



White Paper

High-Level Language Support
for
High Performance Real-Time Java Components

By Dr. Kelvin Nilsen,
CTO, Atego

May 15, 2009

High-Level Language Support *for* *High Performance Real-Time Java Components*

This white paper discusses the rationale for a proposed “Scalable Java” approach to hard real-time software development.

Programming language design and implementation has progressed significantly since introduction of the first digital computers. The first programming languages reduced the burden of detail to simplify the effort associated with the creation of new software. More recently developed programming languages have pursued increasingly aggressive objectives.

Today’s modern programming languages address a variety of software engineering challenges:

- **Abstraction:** high-level programming environments encourage good abstraction by allowing programmers to say more with fewer words. The use of high-level abstraction allows programmers to ignore low-level implementation details when dealing with higher level conceptual issues.
- **Reliability:** modern programming environments improve reliability by enforcing guidelines that reduce the likelihood of programming errors, by reducing the amount of code that developers need to author for each application.
- **Separation of Concerns:** as typical software application sizes grow, the challenge of coordinating the individual contributions of large numbers of software developers grows. Modern programming languages provide facilities to allow developers to isolate their code from other code in the system. This capability, sometimes described as information hiding or data encapsulation, assures programmers that bugs relating to execution of their code are not caused by other components.

Introduction

- **Ease of Maintenance:** modern programming languages facilitate ease of maintenance by assuring that code is portable across different CPU architectures and operating systems, and by offering abstraction facilities that are easy for software engineers to understand.
- **Scalability:** typical software projects begin small and expand over time. Many software projects track Moore's law, doubling in size every 18 to 24 months. Programming languages support scalability by assuring high-level abstraction, separation of concerns, and ease of maintenance. Additionally, they encourage programming practices that support modular composition of components. This assures that individual components always behave the same, regardless of what other components they might be combined with.

It is especially difficult to support scalability for the development of real-time software because component A may consume memory or CPU-time resources that are required by component B in order for it to satisfy its real-time constraints. This is why special programming abstractions are required to facilitate composition of real-time software components.

This white paper discusses the rationale for a proposed "Scalable Java" approach to hard real-time software development. The proposed real-time development guidelines, described more completely in reference K. Nilsen. "Draft Guidelines for Scalable Java Development of Real-Time Systems." March 2005, are structured as a profile of JDK 5.0, extended by the Real-Time Specification for Java (RTSJ). The driving objectives for this effort are to assure:

- Developers of real-time Java components are twice as productive as developers of real-time components in C or C++, matching the productivity benefits of Java vs. C and C++ for traditional (non-real-time) code.
- Maintainers of real-time Java components are five to ten times as productive as maintainers of C and C++ code, matching the productivity benefits of Java vs. C and C++ for traditional (non-real-time) code.
- Generality of the real-time Java technology, supporting the full spectrum of requirements of developers responsible for device drivers, operating system services, and hard real-time functionality.
- Efficiency of implementation that matches C and C++, running over three times faster than traditional Java implementations.

Reliability

Typical mission-critical systems are deployed in a variety of configurations. The software must run reliably in every configuration. Over the life span of a typical embedded software component, that component is typically required to run on a variety of different processor architectures and many different real-time operating systems as well as on multiple versions of architectures and real-time operating systems. Note that the CPU-time and memory resources required for reliable operation of software differ for each configuration. Because errors in the configuration of software for each execution environment may result in catastrophic failures, reliability is improved if the resource requirements to run the software reliably in each configuration is determined automatically.

Typical real-time software development methodologies require substantial software engineering involvement during software configuration to achieve reliable operation. In particular, since each underlying real-time operating system differs in the semantics of scheduling, synchronization, enforcement of deadline and resource overruns, worst-case latency, and jitter behavior, software engineers must carefully characterize the differences between each “target” platform and must determine appropriate configuration parameters to make the code work reliably on each. The natural tendency is for programmers to test their code on a particular test platform and to hard-code the behavior of that platform into the component’s implementation. Various assumptions permeate the software implementation. This creates reliability problems whenever the component must run on platforms that differ from the originally targeted environment.

