Applying COTS Java Benefits to Mission-Critical Real-Time Software

Dr. Kelvin Nilsen
Aonix

As mission-critical, real-time software systems grow in size and complexity, there is increasing pressure to incorporate commercial off-the-shelf (COTS) components as part of the strategy for reducing the total costs of developing and maintaining these systems. In the traditional information technology (IT) space, the object-oriented features of the Java language have proven their value in enabling significant increases in software reuse, and Java has now overtaken C and C++ as the language of choice for most enterprise IT projects. This article discusses the use of Java in mission-critical, real-time systems and emphasizes approaches that address common requirements for portable, efficient, responsive, and predictable real-time systems. These approaches make it possible to develop both soft and hard real-time components, which become COTS real-time components for integration within future real-time systems. Also supported is the ability to integrate non-real-time COTS Java components within systems that have high reliability and real-time constraints.

Moore's law [1], proven by more than four decades of experience, describes the well-known phenomenon that, for a given price point, commercially available computational capacity doubles every 18 months. This increasing computational capacity is used by application software to provide improved functional capabilities, to respond to increasing numbers of service requests with shorter response times, and to support more efficient operations (better fuel efficiency, reduced system failures and down time, and increased productivity). Studies of embedded system trends demonstrate that the size of software in embedded systems also grows exponentially, doubling in size every 18-36 months, depending on the industry [2, 3].

Many of today's typical embedded real-time systems are comprised of hundreds of thousands, if not millions of lines of code. Rather than develop all of the code required for each new product release from scratch, the majority of today's software growth results from integration of third-party software components and the melding of independently developed software systems into larger integrated systems offering the combined capabilities of each individual part.

This article discusses the use of the real-time Java language as an enabling technology to greatly reduce the efforts associated with developing and maintaining reusable real-time software components. The discussion also addresses the need to integrate within real-time systems certain components that were not originally developed with the rigor or discipline typical of real-time software. These non-real-time software components can be successfully deployed within real-time systems as long as systems are carefully constructed to assure that less disciplined components do not steal resources from the allotments for real-time components or compromise the timely execution of them.

What Is Real-Time Software?

The correctness of a real-time system depends not only on delivering correct computational results, but also on delivering these results at the correct time. If results are delivered too early or too late, the real-time system is operating incorrectly. It is the software developer's responsibility to assure that the system operates correctly. Real-time software can be divided into two broad categories: *hard real-time* and *soft real-time*.

> *"The resource needs of each hard real-time component are determined through careful theoretical analysis ... ."*

Hard real-time systems are developed according to the most stringent and conservative practices [4]. The resource needs of each hard real-time component are determined through careful theoretical analysis of the worst-case central processing unit (CPU) time and memory consumption along every worst-case path though the code. On modern CPU architectures, this results in very conservative use of computing resources.

Soft real-time systems comply with real-time constraints using empirical rather than analytical techniques [4]. Characterizing each component's resource needs statistically, developers use probability theory to assess the likelihood that a system integrating multiple independently analyzed components will meet all of its real-time constraints. Since soft real-time systems are not proven with 100 percent certainty to always satisfy all real-time constraints, soft real-time developers must design and implement contingency mechanisms to deal with the occasional missed deadline.

The term *safety-critical* describes software systems that must be certified to the satisfaction of government regulatory auditors because human lives may be lost if the software malfunctions [5]. Such software plays critical roles in commercial avionics, passenger rail systems, nuclear power plants, and certain medical equipment. The rigor of safety certification calls for extremely conservative development practices. As described in this article, safety-critical Java is a proper subset of hard real-time Java technologies.

Using the Java Programming Language for Real-Time Development

The appeal of Java derives from improved developer productivity, reduced software maintenance costs, higher software reliability, enhanced functionality, and improved generality, all of which lead to expanded software longevity. In the traditional business information processing marketplace (financial record keeping, customer relations management, inventory controls, billing, payroll, etc.), the Java programming language has replaced C++ as the predominant programming language, largely because Java programmers are approximately twice as productive when developing new code and are five to 10

times as productive during maintenance of existing code [6-8]. Various real-time Java technologies extend these benefits into embedded real-time systems.

