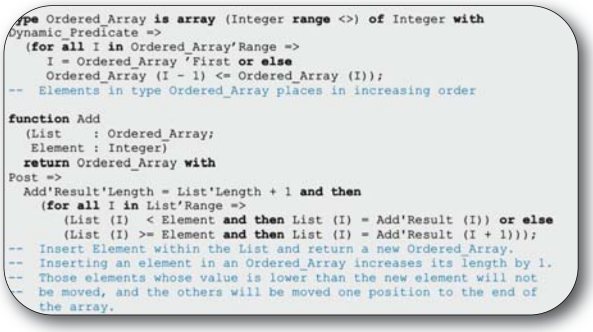
Embedded safety: multicore programming with Ada 2012

By Dr. José F. Ruiz, Adacore

A new version of the Ada language, with nice additions in safety, flexibility, and efficiency has reached the software developers community. Many areas have been improved, but if I have to choose those I like the most, I would mention those directly related to program correctness and the better handling of Ada programs on multicore architectures.



```
ype Ordered_Array is array (Integer range <>) of Integer with
Dynamic_Predicate =>
  (for all I in Ordered_Array'Range =>
     I = Ordered_Array 'First or else
     Ordered_Array (I - 1) <= Ordered_Array (I));
--  Elements in type Ordered_Array places in increasing order

function Add
  (List    : Ordered_Array;
   Element : Integer)
  return Ordered_Array with
Post =>
  Add'Result'Length = List'Length + 1 and then
    (for all I in List'Range =>
       (List (I)  < Element and then List (I) = Add'Result (I)) or else
       (List (I) >= Element and then List (I) = Add'Result (I + 1)));
--  Insert Element within the List and return a new Ordered_Array.
--  Inserting an element in an Ordered_Array increases its length by 1.
--  Those elements whose value is lower than the new element will not
--  be moved, and the others will be moved one position to the end of
    the array.
```

Figure 1: Subprogram contract

■ Ada has always been an attractive choice in application domains where reliability is paramount, and the new Ada 2012 version represents another major advance in the evolution of the language towards safety, namely with the addition of contract-based programming. Another aspect that became critical with the widespread of parallel hardware architectures was the control of task affinities to improve efficiency and analyzability.

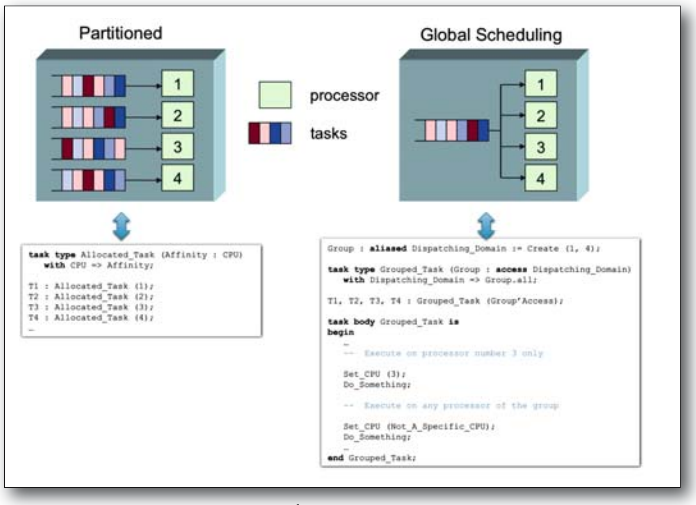
Before explaining the advantages of contracts, let us move to a higher level of abstraction and talk about what they represent: requirements. Software requirements define what needs to be implemented and how. This is usually achieved using natural (informal) language, but with contracts the idea is to define the requirements by formally specifying the exact functionality to implement. This is the cornerstone of Design-by-Contract, which gives precise and verifiable semantics to specifications.

A contract is given by a precondition, which the caller must pay to be entitled to the service provided by the callee, and a postcondition, which is the service the callee must provide to the caller. Ada 2012 includes specific features for contract-based programming: preconditions, postconditions, type invariants, and subtype predicates. A precondition is a logical expression that must be true when a subprogram is called, and analogously a postcondition must be true when the subprogram returns. A type invariant is a postcondition that applies to every public subprogram for a type, and a subtype invariant is a logical expression that characterizes a subset of values for a type. Contracts, which are in effect low-level requirements, may be verified dynamically, and they may also be verified statically using appropriate tools.

Expressing properties in contracts is greatly facilitated by the use of several new Ada 2012 features: conditional expressions, case expressions, universal and existential quantified expressions, and expression functions (an expression function is a simple function whose body is defined by a single expression). Additionally, the expression in a postcondition can refer to the value returned by a function F as F'Result, and to the value in the pre-state (at the beginning of the call) of any variable or parameter V as V'Old. For example, we may want to define the specification (requirements) for an unbounded array whose elements are always ordered in increasing order, which can be modified by a function called Add that inserts an element in the corresponding location of the ordered array. The requirements can be specified using natural language, or better using formal logic formulas relating the input to the output state defining accurately and unambiguously the expected behavior (figure 1).

The Boolean expression representing the postcondition for the subprogram defines the expected effect of the function: we expect a list which has one more element than the one passed as parameter (the one we are inserting as parameter), and with items in the list moved one position to the end if they are greater or equal than the inserted value. Note that in this case, we have defined the properties of the type to ensure it is always ordered, so we do not need to ensure the ordering again in the postcondition. These Ada 2012 capabilities can be exploited by new static analysis and proof tools, which can significantly reduce the time and cost associated with traditional testing approaches, increasing at the same time the level of confidence, and helping detect problems early.

