Intuitive Multitasking in Ada 2005

Dr. Bo I. Sandén
Colorado Technical University

As multiprocessors become common, more software must be multithreaded in order to take advantage of the added processing power. Programming in a language with multithreaded support is easier and less error-prone than with stand-alone thread packages. Multitasking in Ada 2005 builds on the same concepts as Java but is considerably safer.

Ada was first released in 1983 with high ambitions. Unfortunately, it came across as large and complicated. Its approach to multitasking, although innovative, proved unsuccessful. It was built on the rendezvous concept [1], which is elegant in theory, but was an unnecessary departure from the long, practical tradition with threads and shared objects as separate, interacting entity types. Ada 95 remedied this problem and further evolved into Ada 2005. Now, the multitasking facilities in Ada are conceptually mainstream, at the same time much less error prone than in other languages.

This article describes Ada multitasking as it now exists. It is intended for developers of multithreaded systems in other languages and for those who may consider redesigning systems to take advantage of multi-core chips and other types of multiprocessors. Programmers in some languages use POSIX threads [2, 3], which are created by routines with a standardized interface. Other routines operate on mutex variables in order to give threads exclusive access to shared data. Support that is built into the language syntax as in Ada and Java is safer and easier on the programmer because it abstracts low-level operations such as the mutex manipulation. A multitask Ada program will run under symmetric multiprocessing.

From the beginning, Ada was designed to meet the predictability and reliability requirements of dependable systems. Over time, those kinds of systems, which are typically embedded real-time systems, formed Ada's niche. Ada supports *hard* real time. In hard real time, computations must finish by certain, absolute deadlines to avoid dire consequences. If an appropriate scheduling algorithm is used, a set of periodic tasks provably meets its deadlines. Ada supports rate-monotonic and earliest deadline first scheduling.

There is an important category of systems that have *no* conflicting, hard deadlines, but still need to be robust. Many of these systems must perform a periodic function reliably. For example, a cruise controller must adjust the throttle with consistent intervals for a smooth ride, but without absolute deadlines. A building monitoring system may need to check every one of hundreds of sensors every two seconds, but there are no dire consequences if the interval is a little longer. Interactive systems need to respond to human input in a consistent and timely manner, but without hard deadlines.

Ada was always intended for high-integrity systems while Java was originally meant for Windows programming and applets. The real-time specification for Java (RTSJ [4, 5]) does not make the language less error-prone[1]. It is easy to make the case that Ada is much safer than Java in this respect [6, 7]. Unlike Java, but like C++, Ada is a hybrid language; you can program entirely without classes. This can be important in hard, real-time environments where some object-oriented features may be deemed too inefficient.

This article focuses on Ada's support for intuitive multitasking without conflicting, absolute deadlines. This intuitive approach can produce elegant, simple, and efficient programs with tasks modeled on entities in the problem domain [8, 9]. A job that is processed in a flexible manufacturing system (FMS) is an example of such an entity [8, 10]. Flexible manufacturing differs from production lines. Each job task waits for access to one workstation at a time and for access to devices such as forklifts. The task describes the life of a job while resources such as the workstations and the devices can be modeled by means of protected objects. Other aspects of the problem also map directly onto Ada features. The following are various language features and examples of how to use them in intuitive multitasking.

Protected Objects

A protected object is a data structure that is encapsulated to make it *task safe*, which means that multiple tasks can operate on it without risk of conflict; each operation on the object always finishes before the next one starts. Protected objects have lock variables, which are inaccessible to the programmer. The compiler inserts the necessary operations on the locks.

A protected type can have *protected operations* of three kinds: *functions*, *procedures*, and *entries*. These are declared in the protected-type specification as in this example:

```
protected type X is
    function F1( ) return Type1;
    procedure P1 ( );
    entry Acquire ( … );
private
    -- Attribute variables
    -- Private operations including
       interrupt handlers
end X;
```

You can also declare a single protected object. The following are differences between protected functions, protected procedures, and entries:

- *Protected functions* are read-only. They cannot change the protected object's attributes and are subject to a read lock; simultaneous function calls on a given object are allowed, but not during a procedure or entry call on the object.
- *Protected procedures* can change the attribute values. They are subject to a write lock; only one procedure (or entry) call at a time is allowed on a given object, and not during any function call.
- Like protected procedures, entries can change attribute-variable values and are subject to the write lock. In addition, each entry has a barrier condition, which must be true for a call on the entry to proceed. For example, an entry Acquire that is to be performed only when the number of items available is greater than zero can be specified in the body of the protected type as follows:

```
entry Acquire ( ... ) when Available > 0 is ...
```

If a task calls *Acquire* when the attribute variable *Available* is equal to zero, it is queued. Each protected object has one queue *per entry*. This is different from Java, where each synchronized object has a single wait set. Otherwise, a barrier works quite similarly to a wait loop *while (condition) wait( )*, placed at the very beginning of a synchronized operation in Java.

