



Case Study:

Space Station Robot Embeds Ada

Despite rumors of its demise, Adas is at the heart of the International Space Station's in-orbit Canadarm 2 where it assures software safety and reliability.

Robert Dewar, President,
Ada Core Technologies

While safety-critical characteristics have been introduced into the design of many programming languages, Ada is the language specifically targeted at “life-critical” systems. Developed between 1975 and 1984 by the US Department of Defense (DoD), Ada has been classically targeted for use in mission-critical embedded systems that emphasize safety, low cost, and a near-perfect degree of reliability. The most important safety features that make Ada ideal for development of fail-safe software include its information-hiding capability, its ability to provide re-useable code and its “strong typing”, which helps detect and solve many types of coding errors at compile time, very early in the development cycle.

Despite the perception by some that Ada is a dying language, the fact is that Ada's use is on the rise and it's being adopted for some of the most rigorous and critical embedded applications under development today. Under contract to the Canadian Space Agency (CSA), MacDonald Dettwiler (MDA) chose open-source GNAT Ada 95 from Ada Core Technologies to develop control software for the Mobile Servicing System (MSS), an essential component of the International Space Station (ISS).

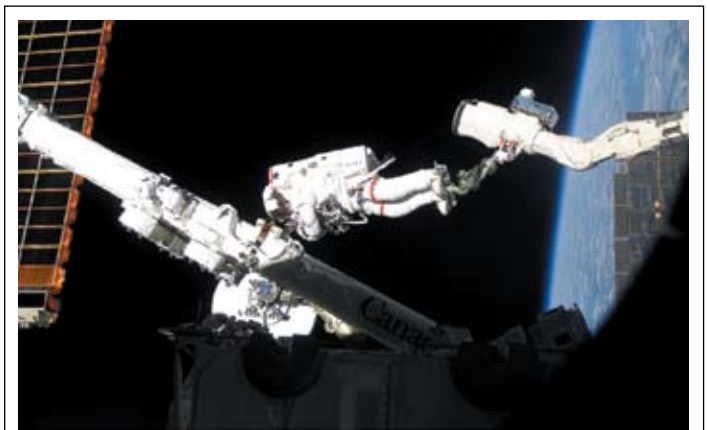
The MSS is a complex robotic manipulator system that plays a key role in space station assembly and maintenance. It helps move equipment and supplies around the station, supports astronauts working in space, and services instruments and other payloads attached to the space station (Figure 1).

Space-Based Robotic Arm

Ideal for a program like the MSS, Ada has clearly carved itself a comfortable and sustainable niche in large, complex high-reliability systems, including safety-critical systems where human life might be at stake. This language, which has little visibility compared with its cousins C and C++, continues to be very effective in developing systems that absolutely must be reliable. So it is no surprise to find Ada in space—a harsh, unyielding environment where the slightest malfunction can lead to death.

The ISS-based, next-generation Canadarm 2, the key element of the MSS, is a bigger, better, smarter version of Canadarm, the robotic arm that operates from the cargo bay of the Space Shuttle. This arm is capable of handling large payloads and assisting with docking the space shuttle to the space station. The new arm, built specifically for the space station, is 17.6 meters (57.7 feet) long when fully extended and has seven motorized joints, each of which operates as a complex real-time embedded control system.

Canadarm 2 is “self-relocatable” and can move around the station's exterior like an inchworm. Each end of the arm is equipped with a specialized mechanism called a Latching End Effector that can lock its free end on one of many special fixtures, called Power Data Grapple Fixtures placed strategically around the station, and then detach its other end and pivot it forward. Unlike the original Canadarm, Canadarm 2 stays in space for its useful life, requiring a high-reliability design that allows astronauts to repair it in-orbit. It nearly goes without saying that the reliability of Canadarm 2 must be unimpeachable, based on high-integrity software and



(Courtesy of: MacDonald Dettwiler.)

Figure 1

Canadian Astronaut Chris A. Hadfield stands on the end of the Space Shuttle's Canadarm while bolting Canadarm 2 together.



