Some Fundamentals of an Event-oriented Control Language for Applications in Parallel Systems

Lambert R.B. Schomaker

NICI Technical Report, Feb. 1988

Nijmegen Institute for Cognition Research
and Information Technology (NICI),
University of Nijmegen, P.O.Box 9104,
6500 HE Nijmegen,
The Netherlands

Abstract

This short technical report informally describes some basic ideas for a command language that naturally incorporates the asynchronous nature of events in the users' working environment. The general idea, system components and problem areas are dealt with briefly. Many concrete examples of the language syntax are given.

1 Introduction

Office metaphors are an extension of the well-known desktop metaphor in a graphical multi-room, multi-operator environment representing parts of the physical office and the information channels and sources. It is clear that handling of events is extremely important in such an environment as opposed to the single PC world where only the user himself determines the occurrence of events. The idea is, that a user defines a **World** in his personal profile file (i.e., "login.com", "profile","autoexec.bat"), defining the actions that the system must undertake if a special event occurs. Events are the reception of electronic mail, the availability of data or of an application program, the occurrence of a specified time of the day and so on. In current systems, functionality that allows for the processing of asynchronous events is virtually lacking. This leads to the situation where sophisticated system users start to build their own server tasks or "demons" to solve the event handling problem by a dumb polling method. Obviously, this is an undesirable situation because of the enormous computational overhead.

This report describes some ideas on a parallel operating system (OS) control language that handles such real-life events, that are considered "asynchronous" from the point of view of a von Neumann machine. The functionality can be considered as supplementary to the basic system functions that a general control language encompasses. The language is built around the concept of

$$Event \rightarrow Task$$

bindings.

An event is an internal (in-memory) or peripheral change in system conditions. Examples of events are: the creation of a file, the pressing of a key, the availability of a data packet from a communication line, some global variable attaining some predefined value, the arrival of an electronic mail message, etc.

A task is an independent program (image), running in its own context. When a program is started, it is able to fetch and parse (a) command line arguments in the interactive case, or to fetch and parse (b) passed arguments from a specified context as defined by the $Event \rightarrow Task$ binding. The code of a normal application program for a task triggered by a (combination of) predefined event types can stay free from dependencies that are related to the parallel system that handles the firing of the $Event \rightarrow Task$ bindings. Only the general argument handling facility would be necessary.

If an event takes place, the $Event \rightarrow Task$ binding is **Instantiated**. A global table $E(Event)$ is filled describing the triggering circumstances and a task or process is

invoked. The context in which this task starts is determined by the binding statement. The event status table $E()$ need not be inspected by the average program if there is no more extra event or status parsing necessary.

2 A preliminary syntax description

There are two main types of defining an $Event \rightarrow Task$ binding.
Syntax: [1]

a Single event, single task, immediate binding, immediate activation:

event_description [**/options**] → **task**[**(context:arguments)**] [**/options**]

Options are /NAME=name to give the event a symbolic name, and

/PRIORITY=number, to give the task execution some priority over or under the priority of other tasks if the event occurs.

b Deferred activation: the binding_table is filled, but bindings only become active after the ENABLE keyword is encountered by the interpreting system:

```
DEFINE_EVENTS: myworld
   event_description_1 [/NAME=nn] → task_1 [/PRIO=mm]
   event_description_2 [/NAME=nn] → task_2 [/PRIO=mm]
          .
          .
          .
END;
ENABLE/WORLD=myworld/ALL_EVENTS
```

The total set of bindings that belong together is called a **World**.

3 Event-handling and context

Context refers to a data structure containing symbols and logical names connected to a task, as well as items such as the name of the current directory. The names of the standard Input and Output ports are contained in symbols. The context is named after the task for which it holds. Since task names are unique, there is no name collision problem for the context names. By entering the context name in an $Event \rightarrow Task$ binding, we declare the context that is to be used by a task. Context may be overlapping, if needed, to provide for a blackboard mechanism. In the latter case, several tasks have simultaneous access to symbols within one and the same context.

[1] Generally, UPPER case words denote pre-defined syntax items, or reserved names, lower case words denote user-defined items.

Context Identifier:	**Meaning:**
∗ (asterisk)	Use current context.
task_nam	Use context of task_nam.
task_nam:symnam	Use context of task_nam and send the contents of symbol "symnam".
task_nam:(symnam1,symnam2)	Use context of task_nam and send the contents of symbols "symnam1" and "symnam2" as argument to the event-related task.

Table 1: context identifiers and their meaning.

Note that the default task creation mechanism determines what symbols there are in a given context. Some symbols are local while others are inherited from parent tasks. The following section give some examples of the principles described earlier.