Protected objects are similar to Java's synchronized objects, but much less prone to mistakes and misunderstandings on the part of the programmer [7]. *All* the operations are protected while in Java. In Java, it is up to the programmer to make selected operations *synchronized*.

Use

Protected objects represent shared resources in the problem domain such as workstations and forklifts in the FMS example. A job task acquires a forklift by the call *Forklift.Acquire* on a protected object *Forklift*. It releases the forklift by calling *Forklift.Release*. Release is a protected procedure that increments the variable *Available* and opens the barrier for other job tasks.

While a job is waiting for a resource, its task is queued on a protected entry. This means that *Acquire*'s task queue models the queue of waiting jobs in the factory. Ada gives the programmer provisions for managing such a queue. For example, a task can be removed from the queue if necessary [10].

Requeuing

The Java style with a wait loop as part of the body of a synchronized operation is flexible. For example, a synchronized method in Java can have multiple wait loops; a thread can effectively execute the synchronized operation in segments separated by calls to *wait* where the thread releases its object lock [7]. Each time it is reactivated from the wait set, it enters a new segment with the lock re-established.

The requeue statement in Ada achieves the same effect. It allows a task that is executing an entry to suspend itself and place itself on the queue of the same or another entry. The syntax for requeuing to an entry called *Wait* is as follows:

```
requeue Wait;
```

There are no parameters. The requeuing entry must have the same parameters as the entry in whose body the requeue statement appears.

Use

As mentioned previously, protected objects with operations such as *Acquire* and *Release* can represent resources in the problem domain. Requeuing is the only way to handle a delay during the resource allocation. A high-priority entity may force an entity holding a resource to relinquish it. The high-priority task can requeue itself while the lower-priority task takes appropriate measures to relinquish the resource and call *Release* [8].

Protected Interfaces

Java and CORBA popularized the interface as a syntactic concept [11]. In Ada 2005, a type can implement any number of interfaces in addition to extending a base type. As in Java, an interface is very similar to an abstract class without data where all the operations are abstract[2].

In Ada 2005, protected types can implement *protected interfaces*[3]. Like other interfaces, they allow polymorphism. This addresses, to a degree, an awkwardness in Ada 95, which introduced both protected types and extensible types, but did not combine the two by allowing protected types to be extended [12].

"Protected objects are similar to Java's synchronized objects, but much less prone to mistakes and misunderstandings on the part of the programmer."

If a number of protected types implement the same protected interface *S*, you can reference their instances by means of access variables with a target of type *S'Class*. Ada distinguishes between a type such as *S* and the polymorphic, *class wide* type *S'Class*, which can refer to instances of *S* and its descendants. An access variable is essentially a pointer to an object of the target type.

Use

In a program that deals with devices of different types where each type needs its own driver, you could declare a protected interface with an operation *Initialize* as in the following example:

```
type Device_Handler is protected inteface;
procedure Initialize (D: in out Device_
  Handler) is abstract;
```

The interface is implemented by various device-handler types as *Device_Type1* in this example:

```
protected type Device_Type1 (Device_
  Number : Natural) is
     new Device_Handler with
     procedure Initialize;
          . . .
end Device_Type1;
```

The body of the protected type contains the logic of the procedure *Initialize* for this particular device type. *Device_Number* is a *discriminant*, which allows you to give unique information such as a device number to each instance.

You can now declare an array of pointers to device drivers as follows:

```
Driver_List : array ( … ) of access
Device_Handler'Class;
```

A loop such as the following invokes the initialization procedure appropriate for each array element:

```
for D in Driver_List'Range loop
     Driver_List(D).Initialize;
end loop;
```

For each value of D, the call *Driver_List(D).Initialize* binds at runtime to the *Initialize* procedure appropriate for the type of device at *Driver_List(D)*.

