Applying Java™ Technologies to Mission-Critical and Safety-Critical Development

Kelvin Nilsen
Aonix Inc, Tucson, USA

Adrian Larkham
Aonix Europe, Henley on Thames, England

Abstract

As the complexity of embedded applications evolves, real-time Java is increasingly being used in large-scale applications that demand higher levels of abstraction, portability, and dynamic behaviour. Examples of such applications include management of network infrastructure, automation of manufacturing processes and control of power generating equipment. To meet these demands, real-time Java has moved increasingly into the mission-critical domain.

With the increased penetration into mission-critical and the expected eventual integration into safety-critical applications, the need to assure that Java can deliver reliable operation without exceeding resource constraints has increased. Ease of development and maintenance, support for dynamic behaviour, high performance, soft and hard real-time constraints, and reduction of physical footprint are just some of the requirements of mission-critical Java developers.

To meet these requirements, standards for both mission-critical and safety-critical software are being developed to assist developers in making the engineering tradeoffs necessary for components of such software.

1 Introduction

Originally designed as a language to support "advanced software for a wide variety of networked devices and embedded systems" [1], the Java programming language has much to offer the community of embedded system developers. In this context, we consider Java as a high-level general-purpose programming language rather than a special-purpose web development tool. Java offers many of the same benefits as Ada, while appealing to a much broader audience of developers. The breadth of interest in Java has led to a large third-party market for Java development tools, reusable component libraries, training resources, and consulting services.

Java borrows the familiar syntax of C and C++. Like C++, Java is object oriented, but it is much simpler than C++ because Java's designers chose not to support compilation of legacy C and C++ code. Because it is simpler, more programmers are able to master the language. With that mastery, they are more productive and less likely to introduce errors resulting from misunderstanding the

programming language. Object-oriented encapsulation reduces interference between independently developed components and object-oriented abstractions reduce name collisions and other interference between components

The Java write-compile-debug cycle is faster than with traditional languages because Java supports both interpreted and just-in-time (JIT) compiled implementations. During development and rapid prototyping, developers save time by using the interpreter. This avoids the time typically required to recompile and relink object files.

Java application software is portable because the Java specification carefully defines a machine-independent intermediate byte-code representation and a robust collection of standard libraries. Byte-code class files can be transferred between heterogeneous network nodes and interpreted or compiled to native code on demand by the local Java run-time environment. The benefits of portability are several-fold:

1. Software engineers can develop and test their embedded software on fast PC workstations with large amounts of memory, and then deploy it on smaller less powerful embedded targets.
2. As embedded products evolve, it is easier to port their code from one processor and operating system to another.
3. Cross compiling is no longer necessary. The same executable byte code runs on Power PC, Pentium, MIPS, XScale, and others. This simplifies configuration management.
4. The ability to distribute portable binary software components provides the foundation for a reusable software component industry.

Certain features in Java's run-time environment help to improve software reliability. For example, automatic garbage collection, which describes the process of identifying all objects that are no longer being used by the application and reclaiming their memory, has been shown to reduce the total development effort for a complex system by approximately 40% [2]. Garbage collection eliminates dangling pointers and greatly reduces the effort required by developers to prevent memory leaks.

A high percentage of the CERT®/CC advisories issued every year are a direct result of buffer overflows in system software. Java automatically checks array subscripts to make sure code does not accidentally or maliciously reach beyond the ends of arrays, thereby eliminating this frequently exploited loophole.

The Java compiler and class loader enforce type checking much more strongly than C and C++. This means programmers cannot accidentally or maliciously misuse the bits of a particular variable to masquerade as an unintended value thus reducing programmer errors.

Developers of Java components can require as part of the interface definition for those components that exceptions thrown by their components be caught within the surrounding context. In lower level languages, uncaught exceptions often lead to unpredictable behaviour.

Another very useful Java feature is the ability to dynamically load software components into a running Java virtual machine (JVM). New software downloads serve to patch errors and add new capabilities to an existing embedded system. Special security checking is enforced when dynamic libraries are installed to