Automated Runtime Verification with Eagle

Allen Goldberg and Klaus Havelund

Kestrel Technology, Palo Alto, California, USA
{goldberg,havelund}@kestreltechnology.com

Abstract. EAGLE is a very powerful logic for expressing behavioral properties about systems that evolve over time. Specifically, the logic can be used to monitor the execution of computer programs. EAGLE is an extension of propositional logic with three temporal kernel operators, with recursion, and with parameterization over formulas in the logic as well as over data values. Formula parameterization allows the user to define new temporal combinators. Data parameterization allows to define properties relating data values from different points in time. A wide range of different notations can be defined on top of EAGLE using these mechanisms, such as future and past time linear temporal logic, extended regular expressions, real-time logics, interval logics, forms of quantified temporal logics, and so on. EAGLE is implemented as a Java library. Monitoring is done on a state-by-state basis, without storing the execution trace. In this paper we demonstrate the logic on an event-based component model of a bounded priority queue.

1 Introduction

Enterprise information systems must meet rigorous requirements for secure, correct, and responsive behavior. In recent years a software verification method called *runtime verification* has been the subject of much research [1]. The target system is instrumented so as to produce a log of relevant events. The resulting event log is analyzed by a module called an *observer* that checks conformity with required functional and performance properties. However, unlike pure performance analysis, runtime verification is oriented towards mainly checking system behaviors. Its ability to do so is the result of the availability of a highly expressive language for asserting system properties, and the translation of these assertions into efficient monitoring code. We shall describe the powerful temporal logic, EAGLE, first introduced in [2], a rule-based language for specifying properties together with an interpreter for the language.

We believe this approach is particularly appealing for monitoring large scale enterprise information systems. It can be used during test, but also on-line during operation, to monitor the system and alert operators and/or developers when a system fails to meet functional or performance requirement. To avoid resource contention, the event log may be streamed to an observer executing on a separate processor. Indeed, because of its sophistication, EAGLE can be used to diagnose faults with sufficient precision to enable an automated response to the detected fault.

Temporal logics are a formal languages for expressing properties of computations i.e. temporally ordered sequences of states or events. Temporal logics were developed in response to the inadequacies of earlier formal program verification logics that focused on proving correct the input/output behavior of a sequential program. These logics are not appropriate for reactive, transaction based systems. Since the temporal logics were designed for proving correctness, usually by model checking, they were designed as a compromise between property expressibility and tractability of the model checker. However model checking has not effectively scaled to large systems. We designed EAGLE for monitoring the behavior of large scale systems. Producing a scalable monitor, while non-trivial, is not as difficult as scaling up model checking, and so our design goal favored maximal language expressibility. EAGLE was designed with a simple core and an extensibility mechanism that allows other temporal logics to be easily defined within it.

The remainder of the paper is organized as follows. Section 2 provides an high-level overview of EAGLE. Section 3 introduces a simple priority queue example, and its informal requirements, such as response time, security and system integrity properties. In Section 4 these requirements are formalized in EAGLE for monitoring. Section 5 concludes the paper.

2 The Eagle Temporal Logic

2.1 The Eagle System

The overall structure of the EAGLE system is shown in Figure 1. A monitored system is instrumented to produce an *event stream*. The structure of the events is user-defined. The state Updater responds to events by invoking a user-defined function that updates the EAGLE state. Recall that temporal logic allows the expression of properties over sequences of computation states. EAGLE does not directly analyze the evolving states of the target system, only the relevant data about the system that get stored in the EAGLE state. When the state is updated in response to an event, the EAGLE engine is invoked to check the validity of the user-defined EAGLE monitors.

EAGLE is implemented in Java. Thus to use EAGLE, the user must define the monitors, specify the EAGLE state as Java class that arbitrarily subclasses an EAGLE state base class, instrument the target system to generate event sequences, and write a method that updates the state in response to the event.

