EE Times: Design News

# Ada enhances embedded-systems development

Ben Brosgol and Jose Ruiz

(01/01/2007 9:00 AM EST)

URL: http://www.eetimes.com/showArticle.jhtml?articleID=196701748

Developing an embedded system is hard work. Reliability is essential; indeed, embedded software may control a safety- or security-critical system where an error can have catastrophic consequences. New requirements will almost surely pop up, so the software must be maintainable. Real-time constraints and memory limitations make time and space predictability and run-time performance important. Many embedded systems comprise activities that are performed concurrently, either with actual parallelism or through multiplexing on a single processor. And most deal at some point with hardware-specific details like interrupt handling and data layout.

To meet those demands, a programming language needs high-level features that support sound software engineering and provide the necessary generality, but without sacrificing run-time performance. Such a language must also provide low-level mechanisms that are more typically found in assembly language programming.

Ada was designed to satisfy these sometimes-conflicting requirements, and recent enhancements in the new Ada 2005 standard have improved it. Ada makes the embedded-system developer's job more manageable, and it can be a better development choice than C, C++ or Java.

Ada was designed from the start to promote reliability and maintainability, with features that emphasize readability over writability and that detect errors early. Many checks are performed by the compiler--for example, to ensure that the uses of a data object are consistent with its type. Errors that are not detectable at compile time--such as an out-of-bounds index (also known as buffer overrun)--are caught at run-time through compiler-generated checks. Those checks can often be eliminated automatically through compiler optimizations or, if the programmer has verified that they will not fail, manually through specific directives.

Ada's emphasis on readability and reliability is in contrast with the C family of languages, including C++. Java detects buffer overrun errors, but its weakly typed primitive type facility allows data misuse errors that would be caught in Ada. And unlike both Java and the C-based languages, Ada allows programmers to specify a constrained range for a scalar variable (for example, integers in the range 0 through 100), which aids both readability and reliability.

The evolution of programming languages has been accompanied by two major development approaches: procedural programming, in which a system's architecture is dictated by the kinds of processing that must be performed, and object-oriented programming, in which a system's architecture is dictated by the kinds of entities that must be processed and their relationships. Some embedded systems can be modeled through a procedural-programming approach; others may best be captured through object orientation in order to facilitate enhancements and maintenance.

## Language support for embedded systems

Wanted: High-level features without sacrificing run-time suppport

| | Ada | C | C++ | Java |
|---|---|---|---|---|
| **Reliability** | | | | |
| Strong typing | Yes | Partial | Partial | Partial |
| Range constraints | Yes | No | No | No |
| Index checks | Yes | No | No | Yes |
| **Methodologies supported** | | | | |
| Procedural | Yes | Yes | Yes | Awkward |
| Object orientation | Yes | No | Yes | Yes |
| **Concurrency features** | | | | |
| Functionality | High | None | None | High |
| Software engineering | High | Not applicable | Not applicable | Low |
| **Low-level support** | | | | |
| Functionality | High | High | High | Low |
| Software engineering | High | Low | Low | Low |
| Subsettability support | High | Low | Low | Low |

Source: AdaCore

Ada, like C++, can be used for both procedural and object-oriented programming. C, by contrast, lacks object orientation, and purely procedural programming in Java is rather clumsy.

Concurrent programming is intrinsically more difficult than sequential programming. Testing is complicated, since there are many more possible control paths and since new sorts of errors can arise, such as race conditions, deadlock and accessing a data object while it is being modified.

Ada has a high-level concurrency model that is designed to help avoid such errors. Concurrent activities (tasks) can communicate with each other either directly (rendezvous) or through protected operations on protected objects. A protected operation is performed with mutually exclusive access to a protected object, and the semantics help prevent race conditions. A predefined object-locking policy can be implemented extremely efficiently because of the requirement that tasks not block while executing protected operations.

Ada 2005 has extended the Ada tasking model. It defines some new task-dispatching policies (including Earliest Deadline First) and provides a framework in which multiple policies can coexist. It also allows applications to monitor and control execution time on a task-by-task basis, with run-time detection of budget overruns.

Many languages (such as C and C++) do not support concurrency directly and instead require the programmer to obtain the desired facilities through libraries. This interferes with portability. Others, most notably Java, have a low-level concurrency mechanism that is error-prone.

Embedded systems often have to perform low-level processing: dealing with storage addresses, laying out data structures with specific fields occurring at specific offsets, querying or specifying the size of data objects, handling interrupts, using specialized hardware instructions, treating data as "untyped" storage elements. All of those capabilities are found in Ada. Moreover, and in contrast to C and C++, the Ada rules make it clear to the reader of the program that such system-specific and perhaps potentially unsafe features are being used. Low-level programming in Java requires native code and a corresponding loss of protection.

Embedded systems typically have stringent requirements for a small memory footprint and high efficiency. "Simple" languages, such as C, that might meet those requirements are problematic with respect to reliability, and they lack some necessary features (such as concurrency support and exception handling). Higher-level languages like C++, Java and full Ada come with run-time libraries that may be large or complex. Ada supplies a mechanism, the Restrictions pragma, that lets the programmer select exactly those features that are needed. No run-time libraries are included for features outside the chosen subset.

A useful subset for embedded high-integrity real-time systems is the Ravenscar profile, a reliable concurrency model standardized in Ada 2005 as a set of Restrictions pragmas. Ravenscar is general enough to express the kinds of concurrency patterns that are needed but is simple enough to be supported by a small, reliable and efficient run-time library. The Ravenscar profile meets the requirements for determinism, schedulability analysis and memory boundedness, suiting it for both hard-real-time and high-integrity embedded systems.

**Safe pointers**

Pointers are important in low-level programming. But if the pointer facility is not strongly typed, an object of one type may be viewed (through a pointer) as having a different type. If an object becomes inaccessible--for example, by being popped off the run-time stack--but a pointer to the object still exists, the result is a notorious bug known as a dangling reference. And with dynamic allocation, storage reclamation is a concern.

Ada addresses those issues via a strongly typed pointer mechanism that includes scope accessibility checks to prevent dangling references. The Restrictions pragma may be

used to indicate a constrained use of pointers (for example, dynamic allocations only occurring during system initialization). Since aggregate data can go on the stack, Ada does not require the overhead of a run-time garbage collector.

Ada is unique in explicitly addressing the mixed-language requirement of modern systems. The Ada standard provides interfacing mechanisms that allow an Ada program to import subprograms or global data from, or export them to, other language environments. Ada also lets the programmer specify that a data structure is to be laid out based on the conventions of a compiler for another language.

Thus, it's possible to combine Ada efficiently, reliably and portably with other languages in the same program. For example, if a scientific library in Fortran is needed in an Ada application, there is no need to convert the library to Ada; it can simply be called from the Ada code. In the other direction, if an existing Ada application is to be extended with new functionality that is to be implemented in another language (such as C++), the simplest approach is to use the Ada interfacing mechanisms so that the new code can call the existing Ada subprograms. There is no reason to undertake the risky and costly strategy of converting Ada to the other language.

*Ben Brosgol (brosgol@ adacore.com) is senior software engineer at AdaCore in New York. Jose Ruiz (ruiz@adacore.com) is senior software engineer at AdaCore in Paris.*