Ada 9X Reusable Components

Jeffrey R. Carter
*Pragm*Ada Software Engineering
10545 East Spanish Peak
Littleton, CO 80127
(303) 972-4091

The inclusion of concurrency in Ada leads to multiple reusable components which implement the same abstraction. The proposed modifications of Ada 9X contain features which may eliminate the need for more than one version of an abstraction. An example of an Ada 9X component illustrates the use of these features.

Introduction

The inclusion of concurrency in the definition of the Ada language leads to the need for language constructs to protect shared data from simultaneous access by multiple threads of control. Because Ada also supports a form of modularity which makes reusable components practical, this protection must be considered when developing reusable components. A number of authors have discussed how this should be done [1, 2, 3, 4]. Their solutions have one thing in common: To ensure that the shared data is protected, all tasks attempting to access the data must rendezvous with another task which protects the shared data.

Although there are exceptions, a rendezvous is a relatively inefficient operation as implemented by most Ada compilers. If a program does not have multiple tasks, it may be detrimental to introduce tasks into the program by reusing a component which contains a task. This leads to the development of two (or more) components which implement the same abstraction, a sequential version for use by systems which do not have multiple tasks, and a concurrent version for use by systems which do. In practice, more than two versions are generally considered necessary. For example, both Booch [1] and I [2, 4] consider multiple concurrent versions of the same abstraction to be needed, although for different reasons.

Having multiple versions of the same abstraction is itself error prone. If a sequential system is modified to use multiple tasks, for example, the modifiers must remember to change reused components from the sequential version to a concurrent version. The concurrent version may have different names and calling conventions than the sequential version, which may mean that every use of the component in the system will need to be changed.

The proposed revision of Ada, called Ada 9X, contains a construct which allows shared data to be protected without the use of a task to protect the data. It also contains features to allow object-oriented programming in Ada with inheritance and polymorphism. Since the basic concept underlying these features has passed through two drafts of the *Mapping Specification* [5] unchanged, it is likely that something very like them will be part of the final Ada 9X standard. The combination of these features may significantly change the way reusable components will be implemented in Ada 9X. Specifically, the "protected record" construct, which provides for shared data protection without introducing a thread of control, may eliminate the need for multiple versions of the same abstraction. A protected record provides for the protection needed by applications with multiple tasks. Since a protected record object does not contain a thread of control, it may be suitable for use by purely sequential systems as well.

Protected Records

Ada 9X introduces the protected record construct to protect shared data from access by multiple tasks without introducing a thread of control. Multiple tasks may safely access a variable of a protected record type simultaneously. However, the type does not create a thread of control, so no rendezvous takes place. Protected record types are limited types, so neither assignment nor the "=" operator are defined for them. A protected record has a private part analogous to a package's private part. The structure of a protected record is:

```
protected [type] identifier [discriminant_part] is
   {protected_operation_declaration}
private
   {protected_operation_declaration}
record
   component_list
end [record] [identifier];

protected body identifier is
   {protected_operation_item}
end [identifier];
```

A "protected operation declaration" may be a subprogram declaration or an entry declaration. A "protected operation item" may be a subprogram declaration, a subprogram body, or a new construct called an "entry body." A user of a protected record may call the operations declared before the reserved word "**private**." The remainder of the structure, including the components, may not be accessed by a user. Procedure and entry operations of a protected record may modify the components of the structure and require exclusive access to the structure. Function operations may not modify the components and do not require exclusive access.

If an operation is an entry, the body of the operation (the entry body mentioned above) has a guard. If the guard is false when a caller calls the operation, the caller is suspended until the guard becomes true, unless the call is part of a selective entry call.

