SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(11), 1025–1049 (NOVEMBER 1994)

A Framework for Execution Monitoring in Icon

CLINTON L. JEFFERY
Division of MCSS, The University of Texas, San Antonio, TX 78249, U.S.A.

AND

RALPH E. GRISWOLD
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

Execution monitors are widely used during software development for tasks that require an understanding of program behavior, such as debugging and profiling. The Icon programming language has been enhanced with a framework that supports execution monitoring. Under the enhanced translator and interpreter, neither source modification nor any special compiler command-line option is required in order to monitor an Icon program. Execution monitors are written in the source language, instead of the implementation language. Performance, portability, and detailed access to the monitored program's state are achieved using a coroutine model and dynamic loading rather than the separate-process model employed by many conventional monitoring systems.

KEY WORDS: Execution monitors Programming languages Interpreters Coroutines Program visualization Program behavior

BACKGROUND

Dynamic analysis, the study of program execution behavior, is a major component of most software development efforts. Tools that perform dynamic analysis, *program execution monitors*, are widely used in debugging, performance tuning, and related program understanding tasks[1].

While software development technology has advanced during the past decade, the tools available for dynamic analysis have remained almost unchanged during this period. The effort required to write an execution monitor is typically very large when done in conventional ways; it requires expert knowledge of the language being monitored, its implementation, and its implementation language. The framework presented in this paper is a means of significantly reducing the effort required to write a broad class of execution monitors.

Three aspects of existing execution monitoring systems balance the difficulty of implementation against the usefulness of the resulting system: information sources and access methods by which monitors observe program behavior, execution models that describe the relationship between the monitor and the program being monitored,

CCC 0038–0644/94/111025–25
© 1994 by John Wiley & Sons, Ltd.
Received 16 November 1993
Revised 20 June 1994

and interaction features that determine how information is presented to the user as well as how the user controls and directs monitor activity.

The most common methods used to obtain information about program behavior are *manual instrumentation*,[2–5] *run-time instrumentation*,[6–13] *interpreter instrumentation*,[14–18] and *instrumenting compilers*.[19–21] Some systems provide additional access to program variables and other execution information.

The most common monitoring technique, despite its limitations, is to manually insert instrumentation code into the program to be monitored. Run-time instrumentation refers to the modification of the monitored program code immediately prior to or during execution. Interpreter instrumentation is the insertion of monitoring code in the language interpreter itself; such instrumentation can provide information about the behavior of any program executed by the interpreter. Instrumenting compilers include preprocessors and code generators that add instrumentation to the code they generate. Such code usually is much larger than the non-instrumented code.

Of the possible relationships between the monitor and the program being monitored, three are commonly used: the *one-process model*,[2,3] the *two-process model*,[6,18] and the *thread model*.[9] In the one-process model, a monitor is a library of procedures linked to the program being monitored or integrated into the run-time system. The one-process model has good performance and access characteristics, but it does not prevent the target program and monitor code from affecting each other in critical ways. In addition, the control flow logic within the monitor is somewhat inverted, since the monitor is activated through callbacks. In the two-process model, the monitor is a separate process from the program being monitored, reducing the problem of intrusion at the expense of complicating monitor access and reducing performance. In the thread model, the monitor is a separate thread in a shared address space occupied by the program and possibly other monitors, providing a reasonable compromise between the characteristics of the one-process and two-process models for many monitoring applications.

Interaction facilities vary both in terms of the kind of execution controls provided to the user, and the techniques used in presenting the user with execution information. Execution controls range from controls that can only start and stop execution to entire languages that can be used to query for execution information or modify program variables. The study of data display techniques used in execution monitors has developed into its own sub-field: *program visualization* refers to the use of graphics to depict execution monitoring information. Examples of program visualization tools include the MemMon system for dynamic storage visualization[22] and the Incense data structure visualization tool.[23] The best-known use of program visualization is in algorithm animation. Some well-known examples are Ronald Baecker's motion picture, 'Sorting Out Sorting',[24] Marc Brown's BALSA[2] and ZEUS,[25] and John Stasko's Tango.[5]

This paper describes an execution monitoring framework for the Icon programming language.[26] Icon is particularly interesting because it is a high-level language with an extensive repertoire of operations on strings and structures, a novel expression-evaluation mechanism and late binding. These language characteristics create a need for exploratory approaches to execution monitoring, a need shared by other non-traditional languages.

This framework simplifies development of monitors in several ways, while avoiding

common pitfalls associated with monitoring. Monitors developed in this system tend to be very short compared with those in other languages, because they are developed in the source language rather than the implementation language, because they have full access to target program's variables, and because monitors can specialize on particular program behaviors of interest. Shorter monitors are in turn easier to understand, to write correctly, and to enhance.

THE ICON PROGRAMMING LANGUAGE

Icon is a high-level, procedural programming language with extensive facilities for processing strings and structures. This section provides an overview of the features of Icon that are relevant to program monitoring.

Expression evaluation

Icon has a sophisticated, goal-directed expression-evaluation mechanism. The evaluation of an expression can produce a result (*succeed*) or produce no result (*fail*). Failure occurs when an expression cannot perform a computation. Success and failure determine whether other computations are performed. For example,

```
line := read()
```

assigns the next line of input to line provided there is one, but fails on an end-of-file. If it fails, the assignment is not performed and the value of line is not changed. Success and failure also control loops, as in

```
while text := read() do
  write(text)
```

which copies the input file to output. The loop terminates when read() fails.

Some expressions can produce more than one result. Such expressions are called generators. The results of a generator are produced in sequence if the context in which the generator is evaluated requires alternative results. For example, the function find(s1,s2) generates the positions at which s1 occurs as a substring in s2.

The iteration control structure causes a generator to produce all of its results in sequence. For example,

```
every i := find(s1,s2) do
  write(i)
```

writes all the positions at which s1 occurs as a substring in s2.

Alternatives are also produced by a generator if they are needed to produce the success of an enclosing expression. For example, in

```
if find(prefix,text)>limit then
  write(&errout,"***prefix out of bounds")
else
  process(text)
```

the control clause causes find(prefix,text) to produce successive results until one is greater than limit, in which case the expression in the then clause is evaluated, or until find(prefix,text) has no more alternatives, in which case the expression in the else clause is evaluated. &errout is a keyword that causes output to go to standard error output. Keywords, indicated by an initial ampersand, are used to denote values with special status.

