Jeffrey R. Carter
PragmAda Software Engineering

Bo I. Sandén
Colorado Technical University

Practical Use of Ada 95's Concurrency Features

The authors compare the new concurrency features in Ada 95 to the concurrency syntax in Ada 83. An example of resource contention in a flexible manufacturing environment illustrates Ada 95's improvements.

When Ada was introduced in 1983, it was one of very few languages with built-in support for concurrency. Unfortunately, because of several shortcomings, its concurrency features were never widely accepted for embedded real-time systems. Exclusive access to shared resources requires additional tasks, which add overhead, and because the tasks provide interrupt handling, they increase interrupt latency. Furthermore, Ada 83 has problems handling asynchronous transfer of control.

The effort to revise Ada 83 and address these shortcomings introduced several requirements related to real-time systems and concurrency.[1] In this article, we examine these improvements and their effect on our Flexible Manufacturing System. In the FMS example, the change from Ada 83 to Ada 95 leads to a shorter and simpler program and reduces tasking overhead. This is primarily because Ada 95 adds a more traditional and pragmatic way of implementing synchronization to the rendezvous mechanism of Ada 83. In the FMS, all the rendezvous are replaced by operations on protected units—a monitorlike construct introduced in Ada 95. With this addition, Ada has become an elegant and efficient tool for concurrent software design.

The Flexible Manufacturing System

Some of the improved Ada 95 features we will discuss might seem to cover rare cases, but the FMS is an example where they are needed in practice. The FMS—which we briefly outline here—represents the problem of resource allocation and concurrency in an automated factory environment.[2,3]

Figure 1 shows the floor layout of a small FMS. *Automated guided vehicles* move parts between workstations, and an *automated storage and retrieval system* stores raw material and finished parts. The ASRS also stores parts that must be removed from the factory floor between job steps. Each job has its own dedicated bin in the ASRS, and an automated forklift truck carries parts between their bins and the storage stands, where they are accessible to the AGVs.

The system can include several of each kind of automatic workstations (mill, lathe, or drill), each one including an input stand, a tool, an output stand, and a robot that moves parts from the input stand to the tool to the output stand. The class diagram in Figure 2 shows the relationship between these objects.

The design of the FMS control software is based on an object-oriented analysis of the problem domain. Each job pursues its completion while calling on the services of workstations, AGVs, and the forklift. The software is patterned closely on this model, with `Job` tasks using the services of a `Workstations` package, an `AGV_Mgr` package,

1092-3063/98/$10.00 © 1998 IEEE

Table 1. Comparing Ada 83 and Ada 95 concurrency features.

Ada 83	Ada 95
Requires guardian tasks to protect shared resources.	Introduces protected units—passive syntactic units providing mutual exclusion.
Treats interrupts as calls to task entry points.	Uses protected procedures as interrupt handlers.
Provides only fixed task priorities.	Offers changeable priorities. The active priority can differ from the task's base priority—for example, when it executes a protected operation with a ceiling priority.
Implements a FIFO (first-in, first-out) queuing policy queued on an entry.	Lets the compiler vendor implement additional policies. Priority-based for tasks queuing must be provided in addition to FIFO.
Has problems handling asynchronous transfer of control.	Lets the asynchronous `select` statement time out a computation. It also lets a task issue two entry calls—and cancel one when the other completes.
Requires instances of task types to be initialized through rendezvous.	Initializes instances of task types and protected types with discriminants.
Allows a task to suspend itself for a given period of time.	In addition, allows a task to suspend itself until a given time.

and an `ASRS` package. The `AGV_Mgr` package includes objects modeled on the physical vehicles, and the `Workstations` package contains objects representing workstation kinds and individual workstations. This allows a job to queue up for a workstation kind, then communicate directly with the workstation to which it is assigned.

The FMS is supervised from a terminal, and once a raw part is present in a storage bin, the supervisor creates a job by associating the part with a process plan. Once created, the job progresses according to the algorithm in Figure 3.

The FMS represents a class of problems where a programming language's concurrency features can directly capture the problem domain's concurrency. The primary issue in the FMS is to guarantee liveness. Each job requires simultaneous exclusive access to multiple resources, which creates opportunities for deadlock, but a static locking order can prevent this.[2,3]

Ada 95 concurrency features

Ada 83 and Ada 95 have several significant differences.[4–6] Before we discuss how this directly affects FMS implementation, we first explain how these changes—summarized in Table 1—translate into increased efficiency for Ada 95. Specifically, Ada 95 simplifies concurrent implementation with protected units, interrupt handlers, dynamic task priorities, priority entry queuing, and asynchronous select statements.

