Eve: An Icon Monitor Coordinator *

Clinton L. Jeffery

March 22, 1996

Abstract

Eve is a program execution monitor coordinator. It coordinates one or more program execution monitors, providing them with requested events and various global services. Eve runs under MT Icon, an extension of Version 9.1 of the Icon programming language.

1 Introduction

Often a monitoring situation calls for not just one monitor but a set of monitors executing concurrently. In such a situation each program event must be *multicast* to the subset of the monitoring programs that are interested in events of the given type. Eve is an Icon program that performs this function for monitors running in the MT Icon [Jeff90] environment. Eve selects the minimal set of events from the monitored program required to satisfy the group of monitoring programs. Eve uses X-Icon for its user-interface facilities [Jeff91] and some of its features are tailored towards program monitors that are themselves X-Icon visualization tools.

This document provides information about Eve for the user and for the program monitor author. It also discusses the event demultiplixing problem and presents Eve's central algorithms.

*This work was supported in part by the National Science Foundation under Grant CCR-8713690 and a grant from the AT&T Research Foundation.

MT Icon and its execution monitoring interface make it easy to develop new EMs. In this model, monitors are free to specialize in particular aspects of program execution, and the user selects the aspects to monitor in a given execution. When multiple EMs come into play, the selection of which EMs to use, the execution of those EMs, and their communication interface are the responsibility of a program called a monitor coordinator (MC).

This chapter presents monitor coordination as another domain within the scope of the exploratory program development features provided by the execution monitoring framework. After a general discussion of monitor coordinators, an example monitor coordinator is presented that implements a generalization of the *selective broadcast* communication paradigm advocated by Reiss [Reis90]. Other paradigms of monitor coordination are possible within the framework. In addition, other generalizations of selective broadcast proposed in the literature may prove complementary to the one presented in this chapter [Garl90].

2 Some monitoring configurations

MT Icon execution events are always reported to the parent program that loaded the TP being monitored. This means that the normal event reporting mechanism handles simple relationships such as monitoring a monitor or monitoring multiple TPs (Figure 1).

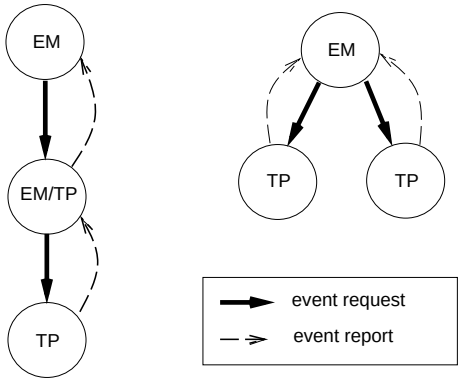

Figure 1: Monitoring a monitor; monitoring multiple TPs

On the other hand, the parental event report relationship means that if more than one EM is to monitor a TP, the TP's parent must provide other EMs with artificial copies of the TP events; MT Icon's `event()` function provides this service. Figure 2 depicts a parent EM that forwards TP events to an assisting EM.

Monitor coordinators are specialized EMs whose primary function is to forward events to other client EMs. A monitor coordinator is an event monitoring *kernel* that integrates and coordinates the operation of multiple stand-alone tools. By analogy to operating systems, the alternative to a kernel design would be a monolithic program execution monitor that integrates all operations into a single program.

Figure 3 depicts some relationships among MCs. Figure 3(a) is similar to Figure 2 and shows that a MC is just an execution monitor that forwards events. Figure 3(b) shows the main purpose for MCs, the execution of multiple EM's on a single TP. Figure 3(c) shows a MC monitoring a MC.