Much of the content presented in this article derives from the general recommendations available in *Guidelines for Scalable Java Development of Real-Time Systems* [9], a document originally developed to guide the use of real-time Java technologies by the European Space Agency. The document details three different approaches to the use of real-time Java, each tailored to the needs of a specific audience: soft real-time, hard real-time, and safety critical real-time. Safe and efficient mechanisms make it possible to build complex systems comprised of components implemented in each of the three different real-time Java profiles.

An important objective of this article is to help engineers understand the trade-offs in selecting between alternative technologies. Soft real-time Java technologies offer the greatest ease of development and maintenance and provide access to the largest existing availability of ready-to-use open-source and COTS Java components. The more constrained hard real-time and safety-critical Java technologies are more difficult for programmers to use, but they offer improved determinism and much more efficient deployment. They also represent a much simpler run-time environment, facilitating the creation of safety-certification artifacts.

Developing Reusable Soft Real-Time Components

One of the key reasons why Java developers are more productive than C and C++ developers is because of automatic garbage collection. According to a study performed by Xerox Palo Alto Research Center in the early 1980s [10], automatic garbage collection reduces programming efforts associated with large, complex software systems by approximately 40 percent. These benefits are amplified significantly in the Java environment because automatic garbage collection is the foundation on which millions of lines of COTS software, including all of the standard Java libraries, are based. Removing garbage collection from the Java language makes it more difficult to develop new software and also precludes the use of nearly all existing Java library code.

However, the power of garbage collection comes with a cost. Traditional Java implementations occasionally pause execution of all Java threads to scan memory in search of objects no longer in use. These pauses can last tens of seconds with large memory heaps. Memory heaps ranging from 100 Mbytes to multiple gigabytes are currently being used in mission-critical systems. The 30-second garbage collection pause times experienced with traditional Java Virtual Machines (VMs) are incompatible with the real-time execution requirements of most mission-critical systems. Special real-time VMs support pre-emptible and incremental garbage collection. This approach is suitable for soft real-time systems with timing constraints as low as a few hundred microseconds.

One of the costs of automatic garbage collection is the overhead of implementing shared protocols between application threads. Application threads continually modify the way objects relate to each other within memory while garbage collection threads continually try to identify objects no longer reached from any threads in the system. This coordination overhead is one of the main reasons that compiled Java programs run at one third to one half the speed of optimized C code.

"When developers speak of hard real-time software, they generally expect that the software be proven to satisfy all real-time constraints prior to execution."

The complexity of the garbage collection process and of any software depending on garbage collection for reliable execution is beyond the reach of cost-effective static analysis to guarantee compliance with hard real-time constraints. Thus, real-time garbage collection is recommended for soft real-time but not hard real-time systems.

Portable soft real-time components should adhere to the following guidelines:

1. Use standard edition Java Application Programming Interface as this provides access to the most widely available set of built-in services.
2. Instrument the component so it can determine its memory and CPU-time requirements on a given deployment target.
3. Deploy the software on a real-time VM that supports fixed-priority scheduling, priority inheritance, priority-ordered queues, and real-time garbage collection.

Software constructed according to these conventions is easily retargeted and integrated into a variety of soft real-time applications. Developers can perform resource needs analysis automatically as part of the dynamic class loading process or manually as part of the software maintenance and integration effort.

Many soft real-time systems have already been fielded using the approaches described in this section. Projects that publicly acknowledge their adoption of real-time Java approaches include Aegis Software System Upgrade [11], Boeing's J-UCAS effort [12], Calix C7 Multi-Service Access System [13], the FELIN (Fantassin à Équipements et Liaisons Intégrés) wearable computer system [14], Nortel's high-end, long-haul fiber-optic switch [15], and Varco's robotic oil exploration system [16]. Developers consistently find these approaches to development and maintenance of soft real-time software result in significant developer productivity increases and software maintenance cost savings in comparison with the use of C or C++.

Developing Reusable Hard Real-Time Components

When developers speak of hard real-time software, they generally expect that the software be proven to satisfy all real-time constraints prior to execution. Programmers who write hard real-time software expect their programming environment to support at minimum the following:

1. Standard libraries precisely constrained by worst-case CPU-time consumption and memory usage.
2. Programming language syntactic features for which the worst-case CPU-time and memory usage of the equivalent machine-language translations are precisely constrained.
3. Programming language and/or library services that allow developers to speak in very precise terms regarding the timing constraints imposed on particular real-time software components.
4. Development tools to assist with the analysis of worst-case CPU time and memory requirements for particular

software components.