As well, real-time developers must determine the CPU time and memory requirements for each real-time component on each targeted platform. Typical real-time development environments provide no support for automatic analysis of these resource requirements. Developers tend to make certain assumptions regarding resource requirements, perhaps based on testing on a particular platform, and to hard-code these assumptions into their real-time source code. This creates reliability problems whenever the program must run on platforms that require different resources than what the programmer assumed.

In addition to the reliability problems that result from incompatibilities between real-time platforms, another source of reliability problems derives from the high probability of human error during the analysis of resource requirements. These analyses are very difficult and tedious. In a developer’s ideal world, this analysis would be performed by a computer. Unfortunately, many of the analyses that must be performed to assure reliable operation of real-time software components represent theoretically undecidable problems, which means the analysis cannot be performed automatically.

Generality

Examples of undecidable analysis problems are made decidable by the restrictive guidelines of the Scalable Java approach to development and maintenance of real-time software are:

1. Determining the memory resources and CPU time required to reliably run a particular software component on a particular platform.
2. Proving that a particular software component will not violate referential integrity rules by guaranteeing the absence of dangling pointers and a very broad class of memory leaks.
3. Proving that discarded temporary memory objects are reclaimed in a timely manner and that the resulting memory allocation pool suffers zero memory fragmentation.
4. Proving the absence of synchronization which might result in priority inversion or deadlock.

The Scalable Java proposal addresses these problems by:

1. Carefully defining the precise semantics for a reliable, portable, and efficient subset of the full real-time extended Java platform,
2. Establishing standard notations to allow programmers to represent resource requirements for particular components within their Java source code,
3. Providing tools that enforce consistency with respect to component resource requirements between independent components, and
4. Providing tools that automatically analyze the CPU time and memory resource requirements of particular software components.

Generality

With proper enforcement of encapsulation abstractions, the use of real-time Java technologies significantly reduces the costs of maintaining software and improves the reliability of deployed real-time systems. During the 9 years that Atego has been commercially supporting the use of Java for real-time development, our customers have consistently reported that, compared with the use of C++, Java developers are approximately twice as productive when developing new code and roughly five to ten times as productive when maintaining existing code.

Even though our customers consistently speak favorably of the benefits of the Java language, many have found that Java lacks the generality required to support 100% of their development needs. According to feedback received from customers, some of the specific reasons why they have found it necessary to use C and C++ for parts of their complex systems are (1) optimized C code runs up to 3 times faster than Java code for certain important functionality; (2) the minimal C footprint is tens of Kbytes, whereas the

Generality

minimal Java footprint is multiple Mbytes; (3) the incremental footprint for new C code and new C objects is one half to one third the incremental footprint for new Java code and new Java objects; (4) Java (even RTSJ Java) does not support certain low-level operations that are required in the implementations of physical interrupt handling and device drivers; and (5) Java latency and determinism is not sufficient to meet the needs of particular real-time constraints.

Among Atego customers, the most noteworthy example of these trade-offs is the software for Nortel's Optera HDX long-haul fiber-optic switch. The management plane software for this product consists of approximately one million lines of Java code running on the PERC virtual machine. The control plane software consists of nearly four million lines of C code. In discussing their experiences with this mixed-language approach, Nortel engineers reported:

1. The Java developers were more productive and their code much more trouble free than C developers, and than the C++ developers who implemented the management plane software for the previous version of this product.
2. Based on their successes with Java, they plan to use higher percentages of Java in future versions of this and similar products.
3. The greatest weakness of the mixed-language approach has been along the interface between Java and C code. Java itself provides excellent self-integrity enforcement as long as all code is written in Java. However, when the JNI protocol is used to combine C code with Java code, the integrity enforcement is compromised. This is where most of the Java bugs were introduced, and these were very difficult bugs to isolate and correct.
4. Nortel wishes they could develop all of the control plane software in Java, but standard Java lacks the required generality.

We believe the community of real-time developers can benefit from the many years of experience of PERC customers on projects like the Nortel Optera HDX. Based on feedback from this and similar customers, the Scalable Java proposal addresses generality requirements by:

1. Assuring that the real-time Java subset can offer performance, latency, determinism, and memory footprint that are comparable to C and C++,
2. Establishing standard real-time Java extensions to allow portable development of device drivers and interrupt handlers, and
3. Enabling robust and efficient coordination between vanilla Java components, soft real-time Java components that depend on reliable operation of a real-time garbage collector, and hard real-time Java components that operate without automatic garbage collection in complex mission-critical software hierarchies.