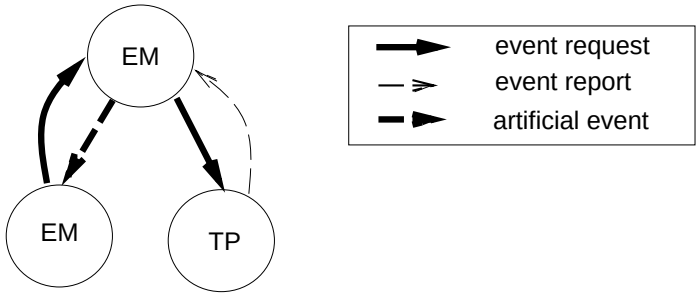

Figure 2: Forwarding events to an assistant

MC configurations and logic generally are limited to and revolve around parent-child relationships. For example, it is impossible to monitor events in a TP loaded and being monitored by another EM or MC unless that parent is configured to forward such events.

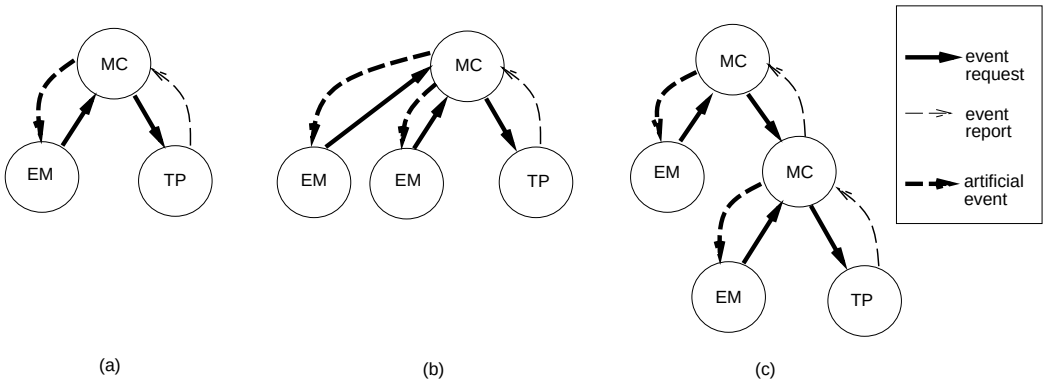

Figure 3: Monitor coordinators

Since event reports also transfer control, MCs also are schedulers for EMs, relinquishing the CPU to them by forwarding events to them. In the simplest case the MC forwards an event and waits for the EM to request another event before continuing; this scheduling is a form of cooperative multi-tasking. If the MC is the parent that loaded the EM in question, it can request event reports (such as clock ticks) from the EM in order to preempt its execution. Since MCs are special-purpose EMs, development of efficient MC designs falls within the scope of exploratory programming support provided by MT Icon.

3 Advantages and disadvantages of the MC approach

The three primary advantages of monitor coordinators are:

Modularity With a MC, monitors can be developed independently of one another and of the MC itself; they can run as stand-alone monitors, directly loading and executing the program to be monitored. This allows monitors to be debugged separately and puts "fire-walls" between monitors when they monitor the same program at the same time.

Specialization Support for multiple monitors allows EMs to be written to observe very specific program behavior and still be used in a more general setting. This in turn reduces the burden of generality placed on EM authors. Specialization also simplifies the task of presenting information, since each EM uses its own window and the user decides how much attention and screen space to devote to each EM.

Extensibility Extensibility refers to the ease with which new tools are added to the visualization environment. Adding a new tool to run under a MC does not require recompiling or even relinking the MC or any of the other visualization tools.

Monitor coordinators do have disadvantages. The implementation of MCs poses serious performance problems that require careful consideration. Although unsuitable for exploratory monitor development and experimental work, a single monolithic EM provides better performance than a MC that loads multiple EMs.

The primary problem with MCs is the number of context switches among tasks; on some architectures, notably RISC architectures such as the Sun SPARC, switching between coroutines is an expensive operation. Minimizing the number of switches required must be a goal of most MC designs.