Failure occurs when a computation is meaningful but cannot be carried out, such as reading when there is no more data. Computations that are not meaningful, such as

```
i := "a" + 1
```

produce run-time errors. Run-time errors normally cause program termination with an error message. However, if the keyword &error is non-zero, run-time errors are converted to expression failure and program execution continues.

Procedures

Procedures can be written to supplement the built-in computational repertoire. Like built-in expressions, procedures can produce a result and succeed, produce no result and fail, or generate a sequence of results. For example, the following procedure generates all the substrings of s1 in s2 that begin in odd-numbered positions.

```
procedure oddfind(s1,s2)
  every i := find(s1,s2)
    if i % 2 = 1 then suspend i
  fail
end
```

The expression suspend i produces the value of i. If an alternative result is needed in the context in which oddfind(s1,s2) is called, execution of the procedure is resumed to continue its computation. When there are no more alternatives for find(s1,s2), fail is executed to terminate the procedure call without producing another result.

Data

Icon supports many different types of data, including integers, real numbers, strings, and several kinds of structures.

Strings and structures are created during program execution and can be arbitrarily large. Storage management is automatic; space is allocated when data objects are created and garbage collection frees space that is no longer in use.

Structures

Structures include records, lists, and associative tables. Structures can be heterogeneous; that is, they can contain values of different types.

Lists are one-dimensional arrays. A list can be created by specifying its size and an initial value for all its elements, as in

```
grades := list(45,0)
```

which creates a list of 45 elements, all of which are zero initially. A list also can be created by specifying its elements explicitly, as in

```
dates := [1066, 1492, 1812, 1914]
```

Lists can be created in a number of other ways. For example, two lists can be concatenated to produce a longer list.

Lists can be accessed by position, as in

```
grades[13] := 95
```

which sets the thirteenth element of grades to 95. Lists also can be accessed as stacks and queues. For example,

```
put(dates, 1929)
```

appends the 1929 to the right end of dates, increasing its size by one. Similarly,

```
first := get(dates)
```

removes 1066 from the left end of dates and assigns it to first. If dates is empty (that is, it has no elements), get(dates) fails and the value of first is unchanged.

Tables provide associative lookup. They resemble lists, but they can be subscripted by values of any type. A table is created by

```
scores := table()
```

which assigns an empty table (with no elements) to scores. Subsequently, elements can be added to a table by assignment to a subscripted references, as in

```
scores["Norma Brown"] := 88
scores["Tom Brady"] := 67
```

Strings

Strings in Icon are first-class data values, not arrays of characters. Strings can be constructed during program execution in several ways, including concatenation, as in

```
heading := prefix || text || suffix
```

Strings can be arbitrarily long. The operation *s produces the length of s.

Strings can be subscripted by position and substrings can be produced by range specifications. Fo example,

```
write(text[3:10])
```

writes the substring of text between positions 3 and 10. Non-positive specifications are relative to the right end of the string. For example,

```
write(text[10:0])
```

writes the substring of text between the tenth and last position.

String scanning provides a high-level pattern matching facility in which a subject string provides the focus of attention for analysis operations. An example of string scanning is

```
line ? {
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
  }
```

The string line provides the subject of scanning. Scanning starts at the beginning of the subject. The function upto(&letters) produces the first location in the subject at which a letter occurs. tab(upto(&letters)) moves the position to this location. In the do clause, many(&letters) produces the position at the end of a sequence of letters. tab() moves to this position and produces the substring of the subject between the previous and new locations, thus matching a ‘word’, which is written.

String scanning expressions can be nested. The current subject and position are saved when a scanning expression is initiated and restored when it is complete.

Co-expressions

In Icon, a co-expression is the expression-level equivalent of a coroutine. A co-expression provides a data value that contains an expression and an environment for its evaluation. The expression can be activated to produce a result at any time and place in the program. If the expression is a generator, it produces a result every time it is activated, failing when there are no more results.

A co-expression is produced by create, as in

```
locs := create find(prefix, line)
```

which assigns a co-expression for find(prefix, line) to locs. The activation expression @locs then causes find(prefix, line) to produce its next result.

An Icon program consists of a co-expession for main(args), the call of its main procedure. This main co-expression, &main, is activated to initiate program execution. During program execution, many co-expressions can be created and activate each other in arbitrary ways.

Co-expressions play a central role in the monitoring framework, as described in subsequent sections.

Graphics

Icon supports a wide range of graphic output operations, including drawing text and geometrical objects in windows.[27] The graphics facilities are geared toward

general-purpose use and ease of programming. For example, window exposure events are handled automatically, so that programs that use graphics can be written in a straightforward manner without having to implement a window redraw operation or to write the program's control structure so that it revolves around window system interactions.

Although graphics are not necessary for program execution monitoring, there are many situations in which the visual presentation of monitoring information is very advantageous. This is illustrated by examples shown later in this paper.

THE FRAMEWORK

The system requirements for Icon's monitoring framework are determined by a set of information sources and access methods, an execution model, and user interface facilities that, taken together, allow a broad class of monitors to be implemented easily and run with acceptable performance.

The first requirement is availability of a wide range of information about program behavior without requiring manual instrumentation of the program to be monitored. This requirement is met by instrumentation of the interpreter.

A second requirement is a tight binding between the execution monitor (EM) and the target program (TP) being monitored. A tight binding ensures that the monitor has access to target program data as well as the interactive control over the progress of execution. Tight binding specifically precludes the two-process execution model. Instead a synchronous thread model is used.

A third requirement is that the framework be useful and usable in practice. This implies both ease of use and acceptable performance on real-world programs. Ease of use is achieved by separating EM and TP code and using dynamic loading to establish the binding and control relationship between them at run-time. Acceptable performance is gained by using the thread-model's light-weight context switch, and by providing a dynamic filtering mechanism to reduce the number of context switches required during monitoring.

Coroutine-based monitoring

The framework uses a multi-tasking model in which the TP and EM are coroutines as shown in Figure 1. The EM and TP execute independently, but not concurrently.

