Concurrent Reusable Abstract Data Types

Jeffrey R. Carter
Senior Engineer, Software
Martin Marietta Astronautics Group
P. O. Box 179
Denver, CO 80201

Abstract

A commonly used form of concurrent abstract data types always leads to erroneous execution. A corrected version of this form, suggested in the article which pointed out this problem, is not considered to be acceptable software engineering practice by the author. An acceptable correct form is presented.

Introduction

In an article in the 1990 Jan/Feb issue of *Ada Letters*, Gonzalez pointed out that the form of concurrent abstract data types used by Booch [1] always leads to erroneous execution, because it relies on the parameter passing mechanism used to pass the state data [2]. This form of concurrent abstract data type is widely accepted, and components of this form are sold commercially as part of the Booch Components from Wizard Software and, in a slightly modified form, as part of the GRACE components from EVB Engineering. The abstract data type in this form is a record with a binary semaphore task component and one or more state-data components.

In an interesting coincidence, the same issue of *Ada Letters* contained an article in which I suggested that abstract state machines be used to to implement concurrent reusable components [3]. Abstract state machines have the advantage over abstract data types that the state data are never passed as parameters. Most concurrent applications, and especially most embedded systems, can be implemented using abstract state machines.

Gonzalez presents a corrected version of the form used by Booch. In this paper, I will address three concerns: Are concurrent abstract data types necessary? If so, is Gonzalez' corrected version of Booch's form acceptable? If not, what is an acceptable, correct form for concurrent abstract data types?

The Need for Concurrent Abstract Data Types

Obviously, I prefer the use of abstract state machines to implement concurrent reusable components. However, there are situations in which the number of instances of a component is not known until run time. When this is the case, the components must be created dynamically, which requires that the component export a type. Therefore, there is a need for concurrent abstract data types.

I have considered the creation of concurrent neural network implementations in Ada. Figure 1 shows the architecture of a typical neural network. The shaded background to some of the entities shows that the icon represents multiple instances of the entity. The array indices in parentheses represent the number of instances which are needed. The actual bounds of these indices are determined from input parameters at run time.

Neural networks are examples of massively parallel systems. There are a number of massively parallel systems which may need to dynamically create concurrent components.

To allow the nodes of the neural network to run as asynchronously as possible, they communicate by means of concurrent queues. To allow the implementation to be as flexible as possible, the number and organization of the nodes are determined at run time. Therefore, the nodes and their concurrent queues must be created dynamically. This cannot be done with abstract state machines, so concurrent types must be used. I therefore need a concurrent queue type for this application.