Additionally, many developers of hard real-time code face additional design constraints beyond the need to prove compliance with real-time constraints. For example, developers face severe CPU time, memory footprint, and battery power constraints; safety certification requirements; marketplace competition that demands challenging functional upgrades; interoperability and maintainability requirements; aggressive completion schedules; and limited development budgets.

Enforcing Static Properties

Java compilers perform more static property analysis to enforce stronger consistency checking within software systems than C and C++ compilers. This stronger consistency checking reduces programming errors, further improving developer productivity. Examples of built-in static property enforcement in Java include the following:

1. Strong type checking prohibits incompatible type coercion between, for example, an integer and a reference type, or between two incompatible reference types.
2. Programmers are prohibited from adding integer values to reference values to create references to new objects.
3. Byte-code verification assures that the types of actual parameters passed to a method invocation exactly match the types of the formal parameters declared for the invoked method.
4. When programmers write code that will throw an exception under certain conditions, the Java compiler enforces that every context that invokes this code is prepared to catch the exception. This reduces the likelihood that error conditions will be overlooked or ignored.

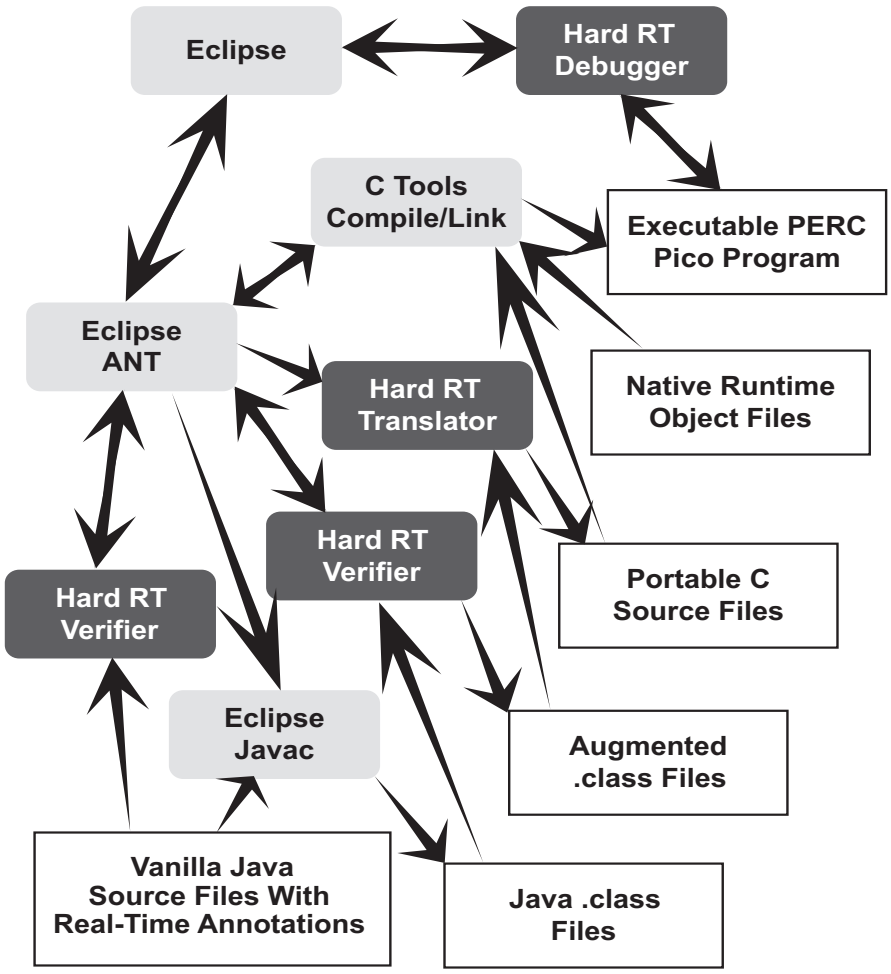
This consistency checking is especially helpful when large software systems are assembled from components independently produced by different teams of developers.

