Writing Icon Program Execution Monitors

Ralph E. Griswold

Department of Computer Science
The University of Arizona

Clinton L. Jeffery

Division of Mathematics, Computer Science, and Statistics
The University of Texas at San Antonio

1. Introduction

The execution of a program in a high-level programming language results in a many computational activities at a lower level. Some of these events mirror the semantics of the language, although superficially simple language operations may involve many low-level events. Other events may not be directly related to the semantics of the language. In Icon, for example, storage allocation accompanies the evaluation of some expressions, but the details of allocation depend on properties of the implementation, not the language, and may depend on the history of program execution as well [1]. Similarly, garbage collection usually occurs at unpredictable times, and what actually happens during garbage collection usually is not directly evident in the results of program execution. There also are lower-level events, such as the execution of instructions for the virtual machine that provides the framework for the interpretive implementation of Icon [2].

One of the purposes of a high-level programming language is to hide low-level events from the programmer. Nonetheless, in order to understand a program, to debug it, or to measure the resources it uses, it may be necessary to go beneath the surface. This report describes facilities that have been added to the MT Icon [3, 4] for such purposes.

Effective use of information about low-level events requires tools that can bridge the gap between the semantics of the programming language and the lower-level events that occur during program execution. MT Icon has been extensively instrumented to report, on request, those events that are most relevant to understanding the execution of an Icon program. This instrumentation is done in a way that does not affect program execution except to slow it down somewhat.

The instrumentation is designed so that an Icon program being monitored (the *source program*, or SP) reports events to another Icon program (the *execution monitor*, or EM). The interface through which events are reported is invisible to the SP except for delays that may occur may occur when the EM is running.

Events

The report of an event consists of two components: (1) a code that identifies the nature of the event, and (2) an associated value. Two typical events are the allocation of space for a newly created string and the subscripting of a list.

Event codes are one-character strings. Symbolic names, which are defined in `evdefs.icn`, are used for referring to event codes. For example, the code for string allocation is `E_String` and the code for a list reference is `E_Lref`.

Event values are Icon values. In the case of `E_String`, the event value is an integer corresponding to the number of bytes allocated. In the case of `E_Lref`, it is the list referenced in the SP. Note that such an event provides an EM with direct access to data in the SP.

Events fall into a few general categories:

- control flow events
- structure access events
- string scanning events

- assignment events
- type conversion events
- allocation events
- garbage collection events
- miscellaneous events

Appendix A contains a list of event codes by category.

The Monitoring Interface

The monitoring interface consists of functions, keywords, a library of support procedures in evinit.icn, and definitions in evdefs.icn.

The procedure EvInit(s) loads the icode file named s for monitoring. EvInit() also performs various initialization tasks. For example,

```
EvInit("concord")
```

loads the icode file concord, creates a thread for it [3], and prepares for monitoring. In addition, the value of &eventsource is set to the SP (the thread for concord, in this case).

If EvInit() is called with a list instead of a string, the first element of the list is taken to be the name of the icode file and the remainder of the list is passed to the icode file as the argument of its main procedure.

EvInit() has three optional additional arguments corresponding to the files for standard input, standard output, and standard error output for the SP. These arguments default to EM's &input, &output, and &errout. In this case the EM and the SP share these files.

EvInit() fails if the SP cannot be loaded. It is important to check for this possibility.

The function EvGet(c) returns the code for the next event, which is one of the characters in the cset *event mask* c. Events with codes not in c are ignored. If EvGet() is called without an argument, any event is returned.

EvGet() also also sets two keywords:

&eventcode	the code for the event (the same as the value returned by EvGet())
&eventvalue	the value for the event

These keywords are variables and values can be assigned to them to, for example, filter a stream of events.

EvGet() fails if there are no more events — that is, when the SP has terminated.

The function

```
event(code, value)
```

produces an event report from the program itself, as opposed to reports from the instrumentation in the interpreter. Such events are called *artificial events*. The value of code is not limited to a one-character string; it can be any value. Normally, only one-character strings event codes are returned by EvGet(). However, EvGet() has an optional second argument, which if nonnull allows EvGet() to accept event codes that are not one-character strings. For example,

```
EvGet('', 1)
```

requests only artificial events.

