1986

Omicron: Events => Action

Balachander Krishnamurthy

Craig E. Wills

Report Number:
86-594

OMICRON: EVENTS => ACTION

Balachander Krishnamurthy
Craig E. Wills

CSD-TR-594
April 1986

Omicron: Events ⇒ Action*

Balachander Krishnamurthy
Craig E. Wills

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD-TR-594

April 14, 1986

Abstract

Omicron allows users to automate the performance of tasks based upon the occurrence of one or more events. It encompasses features available in other time event based programs in UNIX such as *usrcron* and *at*, but also includes time range and non-temporal events. A powerful feature of *Omicron* is the capability to specify that tasks be executed depending on some combination of events.

Omicron is a tool that places emphasis on the user interface to encourage wider use. The two most noticeable features of the user interface are its readable syntax and direct communication with the server that implements *Omicron*. Underlying the workings of *Omicron* is a semantic model based on Petri nets. *Omicron* is an implementation of the philosophy that the system should function as an aid to the user in automation of his work rather than be a simple command interpreter and task executor.

*This work was supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

1 Introduction

In this paper we describe the motivation, design, and implementation of a tool to automate task execution in a system. The tool, called *Omicron*, allows users to automate the performance of actions based on the occurrence of one or more events. Events can be based on time, such as reaching a particular time during the day, or on other actions, such as a user logging in or a file changing.

Omicron is motivated by our belief that many tasks normally performed by a user in interacting with the system can be done automatically without user intervention. Each task the user performs is triggered by the occurrence of one or more events. For example, if the user modifies a source file he recompiles the program, or if a meeting time is approaching he sends mail to all participants. *Omicron* is designed to give the user flexibility to specify different types and combinations of events so that he can easily describe the conditions for automatic performance of these actions.

Our tool has some similarities to two existing tools in the UNIX† environment. *Usrcron*[UNI83] allows users to schedule actions at regular intervals, such as every night. *At*[UNI83] allows users to schedule actions for one particular time. Features of these programs have been incorporated into *Omicron*.

Omicron extends the capabilities of event based tools, such as these, in two important ways. First, *Omicron* not only provides for repeating events and one time events, but it provides two additional event types. Time range events that cause the associated action to be performed within a range, and non-temporal events, that allow Omicron to base actions on the evaluation of predicate functions. Examples of non-temporal events include asking if a user is logged in, or

†UNIX is a trademark of AT&T Bell Labs

if a file has been modified. It can also base the performance of actions upon the success or failure of prior actions. Individually these different types of events give the user more functionality than existing tools but the real power of *Omicron* lies in its capabilities for combining events. Much like pipes in UNIX allow commands to be combined to form more powerful commands, *Omicron* allows events to be combined to form compound events. These compound events give the user flexibility to describe the conditions that will trigger an action.

Second, the user interface to *Omicron* simplifies the specification of events and actions. All events are specified similar to *at* which emphasizes readability and intuitiveness for the user. In addition, *Omicron* is a server that receives requests for event specifications directly from client programs invoked from the shell. Client programs also exist to list, suspend, and remove events in *Omicron*. This style of interaction allows the user to "talk" to the server and immediately confirm that events have been added or removed. In comparison, *at* is a shell level command that puts its specification in a file that is periodically checked, while specifications to *usrcron* are entered directly into a file. Besides using client programs from the shell, we also allow events to be specified in a file.

Underlying the workings of *Omicron* is a semantic model based on Petri nets[Pet77]. The model forms a basis to understand the workings of *Omicron* and is used by us, the designers, in considering the compatibility of any changes to the system.

2 Design and Features

Omicron is a tool that can be used to cause the execution of a task based upon the occurrence of one or more events. In practice, *Omicron* is a single server process

that controls the specification of events and execution of actions for each user in the system. Following the client-server model of interaction, most communication with *Omicron* is done using a set of client routines invoked from the shell.

The dynamic communication routines provided include: *regom* – a program that registers the user with *Omicron*, *addev* – a program to add events to the user's list, *lsev* – a program that lists the current set of events specified by the user, *suspev* – a program that suspends events till they are reactivated by using *fgev*, and *rmev* – a program that can be used to remove events. *Omicron* commands can also be used in shell script files or invoked from other commands. For example, by invoking *Omicron* commands at session creation and completion time, events relative to the session can be scheduled.