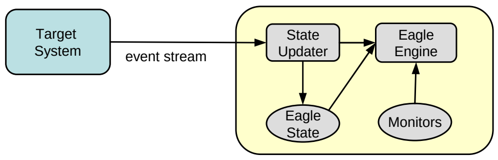


Fig. 1. The EAGLE system

2.2 Eagle Monitors

The logic allows primitive assertions about the state (written as Boolean-valued Java expressions on state), the usual Boolean connectives, and temporal operators that refer to states in the past or future. The logic defines two primitive temporal operators and uses *rules* to define others. The two primitive temporal operators are the next state operator, denoted $\bigcirc F$ and the previous state operator, denoted $\odot F$. The assertion $\bigcirc F$ is true if and only if the assertion F is true in the next state. The assertion $\odot F$ is true if and only if the assertion F is true in the previous state.

A typical temporal operator is $\diamond F$, read *eventually F*, which states that relative to the current state, there is a subsequent state in which the predicate F is true. For example the formula *send()* $\rightarrow \diamond$*ack()* asserts that if in the current state there is a *send* event then in some subsequent state there will be an *ack* event. To express $\diamond F$ in EAGLE we define a rule, written as

$$\underline{\text{min}}\ \texttt{E}(\underline{\text{Form}}\ F) = F \vee \bigcirc\texttt{E}(F)$$

Rules are like parameterized macros. In this case the macro E is defined with a parameter F of type Form (short for Formula). The rule states that $\texttt{E}(F)$ is true if F is true in the current state or if $\texttt{E}(F)$ is true in the next state. Notice that the rule is defined recursively and without a terminating case. To prevent infinite regress, EAGLE requires that any recursive reference to a rule name must be guarded by a next or previous operator. Mutually recursive rules are allowed. The keyword min at the start of the rule has the technical meaning that this rule is defined using minimal fixpoint semantics. In EAGLE to monitor the above temporal formula one writes:

$$\underline{\text{mon}}\ \texttt{SendsAck} = \textit{send()} \rightarrow \texttt{E}(\textit{ack()})$$

Another important operator in temporal logic is *always* (written $\Box F$) meaning that the formula F is true in every subsequent state. This is defined in EAGLE as

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

That is, $\texttt{A}(\underline{\text{Form}}\ F)$ is true if F is true in the current state and $\texttt{A}(\underline{\text{Form}}\ F)$ is true in the next state. Returning to the formula *send()* $\rightarrow \texttt{E}($*ack()*$)$, note that this formula only states that if a *send* event is detected in the *current* state it will eventually be acknowledged. This is a property that we probably want true for *every send* event. This intention is expressed by the EAGLE formula $\texttt{A}($*send()* $\rightarrow \texttt{E}($*ack()*$))$, which can be read as “whenever there is a send event then eventually there is an *ack* event.” The pattern of this formula is common and is called the *response pattern*. It asserts that if an event of one type occurs then the target system (eventually) responds to it.

Now suppose we wish to assert that once there is a *send* event there should not be another *send* until the acknowledgement of the first send is received. We do so by defining a rule $\texttt{U}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2)$ commonly called *until*, asserting that the formula F_1 should remain true in all states forward until one that satisfies F_2. Formally,

$$\underline{\text{min}}\ \texttt{U}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc\texttt{U}(F_1, F_2))$$

The above formulas are called *future time* formulas since the operators refer to subsequent states. EAGLE symmetrically supports past time operators. Built into the logic is the prior state operator, denoted $\odot$. The formula $\odot F$ is true in the current state if and only if F is true in the prior state. By analogy to the *eventually* operator, the *previously* operator, meaning sometime in the past, is defined by

$$\underline{\text{min}}\ \texttt{P}(\underline{\text{Form}}\ F) = F \vee \odot\,\texttt{P}(F)$$

Suppose we wish to assert that if an *ack* event occurred then there must have been a previous *send* event. This is stated with the past time formula $\texttt{A}(\textit{ack()} \rightarrow \texttt{P}(\textit{send()}))$. Past time formulas are often useful in stating security properties that assert that if some activity occurs then there was previously a valid request for that activity.