Masks

Masks serve to limit the events that are reported to those of interest to an EM. The event mask normally is given as the first argument of EvGet() as described above. The event mask also can be set by

```
eventmask(C, c)
```

which associates the event mask c with the thread C (for example, &eventsource). If the second argument is omitted, the function returns the event mask for C.

There also is a mask for selecting a specified set of virtual-machine instructions (‘‘opcodes’’) associated with E_Opcode. The function

```
opmask(C, c)
```

limits the virtual-machine instructions that are reported to those specified in c. If the second argument is omitted, the function returns the opcode mask for C.

Virtual-machine instructions are represented by small non-negative integers. For example, the virtual-machine instruction for removing a bounded expression (given symbolically in the implementation as Op_Unmark) is 78 (hexadecimal 4e). Virtual-machine instructions are given in the opcode mask as characters with corresponding numerical codes. Thus, an opcode mask to limit reporting of virtual-machine instructions to Op_Unmark could be given as \x4e.

The include file opdefs.icn contains definitions for all virtual-machine instructions. For example, as a result of including opdefs.icn, Op_Unmark has the value "\x4e".

An Example

The following EM tabulates procedure events and writes a summary when the SP terminates. The name of the SP is given as the first argument of the EM’s command line. The remainder of the command line is passed to the SP. ProcMask is a mask that includes only procedure events. See Appendix A for an explanation of procedure events.

```
link evinit

$include "evdefs.icn"

procedure main(args)

   EvInit(args) | stop("*** cannot load icode file ***")

   proact := table(0)

#  Tabulate procedure events.

   while EvGet(ProcMask) do
      proact[&eventcode] +:= 1

#  List the results

   write("procedure calls:        ", right(proact[E_Pcall], 6))
   write("procedure returns:      ", right(proact[E_Pret], 6))
   write("procedure suspensions: ", right(proact[E_Psusp], 6))
   write("procedure failures:     ", right(proact[E_Pfail], 6))
   write("procedure resumptions: ", right(proact[E_Presum], 6))
   write("procedure removals:     ", right(proact[E_Prem], 6))

end
```

For example,

```
proact rsg rsg.cfg <rsg.dat
```

causes proact to run rsg as if the command line

```
rsg rsg.cfg <rsg.dat
```

had been used.

Aother example EMs are given in Appendix B.

Programming Guidelines for Monitors

Both SPs and EMs must be compiled using MT Icon.

The ucode file evinit must be linked in the EM.

The include file evdefs.icn must be included in any EM that specifies event codes symbolically.

EvInit() must be called before an event report is requested.

Since a SP usually produces a very large number of events, efficiency is an important consideration in writing EMs. Events requested should be restricted to those of interest. of EvGet()).

Monitors that use visual displays should pay special attention to how graphics facilities are used [5].

A SP and an EM have separate program states and storage regions. Allocation of space in an EM does not affect storage management in the SP. On the other hand, an EM has access to data in the SP through event values. Care should be taken not to modify data in the SP unintentionally.

There are several support procedures for use in EMs. See [6].

Bugs

If the main procedure returns by an explicit return or suspension instead of failing (typically by flowing off the end of the procedure), Icon hangs in a hard loop. When possible, SPs should be examined for this possibility in situations where monitoring may allow them to run to completion.

Disclaimer

The instrumentation of MT Icon for event monitoring is still in process and is subject to change. Some event codes are not listed here because they are subject to change, not presently working, or correspond to events that are too obscure to be useful in monitoring.

Some of the instrumentation is relatively untested.

Acknowledgement

Gregg Townsend assisted in the development of the interface between SPs and EMs. Ken Walker provided help with the instrumentation.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1990.
2. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, 1986.
3. C. L. Jeffery, *The MT Icon Interpreter*, The Univ. of Arizona Icon Project Document IPD169, 1993.
4. C. L. Jeffery, *A Framework for Program Execution Monitoring in Icon*, Doctoral Dissertation, The University of Arizona, 1993.
5. C. L. Jeffery, G. M. Townsend and R. E. Griswold, *Graphics Facilities for the Icon Programming Language; Version 9.0*, The Univ. of Arizona Icon Project Document IPD255, 1994.
6. R. E. Griswold, *Support Procedures for Icon Program Monitors*, The Univ. of Arizona Icon Project Document IPD193, 1994.