Information hiding and resource sharing

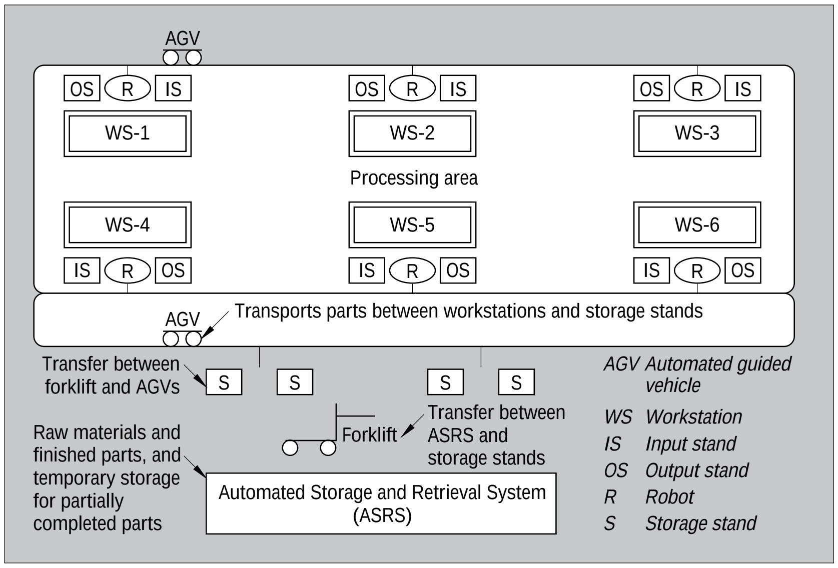
Both Ada 83 and 95 encourage an object-based design based on information-hiding packages and abstract data types.[3] In a concurrent environment, tasks can share objects, and a task might need exclusive access to a particular shared resource. Standard Ada 83 practice was to provide exclusive access by requiring tasks that use a resource to first rendezvous with a *guardian task*. It was also standard practice, for safety, to hide these guardian tasks in package bodies, because a visible task can be aborted.

Ada 95 introduces the *protected unit* to replace guardian tasks.[4,6] A protected unit implements a kind of monitor without introducing additional threads of control. It is safe to make a protected unit visible, so the designer can use the natural syntax of protected units and tasks, which simplifies a system's software architecture and design.[7]

Protected units combine the syntax of tasks and packages, and, as with tasks, we can declare a single protected object or a protected type. Protected units can contain functions, procedures, and entries, each providing a different kind of access to the unit. A protected unit has an *execution resource* associated with it—which is similar to a lock—and protected functions provide read-only access to the unit. Calls to a protected function can proceed concurrently with other function calls—but not with procedure or entry calls—and they can't modify any of the unit's components.

Protected procedures and entries provide exclusive read-write access to the unit; entries, unlike procedures, provide for caller queuing. Calls to a procedure must first acquire the unit's execution resource before executing, ensuring exclusive access. Entry calls first acquire the unit's execution resource, then evaluate the entry body's *barrier*. If the barrier is true, the entry body executes. Otherwise, the calling task queues on the entry and releases the unit's execution resource.

As an example of a protected unit, first consider a guardian task from the Ada 83


AGV
OS
R
IS
WS-1
WS-2
WS-3
Processing area
WS-4
WS-5
WS-6
AGV
Transports parts between workstations and storage stands
Transfer between forklift and AGVs
S
Raw materials and finished parts, and temporary storage for partially completed parts
Forklift
Transfer between ASRS and storage stands
Automated Storage and Retrieval System (ASRS)
AGV Automated guided vehicle
WS Workstation
IS Input stand
OS Output stand
R Robot
S Storage stand
Figure 1. The layout of a flexible manufacturing system.

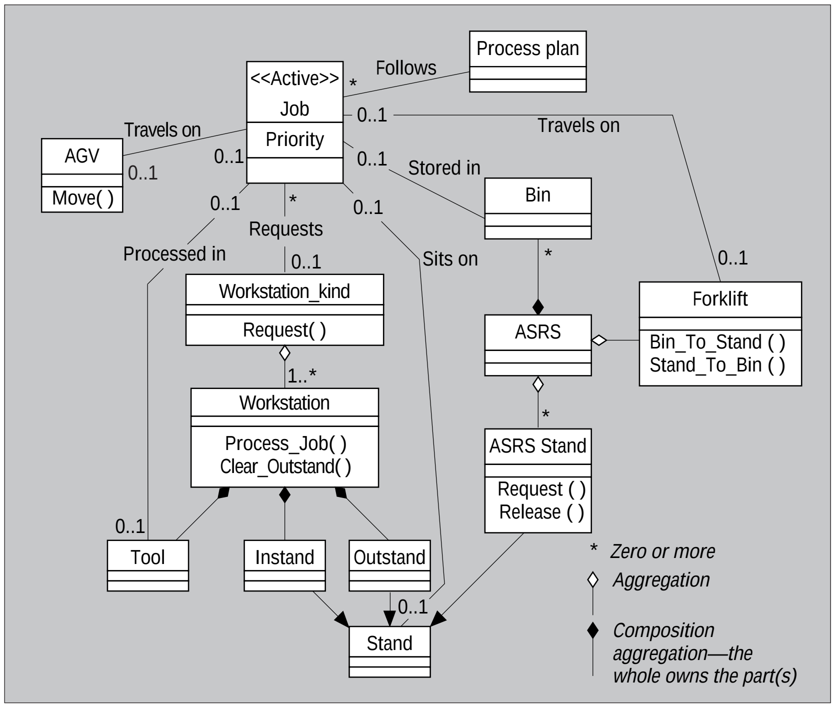


Figure 2. Class diagram of the FMS.

solution to the FMS. `Stand_Control` maintains a queue of available storage stands and provides one when a `Job` task requests it (see Figure 4a).