EAGLE has been designed to work both with continuously operating systems and those that terminate. The former corresponds to online monitoring of an event stream that is potentially infinite, and the latter to monitoring finite traces. In the case of a finite trace, at the end of the trace EAGLE determines whether each monitor succeeds or fails. It does so by evaluating the monitor in a "virtual" final state. Any primitive formula referring to the virtual state or the next state operator evaluates to false; any rule defined with the $\underline{\text{min}}$ ($\underline{\text{max}}$ resp.) keyword evaluates to *false* (*true*), and Boolean connectives are evaluated normally. Similar rules are used to evaluate a formula involving the prior state operator in the first state. For example, *always* (A) is defined as a maximal rule and will evaluate to true at the end of the trace. *Eventually* evaluates to false at the end of a trace. The intuition is that *always* is a *safety* property that was *true* at each encountered state and so should be *true* at the end of the trace, while eventually is a *progress* property which remained unfulfilled at the end of the trace. In temporal logic the *until* operator has a strong form, that was defined above, and a weak form. In EAGLE the weak form is defined as $\underline{\text{max}}\ \texttt{W}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc\texttt{U}(F_1, F_2))$. The only difference is that the weak form is defined as a *maximal* rule. The effect is that a finite trace all of whose states satisfy F_1 but not F_2 will satisfy the weak form but not the strong form.

Data Parameters. Now suppose that each *send* event has an associated identifier and that the *acknowledge* event repeat that identifier. Suppose we would like to check that the identifier of the received *acknowledge* corresponds to the identifier of the *send*. We assume that the identifier is an integer recorded in the EAGLE state and accessed by the method *getId()*. Above we have seen that rules may be parameterized by formulas; here we will see that they can also be parameterized by data values. Data parametrization allows assertions that relate the value of EAGLE state variables at different points in time. To solve the problem at hand, we bind the identifier of the *send* message to a rule parameter and then test the equality of the parameter against the value of the identifier of the *acknowledge* event.

$$\begin{array}{l}\underline{\text{min}}\ \texttt{Ei}(\underline{\text{int}}\ \textit{id}) = E(\textit{ack()} \wedge \textit{id} = \textit{getId()})\\ \underline{\text{mon}}\ \texttt{SendAct} = \texttt{Always}(\textit{send()} \rightarrow \texttt{Ei}(\textit{getId()}))\end{array}$$

The data parameters makes EAGLE richer than propositional temporal logic.

State Machines Many (but not all) properties expressible in EAGLE can be expressed using state machines, with the intuitive graphical notation that many people find easier to comprehend. In this section we shall describe a canonical translation of a rich formulation of state machines into EAGLE. This formulation of state machines implements demonic non-determinism, actions performed on transitions that update a collection of variables local to the state machine, arbitrary EAGLE formulas as transition guards, and the ability to assert a "success formula" or "failure formula" within a state. The machine is non-deterministic in the sense that if two transitions from a state are enabled, then conceptually the machine enters both states. A computation is accepting if *both* transitions lead to an accepting state. It is a simple matter to allow angelic non-determinism or no non-determinism at all. The semantics of a "success formula" is that if a state is entered and it has an associated success formula that evaluates to true, then that branch of the non-deterministic computation is accepted. If a failure formula is true, the state machine fails. If they are both true the success action takes precedence. Note, the success, failure and the formulas guarding transitions are arbitrary EAGLE formulas, and so may include past and future time temporal operators. Similar to many extended state machine notations, transitions may be labelled with actions, in this case assignments of expressions to a (subset) of the variables declared local to the state machine.

The translation of such a state machine to EAGLE maps each state uniformly to two rules: a staging rule and a main rule. We describe the translation for an arbitrary state S with success formula *succ*, failure formula *fail* and transitions $t_1, \ldots, t_n$. For state S the main rule is named S and the staging rule S'. Suppose $\overline{v} = v_1, \ldots, v_m$ are the local variables of the state machine and that they have types $\mathit{ty}_1, \ldots, \mathit{ty}_m$. Each rule is parameterized by the local variables, so the formal parameter list for each rule is $\mathit{ty}_1\ v_1, \ldots, \mathit{ty}_m\ v_m$. The main rule for S is $\underline{\mathrm{max}}(\underline{\mathrm{min}})$ if and only if S is accepting (rejecting). Suppose that transition t_i has destination state D_i, condition c_i, and action $\overline{v} := \overline{e}_i$. The body of the main rule is