Inheritance

Ada 9X provides for a record type to be declared as a "tagged" type by including the keyword "tagged" in the type definition. (Private types may also be declared as tagged, in which case the full type declaration must be a tagged record type.) A new type may be derived from a tagged type while adding new components to the new type. All of the operations of the original type are defined for the new type, new operations may be defined for the new type, and operations of the original type may be redefined. This is the concept of inheritance used in object-oriented languages: The child type inherits all of the components and operations of the parent type, and may add new components and new operations, or redefine existing operations.

If T is a tagged type, T and all types derived from T are called the class of types rooted at T. The type T'CLASS is the universal type of the class of types rooted at T. An object of type T'CLASS may contain a value of any type in the class rooted at T. An access type which has T'CLASS as its designated type may contain any access value which designates a type in the class rooted at T. Operations which operate on type T'CLASS may be dispatching operations, which provide for run-time polymorphism.

An Ada 83 Concurrent Reusable Queue Component

The current standard for Ada is called Ada 83 to differentiate it from Ada 9X. For a reference point in discussing Ada 9X, consider a typical Ada 83 reusable component:

```
generic
   type element is limited private;
   -- things to put on queues

   with procedure assign
      (to : in out element; from : in element) is <>;
   -- have to be able to assign elements
package queue_concurrent_unbounded_blocking is
   type queue is limited private;

   procedure clear (q : in queue);
   -- empties "q"; anything on "q" is lost
   -- all queues are initially empty

   full : exception;
   -- raised by put if insufficient storage is available

   procedure put (onto : in queue; item : in element);
   -- adds "item" to the tail of onto
   -- raises "full" if dynamic storage for a copy of "item"
   -- is not available
   -- if "full" is raised, "onto" is unchanged

   function get (from : queue) return element;
   -- removes the next element from the head of "from" and
   -- returns it
   -- if "from" is empty, the caller is blocked until "from"
   -- becomes un-empty

   function empty (q : queue) return boolean;
   -- returns true if "q" is empty;
   -- returns false otherwise

   -- any other operations of a queue
private -- queue_concurrent_unbounded_blocking
   task type queue is
      entry clear;
      entry put (item : in     element);
      entry get (item : in out element);
      entry empty (status : out boolean);
      -- entries for any other operations
   end queue;
   pragma inline (clear);
   pragma inline (put);
   pragma inline (get);
   pragma inline (empty);
   -- inlines for any other operations
end queue_concurrent_unbounded_blocking;
```

An Ada 9X Reusable Queue Component

To understand the differences between the way reusable components are currently implemented in Ada 83 and the way they may be implemented in Ada 9X, consider the following Ada 9X package specification for a similar queue component:

```
package queue_unbounded_blocking is
   type queue_element  is tagged limited private;
   type access_element is access queue_element'class;

   null_element : exception;
   -- raised if the item passed to queue.put
   -- has the value null

   protected type queue is
      procedure put (item : in access_element);
      -- adds the item designated by "item" to
      -- the tail of the queue
      -- raises "null_element" if "item" = null
      -- the queue is unchanged if "null_element"
      -- is raised

      entry get (item : out access_element);
      -- removes the next element from the head of
      -- the queue and returns it in "item"
      -- if the queue is empty, the caller is blocked
      -- until an element is added to the queue

      function empty return boolean;
      -- returns true if the queue is empty;
      -- returns false otherwise

      -- any other operations of a queue
   private record -- queue
      head : access_element := null;
      tail : access_element := null;
   end queue;
private -- queue_unbounded_blocking
   type queue_element is tagged record
      next : access_element := null;
   end record;
end queue_unbounded_blocking;
```

We can see a number of basic differences from the similar component in Ada 83. For the unbounded structure in Ada 83, the `put` operation performs the storage allocation to obtain the access value which is linked into the list of values on the queue. This means that `put` may raise `storage_error`, which it handles, in which case it propagates `full`, an exception defined in the package specification, instead. Here, the caller performs the storage allocation and passes the access value to `put`, so the caller must deal with `storage_error`.