Figure 4b shows the equivalent protected unit. `Request` is an entry, so tasks attempting to obtain a stand can be queued until one becomes available. (The barrier is between `when` and `is`.) In the guardian-task version, `Release` is also an entry, because that is the only way to communicate with a task. However, calls to `Release` require no queuing, so `Release` becomes a protected procedure in Ada 95.

As with guardian tasks in Ada 83, Ada 95 encourages the use of special-purpose protected units over general-purpose semaphores. We could use a protected type to define a semaphore; however, there are a number of arguments for using special-purpose protected units instead of semaphores—including safety, portability, and runtime efficiency.[8]

The protected unit has a limitation when used to control resources in the problem domain, as in the FMS problem. Operations—such as delay and entry calls—that can block the executing task are prohibited inside protected operations. The `delay` statement suspends the executing task for (or until) a specified time, while an entry call blocks until the call completes. Thus, a protected unit can encapsulate a shared data structure and provide operations that manipulate it, but cannot provide mutual exclusion for external resources that are held for seconds or minutes. For example, `Move_Forklift_To_Stand` cannot be a protected operation, because it involves waiting for the physical forklift to complete its movement. Instead, we are reduced to using a protected unit as a semaphore with operations such as `Request` and `Release`.

```
Jobstep := 1
Loop
   Request workstation of appropriate kind for Jobstep
   (Workstation-1 has been acquired, possibly after queuing)
   Transport part to a storage stand
   Loop
     (Part is on a storage stand or a workstation output stand)
     Transport part to input stand
     Process the part in workstation
     (At this point, the part is on output stand)
     Jobstep := Jobstep + 1
     Exit loop if job is done
     Request workstation of appropriate kind for Jobstep
     Wait for either:
           1. An appropriate workstation is acquired
           2. Another part needs output stand
              In this case, dequeue job; exit loop
   End loop
   Transport part to storage stand
   Transport part to ASRS bin
   Exit loop if job is done
End loop
```

Figure 3. Job progression in the FMS.

Interrupt handling

When an interrupt occurs at a certain level, a CPU transfers control to the interrupt handler for that level. A direct way to capture this in a programming language is to let the programmer specify subprograms as interrupt handlers. Ada 83 takes an indirect approach by attaching interrupts to task entry points. Thus, a task can wait for one of several possible interrupts, A, B, or C, by executing a select statement:

```
select
  accept A;
or
  accept B;
or
  accept C;
end select;
```

In practice, programmers often had to bypass this approach because of the overhead. They would instead use subprograms for interrupt handlers by placing their addresses in the interrupt vector. Ada 95 takes a more practical approach by allowing protected procedures as interrupt handlers:

```
  task Stand_Control is
    entry Request (Stand : out ASRS_Stand_Id); -- Reserve a stand
    entry Release (Stand : in ASRS_Stand_Id);  -- Release a reserved stand
  end Stand_Control;

  task body Stand_Control is
    Queue : ...; -- A queue of Stand IDs
  begin -- Stand_Control
    Forever : loop
          select when not Empty (Queue) =>
            accept Request (Stand : out ASRS_Stand_Id) do
               Get (From => Queue, Item => Stand);
            end Request;
          or
            accept Release (Stand : in ASRS_Stand_Id) do
               Put (Item => Stand, Onto => Queue);
            end Release;
          or
            terminate;
          end select;
    end loop Forever;
  end Stand_Control;
```

(a)

```
  protected Stand_Control is
    entry Request (Stand : out ASRS_Stand_Id);    -- Reserve a stand
    procedure Release (Stand : in ASRS_Stand_Id); -- Release a reserved stand
  private -- Stand_Control
    Queue : ...; -- A queue of stand IDs
  end Stand_Control;

  protected body Stand_Control is
    entry Request (Stand : out ASRS_Stand_Id) when not Empty (Queue) is
    begin -- Request
      Get (From => Queue, Item => Stand);
    end Request;

    procedure Release (Stand : in ASRS_Stand_Id) is
    begin -- Release
      Put (Item => Stand, Onto => Queue);
    end Release;
  end Stand_Control;
```

(b)

Figure 4. (a) A guardian task from the Ada 83 solution to the FMS; (b) the corresponding protected unit in Ada 95.

```
with Ada.Interrupts;
use Ada;
...
Id : constant Interrupts.
     Interrupt_Id := ...;
...
protected Interrupt_Handler
  is
  procedure Handler;
  pragma Attach_Handler
     (Handler, Id);

  ...
end Interrupt_Handler;
```

Here, the pragma attaches the procedure `Handler` to the interrupt `Id`.

Priorities and queuing policies

Ada 83 provides only for static task priorities, established when the tasks are compiled and fixed during program execution. Ada 95 provides for dynamic task priorities, which may change at runtime. To change a task's priority, you refer to the task by means of a system-defined task identifier. It also provides the pragma `Queuing_Policy`, which specifies the order of calls in entry queues.

One possible queuing policy is priority queuing, which orders calls in entry queues based on their priority. If a caller's priority changes, its position in the queue also changes to reflect its new priority. At least 30 priority levels are available.

The FMS example is largely based on priority queuing of jobs vying for access to resources. Data structures for priority job queuing for workstations, and dynamic modification of job priorities, make up a significant part of the Ada 83 solution to the FMS problem (which we further discuss later). Ada 95's priority queuing lets us eliminate these data structures completely.