$$\mathit{succ} \vee (\neg \mathit{fail} \wedge (c_1 \rightarrow {D_1}'(\overline{e}_1)) \wedge \cdots \wedge (c_n \rightarrow {D_n}'(\overline{e}_n)) \wedge \mathit{otherwise})$$

where *otherwise* is a formula

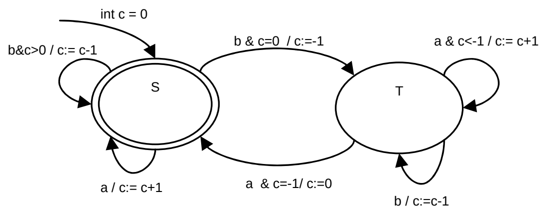
$$(\neg(c_1 \vee \ldots \vee c_n) \rightarrow S'(\overline{v}))$$

The definition of the staging rule is

$$\underline{\mathrm{max}}\ S'(\mathit{ty}_1\ v_1, \ldots, \mathit{ty}_m\ v_m) = \bigcirc S(\overline{v})$$

A state machine accepts a trace if and only if an EAGLE a monitor asserting the main rule of the initial state (parameterized by initial values) is true on that trace. The staging transformations insure that assignments on transitions are evaluated in the current EAGLE state. If the state machine has no data variables the staging rules are not needed (and the primes are dropped in the rule body).

Figure 2 gives an example. This state machine accepts a finite trace if and only if the number of *a* events is greater than or equal to the number of *b* events. S is the initial and only final state. One variable c is defined which counts the relative number of observed *a*'s and *b*'s. Figure 2 also shows the translation to EAGLE rules. The state machine is invoked by referencing the rule of the main rule of the initial state with the initial value of the local variable i.e $S(0)$.



$$
\begin{aligned}
\underline{\text{max}}\ \mathtt{S}(\underline{\text{int}}\ c) &= ((b() \wedge c > 0) \rightarrow \mathtt{S}'(c-1)) \\
&\quad \wedge((b() \wedge c = 0) \rightarrow \mathtt{T}'(-1)) \\
&\quad \wedge(a() \rightarrow \mathtt{S}'(c+1)) \\
&\quad \wedge(\neg((b() \wedge c > 0) \vee (b() \wedge c = 0) \vee a()) \rightarrow \mathtt{S}'(c)) \\
\underline{\text{max}}\ \mathtt{S}'(\underline{\text{int}}\ c) &= \bigcirc \mathtt{S}(c) \\
\underline{\text{max}}\ \mathtt{T}(\underline{\text{int}}\ c) &= ((a() \wedge c < -1) \rightarrow \mathtt{T}'(c+1)) \\
&\quad \wedge((a() \wedge c = -1) \rightarrow \mathtt{S}'(0)) \\
&\quad \wedge(b() \rightarrow \mathtt{T}'(c-1)) \\
&\quad \wedge(\neg((a() \wedge c < -1) \vee (a() \wedge c = -1) \vee b()) \rightarrow \mathtt{T}'(c)) \\
\underline{\text{max}}\ \mathtt{T}'(\underline{\text{int}}\ c) &= \bigcirc \mathtt{T}(c)
\end{aligned}
$$

Fig. 2. A state machine and its translation

3 A Priority Queue and its Properties

To illustrate Eagle we consider a simple event-based component model of computation. In this model, components have input and output ports. Events flow along connectors from an output port in one component to an input port in another using a push or data driven model. That is, arrival of an event on an input port causes computation in the component which in turn can generate additional events on its output ports. Communication is *synchronous* in the sense that a sender component delivers a message to a receiver in theoretically instant time. Events are arbitrary objects.