Maintainability With many mission-critical systems, fielded software must endure many years, often multiple decades. During its useful lifetime, this software evolves in response to changing platform requirements, new communication protocols, integration of new functionality, and so forth. Over the lifetime of a particular system, the costs of software maintenance far exceed the costs of the original software development. Typical maintenance activities include bug fixes, performance improvements, and functional enhancements.

Maintaining real-time software is particularly difficult because the declared interfaces for individual components do not reflect all of the conditions required for reliable composition of components. The developer changing existing software cannot determine by looking at the component interface what rules he must follow for his changes to integrate reliably with other existing software. In particular, he does not know:

1. Whether incoming arguments might refer to temporary objects or permanent objects, and whether the referenced objects might be shared with other threads,
2. Whether memory resources have been budgeted to allow the implementation of a particular service to allocate permanent or temporary objects,
3. Which memory allocation budgets must be increased in order for this revised component to be able to reliably allocate additional memory,
4. Which memory allocation budgets may be decreased in order to make this component run more efficiently, and
5. Which task cost estimates must be modified if changes to this component affect its CPU time requirements.

Maintainers of real-time software must search for all invocations of the components they are trying to modify, and must determine through analysis of that source code what assumptions that software may be making regarding the implementation of this component. For the most part, this approach is impractical.

Scalability

Scalability

Scalability is a generalization of maintainability. Most modern software systems experience evolutionar change that tracks Moore’s Law. As processors and computer memory decrease in cost and increase in capacity, software grows in size and complexity to match the new capacity. Studies of certain commercial embedded software systems have observed that it is common for software size to double every 18 to 24 months!

Java itself has shown tremendous strengths as a platform to support easy integration and economical scalability. This is because all of the Java software is very portable, and because strong object-oriented abstractions mean that independently developed components integrate cleanly, without compromising the integrity of each other’s encapsulation boundaries. Unfortunately, typical real-time components do not offer this same promise. Given two independently developed real-time components, a system integrator would like to know whether it is possible for one component to invoke the methods of another component by studying the interface definitions of the two components. As discussed under the Maintainability heading above, the interface requirements for typical real-time components are not clearly identified. System integrators must carefully scrutinize the implementation of the methods they desire to invoke to determine whether it would be reasonable to invoke those methods from each particular context.

The Atego Scalable Java proposal addresses the Maintainability and Scalability issues by:

1. Maintaining real-time software as “Vanilla Java” source code with JDK 5.0 style meta-data annotations to document the interface requirements associated with each software component, and
2. Providing automatic consistency checking between independent interfaces, assuring that each method invocation satisfies the annotated interface requirements, and
3. Providing automated analysis to determine memory and CPU-time resource requirements to allow automatic configuration of resource budgeting and real-time scheduling each time any system component is modified.

Efficiency

Responding to common misconceptions in the marketplace, real-time scientists often emphasize the fact that “real time is not real fast”. Their point, of course, is that the real-time practitioner is more concerned about predictable compliance with deadline constraints than about getting all work done in the shortest possible time. Even so, experience demonstrates that efficiency concerns cannot be totally ignored. Given two possible approaches to satisfying real-time constraints which are equivalent in every respect except efficiency, almost any business manager would choose the approach that is more efficient. This approach can be deployed less expensively, and the excess CPU and

Efficiency

memory capacity increases future expansion opportunities.

Throughout the nine years that we have been supporting the ability to run Java programs that comply with various real-time constraints, we have frequently encountered situations in which developers desired to exploit the high-level benefits of developing with the PERC platform, but were not able to do so because the resource requirements were too high. They found, for example, that optimized Java throughput was over three times slower than C for certain performance-critical components. And they found that the memory required to deploy Java software far exceeded their hardware budgets.