Ada 95 also provides other priority features, such as ceiling-priority locking.[6]

This allows every task that accesses a protected unit to execute at the ceiling priority defined for that unit, minimizing priority inversion. Priority inversion occurs when a higher-priority task, H, that needs exclusive access to a shared resource must wait for a lower-priority task, L, that currently holds the resource. If a task, M, with intermediate priority preempts L, it exacerbates the problem. If a ceiling priority is defined for the shared resource, then L will execute at that higher priority, preventing M from running. (Because processor scheduling is not a primary concern in the FMS, we don't further discuss ceiling-priority locking and other scheduling features in this article).

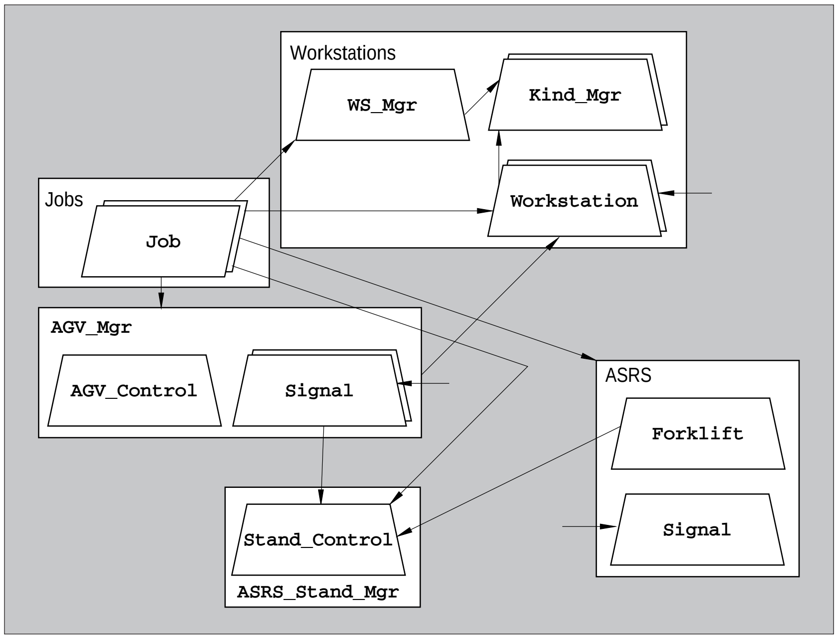


Figure 5. The FMS software design. Rectangles represent packages, parallelograms represent tasks, and trapezoids represent protected units.

Requeing

The issue of preference control arises in a protected unit with multiple entries.[5,9] The Ada policy is to serve the entries with open barriers in arbitrary order and to serve the callers queued for each entry according to the queuing policy in force. This arrangement is insufficient for certain resource-allocation servers. For example, the callers on a memory-resource server might have a parameter indicating the amount of memory requested. This value is unavailable to the server as long as the caller is queued.

Ada 95 provides the `requeue` statement as a blanket solution to this problem. A server can accept a call and then put the caller back on a queue if necessary. In the FMS, we use requeuing to control the jobs' access to workstations. All calls are first made to the entry `WS_Mgr.Request` with the workstation kind as a parameter. `WS_Mgr` requeues the callers on separate entries per workstation kind. With this arrangement, the protected objects representing the different workstation queues need not be made public. This simplifies the `Job` task logic.

Asynchronous transfer of control

Ada 83 cannot easily solve problems that require transferring control out of a sequence of statements when an event occurs.[1,5,6] A common example is a calculation that repeatedly improves an approximate result until it converges to a value with a certain precision. If the calculation does not converge within some time limit, a real-time system might need to terminate the calculation and use the best approximation, even if it hasn't achieved the desired precision. Ada 95 provides an asynchronous `select` statement for such cases:

```
select
  delay until Next_Reading;
then abort
  Some_Computation (...);
end select;
```

This works as follows: `Some_Computation` starts. If it does not end by the time `Next_Reading`, it aborts. `Next_Reading` might be the time when new sensor data arrives, making the current computation obsolete.

The FMS example contains another interesting example of asynchronous transfer of control, which has this general syntax:

```
select
  trigger
  [sequence of statements]
then abort
  abortable sequence of
    statements
end select;
```

The trigger is a `delay` statement or an entry call. If the trigger is a delay that has expired, then the optional sequence of statements executes and control passes to the statement following the `select` statement. If the trigger is an entry call or a delay that has not expired, the abortable sequence of statements begins executing. If the triggering event completes before the abortable sequence of statements, the abortable sequence of statements aborts and the optional sequence of statements execute (as described above). If the abortable sequence completes first, the trigger is cancelled and the next statement after the `select` statement executes.

The FMS control-software design

Although we implemented the FMS software independently of the efforts to revise Ada, its implementation in Ada 83 is something of a catalog of problems that Ada 95 elegantly solves. We structured the FMS software around Job tasks that track each job's progress according to its process plan. The main software concern is allocating shared resources. As soon as the FMS creates a job, it queues the job for the appropriate workstation kind for the job's first operation. After securing a workstation, the job obtains a storage stand and the forklift that will take it to that stand. Once on the storage stand, it releases the forklift and requests an AGV, which is released when the job arrives at the workstation input stand. Similar sequences of resource acquisitions are necessary when the job moves from one workstation to the next or from a workstation output stand to storage.

