

Making Effective Use of the Real-Time Specification for Java

Kelvin Nilsen, CTO, Aonix

Java as a high-level programming language for enterprise and network services applications has been extremely successful. Due to its success in traditional information technology domains, many people have questioned whether Java could also be applied to the domains of real-time development. An effort began in late 1997 as JSR-1 to define a real-time specification for Java. This culminated on Nov. 12, 2001 with the official release of The Real-Time Specification for Java (RTSJ), Version 1.0.

Since that date, only one complying commercial implementation of the RTSJ every reached the market. That product, JTime, has now been withdrawn from the market. Rumors are that the total number of licensees was less than ten, and that none of these licensees progressed beyond the stage of experimentation and pilot studies. There are various partial implementations and uncertified implementations currently available or under development, including:

- A clean-room non-certified product developed by Aicas, a small startup company in Germany
- A partial implementation of RTSJ based on IBM's J9 product
- An ongoing effort to implement RTSJ extensions for Solaris and HotSpot performed by Sun
- An open-source non-certified implementation named jRate, implemented as Angelo Corsaro's Ph.D. research
- An open-source non-certified implementation named OVM, implemented as a collaborative effort between Purdue University, the University of Maryland, SUNY Oswego, and DLTech
- The official reference implementation, provided by TimeSys

1. The Ideals of Real-Time Java

In the traditional information technology domain, Java has demonstrated developer productivity improvements of approximately two-fold over use

of the C++. Once a large software system is built, it enters into a maintenance phase. The maintenance phase typically consists of porting the code to new processors and new operating systems, adding increments of new functionality to the system, and integrating the code with other independently developed software components. Maintenance of Java software is typically measured to require one fifth to one tenth the effort associated with maintenance of code written in C++. There are many reasons why Java software contributes to more efficient software development and maintenance:

- Stronger compile-time type checking reduces programmer errors, especially at the interface between independently developed components.
- More pure object-oriented encapsulation reduces interference between independently developed components.
- Forced exception handling allows programmers to enforce that exceptional conditions are not overlooked or ignored.
- Automatic garbage collection simplifies memory management and reduces programmer errors. During maintenance and integration, automatic garbage collection obviates the need to assign responsibility for reclaiming memory allocated in one component and used by other components.
- Since the Java language is much simpler than C++, it is less likely to be misunderstood and misused by programmers.
- Improved portability allows programmers to master the language, rather than requiring them to independently master each tool chain and RTOS.
- Improved portability also eliminates the need to port components to each new hardware platform or operating system.

The vision of real-time Java is several fold:

1. Ideally, development of real-time programs in Java would be twice as efficient as developing

real-time programs in more traditional languages like C and C++.

2. Likewise, maintenance of real-time programs in Java would be five to ten times as efficient as maintaining real-time programs in more traditional languages like C and C++.
3. Additionally, there is a hope that the use of Java for real-time programming will be able to better leverage the mass-market appeal of Java than is typical with traditional approaches to real-time programming using C, C++, and Ada in combination with various proprietary or commercial real-time operating systems. In particular, there is a desire to exploit off-the-shelf Java developer tools, off-the-shelf reusable software components, and university-based education resources. These mainstream resources generally offer better quality for lower costs than is traditionally available in the niche embedded and real-time marketplaces.

Many of these ideals have already been realized in the soft real-time domain with the PERC virtual machine available from Aonix. This virtual machine supports version 1.3 J2SE-compatible libraries with accurate, paced, defragmenting real-time garbage collection. Many PERC-based products have been commercially deployed in markets including network infrastructure equipment, manufacturing automation, power plant control, and C2. In these products, PERC virtual machine software provides the foundation for soft real-time mission-critical applications with task deadlines ranging from a single ms to approximately 50 ms. Several of these products have successfully demonstrated 5 9's operation in multiple years of commercial service. Development teams who have used these technologies report productivity benefits comparable with those of traditional Java. That's because developing with PERC is essentially developing with standard mainstream J2SE Java. PERC developers enjoy all of the benefits of traditional Java, including the use of standard Java Integrated Development Environments and debuggers, the use of run-time performance profilers, the use of standard J2SE libraries, and access to the huge assortment of J2SE COTS components.

Unfortunately, the RTSJ as currently specified, has not been able to deliver these same benefits to developers of hard real-time software. Among the reasons, we have identified a number of specific shortcomings in the RTSJ specification. We summarize these general areas of weakness immediately below. Each point is discussed in greater detail in subsequent sections of this white paper.