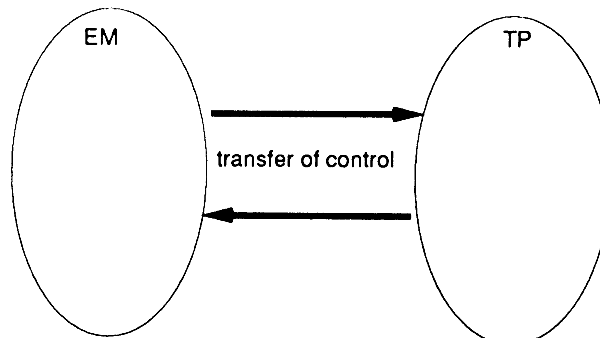

Figure 1. EM and TP are separately loaded coroutines

Consequently, the TP blocks while the EM is running. This is essential for EMs that allow the user to query or control the TP when important events occur. Blocking generally is not a problem in practice even for EMs that do not support user interaction with the TP. The limiting factor is often the speed with which a human user can comprehend EM output. Concurrency could be added to the framework, but at a considerable cost in complexity and difficulty of writing EMs.

Execution takes place within a shared address space, providing the EM with direct access to TP state. The EM accesses data values in the TP the same way it accesses its own data: TP structures and strings are referenced using regular Icon operations. For example, the EM can compare TP strings with its own.

The programming abstraction that supports multi-tasking in the monitoring framework is the *task*; a task is the execution state of an Icon program, including its locus of control as well as its global, stack, and heap data areas. A task in this framework consists of not just a set of machine register values plus a stack (sometimes referred to as a *thread*; a co-expression in Icon is a non-concurrent thread) but a replication of the entire data space employed by the language implementation, with the exception of constant values that may be shared by all tasks. Tasks do not reside inside each other, and names used in one task have no connection with variables in another—references within one task to another task's variable names are always by explicit means. A task called the *root* is created when the interpreter starts execution. Additional tasks can be created dynamically as needed.

Because of the shared address space, the task-switching operation needed to transfer execution between an EM and a TP is fast. This is important because monitoring requires an extremely large number of task switches compared to typical multi-tasking applications. Consider the number of events (potential task switches) in the TP for even the smallest of computations, such as the expression

```
x := a + b[i]
```

The above expression produces of the order of 26 events (possibly more, depending on the types of the values) describing the details of the subscripting, addition, and assignment operations and the underlying language mechanisms used during execution. While a typical EM uses only a small fraction of these events, most of the useful EMs that have been developed thus far perform task switches with a frequency ranging from several times per line of TP source code down to once for every few lines of TP source code. Reference 28 provides task switch counts for several EMs and the relative frequencies of different kinds of events.

In programming terms, a task consists of a main co-expression and zero or more child co-expressions that share a program state. At the source-language level, tasks are loaded, referenced, and activated solely in terms of one of their member co-expressions; the task itself is implicit. The EM and TP are separate tasks, and transfer of control from the TP to the EM is driven by instrumentation in the language run-time system. For this reason, the TP need not be modified in order to be monitored by an EM; neither does an EM need to be modified or recompiled in order to be used on different TPs.

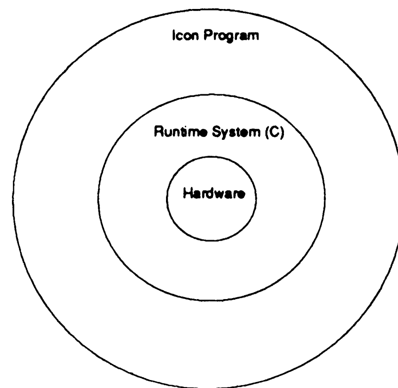

Figure 2. Layers in the Icon implementation

Event instrumentation

The purpose of an EM is to collect and present data from a TP's execution. The nature of the data collection facilities available in a monitoring system also defines and limits the kinds of monitors that can be implemented. Figure 2 depicts the system layers present in running an Icon program under the Icon interpreter. The TP code is executed by an interpreter writtin in C, which in turn calls C language run-time support routines to perform various language operations.[29]

Extensive information about TP execution is available to the EM from locations in the interpreter's run-time system that report significant *events*. At these locations, control can be transferred and information reported to the EM. When execution proceeds through one of these points in the run-time system, an event occurs. Each time an EM resumes execution of the TP, it explicitly specifies what kinds of events are to be reported; other kinds of events are not reported (see Figure 3). *Event masks* provide the mechanism for determining which kinds of events are reported.

Several major classes of events have been instrumented in the Icon interpreter. Most of these events correspond to explicit operations within the source code; but some designate actions, such as garbage collection, performed implicitly by the run-time system of which the programmer may be unaware.

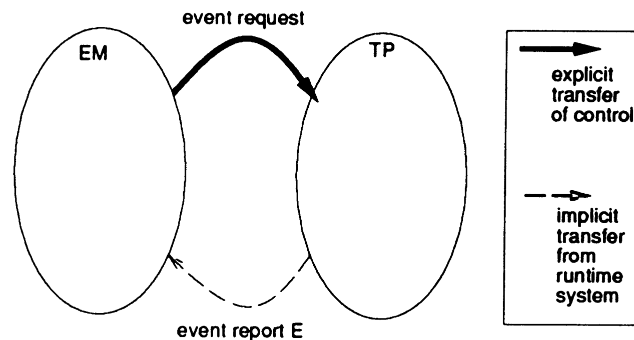

Figure 3. Event-driven control of TP

An event provides the EM with two pieces of TP execution information: an *event code* and an *event value*. The code describes what type of event has taken place. For example, there are different codes associated with the call of a procedure and the creation of a list.

The event value is an Icon value associated with the event. The nature of an event value depends on the corresponding event code. For example, the event value for a procedure call event is an Icon value designating the procedure being called, while the event value for a list creation event is the list that was created. If the event value is an Icon list, table or other structure, the EM accesses it just like any other source-language value. In addition to the information associated with the event itself, when the EM gains control it can interrogate the TP's variables by means of explicit inter-task access functions, discussed below.

There are over 100 different event codes, covering virtually all kinds of events of possible interest in monitoring. Some event codes in turn encompass a wide variety of related events.

Explicit source-related execution events include:

1. Program location changes in terms of line and column numbers.
2. Procedure, function, and operator activity—calls, returns, failures, suspensions, and resumptions.
3. String-scanning activity—scanning environment creation, entry, change in position, and exit.
4. Structure creation and access.

Implicit run-time system events include:

1. Memory allocations including size and type information.
2. Garbage collections including the state of memory after a collection completes.
3. Clock ticks for the passage of CPU time.

