\odot F$), and a concatenation-operator ($F_1 \cdot F_2$). Finally, rules can be applied and their arguments must be type correct. That is, an argument of type Form can be any formula, with the restriction that if the argument is an expression, it must be of boolean type. An argument of a primitive type must be an expression of that type. Arguments can be referred to within the rule body (x_i).

In what follows, a rule N of the form

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

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

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

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

Semantics

The semantics of the logic is defined in terms of a satisfaction relation $\models$ between execution traces and specifications. An execution trace σ is a finite sequence of program

states $\sigma = s_1 s_2 \ldots s_n$, where $|\sigma| = n$ is the length of the trace. The i'th state s_i of a trace σ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of σ from position i to position j, both positions included; if $i \geq j$ then $\sigma^{[i,j]}$ denotes the empty trace. In the implementation a state is a user defined java object that is updated through a user provided *updateOnEvent* method for each new event generated by the program. Given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

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

That is, a trace satisfies a specification if the trace, observed from position 1 (the first state), satisfies each monitored formula. The definition of the satisfaction relation $\models_D \subseteq (\mathit{Trace} \times \mathbf{nat}) \times \underline{\text{Form}}$, for a set of rule definitions D, is presented below, where $0 \leq i \leq n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that the position of a trace can become 0 (before the first state) when going backwards, and can become $n+1$ (after the last state) when going forwards, both cases causing rule applications to evaluate to either true if maximal or false if minimal, without considering the body of the rules at that point.

$\sigma, i \models_D \mathit{expression}$ iff $1 \leq i \leq |\sigma|$ and $\mathit{evaluate}(\mathit{expression})(\sigma(i)) == \mathit{true}$
$\sigma, i \models_D \underline{\text{true}}$
$\sigma, i \not\models_D \underline{\text{false}}$
$\sigma, i \models_D \neg F$ iff $\sigma, i \not\models_D F$
$\sigma, i \models_D F_1 \wedge F_2$ iff $\sigma, i \models_D F_1$ and $\sigma, i \models_D F_2$
$\sigma, i \models_D F_1 \vee F_2$ iff $\sigma, i \models_D F_1$ or $\sigma, i \models_D F_2$
$\sigma, i \models_D F_1 \rightarrow F_2$ iff $\sigma, i \models_D F_1$ implies $\sigma, i \models_D F_2$
$\sigma, i \models_D \bigcirc F$ iff $i \leq |\sigma|$ and $\sigma, i+1 \models_D F$
$\sigma, i \models_D \odot F$ iff $1 \leq i$ and $\sigma, i-1 \models_D F$
$\sigma, i \models_D F_1 \cdot F_2$ iff $\exists j$ s.t. $i \leq j \leq |\sigma|+1$ and $\sigma^{[1,j-1]}, i \models_D F_1$ and $\sigma^{[j,|\sigma|]}, 1 \models_D F_2$

$\sigma, i \models_D N(\overline{F}, \overline{P})$ iff
- if $1 \leq i \leq |\sigma|$ then:
  - $\sigma, i \models_D F[\overline{f} \mapsto \overline{F}, \overline{p} \mapsto \mathit{evaluate}(\overline{P})(\sigma(i))]$
  - where $(N(\underline{\text{Form}}\ \overline{f}, \overline{T}\ \overline{p}) = F) \in D$
- otherwise, if $i = 0$ or $i = |\sigma|+1$ then:
  - rule N is defined as $\underline{\text{max}}$ in D

An expression (a proposition) is evaluated in the current state in case the position i is within the trace ($1 \leq i \leq n$). In the boundary cases ($i = 0$ and $i = n+1$) a proposition evaluates to false. Propositional operators have their standard semantics in all positions. A next-time formula $\bigcirc F$ evaluates to true if the current position is not beyond the last state and F holds in the next position. Dually for the previous-time formula. The concatenation formula $F_1 \cdot F_2$ is true if the trace σ can be split into two sub-traces $\sigma = \sigma_1 \sigma_2$, such that F_1 is true on σ_1, observed from the current position i, and F_2 is true on σ_2 (ignoring σ_1, and thereby limiting the scope of past time operators). Applying a rule within the trace (positions $1 \ldots n$) consists of replacing the call with the right-hand side of the definition, substituting arguments for formal parameters; if an argument is of primitive type its evaluation in the current state is substituted for the associated formal parameter of the rule, thereby capturing a desired freeze variable semantics. At the boundaries (0 and $n+1$) a rule application evaluates to true if and only if it is maximal.