Lack of portability: The RTSJ was designed to run on a variety of different operating systems, supporting a breadth of different real-time scheduling and synchronization mechanisms. In an attempt to deliver to real-time programmers high-level abstractions consistent with the sorts of high-level abstractions that have made Java popular for information technology programming, the RTSJ includes many features that are not commonly provided by modern real-time operating systems (workload feasibility testing, deadline overrun detection, CPU-time overrun detection, asynchronous event handlers). Unfortunately, the specifications of these services are so vague that it is not reasonable to assume every RTSJ implementation will behave the same. Thus, programmers who care about the precise real-time semantics of these services are forced to target a particular implementation of the RTSJ rather than targeting the RTSJ specification itself. This lack of portability also hinders component-based software development because components are not interchangeable between different RTSJ implementations.

Failure to support scalable development: Scalability refers to the notion that small software systems can easily grow into much larger software systems. Scalability of embedded real-time software is very important because the embedded software system analog of Moore's law has demonstrated that the amount of code deployed in a given product family (e.g. television, stereo system, in-vehicle telematics device, smart sensor, or network router) approximately doubles every 18 months. To support scalable software development, it is essential that all of the important characteristics manifest by software components *A* and *B* in isolation are preserved when these components are combined into a larger system. Composability is especially challenging in resource-limited real-time systems, where component *A* may consume

the 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. Unfortunately, the RTSJ does not provide the necessary abstractions. In fact, several of the RTSJ abstractions stand in direct contradiction to the goal of scalable software development.

Performance and footprint issues: Recent trends are to push as much code into higher layers of the abstraction hierarchy as possible. This is because it is much easier to develop code using, for example, full J2SE-style Java rather than the more restrictive subset of Java that is capable of guaranteeing hard real-time performance. In general, the typical reasons code is moved outside the traditional Java domain are because (1) the code has tight hard real-time constraints, (2) it has demanding throughput requirements, or (3) it has severe memory constraints. Unfortunately, RTSJ-style Java is only able to address the first of these requirements. In the best of circumstances, its throughput is slower than traditional Java, which is roughly a third the speed of optimized C. And the memory required for an “optimized” RTSJ implementation is significantly larger than the memory required for the Java platform it extends (J2ME, J2SE, or J2EE), which itself is much larger than the comparable C footprint. Because of these efficiency considerations, RTSJ is only able to address a subset of the low-level software needs of a typical system configuration. This means programmers of low-level software are forced to divide the functionality between components written in Java and those written in C or assembler. This adds considerably to the complexity of the system software and compromises the strong integrity checking that is only available if the entire application is implemented in Java.

Lack of expressive generality: While the RTSJ has an incredible amount of flexibility, it is surprisingly lacking in certain important capabilities. At the high end, for example, it lacks the ability to establish garbage collection pacing parameters. At the other end, it lacks the ability to write interrupt handlers and low-level device drivers. Between these extremes, there are various other missing capabilities.

Dealing with Shortcomings of the RTSJ. Even before the RTSJ was developed, and before the NIST meetings in which requirements for real-time Java were originally gathered, NewMonics (subsequently acquired by Aonix) had developed and was marketing a clean-room virtual machine which provided highly reliable soft real-time support for embedded mission-critical Java systems. This product, which is based on the J2SE standard, is the only real-time Java product to have fielded commercial deployments. Various successfully deployed products in fields spanning market segments such as network infrastructure equipment, manufacturing automation, and C2, have proven the developer productivity benefits and the reliability of this approach to soft real-time development. Given the commercial success of the PERC product family, our perspective is that the sorts of real-time extensions required for soft real-time systems are very different than the extensions that best serve hard real-time developers. In the soft real-time domain, it is important to standardize the pacing of garbage collection, to expose performance monitoring information and tuning parameters, to support high-level real-time thread schedulers, and to provide standard mechanisms to enforce time and memory budgets for particular software components. These are different extensions than are required for low-level hard real-time components. Those need small memory footprint, high performance, guaranteed low latency, low-level abstractions to support hardware I/O and interrupt handling, and simple operation to make practical the creation of safety certification artifacts.