The user communicates directly with the *Omicron* server. He can confirm that any events he adds to *Omicron* have been registered by immediately listing his events and he can discard any events that he no longer wants. This style is in sharp contrast to *usrcron* and *at*. Neither of these tools allow the user to list his current events. In addition, events can be specified to *usrcron* only via a file that is checked periodically for updates. Unfortunately the user cannot be sure how soon the file will be checked and thus must be careful in specifying actions "too soon" in the future. In addition to adding events from the shell, we also have a file (*.omicron* in the user's home directory) that is periodically checked for updates. To avoid similar problems as *usrcron*, we provide an additional command, *readev*, to force *Omicron* to read the user's *.omicron* file. We envision that the user will put permanent events in his *.omicron* file, while he will enter other events from the shell.

The basic types of events are: one time, time range, repeatable, and predicate. The success or failure of a specified action can also be viewed as an event. A few

simple examples of events and their actions are:

```
at 5pm Mail -s "go home" userid
in 20 min Mail -s "reminder of meeting" userid
by 8am make all
between 8am and 4pm do_task
```

The first two examples are one time events and are scheduled relative to the specification time. The latter two examples are time range events and cause *Omicron* to execute the action sometime during the given time range. *By* causes execution of the action sometime between now and the time given, while *between* specifies the range explicitly. The actual time of execution of the task is decided by a heuristic that takes into account the machine load average, the amount of time left in the time range, and the number of events scheduled with *Omicron*. Use of a time range allows *Omicron* to pick a suitable time to trigger an event and doesn't force the user to specify the exact time. Note that specification of a time range only guarantees the *commencement* and not the *completion* of the task in the interval.

More flexible event specifications can be made by combining repeat time events along with other events, such as predicates. One such predicate is *filemod* that checks to see if a file has been modified since the last time the file was checked. Predicate functions are often "driven" by repeat time specifications, as illustrated in the following examples:

```
every 10 min filemod /usr/spool/mail/userid inc
every hour idle 60 min echo "please log me out" > /dev/tty
every day between 2:00 and 6:00 rdist -f /usr/userid/.rdistfile
```

The *every* event causes the next event or action to be triggered at regular intervals. In the first example, it is combined with a predicate event (*filemod*)

that causes mail to be incorporated into the user's mail area if the file has been modified since the last check. The second example repeats every hour to invoke a function that checks if the user has been idle at his terminal for 60 minutes, and if so, writes a message to the screen. The third example combines two time events resulting in a compound event that triggers an action sometime between 2:00 a.m. and 6:00 a.m. daily.

Repeatable events can be extended to events that repeat until a boolean expression becomes true. The outcome of actions can be predicates for other actions. Further, events and actions can be labeled for subsequent usage. These concepts are illustrated with the following examples:

```
every Mon until time ge April 15 Mail -s "send in taxes" userid
every 10 min until loggedout filemod /usr/spool/mail/userid inc
at 5:00pm every 10 min until success(A) A:do_task
every 10 min filemod paper.tex B:tex paper.tex
success(B) dvipr paper.dvi
```

The first example illustrates that repeatable events can be stopped when a particular time is reached. The second example is a refinement of an earlier example and might be invoked in a user's startup file at login time. The third example shows that actions can have labels with a task that will be done beginning at 5:00 p.m. until it succeeds. The last example uses an event (labeled B) to watch for a file being changed that causes text processing to occur that in turn causes output to be printed if the processing succeeds.

These examples illustrate the ease of use and flexibility of *Omicron*. Mnemonic command names aid in easy specification of events. At the same time the advanced user is not forced to specify any extraneous information and can form complex events to cause actions to occur. For these reasons we view *Omicron* as a general tool that can be used by a wide variety of users. It combines many

of the features found in subsystems within UNIX (*cron*, *at*, *leave*, *calendar*, *biff*) into a single tool with a uniform interface.

Another feature of *Omicron* is the execution environment of each action that is performed. In the case of events specified in the *.omicron* file, the user's default login environment is used. In case of events specified from the command line, the current environment is saved and used. Unless otherwise specified, all output and error from commands is saved in a temporary file and mailed to the user. A log file is also maintained that gives the events that have been executed and removed.