4 Eve, an execution monitor coordinator

Eve is an example of a MC that allows the user to execute one or more EMs selected from a list and forwards TP events to those EMs that the user selects. The communication provided by Eve represents a generalization of the selective broadcast communications paradigm, because EMs may change the set of events at any time during execution; in Reiss's FIELD system, tools can specify the set of events they are interested in only when they are started. Unlike Forest's generalization of selective broadcast in which dynamic control is achieved by placing a greater computational load on the coordinating message server, Eve maintains an extremely simple message dispatch mechanism and passes policy changes on to the TP by recomputing the TP's event mask whenever needed. By suppressing events as early as possible, the higher performance required for execution monitoring is attained. This technique of continually minimizing the set of events reported by the TP could be used in conjunction with a Forest-style policy mechanism in the monitor coordinator if that were desired.

Eve is a cooperative multi-tasking scheduler. Figure 4 shows an image of Eve's control window. On the left-hand side are buttons that pause and terminate TP execution and a slider that controls execution speed. The main area of the window consists of a configurable list of EMs, and for each EM a set of buttons allow the tool to be controlled during TP execution. In the figure, two EMs are loaded and enabled.

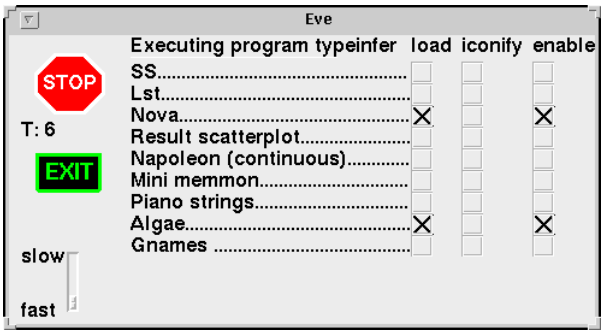

Figure 4: Eve's control window

5 Writing EMs to run under Eve

Eve supplies events to client EMs using the standard `EvGet()` interface [Gris90]. This section describes a few differences between the stand-alone interface and the Eve environment. Note that programs written for the Eve environment run without change or recompilation as stand-alone tools.

Client environment

After each EM is loaded, Eve initializes it with references to its event source (the Eve program itself) and the TP. The former is necessary so that EMs yield control to Eve to obtain each event. The latter is provided so that the state of the TP may be examined or modified directly by all EMs. These references in the form of co-expression values are assigned to the keyword `&eventsource` and the global variable `Monitored`, respectively; the global variable `Monitored` is declared in each EM when it links to the `evinit` event monitoring library.

Since under Eve `&eventsource` is *not* the TP, EMs should always use `Monitored` to inspect program state. For example, to inspect the name of the current source file in the executing program an EM should call `keyword("file", Monitored)` rather than `keyword("file", &eventsource)`.

Aside from the fact that `&eventsource` is not `Monitored` under Eve, from a programmer's standpoint, Eve's operation is implicit. Just as monitoring does not inherently affect TP behavior (other than slowing execution), within the various EMs Eve's presence normally is not visible; the EM can call `EvGet()` as usual.

General-purpose artificial events

Eve sends certain artificial events when directed by the user (in the Eve control window). These include the disable and enable events discussed above, `E_Disable` and `E_Enable`. A tool can pass a second parameter to `EvGet()` in order to receive these pseudo-events, for example `EvGet(mask, 1)`. When an `E_Disable` event is received, a tool is requested to disable itself. Tools that do not maintain any state between events can simply shut off their event stream by calling `EvGet('', 1)`:

```
case &eventcode of {
```

```
   # ...  more frequent events come first
   E_Disable:  while EvGet('', 1) ~=== E_Enable
   }
```

Tools that require events in order to maintain internal consistency might at least skip their window output operations while they are disabled. An `E_Enable` event informs the tool that it should resume operation, updating its display first if necessary.

Monitor communication example

In addition to the use of artificial events for communication between Eve and other EMs, artificial events can be used by EMs to communicate with each other, using Eve as an intermediary. For example, a line-number monitor is more useful if the user can inquire about a section of interest in the line-number graph and see the corresponding source text. This functionality can be built into the line-number monitor, but since many visualization tools can make use of such a service, it makes more sense to construct an EM to display source lines, and use virtual events to communicate requests for source code display from other EMs.

