Writing Execution Monitors for Icon Programs

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 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].

To a large extent, the purpose 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] 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.

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

Events

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 with meaningful symbolic names in the form of global identifiers that are available to the EM. For example, the code for string allocation is named E_String, the code for a list reference is named E_Lref, and the code for the execution of a virtual-machine instruction is named E_Opcode.

An event value can be any Icon value. In the case of E_String, it is an integer corresponding to the number of bytes allocated. In the case of E_Lref, it is the list referenced in the TP. Note that such events provide a EM with direct access to data in the TP.

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

The appendix contains a list of event codes by category.

The Monitoring Interface

The monitoring interface consists of functions, keywords, and a library of support procedures named evinit.

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 TP (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 TP. These arguments default to EM's &input, &output, and &errout. In this case the EM and the TP share these files.

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 TP has terminated.

The function

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

produces an event at the source level (as opposed to in 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 accepts event codes that are not one-character strings.

Programs that need the definitions provided by evinit but that do not perform monitoring themselves (and hence do not call EvInit()) can include evdefs.icn.

Masks

Masks serve to limit the events that are reported to those of interest to a 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. (This string value is automatically converted to a cset by opmask().)

The library include file opdefs.icn contains definitions for all virtual-machine instructions for use in opmask(). 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 TP terminates. The name of the TP is given as the first argument of the EM's command line. The remainder of the command line is passed to the TP. ProcMask is a mask that includes only procedure events. See the appendix for an explanation of procedure events.

```
link evinit

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.

Programming Guidelines for Monitors

The ucode file evinit must be linked in the EM.

Both TPs and EMs must be compiled using MT Icon.

EvInit() must be called at the beginning of the EM. It not only loads the specified icode file, but it initializes global variables and performs other tasks that must be done before monitoring begins.

Since a TP usually produces a very large number of events, efficiency is an important consideration in writing EMs. Events should be restricted to only those of interest (the argument of EvGet()). The argument should be changed if the events of interest change in a specific situation.

Monitors that use visual displays should pay special attention to how graphics facilities [4] are used. In particular, frequent calls to WAttrib() should be avoided. Where foreground or background colors needs to be changed frequently, it is better to create separate graphic contexts for the attributes needed and to use the appropriate graphic context.

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

Support Procedures for Monitors

Several support procedures are available for use in EMs. See [5] for descriptions of these.

Disclaimer

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

Much of the instrumentation is relatively untested.

Acknowledgement

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

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
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, 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.
5. R. E. Griswold, *Support Procedures for Icon Program Monitors*, The Univ. of Arizona Icon Project Document IPD193, 1994.

Appendix — Event Codes and Masks

Global variables for event codes are listed below. The actual event codes are given in evdefs.icn.

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().

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: 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. ScanMask contains the codes for scanning events.

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

Notes: AssignMask contains the codes for assignment events. 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 ^, 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. The E_Value event occurs after the assignment has been made. Thus, an EM can change the value of a variable in a TP following an E_Value event and have the change be effective.

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	input 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 failed, was unnecessary (conversion of a value to its own type), or successful (conversion of a value to another type). The actual output value is not always available in Icon form, so a representative value of the type is used for the value associated with E_Tconv. Note that the event values for the codes E_Fconv, E_Nconv, and E_Sconv are not useful.

Allocation Events

name	*event*	*value*
E_Alien	Alien allocation	bytes alloced
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 alloced
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 Stack Events

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

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	event
E_Opcode	Virtual-machine instruction	operation code
E_Tick	Clock tick	number of ticks

Notes: On a Sun 4, the clock ticks once every 10 milliseconds. The event value for an E_Loc event contains the TP 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: The use of artificial events requires the cooperation of TPs 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 TP source-program location information can be easily communicated between monitors.