Because of the strong static property checking already enforced by the Java programming environment, Java has attracted the attention of software engineers as a possible language of choice for implementation of hard real-time and safety-critical systems. Hard real-time software includes radar and sonar systems, global positioning systems, software-defined radios, and low-level device drivers for various peripheral devices. Safety-critical software includes anti-lock braking systems in consumer vehicles, fly-by-wire control of flight surfaces in commercial aircraft, automatic shutdown of nuclear power plants, computer-controlled switching systems in passenger railroad systems, and weapons fire-control software.

To address the needs of hard real-time developers, the Open Group is sponsoring a Java community process expert group to establish standards for safety-critical development with the Java language. The Open Group is a vendor and technology-neutral consortium that works to enable access to integrated information within and between enterprises based on open standards and global interoperability. The intention is to produce a standard that is endorsed both by the Java Community Process and by the International Organization for Standardization (ISO). This standard is based on the existing Real-Time Specification for Java (RTSJ) [17]. As a key contributor to this standardization effort, Aonix has drafted a set of guidelines for real-time developers who desire to use the Java language [9]. Programmers who develop their code according to these draft guidelines for development of hard real-time and safety-critical Java can rely on assurances from a special byte-code verifier.

1. The maximum amount of CPU time required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program.
2. The maximum amount of stack memory required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program.
3. Execution of a particular method (or of any overriding method) will not allocate any memory in the shared immortal heap.
4. No blocking operations will be attempted while a thread holds a queue-free priority ceiling emulation lock.
5. Execution of particular methods will not result in throwing of RTSJ-defined CeilingViolationException, DuplicateFilterException, IllegalAssignmentError, InaccessibleAreaException,

Figure 1: *Hard Real-Time Java Development Environment*



MemoryAccessError, MemoryScope Exception, MemoryTypeConflictException, OutOfMemoryError, ScopedCycleException, StackOverflowError, or ThrowBoundaryError exceptions.

Enforcement of these static properties is provided by a special byte-code verifier that enforces more stringent constraints than the traditional Java byte-code verifier.

A proposed integrated development environment is illustrated in Figure 1 (see page 21). This hard real-time development environment builds upon the popular open-source Eclipse integrated development environment. COTS technologies, color coded in light grey, provide the foundation of the hard real-time development environment. Hard real-time plug-ins to the open Eclipse architecture, color coded as dark grey, provide special hard real-time development tool capabilities. The *Hard RT (Real-Time) builder* takes responsibility for determining which parts of a large software system need to be retranslated by the *Eclipse Javac* program and reverified by the *Hard RT verifier* each time a programmer modifies existing source code. Errors detected by either *Eclipse Javac* or by the *Hard RT Verifier* are highlighted within the Eclipse Java-syntax-directed editing window, providing immediate developer feedback, and simplifying the development process.

The functional behavior of hard real-time Java code can be exercised and debugged using a traditional Java 5.0 run-time environment. To test the hard real-time Java code on the target hardware, the verified Java class file is translated by the *Hard RT translator* to C code. Then it is compiled and linked with run-time libraries using COTS C-language development tools. C compilers are available to support all popular embedded processor architectures and real-time operating systems (OSs). The C code generated by the *Hard RT translator* can be loaded directly into read-only memory and can execute in place. The *Hard RT debugger* allows developers to debug the executable hard real-time program using a familiar Eclipse Java debugging environment, even though the deployed code was produced by C-language development tools.

Note that the hard real-time Java development environment translates Java code to C before deployment. This provides much higher performance (up to 3.5 times faster than comparable Java code running with the Sun Microsystems HotSpot compiler), much smaller memory footprint (more than 10 times smaller), and tighter real-time latencies (microseconds vs. tens of seconds) than traditional Java. Comparisons with the performance of handwritten C code reveal that this technology generally produces code that runs within 35 percent (either faster or slower) of comparable handwritten C code. The special hard real-time Java verifier automates the static analysis that must be performed by non-standard, third party tools when using less structured languages like C and C++. Because the hard real-time Java verifier is tightly integrated with other components of the development tool chain, the development and deployment process is much smoother. Information flows automatically between the syntax-directed editor, the Java compiler, the hard real-time byte-code verifier, and the C compiler that produces the final machine-code translation of the original real-time Java program.