When NewMonics and Aonix set out to establish “extensions” for real-time Java within the J Consortium, we focused our efforts on the needs of the hard real-time developer. We considered that the needs of soft real-time developers were already fairly well satisfied by technologies such as the PERC virtual machine. Unlike the RTSJ, the Real-Time Core Extensions specification developed by the J Consortium focuses on the following objectives:

1. Achieving memory footprint, performance throughput, and real-time latency comparable (within 5-10%) to C and modern RTOS.

2. Bringing key benefits of Java, such as portable, scalable development, to the domain of of hard real-time developers.
3. Making it possible to implement all layers of a mission-critical system using appropriate combinations of available real-time Java technologies, including even the device drivers and interrupt handlers.
4. Providing efficient and safely partitioned integration of hard real-time Java components, legacy components written in C or C++, soft real-time Java components, and non-real-time Java components. (JNI was considered neither efficient nor safe.)

It seems that the designers of the RTSJ viewed most of these objectives as secondary nice-to-have features. Their focus, however, was on simply making it possible to write real-time programs in Java. A survey of “research papers” on RTSJ makes very clear that the questions currently being addressed by RTSJ researchers focus primarily on real-time latency, and secondarily on performance. The much more ambitious objectives of the J Consortium’s Real-Time Core effort are only touched in passing, if at all.

Aonix has spent many years lobbying the RTSJ expert group in an attempt to influence evolution of the RTSJ towards a standard that would be more relevant to our targeted markets. So far, the committee that is in charge of maintenance of the RTSJ has been slow to address the issues we have raised. In many cases, they have not even acknowledged that these issues need to be addressed. In other cases, when we request specific enhancements to the RTSJ, members of the expert group tell us we should simply extend the base classes and add the desired functionality as proprietary extensions. Of course, following this advice would contradict our desire to support portable and scalable development. It is our belief that if real-time Java cannot offer portable and scalable development, there is not a sufficient motivation for developers to switch to Java from their legacy approaches that are most likely based on C and C++.

While we remain hopeful that a future version of the RTSJ specification will be better suited to the

needs of hard real-time developers, we believe it is necessary for now to work within the confines of the RTSJ as it is currently specified. Our recommended approach is to establish several RTSJ profiles, each targeted to the needs of distinct market segments. Each profile starts with a limited subset of all RTSJ features, carefully constrains the semantics of the supported RTSJ features, and establishes certain standard extensions that must be supported in each conforming implementation of the profile. These extensions include programming style guidelines enforced by static analysis tools, certain developer tools, and carefully defined libraries. We believe that it is possible, through careful definition and standardization of these profiles, to address critical market requirements. Even though programmers who use all of the features of RTSJ cannot expect deployment efficiency comparable to C, portable components, scalable development, the ability to write interrupt handlers, nor safe efficient partitioning of components written for different layers of the system hierarchy, programmers who use particular RTSJ profiles can.

The remainder of this document calls out specific weaknesses of the RTSJ specification and makes recommendations for how these weaknesses should be addressed in particular profiles. The notational convention we use throughout this document is to present our recommendations as indented italicized paragraphs such as this one.

2. Lack of Portability

One of the most important contributions of Java to the information technology market is portable deployment of application software byte-code binaries. Portability is what makes it possible to deploy the same software to run on Macintosh, Windows PC, Solaris, and Linux. It makes possible the creation of 3rd party COTS software targeted to the Java platform. This is also key to enabling embedded developers to use workstation computers with fast processors and abundant memory to develop and test their embedded software, and then deploy that software on slower processors with more limited memory.

Real-time developers concern themselves with a broad class of issues that traditional developers are free to ignore. To support portable deployment of real-time software, this broad class of issues must be handled in a portable way across different virtual machine implementations. In this section, we describe thirteen portability issues that must be considered by developers who are targeting the RTSJ “platform”. Assuming that each of these 13 issues could be resolved in one of two ways, there would be $2^{13} = 8,192$ different RTSJ configurations that would have to be considered as the developer attempts to deliver portable real-time software. Since in fact there are many more than two distinct ways to resolve certain of these portability issues, it is clear that the number of distinct RTSJ configurations is far greater. We view this as a significant impediment to widespread adoption of real-time Java technologies. Our recommendation for addressing this problem is for industry to standardize on a small number of complementary RTSJ profiles. By complementary, we mean to suggest that many of the same development tools could be applied to different profiles, and code targeted to one of the profiles would be upwards compatible with certain other profiles.

2.1. Plug-in Schedulers

The RTSJ leaves placeholders to allow each RTSJ implementer to extend the RTSJ in its own proprietary way, adding custom-tailored schedulers to complement the standard base scheduler. Real-time programmers who make any use of these scheduler extensions must recognize that any such usage is non-portable.