Monitors

An EM first sets up a source of events—a co-expression for the TP, subsequently available to the EM in &eventsource. The act of monitoring then consists of a loop that requests events from the TP and processes the information returned. Execution monitoring is initialized by the procedure EvInit(x), where x is a list that consists of the TP file name followed by arguments passed to it.

```
link evinit                        # monitor support library
procedure main(arguments)
  EvInit(arguments) | stop("can't initialize monitor")
  # ... initialization code
  # ... event processing loop
end
```

Events are requested by the EM using the function EvGet(eventmask), where eventmask specifies the kinds of events that are of interest to the EM. EvGet() activates the TP to obtain an event. The TP executes until an event report takes place; the resulting event code and event value are assigned to &eventcode and &

eventvalue in the EM, after which control is transferred to the EM until it requests its next event. EvGet() fails when execution terminates in TP. The typical processing loop in an EM is

```
while EvGet(eventmask) do
  case &eventcode of {
    # a case for each code in the event mask
    }
```

Inter-program data access

Much of an EM's access to TP data is performed by direct reference to the keyword &eventvalue, which is set by each call to EvGet(). In addition to the primary source of TP information provided by the event stream, EMs may interrogate the target program for values held in local and global variables and keywords. MT Icon provides built-in functions that produce the names of variables in other programs as well as built-in functions that take a co-expression and a variable name and allow assignment or dereferencing of the variable in the scope of the co-expression (typically the TP, when used during monitoring).

The function variable(s, C, i) looks up a variable named by string s in co-expression C. The optional parameter i specifies that the variable should be looked for i levels up in the procedure call chain. At present, there is no direct means to reference local variables in suspended procedures. This limitation is mitigated by the language semantics, which guarantee that such values are not modified across a suspension. A program that is interested in the local values in suspended procedures can maintain a model of the execution state; a library procedure is provided for EMs that need to maintain such information about suspended procedures.

The function keyword(s,C) is similar to variable(), but allows Icon's built-in keyword values to be consulted in other programs.

EXAMPLES

This section contains examples of simple execution monitors that illustrate the kinds of things that can be done within the framework. A larger list-visualization example is presented in an appendix.

Counting procedure calls

The following example is a complete procedure-call counter. The table calls, indexed by procedure, records the number of times each procedure is called. The mask E_Pcall specifies procedure call events.

```
link evinit
procedure main(Args)
  EvInit(Args)
  calls := table(0)
  while EvGet(E_Pcall) do          # collect procedure call events
    calls[&eventvalue] +:= 1
  write("Procedure   calls")       # write results
```

```
  every k := key(calls) do
    write(image(k)[11:0],":",calls[k])  # get procedure name from its image
                                        # and write it, followed by its count
end
```

The keys in the table are procedure values. The image of a procedure consists of the string "procedure" followed by the procedure name, so image(p)[11:0] fetches the name of the procedure for writing. Typical output from monitoring a text processing tool is:

```
Procedure calls
syms: 42
gener: 1600
generate: 4
alts: 4
defnon: 12
main: 1
define: 8
options: 1
```

Interactive error conversion

As mentioned earlier, an error condition that would potentially terminate program execution can be converted to failure to allow program execution to continue. Thus, an EM can trap run-time errors in a TP and allow the user to decide whether to allow the TP to terminate, or to convert the error into expression failure and continue execution of the TP. An E_Error event occurs upon a run-time error. An EM that requests E_Error events gets control before the error is resolved in the TP. Interactive error conversion can be done by including E_Error in the event mask and then processing E_Error events as follows:

```
case E_Error: {
  write("Run-time error", &eventvalue)
  write("File ", keyword("file", &eventsource), ";line ", keyword("line", Monitored))
  write(keyword("errortext", &eventsource))
  write("offending value: ",image(keyword("errorvalue", &eventsource)))
  writes("Convert to failure? ")
  if read()=="y" then
    keyword("error", &eventsource) := 1
  }
```

Inter-program data access functions described earlier provide access to data in the TP via its co-expression value, &eventsource. If an error occurs, the user of the EM is presented with information such as:

```
Run-time error 102
File deadman.icn; line 2
numeric expected
```

```
offending value:"hello"
Convert to failure?
```

If the user responds to the question “Convert to faiure?” with a y, the error in the TP is converted to failure and execution of the TP continues. Otherwise the TP terminates with the run-time error when control is returned to it.

Checking index bounds

List subscripts that are out of range result in failure. Although this feature simplifies loop control, if failure occurs unexpectedly, the result can be cryptic. A user might wish to be informed when and where such failures take place. This can be done with an EM that gets list-subscripting events and then checks the subscript value, which is given in a subsequent event:

```
while EvGet(E_Lref) do {        # monitor references to lists
   size :=*&eventvalue          # remember its size
   EvGet(E_Lsub)                # get the subscript
   if &eventvalue > size then   # check range
      write(&errout,“index out of range:”, &eventvalue,“>”,size)
   }
```

This monitor could be extended to allow the user to decide on continued execution, as in interactive error conversion. Since the event value for E_Lref is the list in the TP, such a monitor also could display the list for the user.

Monitoring string scanning

String scanning is one of the most interesting features of Icon, and it can be monitored in many ways. Even the simplest EM can give useful information. For example, the event value associated with the execution of a new scanning expression is the subject for that expression. Showing just the lengths of successive subjects can provide valuable information about what a program is doing. For example, monitoring a macro expander revealed that typical input contains a large percentage of lines that are too short to contain macro calls. Adding a test for this to the macro expander improved its performance considerably.

The following monitoring loop displays the lengths of subjects in a scrolling window as shown in Figure 4.