"In summary, the hard real-time version of Java allocates all objects on the runtime stack of the current thread rather than using a garbage-collected heap."

When hard real-time Java technologies are used to implement safety-critical systems, the hard real-time Java verifier imposes additional constraints beyond the constraints that are imposed on typical hard real-time software. These additional constraints, motivated by established safety certification guidelines, help enforce that all programmers who contribute to the development of a safety-critical software system adhere to the same conservative development practices.

The most popular alternative approach to development of hard real-time code involves the use of Misra C [18]. In comparison to Misra C, the hard real-time Java approach offers superior portability, scalability, and maintainability. It also provides much easier, more reliable, and more efficient integration with higher level Java software which is typical of the much larger non-real-time and soft real-time layers of many mission-critical software hierarchies. Another key advantage of developing and maintaining hard real-time code with Java tools is that today's graduating software engineering students are much more likely to be proficient in Java than in any other programming language. When compared with Misra C, the key disadvantages of the hard real-time Java approach are that the technology is newer and less familiar to established safety-critical domain experts, and the hard real-time Java approach introduces an additional abstraction layer between source code and deployment. This additional abstraction layer contributes to ease of software maintainability, portability, and software scalability, analogous to abstraction improvements provided by C over assembly language. Some additional effort may be required to trace requirements through source code and multiple intermediate code representations to the final machine code implementation.

Memory for Temporary Objects

Since Java is an object-oriented programming language, all structured data is represented by objects. With a traditional Java run-time environment, all objects are allocated within a region of memory known as the heap, and the memory for these objects is reclaimed by an automatic garbage collector. For hard real-time development, the run-time environment does not include an automatic garbage collector.

The hard real-time Java guidelines allow Java components to allocate objects on the run-time stack using programming constructs similar to those of the C and C++ programming languages. Unlike C and C++, statically enforced programmer annotations guarantee that the use of stack-allocated memory does not create dangling pointers.

In summary, the hard real-time version of Java allocates all objects on the run-time stack of the current thread rather than using a garbage-collected heap. Any running thread may spawn additional threads by dedicating a portion of its stack to represent the run-time stack of the spawned thread. Any objects that need to be shared between multiple threads must be allocated within the stack of some ancestor thread.

The key benefits of safely allocating temporary objects on the run-time stack are the following:

- Temporary memory allocation and

deallocation is very fast.
- Temporary memory allocation is very reliable, because the stack never becomes fragmented, the maximum stack size can be determined through static analysis, and the absence of dangling pointers is guaranteed through the use of enforceable programmer annotations.

For hard real-time software, this gives Java developers capabilities and performance comparable to more traditional languages like Ada, C, and C++.

Synergy Between Java Technologies

The embedded real-time market has been described as a thousand different niches. Each critical software component represents different requirements and economic trade-offs. Figure 2 provides a decision tree to assist system architects in deciding how to implement particular capabilities.

Notably, soft real-time Java development guidelines are generally preferred unless there is a specific constraint that precludes them because soft real-time Java offers the highest developer and software maintenance productivity.

The hard real-time Java technologies do not use automatic garbage collection. Instead, dynamic memory is allocated and deallocated under more explicit programmer control. The safety-critical Java standard supports a subset of the hard real-time Java capabilities.

Note that this decision tree does not distinguish between different security requirements. Security issues are largely orthogonal to real-time issues and are not the focus of this article. Techniques for enforcing multiple independent levels of security are compatible with hard, soft, and safety-critical Java technologies.

Selective Sharing of Control With Traditional Java Components

The recommended approach for providing efficient and reliable integration of hard real-time components with traditional Java components uses a restrictive form of object sharing between the hard real-time and traditional Java domains. The shared objects always reside in the hard real-time domain and do not participate in garbage collection. Since these are hard real-time objects, they are never subject to relocation. This greatly simplifies the implementation and improves execution efficiency.

We describe object sharing as *restrictive* because the traditional Java domain cannot see any of the instance or static variables associated with the hard real-time object. Furthermore, it cannot see the object's regular methods. It can only see methods specially designated as traditional Java methods. These *traditional Java methods* are analogous to OS entry points for application software. In this regard, writing hard real-time Java code is similar to making modifications to an operating system kernel. As with traditional operating system design, greater trust is placed in the implementers of the lower-level OS software, and great care is taken to ensure that errors or malicious intent of application software do not compromise the integrity of lower level components.