Asynchronous Transfer of Control

Like RTSJ, Ada provides for asynchronous transfer of control (ATC). With ATC, the programmer can arrange for a computation to be cut short if a triggering event should occur before the computation is complete. A common example is an algorithm that iteratively improves an approximate result until it converges to a value with a certain precision. If the calculation does not converge within a certain time limit, a real-time system may need to terminate it and use the best approximation available. The trigger in this case is the event that the time limit is reached.

Ada's ATC syntax is considerably simpler than that in RTSJ [4, 5]. Ada implements ATC by means of abortable tasks. It uses an *asynchronous select* statement such as the following:

```
select delay until Next_Reading;
then abort
     -- "Abortable sequence"
end select;
```

This works as follows: The abortable sequence starts. If it has not ended by the time *Next_Reading*, it is aborted.

You can also express the trigger in terms of elapsed central processing unit time. This is useful in real-time systems with hard deadlines, where each periodic task is given a maximum amount of processor time per period. Finally, the triggering event can be the acceptance of a certain call on a protected entry. The following example illustrates the last case.

Use

In the FMS, each workstation has an input stand, a tool, and an output stand. A job can sit on the output stand of a workstation and wait for its next workstation to become available. The job may still be on the output stand when the next job is finished in the tool and needs the stand. In that situation, the first job must clear the stand and be staged in a storage area; the job task must be prepared to handle whichever comes first of two possible events: the next workstation becomes available, or the job is ordered to clear the stand. Both are entry calls. This is expressed by the following statement:

```
select
   WS.Request;
then abort
         X.Clear;
         -- “Additional statements”
         -- (Stage job in storage)
end select;
```

The trigger, *WS.Request*, is the request for the next workstation, which is accepted when that workstation becomes available. The abortable sequence starts with the call *X.Clear*, which is accepted when the next job is done in the tool. This makes for a race between those two events: *WS.Request* is accepted and *X.Clear* is accepted. One will happen first, and then the other one is aborted. If *X.Clear* is accepted, the call *WS.Request* is aborted and the additional statements execute. If *WS.Request* is accepted, the call *X.Clear* is aborted and the additional statements never execute. The ATC logic arbitrates the outcome of the events even if they happen at practically the same time.

Interrupt Handlers and Timing Events

As a language intended for embedded systems from the beginning, Ada allows the programmer to specify interrupt handlers (RTSJ introduces interrupt handling in Java). In Ada 95 and on, the handlers are protected procedures.

Ada 2005 introduces *timing events* as a means to define code to be executed at a certain time. They are similar to *OneShotTimers*, which are a type of asynchronous events in RTSJ [4, 5]. A timing event can be set to go off either at a certain time or after a certain interval and can be canceled. When the event goes off, it causes a handler to execute. Like interrupt handlers, timing-event handlers are protected procedures.

As with interrupt handlers, the system executes the timing-event handlers; the programmer does not need to supply a task. This simplifies things. In earlier Ada versions, you needed tasks with delay statements to achieve the effect of a timing event.

Use

A car driver can manipulate the driver’s side window by means of a lever on the door. Figure 1 is a fragment of a state model of the window. It starts in state *Still*. By pushing the lever down, the driver puts the window in state *Moving_Down*. Releasing the lever takes the window back to *Still*. If the driver holds the lever down for *Time_Amount* milliseconds, the window transitions to the state *Auto_Down*, where it continues down even after the driver releases the lever.

We can define a timing event *Auto_Time* to capture this. It occurs when the window has spent *Time_Amount* milliseconds in *Moving_Down*. It causes the window to enter state *Auto_Down*.

The handlers for the interrupts *Lever_Down* and *Release* and the timing event *Auto_Time* are protected procedures in *Window_Control*, which is a state-machine protected object [8, 9]. It maintains the state of the window in the variable *Wstate* of the enumerated type *State_Type*. *Wstate’s* initial value is *Still*. The following are type and instance declarations and part of the specification of *Window_Control*:

```
type State_Type is
   (Still, Moving_Down, Auto_Down, ... .);
Auto_Time : Timing_Event;
Time_Amount : constant Time_Span := ... .
protected Window_Control is
private
    procedure Lever_Down;
    procedure Release;
    procedure Time_Out
          (Event : in out Timing_Event);
    Wstate : State_Type := Still;
end Window_Control;
```

The interrupt and event handlers need not be visible from other parts of the software so they can be declared *private*. I am leaving out statements that tie the interrupt handlers to certain interrupts. The protected body contains the logic of the procedures [8].