Communication using Eve starts with the definition of an artificial event code for use by the communicating EMs. Some of these codes such as `E_Disable` are defined in the standard library, but in general EMs can use any artificial event codes that they agree upon. In this case, an event code, `E_ALoc`, is defined for artificial location display events. Communicating EMs also agree on the type and meaning of the associated event value. In this case the associated event value is an integer encoding of a source line and column number, similar to that produced by `E_Loc` events.

The source-code display EM is similar to other EMs, except that it is not interested in TP events, but only in `E_ALoc` events. Its main loop is

```
while EvGet('', 1) do
   if &eventcode === E_ALoc then {
      # process requests for source code display
      }
```

Any EM that wishes to request source location display services sends an `E_ALoc` event to Eve. Eve then broadcasts this event to those tools that requested artificial event reports. The code to send a location request event to Eve from within an EM is

```
loc := location(line, column)
event(E_ALoc, loc, &eventsource)
```

6 Eve in operation

This section describes the primary techniques employed in Eve to obtain good performance. The key ideas are to filter events at the source and to precompute the set of EMs to which each event code is distributed.

Different EMs require different kinds of events. After obtaining a list of client EMs to execute, Eve loads each client. It then activates each EM for the first time; when the EM completes its initialization, it calls `EvGet()`, passing Eve an event mask.

6.1 Computation of the minimal event set

Each time an EM requests its next event report from Eve, it transmits a cset event mask indicating what events it is interested in. Eve could simply request all events from the TP, and forward event reports to each EM based on its current mask. The interpreter run-time system is instrumented with so many events that this brute-force approach is too slow in practice. In order to minimize the cost of monitoring, Eve asks the TP for the least set of events required to satisfy the EMs.

From the event masks of all EMs, Eve computes the union and uses this cset to specify events from the TP. The code for this union calculation is

```
unioncset := ''
every monitor := !clients do
   if monitor.enabled === E_Enable then
      unioncset ++:= monitor.mask
```

Although every EM can potentially change its event mask every time it requests an event, constant recomputation of the union mask would be unacceptably expensive. Fortunately, most tools call `EvGet()` with the same event mask cset over and over again. Eve does not recompute the union event mask unless an EM's event mask changes from the EM's preceding event request.

6.2 The event code table

The minimal event set described above greatly reduces the number of events actually reported from the TP. When an event report is received from the TP, Eve dispatches the report to those EMs that requested events of that type. The larger the number of EMs running, and the more specialized the EMs are, the smaller the percentage of EMs that typically are interested in any given event.

Eve could simply test the event code with each EM's cset mask with a call `any(mask, &eventcode)`. This test is fast, but performing the test for each EM is inefficient when the number of EMs is large and the percentage of EMs interested in most events is small. Instead, the list of EMs interested in each event code is precomputed as the union mask is constructed. These lists are stored in a table indexed by the event code. Then, after each event is received, a single table lookup suffices to supply the list of interested EMs. For each enabled monitor, the code for union mask computation is augmented with:

```
every c := !monitor.mask do {
   /EventCodeTable[c] := []
   put(EventCodeTable[c], monitor)
   }
```

6.3 Event handling

Eve requests three types of events whether or not any client EM has requested them: `E_Tick`, `E_MXevent`, and `E_Error`. Eve uses these events to provide basic services while execution is taking place; since these events occur relatively infrequently they do not impose a great deal of overhead.

`E_Tick` events allow Eve to maintain a simple execution clock on the control panel. `E_MXevent` events allow Eve to receive user input (such as a change in the slider that controls the rate of execution) in its control panel. `E_Error` events allow Eve to handle run-time errors in the TP and notify the user when they occur, allowing errors to be converted to expression failure at the user's discretion.