Our experience with various previous real-time Java technologies, both the PERC product and products developed by other vendors, reveals that these technologies actually run even slower than traditional non-real-time Java. This is because various performance tradeoffs have been made to assure predictable execution. One of the key considerations in our proposed standard for high-level high performance Java is the ability to significantly improve upon the performance and memory requirements of traditional Java.

Experimental evaluation of an Atego RTSJ-based translation technology prototype demonstrated the ability to run real-time Java code at over three times the speed of Sun's HotSpot compiler and up to 20% faster than equivalent C code, as shown in Relative Performance of Prototype Translation System.

Benchmark	Vs. C	Vs. HotSpot
Sieve (standalone)	120%	307%
Sieve (CaffeineMark)	90%	148%
Loop (CaffeineMark)	114%	326%
Logic (CaffeineMark)	65%	148%
Method (CaffeineMark)	71%	108%

Table 1: Relative Performance of Prototype Translation System

Footprint is also comparable to C code.

Integrated Development Environment

Integrated Development Environment for Scalable Java

The Atego Scalable Java proposal is based on technology already under development by Atego. The planned development environment consists of the components illustrated in Real-Time Java Translation Environment. In this illustration, the blue components are components that are currently under development at Atego. The others are part of the existing open-source Eclipse development environment. Using this technology, real-time Java developers will be able to use the powerful Eclipse development environment to create real-time Java software components. When a developer saves files, the Eclipse build system will coordinate with the JRTK builder to determine which components need to be processed. Processing consists of invoking the Eclipse Javac compiler and running the JRTK verifier. The JRTK verifier processes the real-time annotations and assures consistency between annotations and implementations. If either the JRTK verifier or the Eclipse Javac compiler finds problems in the Java source code, these problems will be highlighted in the source code and added to the Eclipse task list.

This figure illustrates two possible translation paths. The current focus of Atego is to enable use of the JRTK translator. Experimentation with a preliminary incomplete implementation of this tool proves that JRTK code can run at roughly the speed of C, over three times faster than the HotSpot compiler for certain Java code. It is also possible to translate the augmented class files into generic class files capable of running on any compliant RTSJ system with appropriate libraries and sufficient memory and CPU capacity.