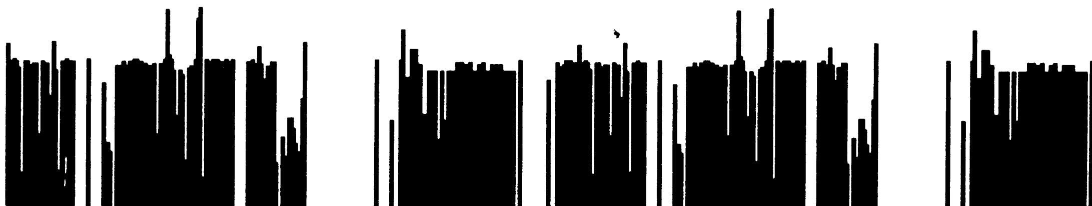

Figure 4. Monitoring string scanning

```
while EvGet(E_Snew) do {                # new subject
  XDrawLine(Width – 1, Height, Width – 1, Height – *&eventvalue)
  XCopyArea(1,0)                        # shift all lines right one pixel
  XEraseArea(Width – 1, 0, 1, Height)   # erase the new line
  }
```

Tabulating storage allocation

Tabulation is not limited to the kinds of information that can be gathered by placing counters at locations within the source code. It is just as easy to monitor behavior that is implicit, such as memory allocation. The following EM, almost identical in structure to the one presented earlier for tabulating procedure calls, counts the number of bytes allocated for each data type during program execution:

```
allocs := table(0)
while EvGet(AllocMask) do               # get event reports for allocations
  allocs[&eventcode] +:= &eventvalue
write("Allocations")                    # write results
every k := key(allocs) do
  write(evname(k)[1:–11],":",allocs[k])
```

The support procedure evname(e) produces a string that identifies the nature of the event e. For allocation events, the strings produced by evname() all end in "allocation". These last 11 characters are removed for the purposes of displaying the results. Typical output is:

```
Allocations
list: 78120
string: 188
record: 38760
table-element trapped variable: 448
cset: 40
list element: 155152
hash header: 120
substring trapped variable: 160
table: 192
```

Visualizing storage allocation

The output from the preceding monitor has two drawbacks: it requires execution of the TP to run to completion before it provides useful output, and it drops all details of individual allocations and their temporal relationships.

On the other hand, such detailed information, if presented textually, is too voluminous and difficult to interpret to be useful. A simple visualization technique can overcome these problems. Each allocation is drawn as a ray whose length corresponds to the amount of allocation and whose color encodes the kind of allocation. Successive rays are drawn at successive angles around the center point,

producing a 'radar sweep' that shows the recent history of allocation and makes it easy to see patterns of allocation. See Figure 5 for an example visualization.

```
degrees := 0
color := table()                # table of colors by allocation type
color[E_List] := "cyan"
color[E_Set] := "red"

      :
      :

while GetEvent(AllocMask) do {
  XFg("white")                             # set to erase previous ray
  XDrawLine(xorg, yorg, radius * cos(radians) + xorg,
    radius * sin(radians) + yorg)
  XFg(color[&eventcode])                   # set color for type
  XDrawLine(xorg, yorg, &eventvalue * cos(radians)+
    xorg, &eventvalue * sin(radians) + yorg)  # draw ray
  degrees +:= 1                            # advance to next ray
  radians := -dtor(degrees)
  }
```

EXPERIENCE

The framework described here was designed to make it easy to develop a broad class of EMs. Ones that perform counting, tabulation, and visualiztion tasks have proven very useful for identifying sources of inefficiency. The usefulness of visualizing individual allocations, for example, is illustrated by the visualization shown on the left-hand side of Figure 6 . Notice the large number of very large allocations. A glance at the program that produced this visualization showed that it concatenated lists frequently, resulting in many very large allocations. After a simple two-line change to replace the list concatenation by more economical incremental list construction, the visual appearance changed radically as shown at the right in Figure 6, and program execution speed doubled.

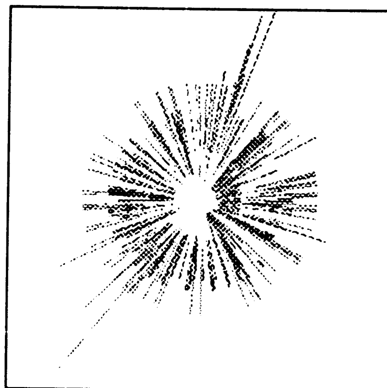

Figure 5. Visualizing individual allocations

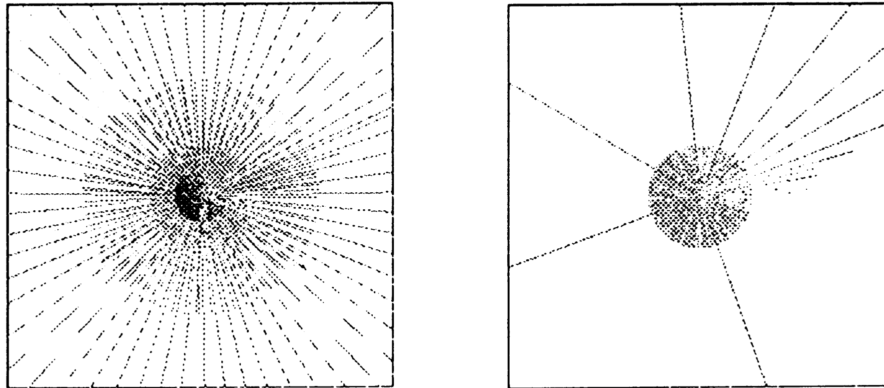

Figure 6. Allocation patterns before and after a two-line change

EMs have also proved useful simply for understanding what a program does. The 'oh, my' kind of experience is common when viewing storage allocation, generator activity, and string scanning.

The ease with which EMs can be written has been confirmed in a graduate workshop on program visualization in which students used the framework described here. Although several students had no prior experience with Icon, they were able to produce quite sophisticated projects including different ways of viewing string scanning, structure creation and access, and procedure activity. The difficult problems they encountered were not in writing EMs, but rather with application design and the presentation of visual information.

Another obvious use for EMs is in program debugging. Although this subject has not yet been explored systematically using the framework, it is clearly possible to write a sophisticated, full-scale debugger as an EM.

Experience with EMs has turned up a number of interesting, unanticipated results. For example, a bug in the implementation of Icon was identified quite by accident when an EM showed an 'impossible' situation in string scanning in which a position appeared beyond the end of the subject. Similarly, testing an EM that monitored list activity suggested an optimization that reduced the memory used in list concatenation.

IMPLEMENTATION

The framework described in this paper has been implemented for Icon's interpreter. The implementation of the framework has four major parts:

1. Providing the capacity to load multiple programs to run under a single invocation of the interpreter.
2. Providing a mechanism for transferring control between programs.
3. Instrumenting events to provide event reports.
4. Extending the source language to allow one program to access information in another.

The nature and difficulty of the implementation depends, of course, on the

language. The major aspects of the implementation for Icon are described in the following sections.

Program state

Loading and running multiple programs requires identifying what constitutes the state of a program and isolating it in the implementation. In the case of Icon, some components of the program state are obvious. Examples are global variables, keywords, and the stack used for expression evaluation. Other components of the state, such as specific C variables in the run-time system, were more difficult to identify.