One other important feature of *Omicron* is its reliability against machine crashes. *Omicron* maintains a copy of each event in the file system and initially reads these events on start up. For relative events (like "tomorrow" or "in 20 minutes"), *Omicron* ensures that events are rescheduled relative to the time the event was specified and not the time *Omicron* was restarted. One issue still not resolved is what to do about actions that should have occurred while the machine was down. The current implementation removes any such events and actions at startup time. An alternative is to execute these actions at startup time. The first choice guarantees that actions are performed at the specified time, but has the potential of not performing some possibly critical action. The second choice guarantees that all actions are performed, but not necessarily at the right time.

Along with the standard set of UNIX commands that can be triggered by *Omicron* events we also have a set of support routines for the user. Examples are an *auto_readnews* routine to automatically read network news the user is interested in (specified by the user in a file), or an *incremental_spellcheck* that can be invoked whenever a file changes. Support routines are not part of *Omicron* itself, but they are an important part in making the system more powerful.

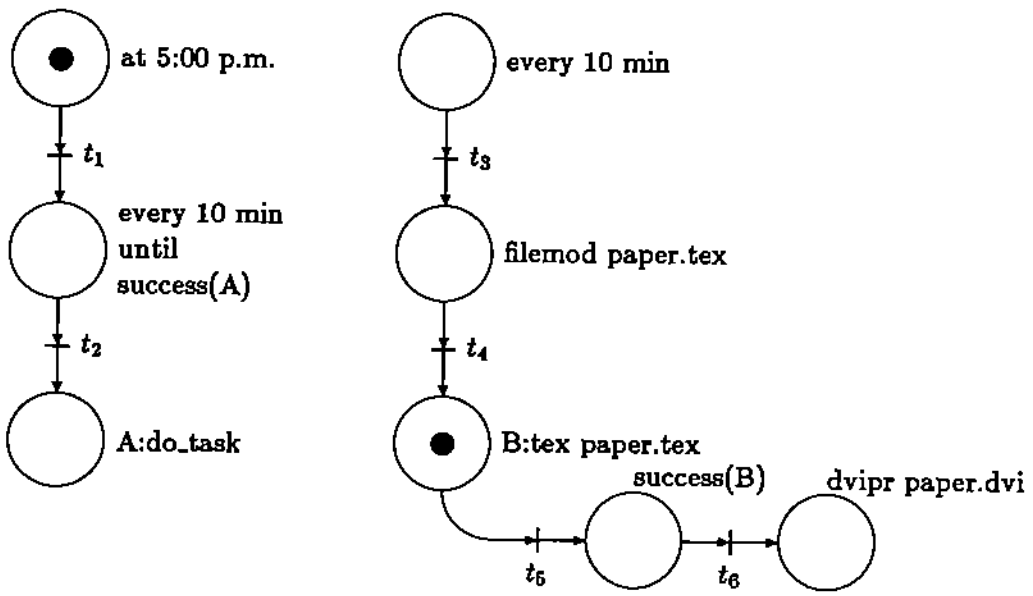
3 Model

The semantic model is an important part of *Omicron* because it guides our design and implementation. As a basis for specifying our model we use Petri nets. This modeling tool allows us to express the semantics of our system in a form commonly used for other event based systems. Modeling *Omicron* with Petri nets not only allows other people to better understand the system, but allows us, the designers, to easily decide whether event specifications can be expressed in terms of the model.

Petri nets are a natural method for modeling systems of events. Briefly, a Petri net is a directed, bipartite graph containing two types of nodes: circles (called *places*) and bars (called *transitions*). The nodes are connected with directed arcs such that each arc goes from a place to a transition (the place is an *input* of the transition), or a transition to a place (the place is an *output* of the transition). The execution of a Petri net is controlled by the position and movement of markers (called *tokens*) that are assigned to places in the graph. The movement of tokens is controlled by the *firing* of transition nodes. A transition node fires when it has a token on each of its input places. A transition firing causes a token to be removed from each input place and one to be deposited on each output place.

Our model is described with these simple rules by adding additional kinds of place nodes to account for different types of events and actions. There are six kinds of place nodes: one for each event type, one for actions, and one as described above. The model can best be illustrated by giving the Petri net for a few examples from Section 2. These are shown in Figure 1.