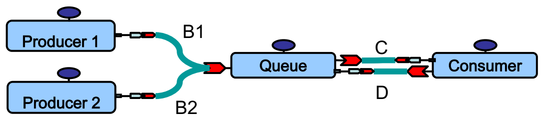


Fig. 3. A simple queue model

Figure 3 shows as an example a multiple-producer/single-consumer architecture mitigated by a priority queue, here with two producers. The figure shows the connections, named B_1, B_2, C and D, between the components as directed arrows, indicating the direction of the message flow. Each producer repeatedly sends messages to the queue on the connections respectively B_1 and B_2. The consumer repeatedly consumes a message on C, processes it, and then acknowledges the receipt of the message on D, thereby asking for a new message. Messages are time stamped and are furthermore identified by an ID. We assume that every message produced by a producer has a unique ID and that the acknowledgement includes this ID. While this example may be simple it is emblematic of a component-based typical pattern.

We will show how behavioral properties of this architecture can be expressed. We assume that each connector is instrumented so that when a message is transmitted on the connector an event is transmitted to the observer that checks conformance with the EAGLE properties. The nine properties stated informally in English below will be formalized in EAGLE in the next section.

requiredResponse *Every message from a producer (communicated on either B_1 or B_2) must be responded to and acknowledged (on D) by the consumer within* 50 *seconds.*

stayAliveQueue *Whenever a message is sent to the queue from a producer (on either B_1 or B_2), a message (not necessarily the same) must be consumed by the consumer (on C) within* 32 *seconds.*

limitedSize *There are never more that* 40 *messages in the queue.*

C_D_Alternation *The consumer should alternate between consuming messages (on C) and acknowledging messages (on D).*

stayAliveConsumer *Every message consumed (on C) by the consumer is processed and acknowledged (on D) within* 30 *seconds.*

noJunkConsumed *Every message consumed by the coonsumer (on C) has previously been produced (on B_1 or B_2).*

noJunkAcknowledged *Every message processed and acknowledged by the consumer (on D) has previously been consumed by the consumer (on C).*

orderPreserved *The queue behaves as a FIFO queue wrt. messages produced by producer 1. That is, the consumer consumes (on C) messages produced on B_1 in the order in which they were produced.*

B1HasPriority *Messages produced by producer 1 (on B_1) have priority over messages produced by producer 2 (on B_2). That is, as long as there are pending B_1 messages in the queue, no B_2 message will be consumed (on C) by the consumer.*

4 Formalization of Buffer Properties in Eagle

As mentioned, the components are assumed too be instrumented to emit events to the observer whenever a message is communicated on a connector. For each such event, the EAGLE monitor state is updated to store the event and its parameters. This section first defines the monitor state and some auxiliary rules, then the properties from the previous section are formalized. Three data items are relevant for each event: the kind of event (B_1, B_2, C or D), the identity of the message, and the time it occurred, measured in seconds since the start of the system. The following shows the state of the observer, including the declaration of a variable for each of these pieces of information.

```
class State extends EagleState {
  static String connector;
  static int ident, clock;
  static void update(String event) {...}
  static boolean B1(){return connector.equals("B1");}
  static boolean B2(){return connector.equals("B2");}
  static boolean C() {return connector.equals("C");}
  static boolean D() {return connector.equals("D");}
}
```

The method `update` is called for each new event, with a string representing the event. The methods `B1`, `B2`, `C` and `D` form the interface to the logic and return true when the corresponding events occur. For each new event, the EAGLE monitors defined below will be evaluated against this state. First, we shall define a few auxiliary application specific rules in addition to the temporal combinators `A`, `E`, `P`, `U` and `W` from Section 2. The following are shorthands for event recognizers (only those needed in this example are listed).

$$\begin{array}{l}\underline{\text{min}}\ \texttt{B12}() = \texttt{B1}() \vee \texttt{B2}() \\ \underline{\text{min}}\ \texttt{B2}(\underline{\text{int}}\ id) = \texttt{B2}() \wedge ident = id \\ \underline{\text{min}}\ \texttt{C}(\underline{\text{int}}\ id) = \texttt{C}() \wedge ident = id\end{array}$$