One important decision concerned the two regions used for dynamic storage allocation in Icon programs. One alternative was to allow all programs to allocate space from the same pair of regions. Although this alternative was simple and easy to implement, it had the disadvantage that allocation by one program (such as an EM) could affect, albeit subtly, the behavior of another program (such as a TP). For example, an EM could cause a garbage collection that otherwise would not occur in a TP. Since the ability to monitor all aspects of storage allocation is important in Icon, the implementation was modified so that each program has its own pair of storage allocation regions.

In order to make changing the program state easy, it was isolated in a vector from which individual state components are accessed indirectly. The cost in terms of execution speed for this generalization is insignificant.

Transfer of control

As described earlier, the framework relies on the use of coroutines to transfer control between programs. Since Icon already had this facility at the source-language level in the form of co-expressions, it was easy to extend this mechanism to transfer control between programs within the run-time system, given the isolation of program state in a single vector.

It is worth noting that it is as easy to transfer control between two programs from within the run-time system as it is to transfer control between two co-expressions in a single program at the source-language level.

Instrumentation

Implementing the instrumentation that reports events first requires the identification of the events to be reported and then locating where in the run-time system these events occur. Most of the events of interest relate to source-language operations, but their manifestation in the run-time system is not always obvious. Consequently, providing the instrumentation for the run-time system required not only a good understanding of the source-language semantics but also of the implementation itself.

Fortunately, many types of events, such as those related to storage allocation, are easy to locate and straightforward to instrument. Others, such as those associated with procedure suspension and resumption, present more problems, both in location and in assuring that the instrumentation is correct.

Macros are used to simplify the implementation process and to avoid coding errors. For example, event reporting for a garbage collection is provided by a single line at the beginning of the function that performs garbage collections:

```
EVVal((word)region, E_Collect)
```

The value of region is a integer that identifies the region in which allocation triggered a garbage collection, and E_Collect is the event code for a garbage collection. EVVal() first checks if the event code is in the event mask for the program that is requesting events; this is a look-up within a bit vector. If the code is not in the mask, nothing is done, the running program continues with garbage collection, and the total cost is the event-mask check. If E_Collect is in the event mask, the values of &eventcode and &eventvalue are set in the requesting program, and control is transferred to it.

Interprogram state access

In Icon, access to information in one program from within another is provided by keywords and functions. How this is done is somewhat a matter of language design but more importantly depends on the information an EM may need about a TP.

Basically, the problem is providing one program access to the state of another. The functions variable() and keyword(), described earlier, are examples of how components of the state can be accessed in terms that are consonant with Icon language semantics. In the case of variable(), an existing function was generalized by adding an optional argument to specify the co-expression for the program of interest. The function keyword(), on the other hand, was an addition to Icon's function repertoire.

In some cases, a decision on interprogram access was influenced by what an EM could do using events as opposed to burdening the source language with additional features. For example, Icon's procedure mechanism that allows procedure suspension may result in a tree of invocations rather than the linear chain that results just from recursive calls. Although it is possible to access, for example, local variables in a chain of calls with the function variable(), this mechanism does not allow a general tree walk of suspended calls. Such a source-language mechanism would be complicated and difficult to use. On the other hand, an EM can build a tree of procedure calls and suspensions from procedure events—indeed, a library procedure that does this is available for use in EMs that need it. With such a structure, an EM can walk the tree with ordinary Icon code, accessing local identifiers or any other information of interest.

CONCLUSIONS

Execution monitors are difficult to implement using traditional approaches. The construction of an execution monitor often involves *ad hoc* modifications to the implementation of the language in which the monitoring is to be done; such modifications are typically specific to the monitor and of little use for other monitors. Many higher-level forms of execution monitoring also require changes to programs that are to be monitored, requiring acces to their source code and creating problems

when the programs are changed. Such monitoring techniques are poorly suited for large programs and situations where the precise behavior of interest is not known in advance.

Because execution monitors under this framework are written in the same language as the programs they monitor, it is easy for a monitor to interpret data in a program it monitors. Tight coupling allows a user to follow and even direct the execution of a monitored program. Execution monitors are sufficiently easy to write in this framework that it is possible to take exploratory approaches to sophisticated monitoring and presentation techniques, including program visualization.

Adding the framework described here to an existing implementation of a programming language is likely to be more difficult than including the framework in the design of a new implementation. Language features, such as those that allow an execution monitor to access the data of the program it monitors, are an important consideration. Ideally, execution monitoring should be part of language design, not just its implementation.

In any event, the implementation of the framework described in this paper requires a sustantial amount of effort. The work required depends, of course, on the language, the nature of its implementation, and the extent of instrumentation that is provided. Many thorny design issues were solved in the Icon implementation and need not be addressed again for other implementations, although, of course, any other language is bound to raise some new issues.

While the overall framework design applies to a wide range of programming languages, the implementation techniques developed for Icon apply most directly to comparable high level languages, such as LISP and SmallTalk. We are investigating implementation techniques necessary to adapt the framework for lower-level languages such as C. Such languages pose additional challenges in the areas of protecting monitors from misfunctioning TP behavior such as stray memory references, and identifying the appropriate granularity at which to report on program execution behavior.

The approach used by the framework described in this paper is applicable, in principle, to compilers as well as interpreters. Adapting the framework to a compiler poses additional problems, such as the necessity that instrumentation code that need occur in only one place in an interpreter must appear everywhere the corresponding event can occur in the code generated by the compiler. The run-time execution slowdown imposed by monitoring is similarly more substantial for compilers than interpreters. The resulting space and time ‘blow up’ factors are problematic issues for a compiler-based implementation of the framework, and are a subject of current research.

Although the implementation of the framework for any language is a substantial undertaking, once it is done, programmers can use it to easily develop a wide variety of monitoring tools. It is not so easy for such programmers to add new instrumentation. This involves at least a basic understanding of the implementation of the language. The instrumentation of some kinds of events can be done easily by imitating the existing implementation of similar events. Instrumenting other kinds of events may require a deeper understanding of the implementation and the framework. For this reason, the instrumentation provided by the framework should be extensive and include all the kinds of events that persons writing monitoring tools may need.

ACKNOWLEDGEMENTS

Gregg Townsend contributed to design discussions, wrote several visualization tools using the framework, and instrumented the Icon memory management system. Nick Kline authored other useful visualization tools during the framework's infancy, and he along with Ken Walker participated in design discussions.