Figure 5 shows the overall FMS software design. Trapezoids represent protected units. The rest of the notation is a simplified version of Buhr's Structure Graph notation[10]—rectangles represent packages and parallelograms represent tasks; arrows represent subprogram or entry calls, and each arrow's direction goes from caller to callee.

In this design, `Job` tasks drive the processing and contend for access to workstation, AGV, and forklift objects. Other designs are also possible; in one workable alternative, workstation tasks contend first for job objects in need of machining, then for access to AGV and forklift objects to transport the jobs.[2]

The forklift

The forklift belongs to the storage facility and is handled in the package ASRS. In the Ada 83 solution, a guardian task, `Fork`, with the entries `Stand_To_Bin` and `Bin_To_Stand`, models the forklift. `Fork` serves three purposes:

- contain the steps necessary for forklift movement from a bin to a stand and from a stand to a bin;
- ensure exclusive access to the forklift, one job at a time; and
- handle interrupts.

```
package ASRS is
  ...
  procedure Bin_To_Stand
    (Stand : out ASRS_Stand_Mgr.ASRS_Stand_Id; Bin : in Bin_Id);
  ...
end ASRS;

package body ASRS is
  task Fork is
    entry Bin_To_Stand
      (Stand : in ASRS_Stand_Mgr.ASRS_Stand_Id; Bin : in Bin_Id);
    ...
    entry Fork_Done;
    for Fork_Done use at ...; -- Fork_Done is an interrupt handler
  end Fork;

  procedure Bin_To_Stand
    (Stand : out ASRS_Stand_Mgr.ASRS_Stand_Id; Bin : in Bin_Id) is
     S :ASRS_Stand_Mgr.ASRS_Stand_Id;
  begin -- Bin_To_Stand
    ASRS_Stand_Mgr.Stand_Control.Request (S);
    Fork.Bin_To_Stand (Stand => S, Bin => Bin);
    Stand := S;
  end Bin_To_Stand;

  task body Fork is
  begin -- Fork
    loop
      select
        accept Bin_To_Stand
          (Stand : in ASRS_Stand_Mgr.ASRS_Stand_Id; Bin : in Bin_Id)
        do
          To_Bin (Bin => Bin); -- Command physical forklift to move to Bin
          accept Fork_Done; -- Wait for physical forklift to finish
          ...
        end Bin_To_Stand;
      ...
      or
        terminate;
      end select;
    end loop;
  end Fork;
end ASRS;
```

Figure 6. Package ASRS (Ada 83).

Figure 6 shows the `Bin_To_Stand` entry (`Stand_To_Bin` is analogous).

The Ada 95 solution places the processing from the guardian task `Fork` in the two procedures, `Bin_To_Stand` and `Stand_To_Bin`, and the protected units `Forklift` and `Signal` replace the guardian task, where `Forklift` is a semaphore and `Signal` handles the completion interrupts from the physical forklift. Figure 7 presents `Signal`.

The steps necessary for forklift movement appear in the procedures `Bin_To_Stand` and `Stand_To_Bin` that the `Job` tasks execute. `Forklift` ensures that only one job at a time uses the physical forklift. `Signal` handles the completion interrupts from the physical forklift and allows the job task executing the procedure to wait for the interrupt to occur. In one sense, the Ada 95 version is inferior to the Ada 83 version, because we replaced a special-purpose guardian task with a semaphore—a low-level and unsafe construct. The semantics prevent a protected operation from waiting for an interrupt, forcing us to this solution. However, the efficiency of eliminating an unnecessary task makes this worthwhile.

Package AGV_Mgr

AGV handling is in a package, `AGV_Mgr`, with one procedure, `Move`, which a job in need of an AGV can call.

In Ada 83, each AGV is similar to the forklift and has its own instance of a guardian task type, `AGV_Task`. An additional guardian task, `AGV_Control`, represents the pool of AGVs. `AGV_Control` has two entries: `Get_Part`, which idle `AGV_Task` instances call, and `Request`, which job tasks in need of transportation can call. As Figure 8a shows, `AGV_Control` manages the two entry queues of job tasks and AGV tasks in a loop.

An additional complication is that each `AGV_Task` instance needs to handle the completion interrupts from one physical AGV. Because Ada 83 does not allow the entries of individual instances of a task type to be associated with different interrupt locations, an additional task, `AGV_Msgs`, communicates with all the physical AGVs and relays each completion entry call to the appropriate `AGV_Task` instance.

In Ada 95, we eliminated all the guardian tasks from the body of `AGV_Mgr`. A protected type, `Signaler`, with discriminants allows an interrupt handler for each AGV (see Figure 8b).

Each AGV has an instance of `Signaler`. As in the Ada 83 solution, `AGV_Control` represents the set of available AGVs. In the Ada 95 solution, it is a protected type with an entry `Request` and a procedure `Release`, which are both called by `Job` tasks. `AGV_Control` controls the set of signaler instances. A `Job` task in need of an AGV executes the procedure `AGV_Mgr.Move`. The procedure obtains an available AGV and a pointer, `S`, to its signaler instance from `AGV_Control`. It then issues successive commands to the AGV and waits for a completion interrupt from the AGV in `S.Wait`. Ultimately, it releases the AGV by calling `AGV_Control.Release`.