In the first example, transition t_1 triggers at 5:00 p.m. when a token becomes available at its input node. This action causes the next node to generate a token



at 5:00p.m. every 10 min
until success(A) A:do_task

every 10 min filemod paper.tex B:tex paper.tex
success(B) dvipr paper.dvi

Figure 1: Petri Net Examples

every 10 minutes until the task succeeds. When the task completes a token is generated at the place labeled A and the success or failure of the task is saved. In the second example a token is generated every 10 minutes that causes a predicate function (*filemod*) to check if the given file has been modified. If the function returns true, a token is generated, and t_4 triggers causing text processing to occur at place B. If the function is false then no token is generated. When the task at place B completes it generates a token (as shown), which causes t_5 to trigger and if the task succeeded then the output of text processing is printed.

These examples show how complex events are modeled. Simple event types can be combined together in a consistent and predictable manner, and any event

specification that can be expressed as a combination of these nodes is part of the model. Each of the examples shows events combined in a linear fashion so that each transition has one input and one output. The model also allows for transitions to have multiple inputs and outputs. For example, the transition t_5 could have multiple outputs designating that multiple actions occur when task B succeeds. Specifications that cannot be modeled with the Petri net primitives given here are not part of *Omicron*. For example, place nodes with more than input or output are not part of our model.

Omicron's model is a dynamic web of Petri nets that grows as the user specifies additional events and actions, and shrinks as non-repeating events occur and nodes are removed. We can express the complex functioning of the system in simple terms as well as have a basis to make decisions about changes in the design by using the defined model. We have already found the model very useful as a guideline for implementation of the complete system and deciding whether user suggestions should be added to *Omicron*.

4 Implementation

Our current implementation is for a VAX 11/780 running UNIX 4.2BSD. It has also been ported to a Sequent DYNIX 2.0, a 68000-based Megatest system, and is being ported to a HP-UNIX system. *Omicron* is a single server that listens on a reserved UNIX TCP[Pos81] port for requests and performs any tasks that need to be done. Communication with the server is done by using a privileged port that prevents unauthorized use. Users may only remove, suspend and list events belonging to themselves.

Our current prototype implements most of the features described. The user

can interact with the server from the shell and also add events in his *.omicron* file. The server automatically recovers from machine crashes and logs all tasks that have been executed in a user's logfile. As described, events and actions specified from the shell cause the user's environment to be saved for execution, and actions from the *.omicron* file are executed in the user's default login environment.

The prototype implements all of the event classes described with limitations on repeatable events, basing events on the outcome of actions, and combining events. We see no problems in extending the prototype to a complete implementation. The prototype has given us a feel for how the system is used and strengthened our conviction that *Omicron* is a powerful tool for both novice and advanced users.

5 Conclusion

We have presented the design and a prototype implementation of a general tool that is based on a straightforward notion of events causing actions to be executed on behalf of the user. The design was guided by the desire to allow the user to automate many of the tasks he normally performs.

Experience with our prototype has been beneficial in three ways. First, adding events from the shell and communicating dynamically with the server gives the user a much better "feel" for the system. Instead of adding events to a file and waiting for a server to eventually read them, he can get immediate feedback. Error messages, if any, are promptly reported to the user, and he can list and remove events.

Second, we have gotten many suggestions for new events from users and incorporated them into the implementation. Any event that can be classified in

one of the four class of events we have defined can easily be added to *Omicron*. Thus we have an extensible tool within the framework of our design.

Third, we are confident that combining events will be a very powerful feature of *Omicron*. Many times we have seen uses for combining events together that cannot be done with our current implementation. A common example is to combine an *every day* event with other events and actions. We expect combining events to be particularly useful for experienced users.

We are encouraged by our prototype and are constantly adding to the capability of *Omicron*. The easy specification, variety of events, mechanism for combining events, and the underlying semantic model makes it a powerful tool for automating many of the tasks that all users perform.

References

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Pos81] J. Postel. Transmission Control Protocol– DARPA Internet program protocol specification. September 1981. RFC 793.

[UNI83] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version.* Computer Science Division, University of California, Berkeley, August 1983.