This work was supported in part by the U.S. National Science Foundation under Grants CCR-8713690 and CCR-8901573 and a grant from the AT&T Research Foundation.

APPENDIX: A TINY LIST VISUALIZER

This appendix presents the Icon source code for Tinylist, an example execution monitor. Tinylist presents all the lists in a program as vertical segments with size proportional to the length of the list. Each element of each list is color-coded with the element's type. As lists are created, and then subsequently grow and shrink, their appearance changes. In addition, accesses to individual list elements are portrayed with a brief flash.

Tinylist conveys numerous pieces of information on a single display and is useful in pointing out unexpected or anomalous behavior. For example, if all the elements of a list are of a single type and a new element of a different type is inserted, it is immediately evident. Sequential access to all the elements of a list results in a smooth motion of a highlight along a line, and has a distinct contrast in appearance with random accesses on multiple lists.

An example of a display produced by this monitor is shown in Figure 7. A program that uses over 200 lists of varying sizes and types is depicted; accesses on individual list elements are still distinguishable.

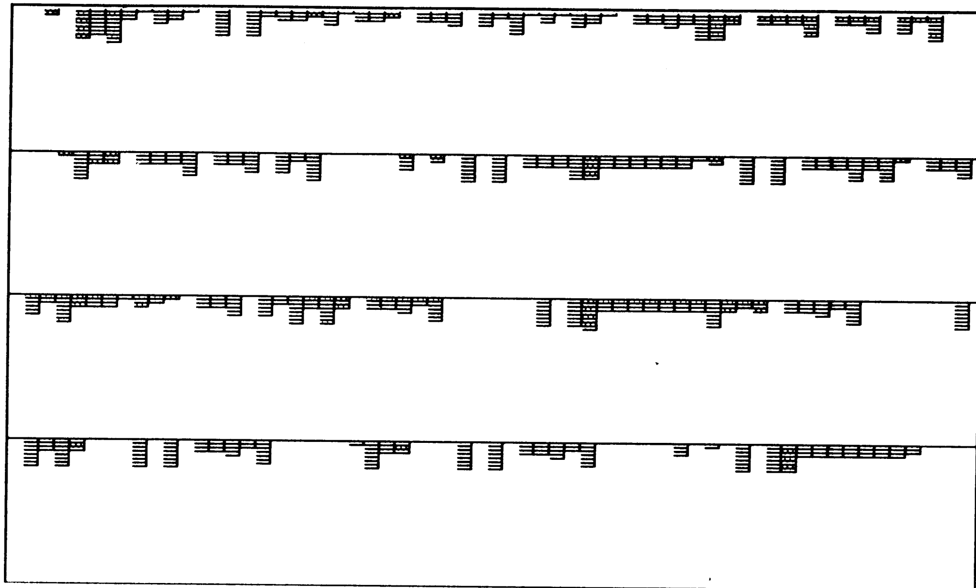

Figure 7. Tinylist shows all the lists in the program

```
#
# obtain event code definitions, and link in library routines
#
$include "evdefs.icn"                # definitions for events
link emsupport                       # support for execution monitors

#
# Global variables. The window is divided into a grid
# of boxes. Within each box, a list is displayed.
#
global
    width,                           # window width
    height,                          # window height
    rows,                            # number of boxes, vertical dimension
    cols,                            # number of boxes, horizontal dimension
    rowheight,                       # number vertical pixels/box for each list
    colwidth,                        # number horizontal pixels/box for each list
    elemheight,                      # number of pixels/elem with individual lists
    numindices,                      # number of screen indices available per list
    lastrow,                         # last row accessed
    lastcol,                         # last column accessed
    contexts                         # table of color bindings indexed by type code


#
# main procedure: initialization, followed by event processing loop
#
procedure main(av)
    local
        L,                           # focus list; the list being operated on
        i                            # index variable
    #
    # initialize execution monitoring system and libraries,
    # open a window for visualization
    #
    EvInit(av) | stop("can't initialize monitoring")
    WOpen() | stop("no window")      # create window for visualization
    contexts := itypebind()          # create contexts for drawing different types

    #
    # initialize variables for screen geometry
    #
    height := WAttrib("height")
    width := WAttrib("width")
    rows := 1
    colwidth := 8
    cols := width / colwidth
    rowheight := height / rows
    elemheight := colwidth
    numindices := rowheight / elemheight

    #
    # Main event processing loop; get the next event from TP.
    #
    while EvGet(ListMask) do {
        #
        # All events whose values are lists change the focus to the list
        # being operated on.
        #
```

```
L := &eventvalue
if type(&eventvalue) == "list" then {
    L := &eventvalue
    lastrow := serial(L) / cols
    lastcol := serial(L) % cols
    }
case &eventcode of {
    E_Lcreate: {                    # a new list has been created

        #
        # If a new list would go past the window border,
        # split the window into enough rows to show all lists
        #
        if lastrow >= rows then {
            rows +:= 1
            rowheight := height / rows
            elemheight := rowheight / numindices
            if elemheight < 1 then {
                elemheight := 1
                numindices := elemheight
                }
            redraw()
            }
        every i := 1 to *L do
            plot(objcolor(L[i]), i, 0)
        }
    #
    # List access events for queue, stack, and subscript operations
    #
    E_Lpop: if *L > 0 then {
        every i := 2 to *L do
            plot(objcolor(L[i]), i - 1)
        plot(contexts["bg"], *L)
        }
    E_Lpull: if *L > 0 then plot(contexts["bg"], *L)
    E_Lpush: every i := 1 to *L do plot(objcolor(L[i]), i)
    E_Lput: plot(objcolor(L[*L]), *L, 0)
    E_Lsub: {
        #
        # convert negative subscripts to positive and plot access
        #
        if &eventvalue <= 0 then
            &eventvalue +:= *L + 1
        plot(objcolor(L[&eventvalue]), &eventvalue)
        }
    #
    # E_Lbang, E_Lrand, and E_Lref are handled above by
    # changing the focus to the list being operated on.
    #
    }
if *Pending() > 0 then
    case Event() of {
        " ": Event()                # pause execution
        "q": break                  # quit
        }
```

```
        }
    EvTerm()
end

#
# plot an individual element access within the focus list
#
procedure plot(w, index, del)
    local x, y, wd, ht
    /del := 40
    #
    # plot indices past the viable range by either reducing the
    # number of vertical pixels per element, or plotting them at
    # the last legal position, if there is only one vertical
    # pixel per element
    #
    if index > numindices then {
        numindices := index
        if elemheight >:= rowheight / numindices then {
            if elemheight = 0 then {
                elemheight := 1
                numindices := index := rowheight
                }
            else
                redraw()
            }
        }
    wd := colwidth
    ht := elemheight
    x := lastcol * colwidth
    y := lastrow * rowheight + (index - 1) * elemheight + 1
    FillRectangle(x, y, wd, ht)
    WFlush()
    delay(del)
    if wd > 3 then wd -:= 1
    if ht > 1 then ht -:= 1
    FillRectangle(w, x, y, wd, ht)
end

#
# Return a foreground color coded by o's type
#
procedure objcolor(o)
    return contexts[type2code(type(o))]
end

#
# Redraw the entire display when required
#
procedure redraw()
    local x, y, wd, ht, r, c, i, L, vs
    EraseArea()
    every i := 1 to rows - 1 do
        DrawLine(0, i * rowheight, width, i * rowheight)
    every type(L := structure( Monitored )) == "list" do {
        r := serial(L) / cols
        c := serial(L) % cols
        vs := *L
```

```
        wd := colwidth
        if wd > 3 then wd -:= 1
        ht := elemheight
        if ht > 1 then ht -:= 1
        every i := 1 to vs do {
            FillRectangle(c * colwidth,
                r * rowheight + (i - 1) * elemheight + 1,
                colwidth, elemheight)
            FillRectangle(objcolor(L[i]), c * colwidth,
                r * rowheight + (i - 1) * elemheight + 1, wd, ht)
            }
        }
end
```