The complexity of both hardware and software quickly increases to cope with ever demanding applications, bringing increasing attention to high-level, abstract development methods. We have just discussed the interest of formal specification as a means to help requirements-based software development. There is another field requiring attention for engineering complex systems, which is software architecture. Writing correct programs efficiently exploiting parallel hardware is not trivial, and by providing the right level of abstraction, programmers are isolated from the need to understand low-level details. The Ada tasking model provides concurrency as a means of decoupling application activities, hence making software easier to design and test. At the same time, it gives different levels of control over the hardware where the application executes. Concurrency has been a first-class citizen in Ada since the beginning, and it has kept improving over time. Already in Ada 83 there were tasks as units of concurrent/parallel execution (same abstraction level as threads), and high-level constructions for message-based synchronizing and communicating them. Then, Ada 95 introduced the notion of synchronization and communication using data-oriented communication. The Ada 2005 standard added support for run-time profiles (for efficiency and simplicity), flexible task-dispatching policies, the



```
task type Allocated_Task (Affinity : CPU)
   with CPU => Affinity;

T1 : Allocated_Task (1);
T2 : Allocated_Task (2);
T3 : Allocated_Task (3);
T4 : Allocated_Task (4);
…
```

```
Group : aliased Dispatching_Domain := Create (1, 4);

task type Grouped_Task (Group : access Dispatching_Domain)
   with Dispatching_Domain => Group.all;

T1, T2, T3, T4 : Grouped_Task (Group'Access);

task body Grouped_Task is
begin
   …
   --  Execute on processor number 3 only

   Set_CPU (3);
   Do_Something;

   --  Execute on any processor of the group

   Set_CPU (Not_A_Specific_CPU);
   Do_Something;
   …
end Grouped_Task;
```

Figure 2: Some partitioning schemes

capability of monitoring and controlling execution time, and a unification of concurrency and object-oriented features. Finally, Ada 2012 improved largely the support for multiprocessor architectures. Ada has always taken into account parallel hardware architectures, supporting concurrent, parallel and interleaved execution, allowing for different partitioning schemes. However, until Ada 2012, there was not a standardized mechanism to control task allocation on processors.

In terms of relationship between tasks and processors, the spectrum goes from global scheduling, where any task can be executed on any processor at any time, to partitioned scheduling, where each task is allocated for its whole lifetime to concrete processors. The schedulability of neither approach is strictly better than the other (there are task systems that are feasible using a global partitioning that cannot be scheduled in a partitioned system and vice versa). Ada 2012 includes a flexible and general-purpose mechanism to handle task affinities in the form of dispatching domains, the abstraction representing groups of processors on which we allocate tasks.

Processors are grouped together into dispatching domains, and tasks may then be allocated to domains. Tasks allocated to a given dispatching domain will be executed on any of the processors of that domain. It is also possible to allocate a task to a concrete processor (either statically or dynamically) for any amount of time. Figure 2 depicts a couple of possible allocation strategies and how to exploit them. The notion of task affinity is supported by mainstream operating systems (such as Linux, Windows, Solaris, VxWorks, …). The Ada model is slightly more restricted than the generic mechanism provided by these operating systems: dispatching domains are non-overlapping, and they can only be created before calling the main subprogram. However, this more static model is flexible enough to support many different partitioning schemes, while at the same time providing for the definition of analyzable software architectures.

Reliable and very efficient execution on multiprocessors can be achieved using the Ravenscar tasking profile. This subset of Ada tasking features embodies a deterministic concurrency model inherently amenable to static analysis and implementable by a small, reliable, and extremely efficient run-time library. The profile has been defined to improve memory and execution time efficiency (removing high overhead or complex features), and to increase reliability and predictability (removing non-deterministic and non-analyzable features).When reliability, predictability, and analyzability are critical, Ada 2012 proposes a simple extension to the Ravenscar profile to support multiprocessor systems using a fully partitioned approach. The implementation of this scheme is simple, and it can be used to develop applications amenable to schedulability analysis. TheRavenscar profile implements fixed-priority pre-emptive scheduling, with tasks statically allocated to processors and no task migration among processors. Apart from the support for task affinities, there are other interesting capabilities allowing for predictability and efficiency on parallel architectures. Ada 2012 added a new effective parallel task synchronization mechanism with which a group of tasks can block and be released at once to work in parallel (mimicking the POSIX barrier mechanism). There is also the possibility to control the behavior of selected objects with respect to their order of loads and stores with multi-level caches. A typical problem when implementing synchronization on multiprocessors (such as wait-free and lock-free) is that of memory consistency, to ensure that the execution does not result in an unexpected order of execution. Ada 2012 allows you to mark variables as volatile so that all tasks of the program (on all processors) that read or update volatile variables see the same order of updates to the variables; it is the responsibility of the compiler to use memory barriers to flush the cache if needed. Ada 2012 takes advantage of decades of experience in using Ada on multiprocessors, and it has become a great language helping to exploit parallelism in an efficient and predictable manner. One of the challenges in software engineering is how to go from high-level specification and design to their actual implementation. Ada 2012 addresses this issue by providing a high level of abstraction exposing concepts that are relevant for the design. It addresses the specification with contracts, that define accurately the required functionality, and that are naturally verifiable by either formal proofs or testing. It targets the design with the Ada tasking model that permits control over aspects such as processor affinity, dispatching mechanism, and memory consistency. ■