VISUAL CONVENTIONS FOR SYSTEM DESIGN USING ADA 9X: *REPRESENTING ASYNCHRONOUS TRANSFER OF CONTROL.*

Jeffrey V. Nickerson
New York University
jnickerson@acm.org

INTRODUCTION

Ada 9X provides a number of features that increase the power of the language to express time-based algorithms. In particular, asynchronous transfer of control allows a complex, time-based behavior to be expressed simply and powerfully.

System design is often accomplished in team meetings in which diagrams are used to explain proposals and explore possibilities. Buhr (1984) advocates the use of a standard set of notations for time-based software systems; Buhr (1990) makes the point that all engineering disciplines with the possible exception of software design use well-defined conventions for visualizing solutions. Buhr has created a system design notation that is closely linked to the concepts of Ada 83. This paper proposes extensions to Buhr's notation to allow for the representation of asynchronous transfer of control.

DEFINITION

Asynchronous transfer of control is defined in the Annotated Ada 9X Reference Manual (1993) in the following way (AARM 9.7.4.2;2.0):

```
ASYNCHRONOUS_SELECT ::=
select
 TRIGGERING ALTERNATIVE
then abort
 ABORTABLE PART
end select;

TRIGGERING_ALTERNATIVE ::=
TRIGGERING_STATEMENT [SEQUENCE_OF_STATEMENTS]

TRIGGERING_STATEMENT ::= ENTRY_CALL_STATEMENT
| DELAY STATEMENT

ABORTABLE_PART ::= SEQUENCE OF STATEMENTS
```

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise or republish, requires a fee and/or specific permission.

©1993 ACM 0-89791-621-2/93/0009--0379 1.50

The transfer of control is accomplished through the use of an *abortable part*. If an entry call is completed while abortable part processing is taking place, the abortable part processing is aborted and control goes to the triggering alternative.

A TEXTUAL EXAMPLE

A user command interpreter can be represented as a loop, in which commands are retrieved from a user's input on a terminal, and then invoked. At any point the user may wish to abort the program by pressing *escape*, *Control-C*, or some other special key combination. This can be written in the following manner (AARM 9.7.4.9;2.0):

```
loop
  select
     TERMINAL.WAIT_FOR_INTERRUPT;
     PUT_LINE("Interrupted");
  then abort
     PUT_LINE("-> ");
     GET_LINE(COMMAND, LAST);
     PROCESS_COMMAND(COMMAND(1..LAST));
  end select;
end loop;
```

Note that `TERMINAL.WAIT_FOR_INTERRUPT` is an entry call meaning that the triggering statement will wait until some event happens on the terminal that allows the accept statement on the terminal to complete.

CREATING THE VISUAL CONVENTION

Early discussion of asynchronous transfer of control described it as being similar to an operating systems fork. More recent discussions have speculated on using a two-thread model to implement the feature. Therefore we first consider using Buhr notation features that deal with the creation and destruction of tasks.

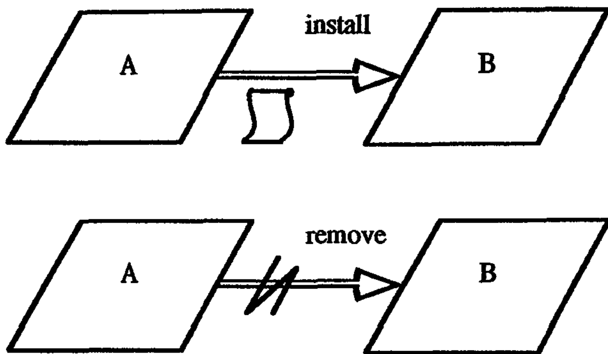


Figure 1.

Buhr(1984) contained the concept of abort - Buhr(1990) supersedes this with the paired concepts of installation and removal. The convention shown in figure 1 shows a machine being installed based on a blueprint, represented as a scroll (we omit this scrolled icon from the remainder of the paper). The second part of the figure shows a machine being removed, which is equivalent conceptually to aborting a task.