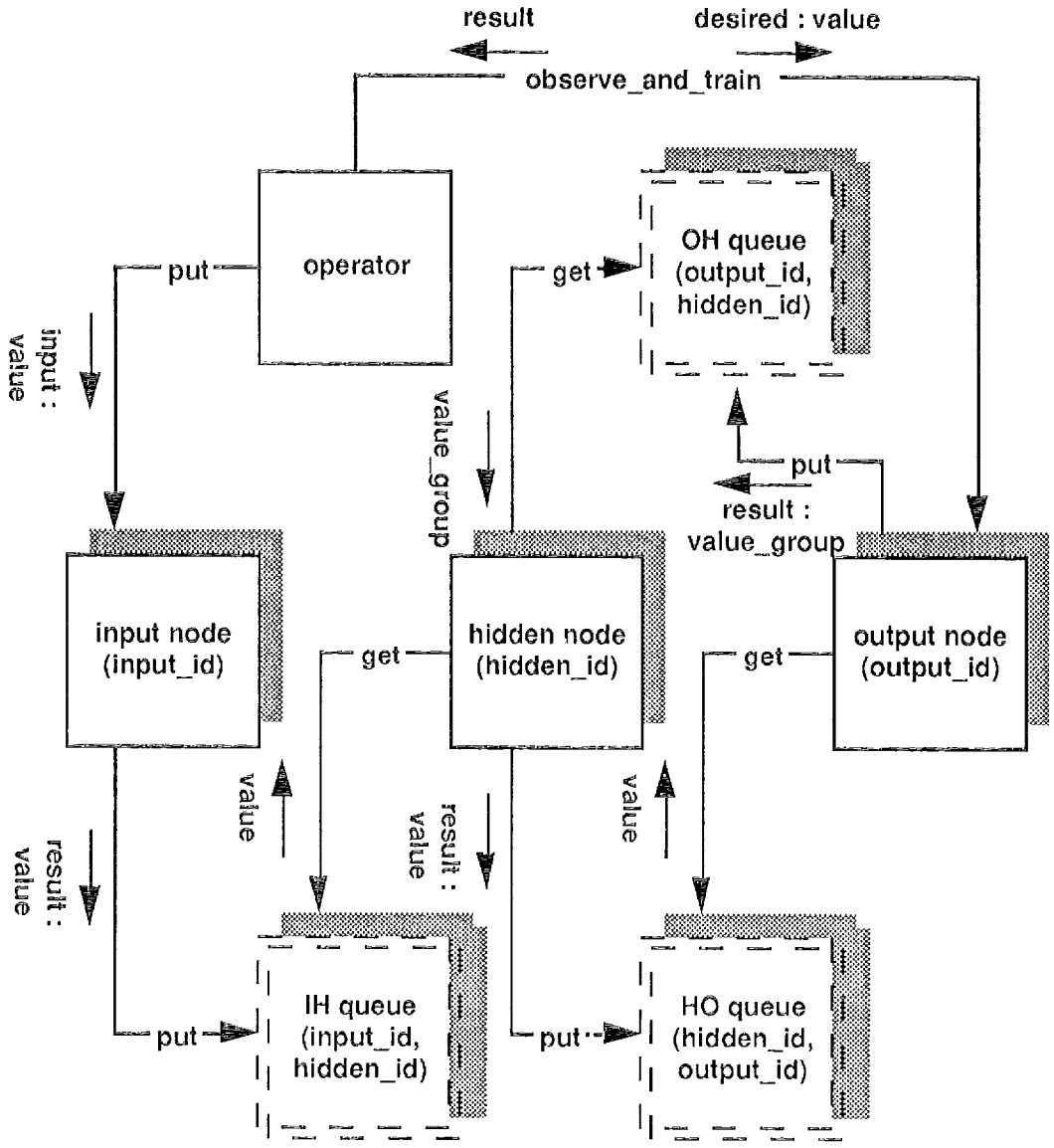

Figure 1. A typical neural network architecture. Shaded backgrounds indicate multiple instances of the entity; the number of instances is given by the array indices in parentheses.

The Corrected Form with a Binary Semaphore

Gonzalez presented a corrected version of Booch's form in his article. Instead of Booch's state-data components, Gonzalez uses a single component of an access type which designates a record containing the state data. This ensures that access to

the state data is always indirect, regardless of the parameter passing mechanism which is used. Therefore, Gonzalez' form does not lead to erroneous execution. However, Gonzalez continues to use a binary semaphore in his correct form. Gonzalez' article addressed the parameter passing dependence of Booch's form, not the form of tasking which it used. A discussion of Booch's form of tasking would not have fit in an article on Booch's parameter passing dependence.

The semaphore is a low-level device for mutual exclusion. Shared data protection is the type of mutual exclusion of interest for concurrent abstract data types such as queues. The high-level form of shared data protection in Ada places the shared data in the body of a task. Access to the shared data is made through the entries of the task. This is the basic form of a monitor.

What Booch did was to use a low-level construct (the semaphore) to implement a high-level construct (the monitor). I have seen no discussion of the suitability of doing this for concurrent systems. This is, perhaps, because many people are still not fully comfortable with concurrency.

In sequential systems, a low-level construct (the conditional GOTO) has sometimes been used to implement a high-level construct (the structured IF statement). Initially, languages only supplied the conditional GOTO, so it had to be used. Now, all major languages supply the structured IF statement. The software engineering community generally agrees that the use of conditional GOTOs to implement an IF statement is unacceptable software engineering practice if the IF statement is available. In general, we consider the use of a low-level construct to implement a high-level construct to be unacceptable software engineering practice if the high-level construct is available.

Should we apply the same standard to concurrent systems? That is, should use of a low-level shared-data protection construct be considered unacceptable software engineering practice if the high-level construct is available? If we accept this concept, we would say that the use of a binary semaphore is unacceptable software engineering practice when Ada's tasks and rendezvous are available.

I believe we should apply the same standard to concurrent systems that we apply to sequential systems. Therefore, even if it had been correct, I find Booch's form for concurrent abstract data types to be unacceptable software engineering practice. Although Gonzalez did the Ada community a great service by pointing out a major problem with Booch's form, his corrected version of the form inherited the binary semaphore from Booch's form, and so I find it to be similarly unacceptable.

The Ideal Form

What is an acceptable, correct form for a concurrent abstract data type? Since I prefer the use of abstract state machines to the use of abstract data types, I will attempt to derive a concurrent abstract data type from a concurrent abstract state machine. What I want ideally is a concurrent abstract data type with all the features of a concurrent abstract state machine, or a concurrent abstract state machine type. An object of such a type coordinates concurrent accesses to itself. An abstract state machine type is not an unreasonable request: The object types of object-oriented languages such as Smalltalk and Turbo Pascal 5.5 are forms of abstract state machine types. For a simplified queue, such a type in Ada might look like:

```
abstract_state_machine  type queue is
      procedure put (item : in element);
      function get return element;
end queue;
```

The body of such a type would have a structure similar to a package body, allowing explicit initialization and the ability to connect the subprograms of the type to a task to handle concurrency. The state data would be located in the task body and so would never be passed as a parameter. Therefore, such a form would be correct.