The rule `B12`() is a shorthand for the occurrence of either a B_1 or a B_2 event. The rules `B2`(<u>int</u> *id*) and `C`(<u>int</u> *id*) represent the occurrence of a production by producer 2, or a consumption by the consumer, of a message that is identified by *id*. These rules are defined as minimal since at the boundaries (before and after execution) no events occur. The following three rules define extensions of the combinators `E` and `P` that were introduced in Section 2. The extensions include constraints on time and message ids.

$$\begin{array}{l}\underline{\text{min}}\ \texttt{Et}(\underline{\text{Form}}\ t, \underline{\text{int}}\ time) = \texttt{E}(t \wedge clock \leq time) \\ \underline{\text{min}}\ \texttt{Eti}(\underline{\text{Form}}\ t, \underline{\text{int}}\ time, \underline{\text{int}}\ id) = \texttt{E}(t \wedge clock \leq time \wedge ident = id) \\ \underline{\text{min}}\ \texttt{Pi}(\underline{\text{Form}}\ t, \underline{\text{int}}\ id) = \texttt{P}(t \wedge ident = id)\end{array}$$

The rule `Et`(<u>Form</u> *t*, <u>int</u> *time*) denotes the property that *t* will eventually occur, and before the *clock* (one of the variables in the state) increases beyond *time*. The parameter *time* will capture this upper future time bound when the rule is applied. The rule `Eti`(<u>Form</u> *t*, <u>int</u> *time*, <u>int</u> *id*) is like `Et` but extended to also put a constraint on the message id. That is, it denotes the property that *t* will eventually occur, before *clock* increases beyond *time*, and with a message id being equal to *id*. The property `Pi`(<u>Form</u> *t*, <u>int</u> *id*) denotes the property that *t* occurred in the past and with a message id being equal to *id*.

The properties can now be formalized as follows.

requiredResponse

This property can be stated as the following response property (every event of type X must be followed later by some other event of type Y).

$$\underline{\text{mon}}\ \texttt{requiredResponse} = \texttt{A}(\texttt{B12}() \rightarrow \texttt{Eti}(\texttt{D}(), clock + 50, ident))$$

The monitor states that whenever a $\texttt{B1}()$ or $\texttt{B2}()$ event occurs (represented by $\texttt{B12}()$ evaluating to true), then $\texttt{Eti}(\texttt{D}(),\mathit{clock} + 50,\mathit{ident})$ holds. This in turns states that eventually within 50 seconds, a $\texttt{D}()$ communication must occur of that same message. This is expressed by passing as arguments to `Eti` the absolute time when it has to occur (the current time plus 50), and the identifier of the current message.

stayAliveQueue

This property can also be stated as a response property.

$$\underline{\text{mon}}\ \texttt{stayAliveQueue} = \texttt{A}(\texttt{B12}() \rightarrow \texttt{Et}(\texttt{C}(), \mathit{clock} + 32))$$

This monitor is expressed in the same style as the previous monitor. However, in this case we are not interested in the identity of the message consumed by the consumer, hence only the upper time bound is passed as parameter to `Et`.

limitedSize

This counting property can be stated as follows.

$$\begin{aligned}
\underline{\text{max}}\ \texttt{CountSize}(\underline{\text{int}}\ \mathit{size}) &= \texttt{B12}() \rightarrow (\mathit{size} < 40 \wedge \bigcirc\texttt{CountSize}(\mathit{size} + 1)) \\
&\quad \wedge \texttt{C}() \rightarrow \bigcirc\texttt{CountSize}(\mathit{size} - 1) \\
&\quad \wedge \texttt{D}() \rightarrow \bigcirc\texttt{CountSize}(\mathit{size}) \\
\underline{\text{mon}}\ \texttt{limitedSize} &= \texttt{CountSize}(0)
\end{aligned}$$

In order to express this property, we need to define an auxiliary rule `CountSize` that counts the number of pending events in the queue. The rule is parameterized with the current count, and is defined by recursion by case: if a $\texttt{B1}()$ or $\texttt{B2}()$ event occurs, then we require that the current count is less than 40, and in the next state we continue with a count increased by 1. $\texttt{C}()$ events cause the counter to be decremented, and $\texttt{D}()$ events have no effect. The monitor starts with a zero counter.