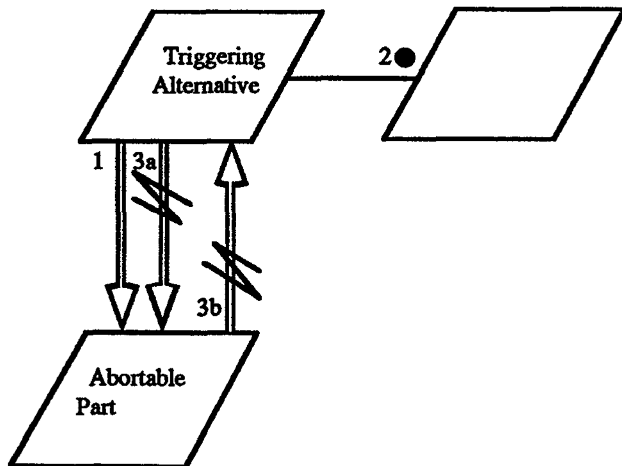


Figure 2.

In figure 2 the triggering alternative installs the abortable part, then makes an entry call (2) and blocks. (The dot at 2 is a Buhr convention indicating a potential waiting.) If the entry call completes, the triggering alternative removes the abortable part (3a). If the abortable part completes first, the abortable part aborts the triggering alternative (Ada 9X calls for the triggering statement to be aborted, and the sequence of statements of the triggering alternative not be executed) (3b).

This diagram makes explicit the two-sided nature of the asynchronous transfer of control - depending on whether the abortable part or the triggering statement complete first, either may end up aborting the sequence of statements or the triggering statement of the other.

However, the diagram implies concepts that do not exist in the language construct. In Ada 9x, there is no sense in which the triggering alternative creates the abortable part. In a more general sense, the triggering alternative is not intended to be an independent task. Also, the abortable part can only abort the triggering statement when the abortable part completes - the diagram implies more symmetry than exists in the language construct.

As an alternate way to model asynchronous transfer of control, Buhr's conventions for exception-handling can be used.

Buhr (1990) calls for a hooked line to be used to indicate propagation of exceptions and alarms. An alarm handler is represented as a rectangle:

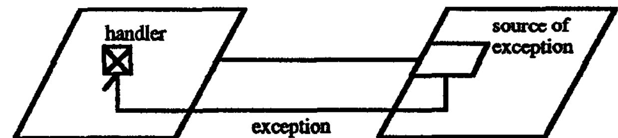


Figure 3.

Using this convention, an asynchronous transfer of control can be shown:

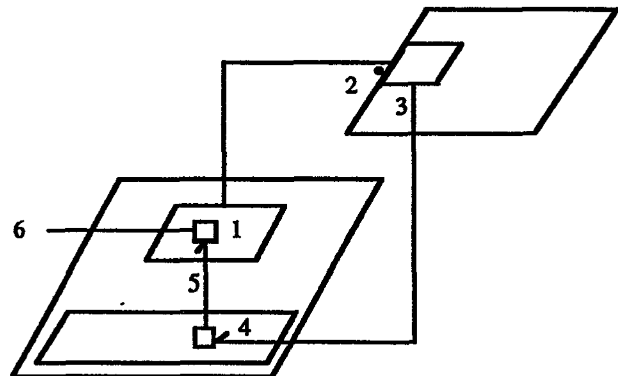


Figure 4.

In the figure above, both the triggering alternative(1) and the abortable part (4) are shown as parallelograms inside a task. First, the triggering statement of the triggering alternative is made. In this case, an entry call is placed to another task (2). While the triggering statement waits, the abortable part is running. So when the accept statement completes, a signal is generated (3) that interrupts the abortable part (4). The abortable part immediately transfers control to the statements following the triggering statement in the triggering alternative (5). This is where the handling really takes place - the triggering alternative may make more calls outside the task (6).

This representation is a fairly complex and not very accurate portrayal of what is happening. A normal occurrence, the completion of an accept statement, is represented here as an exception, as it is necessary to

suggest the interruption in the control of the abortable part. Yet this is deceptive, as the programmer cannot write a handler for an interrupt in the abortable part.