4 Some syntax examples

```
DEFINE_EVENTS: myworld
   KEYBOARD("PF4") → handle_event(*)
      !When the key PF4 is pressed, a user task "handle_event" is executed
      !as a subprocess, using the context of the current process,
      !this is designated by (*). The event is nameless which means
      !that it has a unique name provided by the system

   KEYBOARD("ABC")/NAME=keyabc → handle_event(*:("file.dat",arg2))
      !When the sequence ABC is pressed, the task "process_event" is
      !executed. It takes the arguments "file.dat" and symbol arg2
      !(from the current context). The event is named "keyabc".

   KEYBOARD("Control-Y")/NAME=brkall → breaker(*)/PRIO=100
      !When a Control-Y key is pressed, the task "breaker" is
      !executed at high priority. It uses the current
      !context. The event is named "brkall". This set-up
      !allows e.g. for a one_shot selective removal of processes.

   TIME(+100) → update(process_x:(arg1,arg2))
      !In 100 time ticks, the task "update" will be executed.
      !It uses the symbols arg1 and arg2 from context "process_x"
      !as arguments.
```

TIME(17:00) → logout()
!At five-o-clock, the current process will logout.

CYCLIC(TIME(+1000) → myclock(*))
!Until now, the event definitions were one-shot.
!The keyword CYCLIC enables automatic re-ENABLING of an
!$Event \rightarrow Task$ binding after the event has taken place.
!The example provides e.g. for a user written clock.

CREATE(SYMBOL="process_a:ready") → check_ready(*)
DELETE(SYMBOL="process_a:ready") → check_ready(*)
CHANGE(SYMBOL="process_a:ready") → check_ready(*)
READ(SYMBOL="process_a:ready") → check_ready(*)
WRITE(SYMBOL="process_a:ready") → check_ready(*)
!These examples show how an action is undertaken if something
!happens to a symbol. In this case, symbols from process_a are
!watched. If the specified event occurs, e.g. the creation of a
!symbol named "ready", the task "check_ready" will be executed,
!using the current context. Wild-carding of symbol name is allowed.

CREATE(FILE="dum.dat") → some_action(*)
DELETE(FILE="dum.dat") → some_action(*)
CHANGE(FILE="dum.dat") → some_action(*)
READ(FILE="dum.dat") → some_action(*)
WRITE(FILE="dum.dat") → some_action(*)
!These examples show how an action is undertaken if something
!happens to a file. In this case, files from the current directory
!will be watched. If the specified event occurs, e.g. the creation
!of a file named "dum.dat", the task "some_action" will be
!executed, using the current context. Wild cards allowed.

WRITE(FILE="mail.mai") → warn(:"USR$TERM")
!In this case the task warn is a context-less task, that uses
!the current user terminal as an output port to give a warning
!when something is being written into the mail.mai file.
!In combination with CYCLIC, e.g., CYCLIC(WRITE(FILE="MAIL.MAI"))
!we can create a mail handler.

DATA_RDY(PORT=ddv:) → read_from_DDV(*)
DATA_REQ(PORT=ddv:) → write_to_DDV(*)
!These examples show how an action is undertaken if something
!happens to a device.

BIRTH(task_calcul) → log(*:"task_calcul starts")
SLEEP(task_calcul) → log(*:"task_calcul sleeps")

```
    WAKEUP(task_calcul) → log(*:"task_calcul works again")
    EXIT(task_calcul) → log(*:"task_calcul stops")
        !Here a task is executed if something changes in the state of
        !a task with the name "task_calcul". The task "log" stores
        !the given string constants somewhere.
        !Note that we talk about transitions: so SLEEP() means
        !transition from any state to SLEEPing. Furthermore, these
        !events put more detailed process status information into
        !the global event table E().

END; !now these task bindings are defined.
ENABLE/WORLD=myworld/ALL_EVENTS;
```

After the final ENABLE command, all the task bindings defined between DEFINE_EVENTS and END are open to instantiation.

5 More complex constructs: chaining and task-generated events.

Complex systems can be built by using chaining: the definition of $Event \rightarrow Task$ bindings that are made dependent on events that are generated by the tasks themselves instead of being generated by external events. The most simple solution to chaining is the attachment of a task to the occurrence of an EXIT event, e.g.:

```
EXIT(spreadsheet) → plot(*:"result.fil")
EXIT(plot)        → message(:"USR$TERM")
```

Tasks may also create **Autogene Events** to control the subsequent flow of processing. Autogene events differ from other events in that they occur artificially by deliberate action of a task. A typical task that causes an autogene event to occur picks up the associated symbolic event name from the invocation argument list, as in the following example.

$$\begin{aligned} \text{...some event...} &\rightarrow task_1(* : ..., event_{11}, event_{12}, event_{13}) \\ event_{11} &\rightarrow task_{11}(* : ..., event_{111}) \\ event_{12} &\rightarrow task_{12}(* : ..., event_{112}) \end{aligned}$$

The task & event numbering is only done for the sake of clarity. In any case, event names must be unique: they must not collide with any pre-defined or autogene event name.

Furthermore, detached tasks (demons) can be created to invoke user-defined autogene events. Such a demon may be active all the time, and are initiated by the pre-defined autogene event named RUN_WATCHDOG. Ideally, such a watchdog runs on a separate CPU. The mechanism is necessary for the detection of complex circumstances, such as an intrusion to the system.