In Ada 83, we need an extra task to handle the completion interrupts from the physical AGVs, because multiple instances of a task type cannot handle different interrupts. In Ada 95, discriminants allow objects to be parameterized when declared. This replaces Ada 83's inefficient and serial initialization rendezvous and allows instances to each handle different interrupts, as in the case of `Signaler`.

```
protected Signal is
  entry Wait;
private -- Signal
  procedure Handle;  -- Handle is an interrupt handler
  pragma Attach_Handler (Handle, ...);

  Occurred : Boolean := False;
end Signal;

protected body Signal is
  entry Wait when Occurred is
  begin -- Wait
    Occurred := False;
  end Wait;

  procedure Handle is
  begin -- Handle
    Occurred := True;
  end Handle;
end Signal;
```

Figure 7. In Ada 95, the protected unit `Signal`, which handles forklift completion interrupts.

```
loop
  -- Wait for call from free AGV
  accept Get_Part (AGV_Handle : in ...) do
    Avail_Handle := AGV_Handle;
    -- Avail_Handle is the first free AGV
  end Get_Part;

  -- Wait for call from Job
  accept Request (AGV_Handle: out ...) do
    AGV_Handle := Avail_Handle;
  end Request;
end loop;
```
(a)

```
protected type Signaler (Interrupt :Interrupts.Interrupt_Id) is
  entry Wait;
private -- Signaler
  procedure Handle;
  pragma Attach_Handler (Handle, Interrupt);

  Occurred : Boolean := False;
end Signaler;
```
(b)

Figure 8. (a) AGV handling in Ada 83; (b) in Ada 95, protected type `Signaler`, which handles AGV completion interrupts.

Package Workstations

The Package `Workstations` changes the most between Ada 83 and Ada 95, because the Ada 83 task-queuing mechanism is inadequate for jobs waiting for access to workstations. Instead, the Ada 83 solution explicitly models workstation queues as data structures because

- the supervisor must be able to change the priority of a queued job and change its queue position accordingly, and
- a job on an output stand must be able to wait for the first of two events: either it is assigned a workstation, or it is bumped off the stand (in which case it is temporarily removed from the workstation queue).

In Ada 95, we met these two requirements by means of priority queuing and asynchronous transfer of control. A protected unit represents each workstation, handling all the operations. A further concern in `Workstations` is that jobs are queued per workstation kind, so a protected unit also represents each work-

```
protected WS_Mgr is
  entry Request (Kind : in WS_Kind; Id : out WS_Id; Ptr : out WS_Ptr);
end WS_Mgr;

protected body WS_Mgr is
  entry Request (Kind : in WS_Kind; Id : out WS_Id; Ptr : out WS_Ptr)
  when True is
  begin -- Request
    -- Just direct the request to the correct kind of workstation
    requeue Kind_Mgr (Kind).Request with abort;
  end Request;
end WS_Mgr;
```

Figure 9. `WS_Mgr` in Ada 95.

station kind. We use requeuing to handle workstation requests: only a protected unit, `WS_Mgr` (see Figure 9), is externally visible. A job requests exclusive access to the input stand of an appropriate workstation by calling `WS_Mgr. Request`. The protected unit `WS_Mgr` protects no data, and its entry `Request` has a barrier that is always true. `Request` requeues the entry call on the entry for the appropriate kind of workstation. This hides the collection of workstation managers from the caller.

Each kind of workstation has an instance, `Kind_Mgr (Kind)`, of the protected type `Id_Queue`, containing a set of IDs for workstations of that kind. It has the entry `Request` and the procedure `Release`. The requeue statement queues the caller on the correct `Id_ Queue. Request` entry as if the caller had called that entry directly. The caller then remains suspended until entry body `Id_Queue.Request` has completed.

On return from `WS_Mgr. Request`, the job task has a pointer to a `Workstation` unit. When the AGV delivers the job to the workstation's input stand, the job calls the workstation's `Process_Job` entry. The workstation's protected unit then tracks the job's progress through the tool in response to messages from the workstation's microprocessor.

Task Job

The `Job` task type describes the life of a job in the FMS at a high level of abstraction. Together, the `Job` instances drive the software by contending for resources. In the Ada 95 solution, each `Job` receives an identity to allow the supervisor to change its priority. The identity is set in a record `Q` that is given as a discriminant when the supervisor creates the `Job` instance. The task type specification is

```
task type Job (Q : access
        Queue_Element);
```

`Job`'s body (see Figure 10) is structurally similar in Ada 83 and Ada 95: it begins in the "newly created" state. At the top of the `Completion` loop, the job is in its storage bin. It requests a workstation; when one is granted, it obtains the forklift and travels to a storage stand.

At the top of the `Floor` loop, the job is on a stand and has been assigned a workstation for its next step. It obtains an AGV and travels to the workstation's input stand. It informs the workstation that it has arrived by calling `WS_Ptr. Process_Job`. Next, the task waits for the job to move to the output stand. Then, if the job is not finished, the task retrieves the next job step and queues up the job for an appropriate workstation kind.