There is another issue with the above representation. The relation between the final part and the triggering alternatives is not made clear. There is no way to gather from the diagram that the two inner parallelograms are part of a single select statement. Nor is there a way to recognize the construction as being an asynchronous transfer of control as opposed to a normal exception propagation.

We propose the following convention:

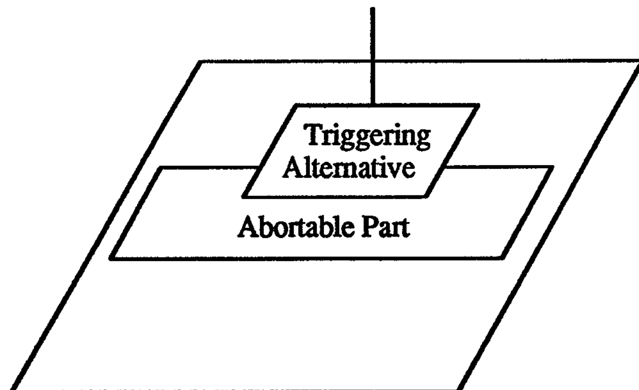


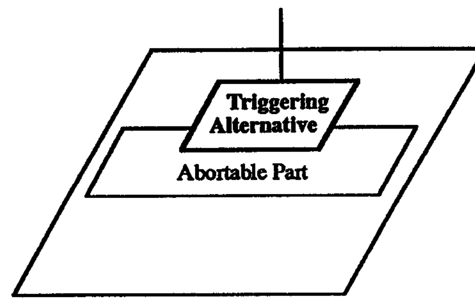
Figure 5.

Note that overlap is used to indicate a form of precedence. The triggering alternative can interrupt and abort the abortable part. Overlap was chosen as it:

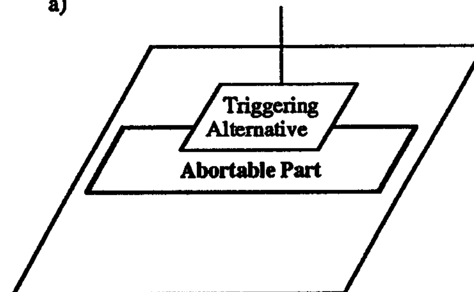
- suggests the triggering alternative as interrupting the abortable part
- establishes an association between the triggering alternative and the abortable part of the select statement.
- can be drawn easily.
- does not conflict with other Buhr conventions

The non-terminated vertical line is assumed to connect up to an entry call. In figure 6, the different stages of a task using an abortable part are shown.

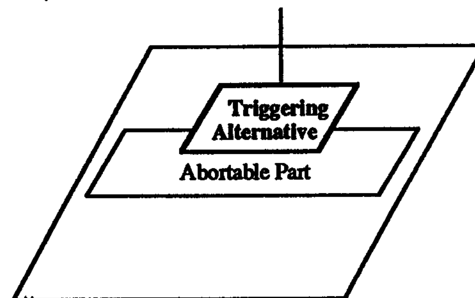
In Figure 6A, the triggering alternative places an entry call. In Figure 6B, the entry call has not returned, so the abortable part begins running. In 6C, the entry call has returned, and the abortable part is aborted. Control has gone to the triggering alternative.



a)



b)



c)

Figure 6.

A VISUAL EXAMPLE

Figure 7 follows the text of that it represents:

```
loop
 select
   TERMINAL.WAIT_FOR_INTERRUPT;
   PUT_LINE("Interrupted");
 in
   PUT_LINE("-> ");
   GET_LINE(COMMAND, LAST);
   PROCESS_COMMAND(COMMAND(1..LAST));
 end select;
end loop;
```