C_D_Alternation

This property can be formulated as a state machine with two states, `S1` and `S2`, with the initial state being `S1`. The state machine expresses the alternation between consumptions on `C` and acknowledgements on `D`.

$$\begin{aligned}
&\underline{\text{max}}\ \texttt{S1}() = (\texttt{C}() \rightarrow \bigcirc\texttt{S2}()) \wedge (\neg\texttt{C}() \rightarrow \bigcirc\texttt{S1}()) \wedge \neg\texttt{D}() \\
&\underline{\text{min}}\ \texttt{S2}() = (\texttt{D}() \rightarrow \bigcirc\texttt{S1}()) \wedge (\neg\texttt{D}() \rightarrow \bigcirc\texttt{S2}()) \wedge \neg\texttt{C}() \\
&\underline{\text{mon}}\ \texttt{C\_D\_Alternation} = \texttt{S1}()
\end{aligned}$$

For example, when in state `S1`, and a $\texttt{C}()$ event occurs, the next machine state will be $\texttt{S2}()$. A non $\texttt{C}()$ event causes the machine to stay in the currrent state. The fail condition for state $\texttt{S1}()$ is the occurrence of a $\texttt{D}()$ event. There is no success condition since the alternating property never gets satisfied finally, it's a property being monitored continuously. Note that state $\texttt{S1}()$ is defined as a maximal rule, while state $\texttt{S2}()$ is defined as a minimal rule, reflecting that state $\texttt{S1}()$ is an *accepting* state, while state $\texttt{S2}()$ is a *rejecting* state, that has to be left before the application terminates. This reflects the extra requirement that every $\texttt{C}()$ event has to be followed by a $\texttt{D}()$ event.

stayAliveConsumer

This property can be stated as a simple response property as follows.

$$\underline{\text{mon}}\ \texttt{stayAliveConsumer} = \texttt{A}(\texttt{C}() \rightarrow \texttt{Et}(\texttt{D}(), \mathit{clock} + 30))$$

noJunkConsumed

This property can conveniently be expressed with past time logic as follows.

$$\underline{\text{mon}}\ \texttt{noJunkConsumed} = \texttt{A}(\texttt{C}() \rightarrow \texttt{Pi}(\texttt{B12}(), \mathit{ident}))$$

The formula states that whenever a C() event occurs, then in the past a B1() or B2() event has occurred with the same message id as that of the C() event. Note that in order to check the property above, the monitor will have to keep track of all ids of messages produced by the two producers. EAGLE does this automatically.

noJunkAcknowledged

This property can be stated in a similar way as a past time formula.

$$\underline{\text{mon}}\ \texttt{noJunkAcknowledged} = \texttt{A}(\texttt{D}() \rightarrow \texttt{Pi}(\texttt{C}(), \mathit{ident}))$$

orderPreserved

This property can be formulated in a mixture of state machine notation and temporal logic. The state machine is slightly more complicated than the state machine for C_D_Alternation due to the need of two machine state variables (rule parameters) *id1* and *id2* for capturing message identifiers, leading to the extra rules R1′ and R2′ for assigning new values to these variables when transitioning from one state to another, as explained in Section 2.

$$\begin{array}{ll}
\underline{\text{max}}\ \texttt{R0}() & = (\texttt{B1}() \rightarrow \texttt{R1}'(\mathit{ident}, 0)) \wedge \bigcirc\texttt{R0}() \\
\\
\underline{\text{max}}\ \texttt{R1}'(\underline{\text{int}}\ \mathit{id1}, \underline{\text{int}}\ \mathit{id2}) & = \bigcirc\texttt{R1}(\mathit{id1}, \mathit{id2}) \\
\underline{\text{max}}\ \texttt{R1}(\underline{\text{int}}\ \mathit{id1}, \underline{\text{int}}\ \mathit{id2}) & = \texttt{C}(\mathit{id1}) \vee \\
 & \quad ((\texttt{B1}() \rightarrow \texttt{R2}'(\mathit{id1}, \mathit{ident})) \\
 & \quad \wedge (\neg\texttt{B1}() \rightarrow \bigcirc\texttt{R1}(\mathit{id1}, \mathit{id2}))) \\
\\
\underline{\text{max}}\ \texttt{R2}'(\underline{\text{int}}\ \mathit{id1}, \underline{\text{int}}\ \mathit{id2}) & = \bigcirc\texttt{R2}(\mathit{id1}, \mathit{id2}) \\
\underline{\text{max}}\ \texttt{R2}(\underline{\text{int}}\ \mathit{id1}, \underline{\text{int}}\ \mathit{id2}) & = \texttt{W}(\neg\texttt{C}(\mathit{id2}), \texttt{C}(\mathit{id1})) \\
\\
\underline{\text{mon}}\ \texttt{orderPreserved} = \texttt{R0}() &
\end{array}$$