Appendix A — Event Codes and Masks

Control Flow Events

name	*event*	*value*
E_Fcall	Function call	function
E_Ffail	Function failure	–1
E_Fresum	Function resumption	0
E_Fret	Function return	value produced
E_Fsusp	Function suspension	value produced
E_Frem	Function suspension removal	0
E_Ocall	Operator call	operation
E_Ofail	Operator failure	–1
E_Oresum	Operator resumption	0
E_Oret	Operator return	value produced
E_Osusp	Operator suspension	value produced
E_Orem	Operator suspension removal	0
E_Pcall	Procedure call	procedure
E_Pfail	Procedure failure	procedure
E_Prem	Suspended procedure removal	procedure
E_Presum	Procedure resumption	procedure
E_Pret	Procedure return	value produced
E_Psusp	Procedure suspension	value produced

Notes: FncMask, OperMask, and ProcMask contain the codes for function, operation, and procedure events, respectively. The event values for E_Fcall, E_Ocall, and E_Pcall all have type procedure. More specific information can be obtained using image() on the event value. Note that the event values for E_Ffail, E_Fresum, E_Frem, E_Ofail, E_Oresum, and E_Orem are not useful. Useful values are not provided because the necessary information is not available when these events occur.

Structure Access Events

name	*event*	*value*
E_Lbang	List generation	list
E_Lcreate	List creation	list
E_Lpop	List pop	list
E_Lpull	List pull	list
E_Lpush	List push	list
E_Lput	List put	list
E_Lrand	List random reference	list
E_Lref	List reference	list
E_Lsub	List subscript	subscript
E_Rbang	Record generation	record
E_Rcreate	Record creation	record
E_Rrand	Record random reference	record
E_Rref	Record reference	record
E_Rsub	Record subscript	subscript
E_Sbang	Set generation	set
E_Screate	Set creation	set
E_Sdelete	Set deletion	set
E_Sinsert	Set insertion	set
E_Smember	Set membership	set
E_Srand	Set random reference	set

E_Sval	Set value	value produced
E_Tbang	Table generation	table
E_Tcreate	Table creation	table
E_Tdelete	Table deletion	table
E_Tinsert	Table insertion	table
E_Tkey	Table key generation	table
E_Tmember	Table membership	table
E_Trand	Table random reference	table
E_Tref	Table reference	table
E_Tsub	Table subscript	subscript
E_Tval	Table value	value

Notes: ListMask, RecordMask, SetMask, and TableMask contain the codes for list, record, set, and table events, respectively. StructMask contains all structure events. In most cases, structure reference events occur in pairs with the referencing event first and the corresponding subscript or value next.

String Scanning Events

name	*event*	*value*
E_Sfail	Scanning failure	old subject
E_Snew	Scanning environment creation	new subject
E_Spos	Scanning position	position
E_Sresum	Scanning resumption	restored subject
E_Ssusp	Scanning suspension	current subject
E_Srem	Scanning environment removal	old subject

Notes: ScanMask contains the codes for scanning events. E_Spos events occur for all changes to the scanning position except when a new scanning environment is created. An E_Snew event implies changing the scanning position to 1.

Co-Expression Events

name	*event*	*value*
E_Coact	Co-expression activation	co-expression
E_Cofail	Co-expression failure	co-expression
E_Coret	Co-expression return	co-expression

Assignment Events

name	*event*	*value*
E_Assign	Assignment	variable name information
E_Value	Assignment value	value assigned
E_Ssasgn	Assignment to substring	length of resulting string

Notes: AssignMask contains the codes for E_Assign and E_Value, but not for E_Ssasgn. The event value for E_Assign is based on the string produced by name(). In the case of identifiers, the event value for E_Assign contains additional information about the type of identifier, and in the case of local and static identifiers, the procedure name is listed also. A + after an identifier name indicates a global variable, :, a static variable, –, a local variable, and ˆ, a parameter. In the last three cases, the procedure name follows the symbol, as in

```
count–tabulate
```

which identifies the local identifier count in the procedure tabulate. An E_Value event occurs after the assignment has been made. Thus, an EM can change the value of a variable in a SP following an E_Value event and have the change be effective.

E_Ssasgn events occur as the result of evaluating expressions such as

```
s1[i:j] := s2
```

which is equivalent to

```
s1 := s1[1:i] || s2 || s1[j:0]
```

(assuming i and j are positive and in nondecreasing order).