Given such a type, we could declare queues:

```
q : queue;
```

and operate on them:

```
q.put (item => item);
item := q.get;
```

The Exposed Task Form

Unfortunately, abstract state machine types are not available in Ada. I have chosen the form in which I presented them to remind us of something which is:

```
task type queue is
      entry put (item : in  element);
      entry get (item : out element);
end queue;
```

Such a type could be used in a manner very similar to the ideal abstract state machine type:

```
q : queue;
. . .
q.put (item => item);
q.get (item => item);
```

At first glance, this is exactly what I want. It has the correct form and explicitly handles concurrency. The state data of this form is stored in the task body. This form is correct and acceptable. In this article, we are discussing concurrent abstract data types for reusable components. One of the desirable characteristics of reusable components is robustness: The component should be able to handle any operation the user may perform on it, provide reasonable results when an incorrect operation is performed, and continue to provide reasonable results to future operations after an incorrect operation. For example, if the user attempts to get an item from an empty queue, the queue component should provide a reasonable result (for example, raising an appropriate exception), and provide reasonable results to future operations on the queue (for example, an immediately following attempt to put an item on the queue should result in a queue of one item). An exposed task can be aborted:

```
abort q;
```

Since the component cannot handle the abort to provide reasonable results to future operations on "q," an exposed task should not be used for a reusable component.

The Hidden Task Form

To prevent the abort statement from being used on an object of the type, we must hide the fact that the object is a task. This changes the structure from the abstract-state-machine-type form to the abstract-data-type form:

```
      type queue is limited private;
      procedure put (onto : in queue; item : in  element);
      procedure get (from : in queue; item : out element);
private
      task type queue is
            entry put (item : in  element);
            entry get (item : out element);
      end queue;
```

Since this is an abstract data type, it is used somewhat differently:

```
q : queue;
. . .
put (onto => q, item => item);
get (from => q, item => item);
```

Internally, though, it is still treated as an abstract state machine type:

```
procedure put (onto : in queue; item : in element) is
      -- null;
begin -- put
      onto.put (item => item);
end put;
```

This form is robust, correct, and acceptable. It is correct because the Ada Reference Manual [4] states that access to a task formal parameter ("onto") is access to the task actual parameter ("q"), regardless of the parameter passing mechanism [ARM 6.2(15) and 9.2(2)]. It also preserves the use of an abstract state machine type as much as possible without sacrificing robustness.

Comparison

The hidden task form allows the bounded form of components. Gonzalez states that his corrected version of Booch's form does not allow the bounded form.

The hidden task form presents some challenges to the implementor of the component. Booch's form allows the bound of a bounded form to be provided as a discriminant; this form does not. Implementing an "assign" procedure for the hidden task form is an interesting exercise. Luckily, most applications do not need to assign concurrent components, so this feature may be omitted without serious loss of generality.

From the point of view of time complexity, Booch's form requires two rendezvous, one to obtain the semaphore and one to release it, while the hidden task form requires only one. Although the time required for a rendezvous is very implementation-dependent, the hidden task form will have about half the tasking overhead of Booch's form.

Conclusion

An abstract state machine remains the preferred form for reusable components for concurrent use. For embedded, real-time systems, for which the number of instances of a component is likely to be fixed at compilation, this is no problem. However, there are a number of massively parallel systems for which it is desirable to dynamically create concurrent components. Neural networks are one such massively parallel system. For such systems, concurrent abstract data types are necessary.

By analogy to sequential systems, I consider Booch's form and Gonzalez' corrected version of Booch's form to be unacceptable software engineering practice, since both use a low-level construct, the semaphore, when a high-level construct, the monitor, is available. Beginning with the concept of an abstract state machine, we have derived an alternate form of concurrent abstract data type, the hidden task form. The hidden task form is robust, correct, and acceptable. It allows the bounded form of components and has less tasking overhead than Booch's form. However, it does not allow the bound of a bounded form to be specified as a discriminant, and implementing such operators as assignment is more difficult than for Booch's form. The hidden task form is the preferred form when concurrent components must be dynamically created.

References

1. Booch, G., *Software Components with Ada*, Benjamin Cummings, 1987
2. Gonzalez, D., "Multitasking Software Components," *Ada Letters*, 1990 Jan/Feb
3. Carter, J., "The Form of Reusable Ada Components for Concurrent Use," *Ada Letters*, 1990 Jan/Feb
4. Department of Defense, *ANSI/MIL-STD-1815A: Ada Programming Language*, Government Printing Office, 1983