The monitor starts in state R0. Upon recognition of a B1() event, state R1 is entered, remembering that value of the current message identifier in machine state variable *id1* (*id2* is initialized to 0). Independently of whether a B1() event occurs, a transition leads back to state `R0` such that this property is checked in every state. An alternative would have been to formulate state `R0` as having one success formula, being a temporal always-property (using the `A` combinator) of the form

$$\underline{\text{max}}\ \texttt{R0}() = \texttt{A}(\texttt{B1}() \rightarrow \texttt{R1}'(\mathit{ident}, 0))$$

In state `R1`(*id1*,*id2*), as long as no new message is produced by producer 1, a success criteria is that the previous message produced by producer 1, and identified by *id1*, gets consumed. Alternatively, in case of a new `B1`() event, machine state `R2`′(*id1*,*ident*) is entered, now remembering the new message identifier in *id2*. State `R2`(*id1*,*id2*) is formulated as a temporal property stating that there should not be a consumption of *id2* before a consumption of *id1*. This is expressed with the Weak Until combinator `W`, stating that either the consumer will never consume the message identified by *id2*, or (in case it does), the message with *id1* will be consumed first (no consumption of *id2* until *id1* is consumed). Observe how this property has been expressed in a mixture of state machine notation (identifying the main states and transitions) and temporal logic notation (expressing a success coondition). The last property expresses the priority aspect of the queue.

B1HasPriority

This property can be stated as follows in EAGLE.

$$\begin{array}{l}
\underline{\text{min}}\ \texttt{CForB1BeforeCForB2}(\underline{\text{int}}\ \mathit{id}) = \texttt{U}(\neg(\texttt{C}() \wedge \texttt{identFromB2}(\mathit{ident})), \texttt{C}(\mathit{id})) \\
\underline{\text{min}}\ \texttt{identFromB2}(\underline{\text{int}}\ \mathit{id}) = \texttt{P}(\texttt{B2}(\mathit{id})) \\
\underline{\text{mon}}\ \texttt{B1HasPriority} = \texttt{A}(\texttt{B1}() \rightarrow \texttt{CForB1BeforeCForB2}(\mathit{ident}))
\end{array}$$

The monitor reads as follows. Whenever a message is produced by producer 1, then that message must be consumed before any message produced by producer 2 is consumed. This latter property is exppressed by `CForB1BeforeCForB2`(*ident*). This rule states (using the `U` combinator) that eventually the message is consumed (`C`(*id*)), and until then there is no consumption of a message (`C`()) that has been produced by producer 2 in the past (`identFromB2`(*ident*)).

5 Conclusion

We have shown how EAGLE can be used to monitor behavioral properties in a component architecture setting. For presentation purposes the example is simple, but the extensibility mechanism of EAGLE allows construction of a library of rules formalizing domain concepts. With this very complex behavioral properties can be expressed.

References

1. *1st, 2nd, 3rd and 4th Workshops on Runtime Verification (RV'01 - RV'04), 2001–2004*, volume 55(2), 70(4), 89(2), 113 of *ENTCS*. Elsevier Science Direct.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In B. Steffen and G. Levi, editors, *Proceedings of Fifth International VMCAI conference (VMCAI'04)*, volume 2937 of *LNCS*. Springer, January 2004.