REFERENCES

1. Bernhard Plattner and Jurg Nievergelt, 'Monitoring program execution: a survey', *IEEE Computer*, November 1981, pp. 76–93.
2. Marc H. Brown and Robert Sedgewick, 'A system for algorithm animation', *Computer Graphics*, **18**, (3), 177–186 (1984).
3. R. A. London and R. A. Duisberg, 'Animating programs using Smalltalk', *IEEE Computer*, August 1985, pp. 61–71.
4. M. H. Brown, *Algorithm Animation*, ACM distinguished dissertation series, MIT Press 1988.
5. John T. Stasko, 'Tango: a framework and system for algorithm animation', *Computer*, September 1990, pp. 27–39.
6. Mark A. Linton, 'The evolution of Dbx', *Proceedings of the Summer 1990 USENIX Conference*, June 1990, pp. 211–220.
7. Ronald A. Olsson, Richard H. Crawford and W. Wilson Ho, 'Dalek: a GNU, improved programmable debugger', *USENIX Summer '90 Confernce*, June 1990, pp. 221–231.
8. Ronald A. Olsson, Richard H. Crawford and W. Wilson Ho, 'A dataflow approach to event-based debugging', *Software—Practice and Experience*, **21**, 209–229 (1991).
9. Ziya Aral and Ilya Gertner, 'Non-intrusive and interactive profiling in Parasight', *Proceedings of the ACM/SIGPLAN PPEALS 1988*, September 1988, pp. 21–30.
10. Ziya Aral and Ilya Gertner, 'High-level debugging in parasight' *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging (published in ACM SIGPLAN Notices),* **24**, 151–162 (1989).
11. Steven P. Reiss, 'Connecting tools using message passing in the FIELD environment', *IEEE Software*, July 1990, pp. 57–66.
12. Steven P. Reiss, 'Interacting with the FIELD environment', *Software—Practice and Experience*, **20**, 89–115 (1990).
13. David Garlan and Ehsan Ilias, 'Low-cost, adaptable tool integration policies for integrated environments', *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, December 1990, pp. 1–10.
14. Steven P. Reiss, 'Graphical program development with the PECAN development systems', in Peter Henderson (ed.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, May 1984, Volume 19, pp. 30–41.
15. Heinz-Dieter Bocker, Gerhard Fischer and Helga Nieper, 'The enhancement of understanding through visual representations', *CHI '86 Proceedings*, June 1986, pp. 44–50.
16. A. D. Dewar and J. G. Cleary, 'Graphical display of complex information within a Prolog debugger', *International Journal of Man–Machine Studies*, **25**, 503–521 (1986).
17. Siamak Masnavi, 'Automatic visualization of the dynamic behavior of programs by animation of the language interpreter', *Proceedings of the 1990 IEEE Workshop on Visual Languages*, 1990, pp. 16–21.
18. Roc Sosic, 'Dynascope: a tool for program directing', *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992, Volume 27, pp. 12–21.
19. David Socha, Mary L. Bailey and David Notkin, 'Voyeur: graphical views of parallel programs',

Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging (published in ACM SIGPLAN Notices), **24**, 206–215 (1989).

20. R. R. Henry, K. M. Whaley and B. Forstall 'The University of Washington illustrating compiler, *Proc. ACM SIGPLAN' 90*, White Plains, NY, June 1990, pp. 223–233.
21. Ricardo A. Baeza-Yates, 'Another look at program visualization (extended abstract)', *Technical Report*, Depto. de Ciencias de la Computation, Universidad de Chile, 1991.
22. Ralph E. Griswold and Gregg M. Townsend, 'The visualization of dynamic memory management in the Icon programming language', *Technical Report 89-30*, Department of Computer Science, University of Arizona, December 1989.
23. Brad A. Myers, 'Incense: a system for displaying data structures', *Computer Graphics*, **17**, 115–125 (1983).
24. Ronald Baecker and David Sherman, 'Sorting out sorting', 16mm color sound film shown at *SIGGRAPH '81, Dallas, TX*, 1981.
25. Marc H. Brown and John Hershberger, 'Color and sound in algorithm animation', *Technical Report 76a*, Digital Systems Research Center, August 1991.
26. Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, second edition. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
27. Clinton L. Jeffery, Gregg M. Townsend and Ralph E. Griswold, 'Graphics facilities for the Icon programming language', *Technical Report IPD 255*, Department of Computer Science, University of Arizona, 1994.
28. Clinton L. Jeffery, 'A framework for monitoring program execution', *Technical Report 93-21*, Department of Computer Science, University of Arizona, July 1993.
29. Ralph E. Griswold and Madge T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, Princeton, New Jersey, 1986.