An asynchronous transfer of control (`select … then abort …`) lets the job remain on the output stand until either a workstation has been assigned or another job needs the stand. If a workstation is assigned, the `Job` task loops back to the beginning of the `Floor` loop.

If the part is bumped from the output stand, or if the job finishes, the part moves to the ASRS. This happens after `end loop Floor` and involves a request for a storage stand, AGV transportation, and forklift transportation. If the job is not finished, it then loops back to the beginning of the `Completion` loop.

`Job`'s discriminant, the record `Q`, provides initialization information to instances of the type, such as its production plan and ASRS bin. `Job` tasks arrange for the parts to be moved from place to place and spend the rest of their time waiting for resources—such as AGVs—or for the movement or machining to finish.

Bumping

The bumping situation, in which a job can be bumped off an output stand before its next workstation has been assigned, presents a particular problem in the Ada 83 solution. It involves three jobs:

- J1 sits on workstation W's output stand, waiting for a workstation of a certain kind K, which are all occupied.
- J2 is on the input stand of a workstation X of kind K.
- J3 occupies workstation W's tool and is about to finish.

J1 thus waits for one of two events:

- The assignment of workstation X, caused when X moves J2 into its tool and releases its input stand.
- Bumping, caused when W finishes machining J3, which needs the output stand.

Figure 11 shows the awkward construct in the Ada 83 `Job` task body that accomplishes the multiple wait for either of these events.

The select statement allows J1 to wait for an entry call to either `Assigned` or `Clear_Out_Stand`.

- `Assigned` is called when J2 releases workstation X's input stand.
- `Clear_Out_Stand` is called when J3 finishes.

What happens depends on which entry call comes first:

- If `Assigned` comes first, J1 moves from workstation W's output stand to X's input stand.
- If `Clear_Out_Stand` comes first, J1 is staged in the ASRS.

There is a small likelihood that the calls could be nearly simultaneous. For example, the call to `Clear_Out_Stand` could occur just after the `Assigned` call, and before the data structures reflect the new assignment. The call to `Clear_Out_ Stand` might then become a dangling rendezvous—a call that is never served. To

```
task body Job is
  In_Stand   : Stand_Id;
  Out_Stand: Stand_Id;
  WS        : WS_Id;
  WS_Ptr    : Workstations.WS_Ptr;
  Next_Ptr  : Workstations.WS_Ptr;
  Step      : Step_Record;
begin -- Job
  Step := Get_Step_Info (Q.Plan, Q.Step); -- Get info for 1st step

  Completion : loop
    -- Job is in ASRS bin. Request an appropriate workstation
    Workstations.WS_Mgr.Request
    (Kind => Step.WS_Kind, Id => WS, Ptr => WS_Ptr);
    -- Workstation has been assigned. Move from bin to stand

    ASRS.Bin_To_Stand (Stand => Out_Stand, Bin => Q.Bin);

    Floor : loop
      -- Job is on output stand. Move to input stand.
      In_Stand := Get_In_Stand (Step.WS_Kind, WS);
      AGV_Mgr.Move (From => Out_Stand, To => In_Stand);

      WS_Ptr.Process_Job;
      -- Inform WS that Job is on its input stand.
      -- Job is on output stand on return from this entry call.

      Out_Stand := Get_Out_Stand (Step.WS_Kind, WS);

      Q.Step := Q.Step + 1; -- Get info for next step
      Step := Get_Step_Info (Q.Plan, Q.Step);

      Q.Done := Step.Done;
      exit Floor when Q.Done; -- Exit if finished

      Next_Ptr := null;
      select -- Get workstation for next step, or be bumped back
            to ASRS
        Workstations.WS_Mgr.Request    -- Get workstation
          (Kind => Step.WS_Kind, Id => WS, Ptr => Next_Ptr);
      then abort
        WS_Ptr.Clear_Out_Stand; -- Be bumped
      end select;
      exit Floor when Next_Ptr = null; -- Exit if bumped
      WS_Ptr := Next_Ptr;
    end loop Floor;

    -- Move job to ASRS
    ASRS_Stand_Mgr.Stand_Control.Request (Stand => In_Stand);
    AGV_Mgr.Move (From => Out_Stand, To => In_Stand);
    ASRS.Stand_To_Bin (Stand => In_Stand, Bin => Q.Bin);

    exit Completion when Q.Done; -- Exit if finished
  end loop Completion;
end Job;
```

Figure 10. The task body of `Job` (Ada 95).

avoid this, the calls to `Clear_Out_ Stand` and `Assigned` are both timed entry calls that are kept outstanding for only a short time. Task `WS_Msg` executes the following timed entry call when J3 needs the output stand (`WS_Msg` in the package `Workstations` receives messages from the workstations):

```
select
   T.Clear_Out_Stand;
  -- T is the task of J1
or
  delay Short;
end select;
```

A `Job` task executing the subprogram `Workstations.Rel_In_Stand` makes a similar timed call to the entry `Assigned` after it has released the input stand J1 needs.[3]

There is yet another complication. Assume that job J1 is to be bumped off the output stand. First, a call to `Dequeue` takes it off the workstation queue, which is necessary because a part that is being transported to the ASRS can't be rerouted to a workstation. (Once in storage, the job is requeued.) In `Dequeue`, we again account for the slight chance that the job might have been assigned a workstation just after the task received the `Clear_Out_Stand` call. If so, `Dequeue` returns a valid workstation number, and the part moves to the next workstation rather than to the ASRS.