RUN_WATCHDOG → intrusion_checker(*:intrusion_occurs)
intrusion_occurs → warn(:"USR$TERM","Intruder",$E(intrusion_occurs)$)

The task intrusion_checker infinitely checks for some kind of intrusion event and invokes the event called "intrusion" if it happens. When the event "intrusion_occurs" takes place, the task "warn" is executed, sending some message to the user terminal. Note that the warning task uses the event table $E()$ under index "intrusion_occurs" to see what actually happened.

6 Preventing avalanches.

If events occur at a rate, faster than the rate at which tasks can be finished, essentially two modes can be described. These modes are determined with an optional qualifier when defining an $Event \rightarrow Task$ binding. An option called /DUPLICATE means activation, even if the task is already active. This consumes system resources so it seems reasonable to take /NODUPLICATE as the default mode. An example:

EVENTX → TASKY(*)/NODUPL

prevents instantiation of a binding when the bound task is already active and has not finished yet. Bindings that cannot be instantiated at event time are put in a queue, possibly with a time-out value attached. This queue is implemented in the global table $E()$.

Other safety mechanisms involve the deletion of $Event \rightarrow Task$ task bindings,

DELETE/EVENT=brkall

or the temporary enabling or disabling of an $Event \rightarrow Task$ binding:

DISABLE/EVENT=brkall/WORLD=myworld
ENABLE/EVENT=brkall/WORLD=myworld

Temporarily enabling or disabling of a whole binding world:

DISABLE/WORLD=myworld/ALL
ENABLE/WORLD=myworld/ALL

7 Temporal logic.

Needed are logical operators that include temporal specifications of order and temporal vicinity. The following syntax refers to the description of the occurrence of events within a specified time window.

(within_time_window(1sec(event1.and.event2))→ my_task
(within_time_window(24hour(event1.or.event2))→ my_task

The occurrence of a predefined sequence of events can be described and implemented in one of the following two ways:

(event1.then.event2.then.event3)→ my_task

or:

(event1 →
  (event2 →
    (event3 → my_task)))

In the latter nested set-up (event2 → ...) is a **Nameless Task** that activates (event2 → ...) that activates the user-defined task "my_task" in turn.

8 What event types should be hard-wired into the design?

Event Types are pre-defined types of events. Since the handling of a large amount of events is costly in terms of computation, a dedicated architecture must be designed that allows for the efficient processing of a set of such pre-defined event types.

Question: should the number of event types be large (Rich and Specific Set) or be reduced (Reduced and General Set), delegating fine event characterization to the task once started? Example: should we use a single event called "FILE_ACCESS" for any kind of access to a file and let the task decide what access type actually happened, or should we use a detailed event description at the top level as depicted earlier?

Event types	**Advantages**	**Disadvantages**
Rich and Specific Set	Less spurious task activation Decisions left to OS&hardware Top level shows dependencies	Top-level is complex Preferably impl. in hardware Inflexible through hard-wired nature
Reduced and General Set	Easier implementation Top level has limited complexity Flexible: more things possible if the user is able to program.	Tasks are activated often Tasks have to take decisions Less predictable behaviour

Table 2. Advantages and disadvantages of the Rich Set approach and the Reduced (general) Set approach.

Solution: provide as much hard-wired event types as possible so the implementation can be fast, but give hackers the opportunity to create their own events if these cannot be produced by any combination of given event types. This is done by means of the autogene event mechanism.

9 Security

It must be possible to protect **Context** and **Event Type** from access by tasks of other system users.

10 Problem areas

The general disadvantage of the proposed system is the reduced predictability of system behaviour if the number of bindings increases. Also, the consumption of systems resources will be much higher than in in existing systems.

11 Implementation aspects

This section is written without the claim that the author has specific hardware expertise. Ideally, bindings are delegated to different watchdog CPU's, DMA connected to the general bus. Thus, such a CPU can handle several bindings while the big central processing unit is not bothered by this low-level polling. These processors scan memory structures for events and generate an interrupt if an event happens, or just pass device interrupts to the central CPU. Before the interrupt is generated or passed, a table is filled with the event description. A equalizing algorithm assigns watchdog functions to these CPU's.

Dedicated hardware may provide for task activation: normally some base pages of each task already reside in memory, but the copying of additional pages can be done by a dedicated CPU or chip. Maybe in large-memory systems these problems are not so important as they are now.

In any case, testing of the ideas above can be done fully in software, e.g.in an existing VMS or Unix environment, starting with a simulating top layer first, before descending to the lower levels of these existing operating systems.

12 An application example

The following is an example in which a customized intelligent mail handling and sorting facility is implemented. Assuming that the arrival of an electronic mail message is identical to the creation of a new file, we can set up the following application.

CREATE(FILE="/mydir/mailfiles/mail.*")/NAME="new_mail_arrived"
→ mail_sort(*:E("new_mail_arrived"))

The user application "mail_sort" is a program that checks the contents of the newly received mail file. The name of this file is looked up in the global event table $E()$ under event "new_mail_arrived". Typically, the program will search for keywords in the "sender" and "subject" fields. This way, the system can sift out important mail from trivial messages ("OPERATOR"-originating, e.g.).

(finished 2 feb 1988)

Contents