In Ada 83, the package is generic so that the type of the elements of the queue may be imported. In Ada 9X, the type `queue_element` is a tagged type, which means that the type may be extended by inheritance. If we want to put integers on a queue, we may use this package by extending `queue_element` with an integer value:

```
with queue_unbounded_blocking;
procedure demo is
   type integer_element is
   new queue_unbounded_blocking.queue_element
```

```
      with record
         value : integer := 0;
      end record;

      value : queue_unbounded_blocking.access_element;
      queue : queue_unbounded_blocking.queue;
   begin -- demo
      value := new integer_element'(value => 10);
      queue.put (item => value);
      value := new integer_element'(value => 42);
      queue.put (item => value);
      queue.get (item => value); -- value.value = 10
   end demo;
```

Type integer_element extends queue_element to add a component of type integer. Since type access_element designates queue_element'class, an object of type access_element may contain any access value which designates any type derived from queue_element, such as the value returned from the allocator **new** integer_element. This is why the package does not need to be generic. In addition, this allows the queue to contain values of different types, making it what is called a "heterogeneous data structure." The reader is referred to the *Mapping Specification* [5] and the *Mapping Rationale* [6] for the details of heterogeneous data structures.

The package body provides an example of an entry body with its guard:

```
   package body queue_unbounded_blocking is
      protected body queue is
         procedure put (item : in access_element) is
            -- null;
         begin -- put
            if item = null then
               raise null_item;
            end if;
            if tail = null then -- queue is empty
               head := item;
            else
               tail.next := item;
            end if;
            tail := item;
            tail.next := null; -- avoid spurious links
         end put;

         entry get (item : out access_element)
         when head /= null is
            -- "when head /= null" is the guard
         begin -- get
            item := head;
            -- we know head /= null from the guard
            head := head.next;
            if head = null then -- queue is now empty
               tail := null;
            end if;
            item.next := null;
            -- this avoids returning a link into the queue
         end get;
```

```
        function empty return boolean is
            -- null;
        begin -- empty
            return head = null;
        end empty;
    end queue;
end queue_unbounded_blocking;
```

The guard on `get`, which is analogous to a guard in a select statement, prevents the operation from being executed when the guard is false. This allows us to use the value of head without worrying if it might be `null`.

The effects of these Ada 9X features on reusable components may be profound. The Ada 9X queue component presented may be usable by all applications which need an unbounded queue, regardless of their tasking characteristics. This eliminates the likelihood of errors when a system's tasking characteristics change. Companies which develop and sell reusable components, such as *Pragm*Ada Software Engineering, may be able to offer fewer components, developed at less expense. Those who make use of reusable components may not need to acquire as many components, reducing system development costs even further.

Conclusion

A developer will use a different version of an Ada 83 reusable component to produce a sequential system than to produce a concurrent system. This can lead to errors if the system's tasking characteristics change. Ada 9X may have features which will allow one component to be useful for all systems. The examples of a queue component in Ada 83 and using the proposed Ada 9X features shows how different an Ada 9X component is from its Ada 83 counterpart. These differences may result in Ada 9X reusable components being less costly to develop, to acquire, and to use than their Ada 83 counterparts.

References

1. Booch, G., *Software Components with Ada*, Benjamin Cummings, 1987

2. Carter, J., "The Form of Reusable Ada Components for Concurrent Use," *Ada Letters*, 1990 Jan/Feb

3. Gonzalez, D., "Multitasking Software Components,"*Ada Letters*, 1990 Jan/Feb

4. Carter, J., "Concurrent Reusable Abstract Data Types," *Ada Letters*, 1991 Jan/Feb

5. Ada 9X Mapping/Revision Team, *Ada 9X Mapping Document, Volume II: Mapping Specification* (draft, 1991 Aug), Intermetrics, Inc., 1991

6. Ada 9X Mapping/Revision Team, *Ada 9X Mapping Document, Volume I: Mapping Rationale* (draft, 1991 Aug), Intermetrics, Inc., 1991