In a state machine-protected object such as *Window_Control*, the timing-event handler fits in particularly well among the various interrupt handlers. In this example, where there is no real computation, you need no task at all. A pre-Ada 2005 solution would require a task with a delay statement that calls *Time_Out* when the delay expires.

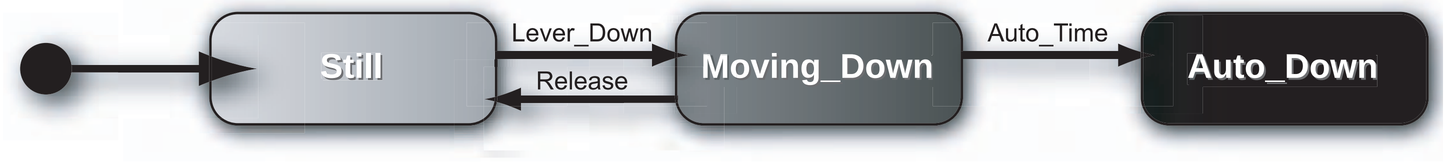
Conclusion

In the mid-80s, soon after Ada 83 first appeared, programmers switched from secure Pascal-like languages such as Ada to insecure C-like languages [13]. Had Ada been an immediate success, things might have gone differently. The original tasking model worked against Ada. That awkwardness is now long gone. Tasking in Ada 2005 is conceptually similar to Java threading, but much safer. Those responsible for critical software development no longer have an excuse to gamble on languages that leave ample room for programmer mistakes.◆

References

1. Burns, A., and A.J. Wellings. Concurrency in Ada. 2nd ed. Cambridge University Press, 1998.
2. Nichols, B., D. Buttlar, and J. Proulx Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. O’Reilly and Associates, 1996.
3. Butenhof, D.R. Programming With POSIX Threads. Addison-Wesley, 1997.
4. Bollella, G., and J. Gosling. “The Real-

Figure 1: *State Diagram Fragment of the Car Window*



Time Specification for Java." IEEE Computer 33:6 (2000): 47-54.

5. Wellings, A.J. Concurrent and Real-Time Programming in Java. John Wiley & Sons, 2004.
6. Nilsen, K. "Applying RAMS Principles to the Development of a Safety-Critical Java Specification." CROSSTALK Feb. 2006 <www.stsc.hill.af.mil/crosstalk/2006/02/0602Nilsen. html>.
7. Sandén, B.I. "Coping With Java Threads." IEEE Computer 37:4 (2004): 20-27.
8. Sandén, B.I. Multithreading. Colorado Technical University, 2006 <http://home.earthlink.net/~bosanden/Multithreading>.
9. Sandén, B.I., and J. Zalewski. "Designing State-Based Systems With Entity-Life Modeling." Journal of Systems and Software 79:1 (2006): 69-78.
10. Carter, J.R., and B.I. Sandén. "Practical Uses of Ada 95 Concurrency Features." IEEE Concurrency 6:4 (1998): 47-56.
11. Siegal, J. "OMG Overview: CORBA and the OMA in Enterprise Computing." CACM 41:10 (1998): 37-43.
12. Wellings, A.J., R.W. Johnson, B.I. Sandén, J. Kienzle, T. Wolf, and S. Michell. "Integrating Object-Oriented Programming and Protected Objects in Ada 95." ACM TOPLAS 22:3 (2000): 506-539.
13. Brinch Hansen, P. "Java's Insecure Parallelism." ACM SIGPLAN Notices 34:4 (1999): 38-45.

Notes

1. A safety-critical Java specification is being proposed that carefully defines a small subset of Java and RTSJ to meet stringent reliability, availability, maintainability, and safety requirements [6].
2. Ada interfaces can also have *null* operations, which are concrete but have no effect.
3. *A synchronized interface* can be implemented by protected types and tasks.

About the Author

Bo I. Sandén, Ph.D., has 15 years experience as a software developer and 20 years teaching sofware engineering. He is now a professor of computer science at Colorado Technical University in Colorado Springs. Sandén's main research interest is language support for multithreading and the design of multithread software. He has a master of science in engineering physics from the Lund Institute of Technology, Sweden, and a doctorate in computer science from the Royal Institute of Technology, Stockholm.

Colorado Technical University
4435 North Chestnut ST
Colorado Springs, CO 80907-3896
Phone: (719) 531-9045
Fax: (719) 598-3740
E-mail: bsanden@acm.org