Each standard profile must precisely describe the set of scheduling extensions, if any, that are supported by that profile. Besides naming the alternative schedulers that are available in the profile, the standard must describe how the alternative scheduler interacts with the default scheduler, and must describe how priority inheritance and priority ceiling and any other supported synchronization techniques resolve contention between threads that are scheduled according to different scheduling abstractions. Programmers should

refrain from using scheduling extensions that are not standardized by a particular profile definition.

2.2. Plug-in Synchronization

The RTSJ describes (loosely) the availability of PriorityCeilingEmulation protocol, but specifies that this capability is optional. The RTSJ leaves placeholders to allow each RTSJ implementer to extend the RTSJ in its own proprietary way, adding custom-tailored synchronization mechanisms to complement the standard consisting of PriorityInheritance and PriorityCeilingEmulation protocols. Real-time programmers who make any use of these scheduler extensions must recognize that any such usage is non-portable.

Each standard profile must precisely describe whether it includes support for PriorityCeilingEmulation and must describe the set of synchronization extensions, if any, that are supported by that profile. Besides naming the alternative synchronization mechanisms that are available in the profile, the standard must describe how the alternative synchronizers interact with the default synchronizers, and must describe how priority inheritance and priority ceiling and any other supported synchronization techniques resolve contention between threads that are scheduled according to the different scheduling abstractions supported by the profile.

2.3. Plug-in Garbage Collection Interface

The RTSJ provides a GarbageCollector class which provides an interface between application code and operation of the garbage collector. Unfortunately, the only information made available by way of this method is the maximum garbage collection pre-emption latency experienced by RealtimeThread. In order to support portable, efficient, reliable deployment of soft real-time code that makes use of a real-time garbage collector, the application must coordinate with the garbage collector to identify pacing parameters.

Each standard profile must precisely describe any extensions of the `GarbageCollector` class. To support reliable, portable pacing of garbage collection, extensions must allow the application to (1) specify the priority at which garbage collection is expected to run, (2) specify the percentage of CPU time that must be reserved for application threads both above and below the garbage collector's priority, (3) specify (or support measured self-discovery) the rate at which the application intends to allocate memory, (4) specify (or support measured self-discovery) the maximum amount of memory that the application intends to retain as live at any instant in time, and (5) specify the desired size of the safety buffer from which objects can be allocated if/when garbage collection pacing heuristics fail to keep up with the application's allocation rate.

2.4. Libraries

The RTSJ is not a platform. Instead, it is a set of extensions that can be used to modify any one of the standard Java platforms (J2ME CLDC or CDC, J2SE, J2EE). It is a common error for system architects and designers to speak of RTSJ as if this were sufficient to constrain the set of available libraries. Instead, programmers must describe their target platform as, for example, J2ME CDC with RTSJ extensions. Unfortunately, it turns out that even this is not enough to characterize the set of libraries that can be called from real-time threads. The great difficulty is that the RTSJ defines two new thread types, each of which implements a different memory model. Of all the standard libraries available in J2ME CDC, which of these may be called reliably from a `RealtimeThread`? Which of these may be called from a `NoHeapRealtimeThread`? The answer depends on how the libraries are implemented, and it is likely to be different for every RTSJ implementation.

Each standard profile must precisely describe what set of libraries are available for execution on that platform. The memory allocation, access, and modification behaviors of all the available libraries

must be described in sufficient detail to allow programmers to reliably determine which libraries can be called from `RealtimeThread` and which can be called from `NoHeapRealtimeThread`. Programmers must refrain from invoking libraries from thread types for which the specified memory behavior cannot be demonstrated consistent with that thread type from the specification alone. Otherwise, their code will not behave consistently across all implementations of the given profile.

2.5. Number of Priorities

The RTSJ specifies that there shall be at least 28 priorities. Any use of priorities above 28 is non-portable.

Each standard profile must specify exactly how many priorities are supported by that profile. Programmers must recognize that any use of priorities greater than 28 is non-portable across profiles.

2.6. Task Cost Overrun Enforcement

The RTSJ specifies that detecting when a task has consumed more than its budgeted CPU-time resource is an optional capability. Note that for certain systems, enforcement of task overruns may be a critical part of assuring reliable operation. Without cost overrun enforcement, it is difficult to guarantee that one component's reliable operation will not be compromised by another component that has consumed the CPU time resources that were supposed to have been reserved for the first component.