Type Conversion Events

name	*event*	*value*
E_Aconv	Conversion attempt	input value
E_Fconv	Conversion failure	input value
E_Nconv	Conversion not needed	input value
E_Sconv	Conversion success	output value
E_Tconv	Conversion target	representative value of type

Notes: ConvMask contains the codes for conversion events. Each conversion consists of three events. The first is E_Aconv, which is followed by E_Tconv. Next is one of the other events depending on whether the conversion fails, is unnecessary (conversion of a value to its own type), or is successful (conversion of a value to another type). Since the potential output value is not available when a E_Tconv event occurs, a representative value of the type is used. This allows the types for an attempted conversion to be determined in cases where the conversion fails. Note that the event values for the codes E_Fconv and E_Nconv are not particularly useful.

Allocation Events

name	*event*	*value*
E_Alien	Alien allocation	bytes allocated
E_BlkDeAlc	Block deallocation	bytes deallocated
E_Coexpr	co-expression allocation	bytes allocated
E_Cset	Cset allocation	bytes allocated
E_External	External allocation	bytes allocated
E_File	File allocation	bytes allocated
E_Free	Free allocation	bytes allocated
E_Lelem	List element allocation	bytes allocated
E_List	List allocation	bytes allocated
E_Lrgint	Large integer allocation	bytes allocated
E_Real	Real allocation	bytes allocated
E_Record	Record allocation	bytes allocated
E_Refresh	Refresh allocation	bytes allocated
E_Selem	Set element allocation	bytes allocated
E_Set	Set allocation	bytes allocated
E_Slots	Hash header allocation	bytes allocated
E_StrDeAlc	String deallocation	bytes deallocated
E_String	String allocation	bytes allocated
E_Table	Table allocation	bytes allocated
E_Telem	Table element allocation	bytes allocated
E_Tvsubs	Substring trapped variable allocation	bytes allocated
E_Tvtbl	Table-element trapped variable allocation	bytes allocated

Notes: AllocMask contains the codes for all allocation events (but not deallocation events). See also the next section on garbage collection events.

Garbage Collection Events

name	*event*	*value*
E_Collect	Garbage collection	region number
E_EndCollect	End of garbage collection	null value
E_TenureBlock	Tenure block region	size
E_TenureString	Tenure string region	size

Notes: If E_EndCollect is in the event mask for EvGet(), the data objects saved by garbage collection are reported as allocation events using the same event codes as for allocation. Such events occur after the E_Collect event but before the E_EndCollect event. This dual use of event codes occurs only if E_EndCollect is in the event mask. Monitors that request E_EndCollect events need to take into account the context in which allocation events are reported.

Interpreter Evaluation Stack Events

name	*event*	*value*
E_Intcall	Call of interpreter procedure	interpreter signal
E_Intret	Return of interpreter procedure	interpreter signal
E_Stack	Stack depth change	stack depth

Notes: The stack depth reported in the event value for E_stack is erroneously large. Use the event value of the first E_Stack event as a base for subsequent values.

Other Events

name	*event*	*value*
E_Error	Run-time error	error number
E_Exit	Program exit	exit code
E_Loc	Program location change	line/column number
E_MXevent	Event in EM window	window event
E_Opcode	Virtual-machine instruction	operation code
E_Tick	Clock tick	number of ticks

Notes: E_Tick events are obtained by checking the system clock during program interpretation. During garbage collection and other time-consuming activities the suspend interpretation, several clock ticks may occur before they are reported. This is reflected in the event value for E_Tick. On a Sun 4, the clock ticks once every 10 milliseconds. The event value for an E_Loc event contains the SP source-program column number in the high-order 16 bits and the line number in the low-order 16 bits.

Artificial Events

name	*event*	*value*
E_Disable	Disable monitoring	varies
E_Enable	Enable monitoring	varies
E_ALoc	Program location change	line/column number

Notes: These events are provided for communication between EMs running under the control of a monitor coordinator. The use of artificial events requires the cooperation of EMs and their production of appropriate event values. The E_ALoc event is an artificial version of the E_Loc event and is provided so that SP source-program location information can be communicated between monitors.