Appendix B: Buffer Example

Source specification

```
/* LTL with past and future: */
// Future:
max Always(Term t) = t /\ @ Always(t) .
min Eventually(Term t) = t \/ @ Eventually(t) .
min Until(Term t1,Term t2) = t2 \/ (t1 /\ @ Until(t1,t2)) .
// Past:
max Sofar(Term t) = t /\ # Sofar(t) .
min Previously(Term t) = t \/ # Previously(t) .
min Since(Term t1,Term t2) = t2 \/ (t1 /\ # Since(t1,t2)) .

observer BufferObserver {

  classPath = C:/tests/eaglepp
  targetPath = C:/tests/eaglepp
  terminationMethod = bufferexample.Barrier.end()

  var bufferexample.Buffer b ;
  var Object o ;
  var Object k ;

  mon M1 = Always( [b?.put(o?)]
                   Eventually ( <b.get() returns k?> (o == k) ) ) .

}
```

Generated Specification

```
/* LTL with past and future: */
// Future:
max Always(Term t) = t /\ @ Always(t) .
min Eventually(Term t) = t \/ @ Eventually(t) .
min Until(Term t1,Term t2) = t2 \/ (t1 /\ @ Until(t1,t2)) .
// Past:
max Sofar(Term t) = t /\ # Sofar(t) .
min Previously(Term t) = t \/ # Previously(t) .
min Since(Term t1,Term t2) = t2 \/ (t1 /\ # Since(t1,t2)) .

max r_5(Object o, Object k)  = m_7(htable, o, k) .
max r_1(Object o, Object b) = Eventually(m_4(htable, b) /\ r_5(o, getValue(htable,c6) )) .
mon M1 = Always(m_0(htable) -> r_1(getValue(htable,c2) , getValue(htable,c3) )) .
```

Generated Instrumentation Aspects

```
package monitors;

import eagle.parser.RuleBase;
import eagle.rbmf.Observer;
import eagle.rbmf.EagleState;
import eaglepp.*;

public aspect BufferObserverAspect {

  BufferObserverState state = new BufferObserverState();
  Observer observer =
    new Observer(RuleBase.parse("C:/tests/eaglepp/bufferexample/buffer_compiled.spec"));
  Object lock = new Object();

  pointcut put_ ( bufferexample.Buffer caller  , Object arg0) :
      target(caller) && args(arg0) &&
      execution(* bufferexample.Buffer.put(Object));

  before  ( bufferexample.Buffer caller  , Object arg0) returning :
     put_ ( caller  , arg0 ){
     synchronized (lock) {
       MethodCall mcall = new MethodCall("caller", caller,
         new EagleMethod("bufferexample.Buffer","put", new String[]{"Object"}));
       mcall.addActualParameter("arg0",arg0  );
       state.setCurrentEvent(mcall);
       state.eventMessage();
       observer.handle(state);
     }
  }

  pointcut get_  ( bufferexample.Buffer caller ) :
    target(caller) && execution(* bufferexample.Buffer.get() );

  before  ( bufferexample.Buffer caller ) returning (Object result) :
    get_ ( caller  ){
      synchronized(lock) {
        MethodReturn mret = new MethodReturn(caller, new
          EagleMethod("bufferexample.Buffer","get", new String[]{}) , result );
        state.setCurrentEvent(mret);
        state.eventMessage();
        observer.handle(state);
      }
    }

  pointcut end_ () : call(* bufferexample.Barrier.end(..));

  before() : end_(){
    state.terminate();
    observer.end();
  }

}
```

Generated EAGLE State

```
package monitors;

import eaglepp.*; import java.util.*; import java.io.*;

public class BufferObserverState extends EaglePPState  {

   public static boolean m_0(Hashtable htable) {
      return (((String)getValue(htable,"methodName"))!=null &&
       ((String)getValue(htable,"methodName")).equals("put")) &&
       (((String)getValue(htable,"targetType"))!=null &&
       ((String)getValue(htable,"targetType")).equals("bufferexample.Buffer"));}

   public static boolean m_4(Hashtable htable, bufferexample.Buffer b) {
     return (((String)getValue(htable,"methodName"))!=null &&
      ((String)getValue(htable,"methodName")).equals("get")) &&
      (((String)getValue(htable,"targetType"))!=null &&
      ((String)getValue(htable,"targetType")).equals("bufferexample.Buffer")) &&
      ( getValue(htable,"caller") == b );}

   public static boolean m_7(Hashtable htable, Object o, Object k) {
      return (o == k) ; }

   public static final String c2 = "arg0";
   public static final String c3 = "caller";
   public static final String c6 = "retObject";
   private static File logFile =
      new File("bufferexample/errors.BufferObserverState");
   private static StringBuffer errorMessages = new StringBuffer();
   private static StringBuffer errorWarningMonitors = new StringBuffer();
   private static StringBuffer warningMessages = new StringBuffer();

   public void eventMessage() {
      errorWarningMonitors.append(printEventAsString()+"\n");
   }

   public void error(String args) {
     errorWarningMonitors.append("error: " + args + " was violated\n");
     errorMessages.append("error: " + args + " was violated\n");
   }

   public void warning(String args) {
     errorWarningMonitors.append("warning : monitor " + args + " was not validated.\n");
     warningMessages.append("warning : monitor " + args + " was not validated.\n");
   }

   public static void terminate() {
      System.out.println("----------------------------------------");
      System.out.println("SUMMARY FOR MONITORS");
      if (errorMessages.length()>0) {
        System.out.println(errorMessages.toString());
      } else {
        System.out.println("  no violation");
      }
      if (warningMessages.length()>0) {
        System.out.println(warningMessages.toString());
      } else {
        System.out.println("  eventualities validated");
      }
      System.out.println("----------------------------------------");
      try {
        PrintWriter pwriter = new PrintWriter(new FileWriter(logFile));
        pwriter.print(errorWarningMonitors.toString());
        pwriter.flush();
        pwriter.close();
       } catch (IOException ioException) {
        System.err.println("Could not write to the file."); } } }
```