In most real-world systems, task overrun enforcement is only an approximation. When task overrun enforcement is in place, it is very important that the application developers understand exactly how the CPU-time approximations are calculated. This is necessary in order for them to know how much trust they can place in the run-time environment's ability to detect and prevent task overruns.

Consider, for example, the common approximation of adding the duration of the timer tick period to the CPU-time accumulation of whatever thread

happens to be running when the timer tick arrives. This approach will not work reliably for task cost overrun enforcement because:

1. This will overestimate the CPU time consumed by a thread that spends most of its time blocked on I/O, but happens to unblock just before the timer tick arrives.
2. This will underestimate the CPU time consumed by a thread that spends most of its time in computation, but happens to block on I/O just before the timer tick arrives.
3. Even in the case that a thread is CPU time bound and runs without interruption, you generally must count two extra ticks before you can reliably announce that a thread has overrun its budget. For example, suppose the tick period is 1 ms and the CPU time budget is 150 μ s. First, you must round 150 μ s up to 1 ms because that's the limit of the system's timing accuracy. Then, before announcing a cost overrun, you must count two ticks to make sure the thread actually ran the duration of a full tick, rather than simply becoming unblocked immediately before the first tick arrived.
4. To be fully accurate, any CPU time spent handling hardware interrupts must be subtracted from the CPU time charged to this particular thread.

Clearly, there are many different ways to approximate CPU time consumption. Developers who are relying upon task cost overrun detection need to understand the limits of accuracy in order to write code that runs portably on a variety of RTSJ implementations.

Each standard profile must specify whether it supports cost overrun detection. If so, it must precisely characterize the limits of accuracy associated with the enforcement heuristics.

2.7. Workload Feasibility Analysis

The RTSJ specifies that the ability to determine whether the CPU capacity is sufficient to handle a particular workload (known as feasibility testing) is an optional capability. Well-known techniques for performing feasibility testing are available.

However, all make various idealistic assumptions that are not necessarily representative of the real world. Examples of the simplifying assumptions include:

1. The cost of a context switch is zero.
2. The cost of a preemption does not affect the CPU time required to complete the preempted task's work (which is not true if the cache contents is modified by the preempting task).
3. The time required by the scheduler to decide which task to dispatch next is zero or constant or negligible. In reality, however, many schedulers require time that is proportional to the number of ready tasks in the system.
4. All tasks will honor their CPU-time budgets. Clearly, workload feasibility analysis depends on accurate knowledge of the amount of CPU time required to run each of the tasks. If any task violates its budget, the entire workload's reliability is compromised. One would think that we could combine scheduling feasibility testing with cost overrun enforcement to guarantee that all tasks in the system will run reliably. Unfortunately, the ability to accurately enforce cost overrun is not generally available. See Section 2.6. So should our feasibility analysis be based on stated costs of each thread, or on the enforceable costs of each thread?

Clearly, there are many different ways to approach workload feasibility testing and every RTSJ implementation is likely to approach this differently. If feasibility testing is provided, programmers need to know exactly how it interacts with real-world scheduling overhead and cost overrun enforcement.

Each standard profile must specify whether it supports workload feasibility analysis. If so, it must precisely characterize the semantics of workload feasibility analysis as defined for that profile.

2.8. Semantics of ImportanceParameters

The RTSJ describes a data type named ImportanceParameters, but does not precisely specify what it means. Every RTSJ implementation is allowed to

assign different meaning to the use of this parameter.

Each standard profile must specify whether it supports ImportanceParameters. If so, it must precisely define the semantics.

2.9. Real-Time Clocks

The RTSJ defines an abstract Clock class, which represents the ability to keep track of high-precision timing information. Each compliant RTSJ implementation must provide a default real-time clock and may optionally support additional real-time clocks to represent alternative representations of time. One clock might be driven by a 1 ms tick period, another by a 100 ms tick period, one could periodically synchronize with time as calibrated from Global Positioning satellite readings, and another might calibrate with atomic time as broadcast from Ft. Collins, Colorado. The RTSJ tells us: “Clocks differ because of monotonicity, synchronization, jitter, stability, accuracy, and resolution.” Speaking of the default real-time clock, the RTSJ states that it must “advance in sync with the external world”. It is not clear exactly how this is defined. Will it be monotonically increasing? What level of tolerance is expected for a “compliant implementation”.