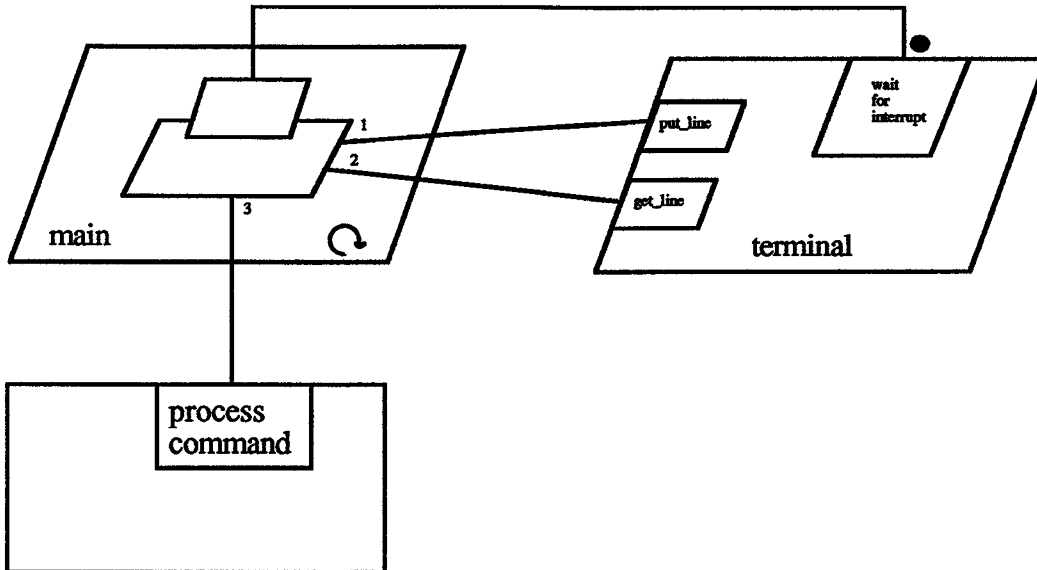


Figure 7.

The triggering alternative waits for an interrupt from the terminal. At the same time, the abortable part puts and gets the command line to and from the terminal, and processes the command received. The circular arrow indicates that the main task is persistent and will continue to loop.

The example given in the AARM treats command processing as a procedure. In many multi-tasking systems, shells spawn tasks or processes to perform the work of the command. In Ada 9X, the textual code and its visual representation might be:

```
loop
   select
      TERMINAL.WAIT_FOR_INTERRUPT;
      PUT_LINE("Interrupted");
   in
      declare
         SHELL: COMMAND_INTERPRETER;
      begin
         GET_LINE(COMMAND);
         SHELL.EXEC(COMMAND);
      end
   end select
end loop
```

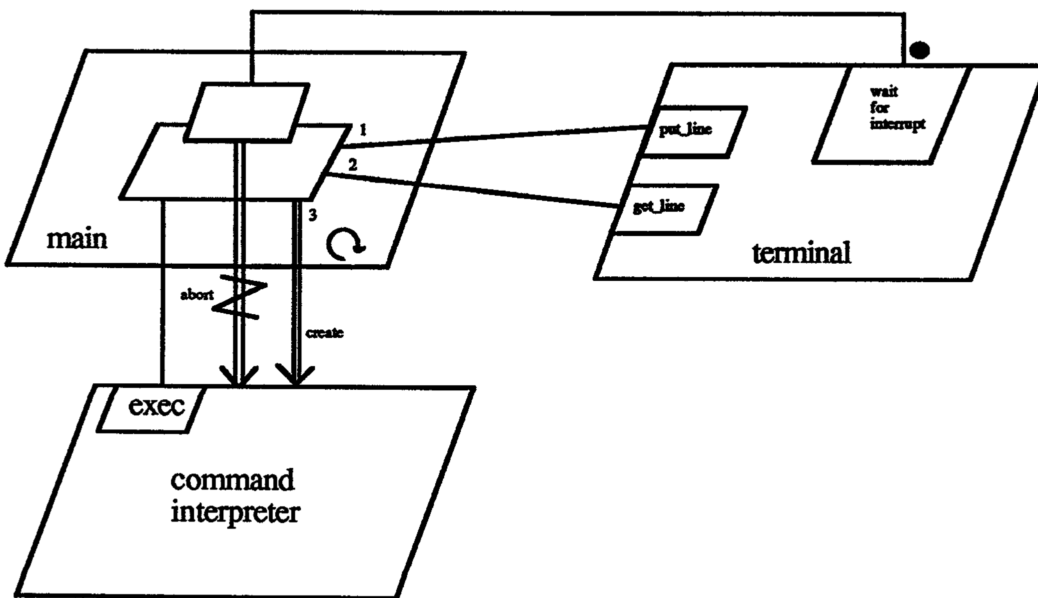


Figure 8.