(Courtesy of: MD Robotics.)

Figure 2

The Mobile Servicing System's Robotic Work Station (RWS) features a display and control panel, hand controllers, video monitors and computers to provide a highly reliable, seamless interface between man and machine.

safety-critical design constructs that need to be formally and rigorously specified, designed and implemented.

Safety-critical software systems like Canadarm 2 often employ multiple levels of safety criticality. Historically, these have been deployed on systems where each function executes on a dedicated processor. The need to lower maintenance costs and reduce the size, weight, and power consumption of older embedded computer systems, combined with the availability of modern processor technology, has created the demand for commercial run-time systems.

Ada Core's GNAT High-Integrity Edition no-run-time instantiation of Ada 95 was used to develop Canadarm 2, allowing processing tasks with multiple threads of activity, as well as coordination and conformance of multiple processors communicating over well-defined interfaces. GNAT also facilitates a software architecture and implementation that is seamlessly extensible, allowing the integration of future phases of the robotic system.

The main Ada target processor on the Canadarm 2 is an embedded Intel386 device running on a proprietary board that sits at the heart of the arm's Robotic Work Station (RWS). The RWS controls the seven-joint servo mechanisms on the arm, sending commands to motor-control microprocessors, and closing the control loop by reading feedback from position sensors (Figure 2).

The RWS computer also handles the task of scanning user-interface inputs that allow astronauts to control the arm. Software for each joint-control processor is also written in Ada. I/O communications between the RWS and the joint processors occurs over MIL-STD-1553, known as the Aircraft Internal Time-Division Command/Response Multiplex Data Bus (handled by one processor).

Performance Legacy

While the concept of mission-critical systems has been widely adopted throughout the computing industry, it was the DoD that created the original concept. Ada, named after Ada Byron, founder of scientific computing, was born out of the DoD's mid-1970s

concern over the proliferation of programming languages. The DoD advocated a single language, based on solid software engineering principles. Following its 1983 unveiling, the DoD mandated Ada as the official language in 1986. Ada's demise was widely predicted after the DoD removed the mandate in 1997. However, Ada's use has actually risen since then. Ada increasingly has its place as a model of software reliability, reusability, readability, and portability among embedded systems developers.

Ada was the first mainstream programming language to incorporate constructs for multi-tasking and real-time programming, with well-defined interactions between complex features such as tasking and exceptions. It also includes asynchronous transfer of control, protected records, better user access to scheduling primitives, additional forms of delay statements, and parameterized tasks.

In the 1980s, Ada consistently outperformed established programming languages like Pascal, Fortran, and C. In the 1990s, Ada continued to surpass C++ in performance evaluations measuring capability, efficiency, maintenance, risk, and lifecycle cost. In a 1993-94 study of the 10-year history of Verdix Corporation's use of Ada and C development, preliminary findings indicated that Ada was twice as cost effective as C, said Stephen F Zeigler, Ph.D. in his document "Comparing Development Costs of C and Ada".

In-depth analysis of Verdix's general development methodology ultimately showed that when the effects of makefiles and of external costs were factored in, Ada costs were on the order of half the cost of "carefully crafted" C code. The simplest reason for this is that Ada inherently encourages developers to spend more time notating their code in writing, communicating information to future readers of that code. The benefits derived from this include:

Improved Error Locality. Ada bugs are often indicated directly by the compiler, giving developers fewer places to look for bugs. Bugs are often discovered very soon after they are created, and are considered "local in time." If a bug is easily identified as being close to the place in the code where it actually occurs, it is considered "local in space." Ada compilers are good at both time and space locality for bugs, greatly reducing development time.

Better Tool Support. Ada development intrinsically causes a great deal of information to be sent to the development tools. This characteristic is most pronounced for tasking, where the Ada tools can create parallel, distributed multiprocessing versions of ordinary-looking programs without developers having to do much more than they did for a single processor. Another big win is in machine code, where users can get free access to the underlying hardware without having to give up the semantic checks and supports of the high level language.