Appendix B — Example EMs

Virtual-Machine Presentation

This EM lists every virtual-machine instruction followed by all events that occur before the next virtual-machine instruction.

```
link evinit
link evsyms
link opnames
link options

$include "evdefs.icn"

procedure main(args)
   local codes, esmap, opmap, mask, opts, output

   opts := options(args, "o:")
   output := open(\opts["o"], "w") | &output

   EvInit(args) | stop("*** cannot load SP")

   opmap := opnames()          # table to map opcodes to their names
   esmap := evsyms()           # table to map event codes to their symbols

   mask := cset(E_Opcode)

   #  When a program starts, there are a few pseudo opcodes before real ones
   #  Skip these.

   while EvGet(mask) do {
      if opmap[integer(&eventvalue)] == "Invoke" then {
         writes(output, "Invoke     |")
         break()
         }
      }

   while EvGet() do {
      if &eventcode === E_Opcode then {
         write(output)
         writes(output, left(opmap[integer(&eventvalue)], 10), "|")
         }
      else writes(output, " ", esmap[&eventcode])
      }

   write(output)

end
```

Typical output:

```
Invoke     | E_Pcall E_Loc
Mark       |
Pnull      |
Global     |
Pnull      |
Global     |
Global     | E_Loc
Keywd      | E_Loc
Invoke     | E_Ecall E_Fcall E_Aconv E_Tconv E_String E_Sconv E_Fret E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret
Unmark     |
Mark       |
Pnull      |
Global     |
Pnull      |
Global     |
Pnull      |
Pnull      |
Global     | E_Loc
Size       | E_Ocall E_Oret
Int        | E_Loc
Div        | E_Ocall E_Oret E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret
Unmark     |
Mark       |
Pnull      |
Global     |
Str        | E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret
Unmark     |
Mark       |
Pnull      |
Global     |
Global     |
Str        |
Global     | E_Loc
Invoke     | E_Ecall E_Fcall E_Aconv E_Tconv E_Nconv E_String E_Fret E_Loc
Asgn       | E_Ocall E_Assign E_Value E_Oret
Unmark     |
```

Summary of Numeric Computation

This EM summarizes numerical computation, listing the number of times each operation is performed. The output is divided into integer and real arithmetic

```
link evinit
link options
link procname

$include "evdefs.icn"

procedure main(args)
   local opts, itime, output, inttbl, reltbl, cmask, rmask, numlist, op
   local pos, neg, plus, minus, mpy, div, pwr, mod

   opts := options(args, "o:t")

   output := open(\opts["o"], "w") | &output

   if \opts["t"] then itime := &time

   EvInit(args) | stop("*** cannot load SP")

   inttbl := table(0)
   reltbl := table(0)

   cmask := E_Fcall ++ E_Ocall
   rmask := E_Fret ++ E_Oret ++ E_Ffail ++ E_Ofail

   pos := proc("+", 1)
   neg := proc("+", 1)
   plus := proc("+", 2)
   minus := proc("+", 2)
   mpy := proc("*", 2)
   div := proc("/", 2)
   mod := proc("%", 2)
   pwr := proc("^", 2)

   while EvGet(cmask) do {

      #  Check to see if the operation is a numeric one.

      if (op := &eventvalue) === (
         plus | minus | mpy | div | neg | pwr | mod |
         iand | ior | ixor | icom | ishift | pos)
```

```
          #  If it is, look for the return event.

          then {
                EvGet(rmask)
                if &eventcode === (E_Ofail | E_Ffail) then next# skip failures
                case type(&eventvalue) of {
                   "integer":  inttbl[op] +:= 1
                   "real":     reltbl[op] +:= 1
                   }
                }
          }
       write(output, "\nInteger computation:\n")
       numlist := sort(inttbl, 3)
       while write(output, left(procname(get(numlist)), 6), right(get(numlist), 9))

       write(output, "\nReal computation:\n")
       numlist := sort(reltbl, 3)
       while write(output, left(procname(get(numlist)), 6), right(get(numlist), 9))

       write(output, "\nelapsed time: ", &time – \itime, "ms")

    end
```

Typical output:

```
    Integer computation:

    +1            1
    %2           16
    *2           90
    +2        28324
    –2        23194
    /2           16
    ior        6415
    ishift    21876
    ixor       7730

    Real computation:

    /2            1
```