In traditional OS design, invocation of kernel services generally crosses a memory protection barrier, and hardware memory management units assure that application code cannot see or modify kernel code and data structures. The restrictive object sharing architecture supports the same abstraction guarantees, but it does so using static bytecode verification techniques which allow much more efficient integration of the hard real-time and non-real-time software. The performance benefits of this architecture have already been demonstrated, for example, in studies conducted by Calton Pu on the Synthesis Kernel [19].

When systems are comprised of a combination of hard and soft real-time components, the hard real-time components will typically run with footprint and throughput efficiency very close to that of optimized C. This represents a three-fold improvement over typical optimized Java performance and footprint. There are many important mission-critical needs that can be addressed by this configuration, such as the following:
- Portable and very efficient device drivers (possibly, but not necessarily, having hard real-time constraints).
- Compared with the use of the Java Native Interface (JNI), interfaces to legacy (*native*) components written in other languages are much more efficient and much safer if implemented using the hard real-time Java technologies as an intermediary between traditional Java and the native code.
- Performance-critical code such as Fourier analysis and matrix manipulation can be provided much more efficiently as hard real-time Java components than as traditional Java code or as legacy code interfaced to Java through JNI.

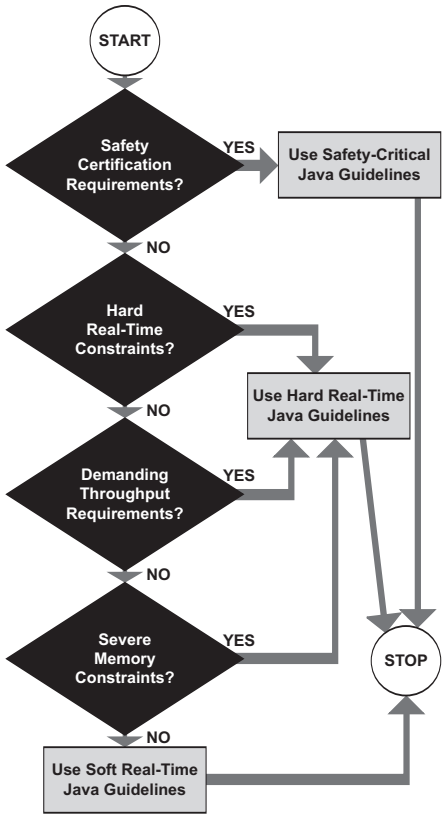
The hard real-time Java approaches described here are only now becoming commercially available, so current experience is limited to research experiments. A number of commercial and defense projects are beginning development based on these technologies. Information should be available in another year or so.

Using Off-the-Shelf Components in Real-Time Systems

The key to using traditional off-the-shelf Java components in real-time systems involves careful partitioning of capabilities to ensure that reliable operation of real-time components is not compromised by the less-disciplined behavior of non-real-time components. System architects and integrators need to evaluate which partitioning approaches are most appropriate for each particular application's requirements. Among the practices in common use today are the following:

1. Structure the application so non-real-time code runs in a different VM (possibly even on a different processor) than real-time code. By isolating the non-real-time code within a distinct VM, it is possible to enforce strict memory budgets and limit the sched-

Figure 2: *Decision Tree for Selecting Between Alternative Java Technologies*



uling priorities at which the non-real-time components consume CPU time.
2. Allow selected non-real-time components to run on the same VM as more carefully constructed soft real-time components only after thorough testing of the non-real-time software to establish sufficient confidence that the resource needs of this non-real-time software are well understood.
3. Take extra care to assure that real-time software does not have to wait indefinitely for interactions with non-real-time software. For example:
   a. Run the non-real-time software within a different VM than the real-time software.
   b. Avoid blocking on synchronization for objects that are shared between non-real-time and real-time components.
   c. OR make use of alternative information sources (e.g. approximations) whenever a non-real-time component does not deliver critical information to the real-time component within the window of time that is required in order to satisfy end-to-end timing constraints.
4. Run components with the most stringent real-time constraints within a hard real-time environment at priorities higher than all soft real-time components. The hard real-time environment enforces strict memory partitioning to prevent memory contention from non-real-time components running in the traditional Java VM environment.