6.4 Eve's main loop

Eve's main loop activates the TP to obtain an event report, and then dispatches the report to each EM whose mask includes the event code. Since this loop is central to the performance of the overall system, it is coded carefully. Event dispatching to client EMs costs one table lookup plus a number of operations performed for each EM that is interested in the event – EMs for whom an event is of no interest do not add processing time for that event. The code for Eve's main loop is:

```
while EvGet(unioncset) do {
   #
   # Call Eve's own handler for this event, if there is one.
   #
   (\ EveHandlers[&eventcode]) ()
   #
   # Forward the event to those EM's that want it.
   #
   every monitor := !EventCodeTable[&eventcode] do
      if C := event( , , monitor.prog) then {
         if C ~=== monitor.mask then {
            while type(C) ~== "cset" do {
               #
               # The EM has raised a signal; pass it on, then
               # return to the client to get his next event request.
               #
               broadcast(C, monitor)
               if not (C := event( , , monitor.prog)) then {
                  unschedule(monitor)
                  break next
                  }
               }
            if monitor.mask ~===:= C then
               computeUnionMask()
            }
         }
      else
         unschedule(monitor)
      # if the slider is not zero, insert delay time
```

```
}
```

7 Interactive error conversion

Normally execution terminates when a run-time error occurs. Icon supports a feature called *error conversion* that allows errors to be converted into expression failure. Error conversion can be turned on and off by the source program by assigning an integer to the keyword `&error`. `&error` indicates the number of errors to convert to failure before terminating the program; on each error the value of `&error` is decremented and if it reaches zero the program terminates. A program can effectively specify that all errors should be converted by setting `&error` to a small negative integer. The mechanism is limited in that it does not allow the user or the program to inspect the situation and determine whether error conversion is appropriate: error conversion is either on or it is off.

Eve catches run-time errors in the TP and allows the user to decide whether to terminate execution, or convert the error into expression failure and continue execution (Figure 5).

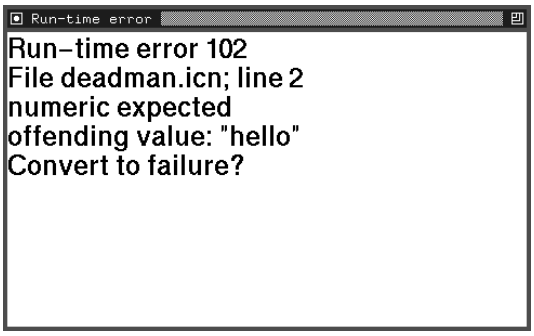

Figure 5: Eve's interactive error converter

An `E_Error` event occurs upon a run-time error. A monitor that requests `E_Error` events is given control before the error is resolved. Eve requests these events, presents the user with the error, and asks for an appropriate action. The code in Eve that does interactive error conversion is:

```
procedure eveError()
   win := open("Run-time error " || &eventvalue, "g")
   write(win, "Run-time error ", &eventvalue)
   write(win, "File ", keyword("file", Monitored), "; line ", keyword("line", Mo
   write(win, keyword("errortext", Monitored))
   writes(win, "Convert to failure?  ")
   if read(win)=="y" then
      keyword("error", Monitored) := 1
   close(win)
end
```

8 Summary

Eve is a simple monitor coordinator that enables multiple monitors to operate on a single subject program. In event monitoring performance considerations are serious, and even more so when running multiple monitors. Nevertheless, Eve is written in the same high-level interpreted language as the monitors and the subject program, and runs acceptably on contemporary hardware. This demonstrates the efficiency of MT Icon.

References

[Garl90] Garlan, D. and Ilias, E. Low-cost, Adaptable Tool Integration Policies for Integrated Environments. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 1–10, December 1990.

[Gris90] Griswold, R. E. Processing Icon Event Streams. Technical Report IPD152, Department of Computer Science, University of Arizona, December 1990.

[Jeff90] Jeffery, C. L. The MT Icon Interpreter. Technical Report IPD169c, Department of Computer Science, University of Arizona, July 1990.

[Jeff91] Jeffery, C. L. X-Icon: An Experimental Icon Windows Interface. Technical Report 91-1, Department of Computer Science, University of Arizona, January 1991.

[Reis90] Reiss, S. P. Connecting Tools Using Message Passing in the FIELD Environment. *IEEE Software*, pages 57–66, July 1990.