Bumping is an inelegant part of the Ada 83 solution, which cannot be well encapsulated but appears as the interaction of several statements in the Job task body. In the Ada 95 solution, we improved this through asynchronous transfer of control (see Figure 12).

In this asynchronous `select` statement, the abortable sequence is

```
WS_Ptr.Clear_Out_Stand;
```

and the trigger is the entry call

```
Workstations.WS_Mgr.Request
(...);
```

If the `Request` call is completed, the asynchronous `select` statement aborts the abortable sequence. On the other hand, if the abortable sequence completes, it aborts the `Request` entry call.

ADA 95'S CONCURRENCY FEATURES ARE primarily about efficient implementation. Most of the abstractions that Ada 95 allows are implementable in Ada 83. A more conceptual change that did not affect the FMS solution is the inclusion of tagged types in Ada 95. This feature allows abstractions to build on each other through subclassing. Ada could have had tagged protected types, but the language designers did not combine the new, object-oriented features with the new concurrency features in that way.

Unfortunately, Ada has a reputation in some quarters as a special-interest language ill-suited for general use. With the Ada 95 improvements, it is versatile enough for a variety of real-time systems ranging from structurally simple problems with periodic tasks to complex, safety-critical control systems. Examples of successful Ada implementations include systems for air-traffic control, railroad signaling and train control, the control of radio telescopes, process supervision, and jet-airliner avionics.

References

1. Dept. of Defense, *Ada-9X Requirements*, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., 1990.
2. B. Sandén, "Modeling Concurrent Software," *IEEE Software*, Vol. 14, No. 5, Sept./Oct. 1997, pp. 93–100.
3. B. Sandén, *Software Systems Construction with Examples in Ada*, Prentice Hall, Upper Saddle River, N.J., 1994.
4. *Ada 95 Reference Manual*, ISO/IEC (Int'l Organization Standardization/Int'l Electrotechnical Commission) Vol. 8652, 1995; http://www.adahome.com/rm95/.
5. *Ada 95 Rationale*, Intermetrics Inc., Cambridge, Mass., 1995; http://www.adahome.com/Resources/refs/rat95.html.
6. A. Burns and A. Wellings, *Concurrency in Ada*, Cambridge Univ. Press, Cambridge, UK, 1995.
7. J. Carter, "The Form of Reusable Ada Components for Concurrent Use," *Ada Letters*, Vol. 10, No. 1, Jan./Feb., 1990, pp. 118–121.
8. J. Carter, "Concurrent Reusable Abstract Data Types," *Ada Letters*, Vol. 11, No. 1, Jan/Feb, 1991, pp. 96–101.
9. T. Elrad, "Comprehensive Scheduling Controls for Ada Tasking," *Proc. Second Int'l Workshop Real-Time Ada Issues*, *Ada Letters*, Vol. 8, No. 7, Fall 1988, pp. 12–19.
10. R. Buhr, *System Design with Ada*, Prentice Hall, 1984.

```
Enqueue (WS_Kind, WS_Num, Q); -- Queue up J1 for its next
      workstation
if WS_Num = No_WS then
  -- Workstation not immediately available.
  -- Wait for either assignment or bumping:
  select
    accept Assigned (WS_N : in WS_Num_Type) do
      WS_Num := WS_N;
    end Assigned;
  or
    accept Clear_Out_Stand;
    Dequeue (WS_Num, Q); -- Remove J1 from workstation queue
    exit Floor when WS_Num = No_Ws; -- move J1 to storage
  end select;
end if;
```

Figure 11. Multiple wait in the Ada 83 `Job` task body.

```
Next_Ptr := null;
select -- Get workstation for next step, or be bumped back to
       ASRS
  Workstations.WS_Mgr.Request  -- Get workstation
    (Kind => Step.WS_Kind, Id => WS, Ptr => Next_Ptr);
then abort
  WS_Ptr.Clear_Out_Stand;  — Be bumped
end select;
exit Floor when Next_Ptr = null;
WS_Ptr := Next_Ptr;
```

Figure 12. Bumping in Ada 95 through asynchronous transfer of control.

Jeffrey R. Carter is the president of PragmAda Software Engineering. He is interested in software engineering; software-development methods—especially as applied to concurrent, real-time software; and software reuse. He holds an MS in software systems engineering from George Mason University. He is a member of the ACM and IEEE Computer Society. Contact him at PragmAda Software Eng., 1540 Coat Ridge Rd., Herndon, VA 20170; jrcarter@computer.org.

Bo I. Sandén is a professor of computer science at Colorado Technical University. His research and teaching interests include software engineering and software design, especially for concurrent real-time systems, object-oriented analysis, and concurrent object-oriented systems. He received his MS in engineering physics from the Lund Institute of Technology and his PhD in computer science from the Royal Institute of Technology in Stockholm. He is a member of the ACM and IEEE Computer Society. Contact him at Colorado Technical Univ., 4435 N. Chestnut St., Colorado Springs, CO 80907-3896; bsanden@acm.org; http://www.pcisys.net/bsanden.