These are the approaches that have been successfully deployed in the various projects mentioned in [11] through [16].

Conclusions

By restricting the use of Java programming language features and libraries, and by exploiting special static analysis tools, it is possible to apply many of the benefits of the Java programming language to the specialized domain of real-time software. Standards to support these development approaches are being developed under the auspices of the open group, which is working to establish standards that can be endorsed both by the Java Community Process and by ISO.

By carefully partitioning functionality between high and low-level software, it is possible to leverage the best strengths of Java within each respective programming domain. Lower-level Java technologies are most appropriate for implementing low-level device drivers, interrupt handlers, safe and efficient interfaces to legacy (*native*) code, and certain performance-critical components such as Fast Fourier transforms. Higher-level Java technologies are most appropriate for larger, more complex functionality, especially subsystems that need to support dynamic reconfiguration and/or generic reuse of existing off-the-shelf Java software components.◆

References

1. Moore, G. "Cramming More Components Onto Integrated Circuits." Electronics Magazine Apr. 1965.
2. Bourgonjon, R. "The Evolution of Embedded Software in Consumer Products." International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, 1995.
3. McMillan, R. "GM CTO Sees More Code on Future Cars." InfoWorld Oct. 2004.
4. Jensen, E.D. "Overview of Fundamental Real-Time Concepts and Terms." Real-Time 19 Feb. 2007 <www.real-time.org/realtimeoverview.htm>.
5. "Software Considerations in Airborne Systems and Equipment Certification." RTCA/DO-178B. RTCA, Inc., 1992.
6. Dios, Chen, et al. "An Empirical Study of Programming Language Trends." IEEE Software 22.3 (2005).
7. Nilsen, K. "Quantitative Analysis of Developer Productivity in C vs. Real-Time Java." Defense Advanced Research Projects Agency Workshop on Real-Time Java, 13 July 2004, Arlington, TX: Aonix Research and Development, 2004.
8. Phipps, G. "Comparing Observed Bug and Productivity Rates for Java and C++." Software Practice and Experience 29.4 (1999): 345-358.
9. Nilsen, K. "Guidelines for Scalable Java Development of Real-Time Systems." Aonix, 2006. <http://research.aonix.com/jsc/rtjava.guidelines.3-28-06.pdf>.
10. Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly Typed, Statically Checked, Concurrent Language." CSL-84-7. Xerox Palo Alto Research Center, 1985 <www.parc.xerox.com/about/history/pub-historical.html>.
11. "Lockheed Martin Selects Aonix PERC Virtual Machine for Aegis Weapon System." Space War 13 Oct. 2006.
12. "Boeing Selects Software for J-UCAS X-45C." Defense Industry Daily 31 Oct 2005.
13. "Case Study: Centralized Network Element Platform, Calix Networks." Aonix <www.aonix.com/pdf/PERC_CalixSuccess.pdf>.
14. McHale, J. "Wearable Computers and the Military: The Smaller the Better." Military and Aerospace Electronics Nov. 2005.
15. "Case Study: Optical Network Switch, Nortel." Aonix <www.aonix.com/pdf/PERC_NortelSucces.pdf>.
16. "National Oilwell Varco Selects Aonix PERC for Java-Based Robotic Drilling." Rigzone 25 Sept. 2006 <www.rigzone.com>.
17. Bollella, G., et al. The Real-Time Specification for Java. Addison-Wesley, 2000.
18. "MISRA-C: 2004 – Guidelines for the Use of the C Language in Critical Systems." MISRA, 2004.
19. Pu, C., H. Massalin, and J. Ioannidis. "The Synthesis Kernel." Computing Systems 1.1 (1988): 11-32.

About the Author

Kelvin Nilsen, Ph.D., is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and COTS safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. In 2003, Aonix acquired NewMonics. Nilsen has a bachelor's degree in physics from Brigham Young University and a master's degree and a doctorate in computer science from the University of Arizona.

Aonix
5930 Cornerstone Court West
STE 250
San Diego, CA 92121
Phone: (801) 756-4821
Fax: (801) 756-4839
E-mail: kelvin@aonix.com