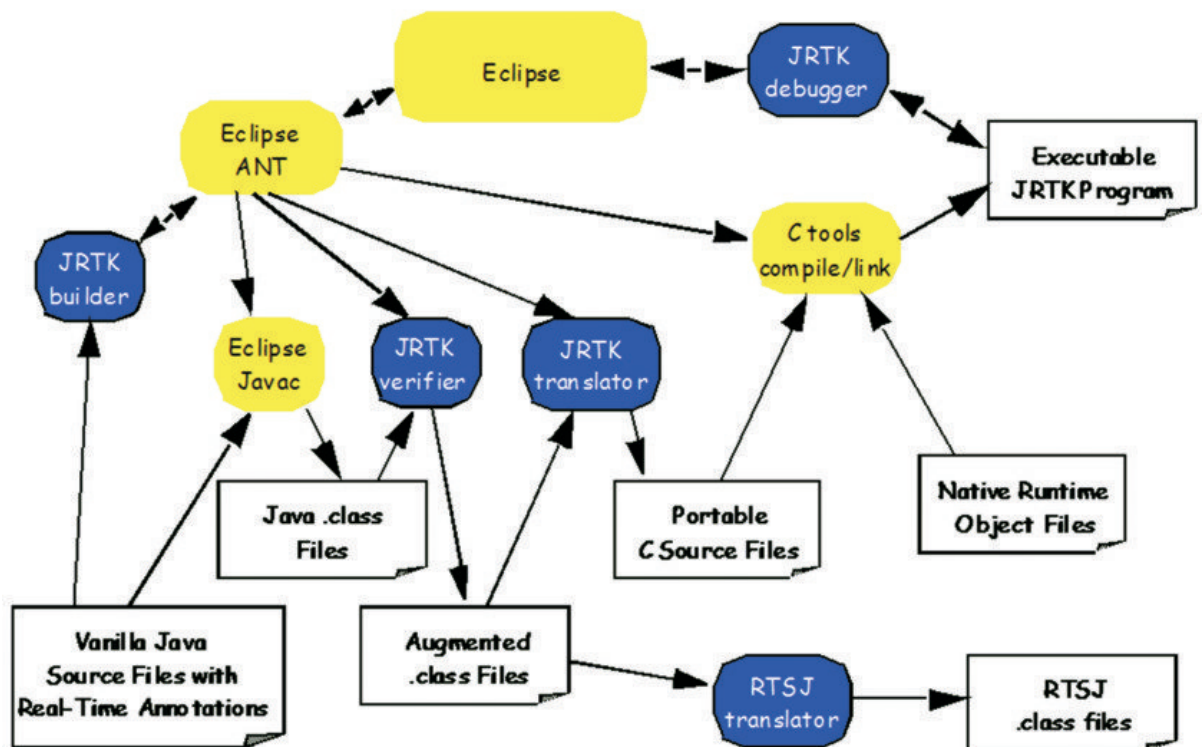


Figure 1: Real-Time Java Translation Environment

Bibliography

Bibliography

1. Sun Microsystems Inc., The Java Language Environment: A White Paper. 1995, Sun Microsystems, Inc.: Mountain View, CA. (<http://java.sun.com/docs/white/langenv/>)
2. P. Rovner. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed Statically Checked, Concurrent Language", CSL-84-7, Xerox Palo Alto Research Center. 1984. (See <http://www.parc.xerox.com/about/history/pub-historical.html>)
3. K. Nilsen. "Issues in the Design and Implementation of Real-Time Java", Real-Time Magazine. March 1998. (http://www.realttime-info.be/magazine/98q1/1998q1_p009.pdf)
4. K. Nilsen. "Adding Real-Time Capabilities to the Java Programming Language." Communications of the ACM. Vol. 41, no. 6, pp. 49-56, June 1998. (<http://doi.acm.org/10.1145/276609.276619>)
5. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull. The Real-Time Specification for Java. 195 pages. Addison-Wesley Publishing Company. January 2000. (<http://www.rtg.org/>)
6. "Real-Time Core Extensions." 154 pages. J Consortium. September 2000. (<http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf>)
7. K. Nilsen, A. Klein. "Issues in the Design and Implementation of Efficient Interfaces Between Hard and Soft Real-Time Java Components." Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems. Springer-Verlag. November 2003.
8. K. Nilsen. "Doing Firm-Real-Time With J2SE APIs." Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems. Springer-Verlag. November 2003.
9. M. Klein, T. Ralya, B. Pollak, R. Obenza. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. 712 pages. Kluwer Academic Publishers. November 1993. (<http://www.sei.cmu.edu/publications/books/other-books/rma.hndbk.html>)
10. Calix Success Story, 2 pages, May 2003. (http://www.Atego.com/pdf/PERC_CalixSuccess_e.pdf)
11. Nortel Success Story, 3 pages, October 2003. (http://www.Atego.com/pdf/PERC_NortelSuccess_e.pdf)
12. K. Nilsen. "Making Effective Use of the Real-Time Specification for Java." Oct. 2004. (<http://research.Atego.com/jsc/rtsj.issues.9-04.pdf>)
13. K. Nilsen. "Draft Guidelines for Scalable Java Development of Real-Time Systems." March 2005. (<http://research.Atego.com/jsc/rtjava.guidelines.3-26-05.pdf>)
14. K. Nilsen. "Questions and Answers Regarding Proposed Static Analyzable Memory Model." Oct. 2004. (<http://research.Atego.com/jsc/jsc.mem.model.qa.pdf>)

To obtain more information, please contact Atego at www.atego.com or call one of our sales offices

North America

Phone: (888) 91-ATEGO
Fax: (858) 824-0212
E-mail: info@atego.com

United Kingdom

Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 575033
E-mail: info@atego.com



France

Phone: +33 (0) 1 4146-1999
Fax: +33 (0) 1 4146-1990
E-mail: info@atego.com

Germany

Phone: +49 7243 5318-0
Fax: +49 7243 5318-78
E-mail: info@atego.com

© 2010 Atego. All rights reserved. Atego™ is a trademark of Atego. PERC® is a registered trademarks or service mark of Atego. Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. All other company and product names are the trademarks of their respective companies.