In figure 8, note that a command task is created on every loop through the shell. In the event that the abortable part is interrupted, the command task will be aborted. This happens automatically as a result of the abort of the sequence of statements in the abortable part. We explicitly represent this as a removal arrow originating from the triggering alternative.

ASYNCHRONOUS TRANSFER ON EXPIRATION OF A DELAY

An abortable part can also be interrupted by a delay statement. This allows for code to be written that will be interrupted if it exceeds some time boundary, as in the example shown below (AARM 9.7.4.11;2.0) :

```
select
 delay 5.0
    PUT_LINE("Calculation doesn't converge");
 then abort
    HORRIBLY_COMPLICATED_RECURSIVE_FUNC(X,Y);
end select;
```

The visualization of this is analogous to the visualization shown before of an entry call statement. Instead of waiting for an accept statement to complete, the triggering statement is waiting for a timer to expire.

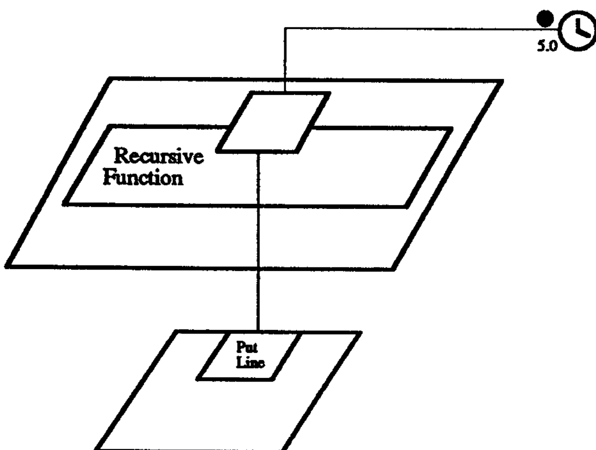


Figure 9.

The clock symbol used here is consistent with Buhr (1990) notation. When the timer expires, control will transfer to the triggering statement, which in this case will put a message out to the terminal.

In systems with many time-outs, it is easy to imagine a short-hand emerging, in which the triggering statement itself contains the clock symbol:

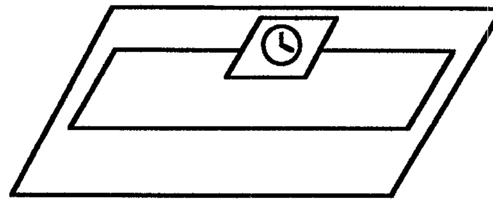


Figure 10.

The intended meaning is that of figure 9: when the timer runs out, if the abortable part is still running, abort the sequence of statements in the abortable part and transfer control to the triggering alternative.

CASCADING TRANSFERS

Given this new convention, it is worth considering how the visual representation of asynchronous transfer of control can contribute to the system design process. As Buhr (1990) points out, the visual can sometimes aid in capturing the essence of a problem that may be otherwise be represented as many lines of disparate code. One point of visual representation is to allow a concept to be represented in such a way that it can be taken in instantly. Another point specific to system design is to allow the multiple potential sequences of interaction to be walked through and discussed. Much of the utility of system design notations come from the work that a team can do with a shared convention and an interactive visual medium such as a white board. This kind of conversation often involves using the diagram as a map, and sequentially stepping through an event and its implications on the process it directly touches, along with the ripple effect on other dependent or synchronized tasks. In a sense, in early design stages, the diagrams are used as the basis for informal simulations of the working system. Buhr proposes the use of graphic user interfaces as an alternate way of doing these simulations. With the proper design tools, and with the design diagrams linked with underlying code, the hope is that time-based systems design can benefit in the same way as other design fields have benefited from CAD tools.

With this in mind, we consider a more complicated instance of asynchronous transfer of control. In the example of figure 8, a new task is created by an abortable part. It is certainly possible that a task that is created may itself include an asynchronous transfer of control. It may not be immediately obvious in a system with many lines of code how deep this cascading of asynchronous transfers of control goes. Yet the end effect is a set of tasks that are closely linked together - the completion of an accept statement on any of the blocked entry calls will affect all the tasks at a deeper level of the cascade.