Improved program design. One of Ada's more subtle effects is to encourage better program design, avoiding the "quick-fix" habits unfortunately common to C programming. Examples include: C allows users to create a global variable without registering it in a ".h" file; C allows users to avoid strong typing easily. The effects of such details are subtle, but can account for some of the "progressive design deterioration" effects that can lead to many extra hours of debugging and interpreting old code.

Ada Embedded

With embedded systems residing in a variety of applications ranging from satellite and telecommunication systems to networking routers and passenger aircraft, an increasing number of lives and dollars now depend on the reliability of embedded systems software. Joint Strike Fighter, various armored vehicle programs, the Apache helicopter—dozens of programs, maybe more—are still actively developing in Ada and spending millions of dollars. Boeing's choice of Ada for high-profile commercial projects like the Boeing 777 is a good example. The 777 is an all-Ada plane except for the entertainment system which is coded in C++.

Penetrating markets long dominated by C++, Ada is surfacing outside the traditional realm of aviation and aerospace, in applications as diverse as meteorological imaging and yachting security. Compaq, for example, used Ada to implement its IN7 project, first on SCO Unix, then on Tru64 Unix v5.1, replacing DECada. IN7 is a relatively complex code mixing several languages and compilation tools, and one of the major difficulties was to get the generated code to run correctly.

IN7, one of the major implementations of the telephone central office switch Signaling System Number 7 (SS7) standard, needed to be fully distributed across multi-computer systems for high availability and high performance to telecom equipment manufacturers and software developers. Compaq turned to Ada to meet these demands, and GNAT specifically because it is targeted at many platforms supported by IN7. While Ada's primary advantage is its support for open systems and interoperability, Table 1 shows several other factors also contribute to higher productivity and lower cost.

Ada's Lifecycle

Research has shown that between 60 and 80 percent of software costs occur in code maintenance—after a system is actually developed and implemented. MacDonald Dettwiler knew these post-project costs could be even higher for the space-based Canadarm 2. Thus, the reliability and long-term robustness of the Ada code took on a very real economic incentive as the MDA engineers considered project lifecycle costs that went far beyond issues pertaining to the development phase of the project.

But will Ada survive? Despite the steady increase in Ada use for safety-critical embedded systems, inaccurate perceptions still linger about this stalwart language. Some programmers remain convinced that Ada will eventually fade away without its DoD mandate.

But, the truth is that Ada is not going to go away. Instead, it will thrive as a language-of-choice for the most demanding life-critical applications. Ada's founding principle was based on rigorous software engineering practices, and Ada is still the only internationally standardized programming language specifically designed for large, complex applications. For real-time, safety-critical applications Ada is still the undisputed leader. ■■

Ada Core Technologies
New York, NY.
(212) 620 7300.
[www.gnat.com].

Metric	Reasons
Availability/Reliability	Ada is highly reliable because its compilers rigorously check the code at compile-time, enabling the programmer to locate and remove defects early in the programming process.
Code Size	First generation Ada compilers produced large executables, and the stigma of large executables stuck with the language. Today's optimizing Ada compilers produce the tightest embedded system code available.
Isolation	Ada effectively compartmentalizes code at run-time, eliminating unpredictable interactions between code modules.
Portability	Ada code is easily portable across microprocessor architectures, making hardware migration and upgrades cheaper and faster.
Readability	Ada code is inherently very readable; errors are easy to locate and correct before compile-time.
Reusability	Ada uses a modularized structure with generic procedures and data abstractions. The result is exceedingly reusable software components, reducing costs for each new project. NASA did a systematic assessment that measured between 60 percent and 80 percent Ada code reuse from one satellite system to the next. This was comparing reuse against past use of Fortran, which was NASA's primary language for satellite systems. With Fortran, code reuse was about 25 percent to 30 percent.
Verifiability	All Ada compilers verify code against the Ada standard using the Ada Conformance Assessment Test Suite (ACATS formerly known as the ACVC tests).

Table 1

A list of factors making Ada increasingly popular in military and high-availability commercial systems.