An instance of this sort of situation can be represented in the following way:

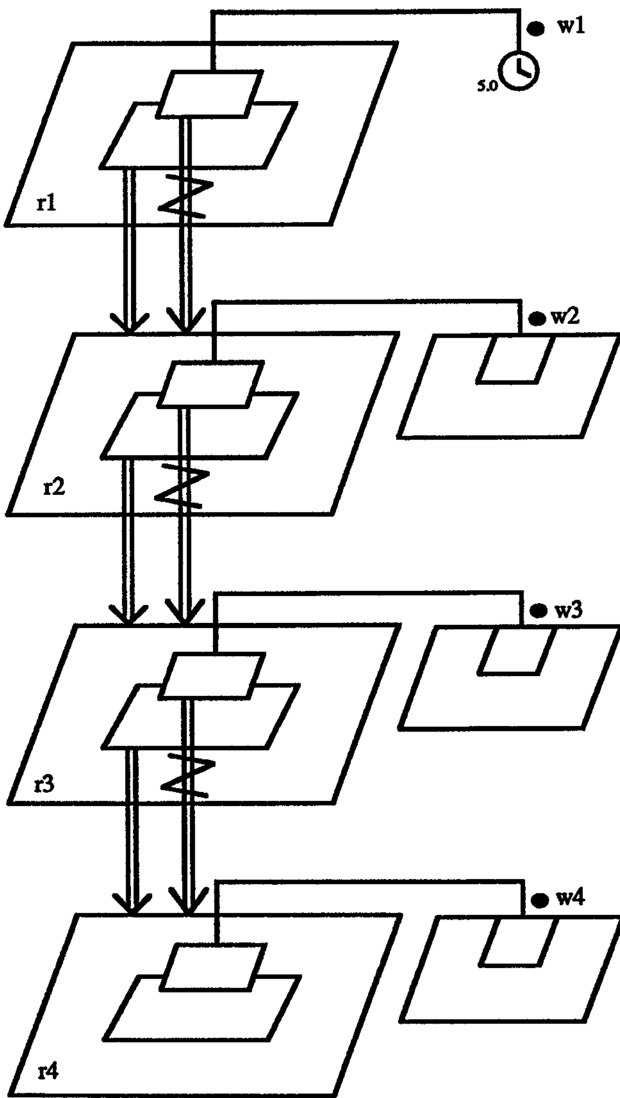


Figure 11.

In figure 11, the first robot (r1) creates r2, which in turn creates r3, which in turn creates r4. Each robot contains an asynchronous transfer of control with a corresponding waiting place (w1, w2, w3, w4). In the event that any triggering statement completes, the task dependent on the abortable part will be aborted. This is shown through a set of abort arrows from the triggering alternative to the created tasks.

In the event that a task is aborted, all tasks dependent on it are also aborted. If, say, the accept completes at w2, then not only will the abortable part of r2 be aborted, but r3 will also be aborted. The abort of r3 will in turn cause the abort of r4.

So, from this diagram it becomes clear that, depending on which triggering statement completes first, a chain reaction of aborts is possible. It is also clear, that no matter what else happens, r2, r3, and r4 will be aborted when the timer at w1 expires after 5 seconds.

For the system designer, the extension to the Buhr notation shown here makes it possible to visually trace the possible ramifications of a design that uses asynchronous transfer of control.

ACKNOWLEDGMENTS

I want to thank Edmond Schonberg, my thesis advisor, for his many contributions to this paper, especially his insights into how Ada 9X features work and how they should look.

REFERENCES

Buhr, R. J. A. 1990. *Practical Visual Techniques in System Design.* Englewood Cliffs, NJ: Prentice Hall.

Buhr, R. J. A. 1984. *System Design With Ada.* Englewood Cliffs, NJ: Prentice Hall.

Ada 9X Mapping/Revision Team. 1993. *Annotated Ada 9X Reference Manual.* Cambridge, Mass: Intermetrics Inc.