Clearly, real-time code makes certain assumption regarding the accuracy of the computer system’s notion of time. Within certain tolerances, the real-time code will function correctly. If the real-time clock’s accuracy is outside the assumed tolerances, real-time operation will fail. Real-time programmers who are targeting a particular profile need to understand the precise semantics of each Clock service provided within that profile.

Consider defining a profile that standardizes a particular set of available Clock implementations, and for each, describes the required accuracy, tick frequency, and scalability with respect to pending timer events.

2.10. Deadline Overrun Enforcement

The RTSJ requires the ability to detect and report an overrun condition whenever a real-time thread

misses its deadline. However, it does not provide any clarification as to the accuracy with which deadline overruns are enforced, and each RTSJ implementation is likely to deal differently with deadline enforcement. Note, for example, that deadline enforcement first depends on accurate recording of the time at which the task is triggered for execution. This is no more accurate than the tick period of the default real-time clock. It is not entirely clear from the RTSJ specification whether checking for deadline overrun only after a real-time thread has completed its execution is a compliant implementation. If the deadline overrun is expected to be triggered asynchronously, we need to know when it will be triggered.

Each standard profile must specify the timing and accuracy with which task deadline overruns are enforced.

2.11. Interruption of Blocked Treads

The RTSJ says that if a thread is blocked when the thread is interrupted by an asynchronous transfer of control request, the RTSJ run-time environment will either:

1. “unblock the blocked call,
2. raise an InterruptedException on behalf of the call, or
3. allow the call to complete normally if the implementation determines that the call would eventually unblock.”

Note that there is no timing constraint on “eventually”. Note also that every RTSJ implementation is free to deal differently with every particular blocking call situation.

Each standard profile must specify for each of the blocking libraries that it supports exactly how that library method behaves if an asynchronous transfer of control request is delivered while a thread is blocked waiting for that blocking operation to complete.

2.12. AsyncEventHandler Semantics

The RTSJ states that asynchronous event “handlers are bound to an execution context dynamically

when the instances of AsyncEvent to which they are bound occur.” Furthermore, the RTSJ tell us:

“It is intended that the system can cope well with situations where there are large numbers of instances of AsyncEvent and AsyncEventHandler (tens of thousands). The number of fired (in process) handlers is expected to be smaller.”

Hard real-time programmers take great care to understand precisely what resources are required to reliably run their application. The RTSJ specification places very few constraints on how exactly these unbounded asynchronous event handlers are implemented. Is there a pool of proxy threads? How many threads? How large are their run-time stacks? Assume, for example, that there is a pool of proxy threads, one thread for each different priority level. This would work well until one of the asynchronous event handlers blocks, at which time no further asynchronous event handlers would execute at that particular priority level. Would the run-time environment then spawn another proxy thread? How much memory will this require? What will happen if the memory is not available? And if the RTSJ run-time does not spawn another proxy thread, we run the risk that the system will deadlock when asynchronous event handlers block under certain difficult-to-replicate situations.

Each standard profile must describe the implementation technique (or at least the behavioral semantics) required for compliant implementations of the AsyncEventHandler for the particular profile. The profile specification must require implementers to describe how many resources the implementation of AsyncEventHandler must allocate and when those resources will be allocated and reclaimed.

In some profiles, it might be desirable to prohibit the use of unbound AsyncEventHandler. In other profiles, it might be most appropriate to provide an explicit API to allow application software to monitor and control the number of threads in the proxy pool and their status.

2.13. Object Finalization

No-heap real-time threads cannot allocate garbage collected objects, but they can allocate objects within special ScopedMemory contexts. There is some imprecision in exactly when objects are finalized, and when the memory for a given ScopedMemory context is reclaimed. Thus, each RTSJ implementation may behave differently in this regard. Consider the following quote from the 1.0.1 draft of the RTSJ:

“When the last reference to the [ScopedMemory] object is removed..., finalizers are run for unfinalized objects in the memory area, and if the reference count is still zero after finalization completes, the memory area will be emptied before it is used again.”

Note that it is not specified what thread performs the object finalization. Note also that there is a race condition between finalization of the dead objects and reentry by some other thread into the memory area’s scope. If some other thread enters the scope before finalization has completed, the scope entered by that new thread will not have been cleaned up, and the thread will therefore not be able to reliably allocate all of the memory that is required for its successful execution.

Each standard profile must describe the exact semantics of when ScopedMemory objects will be finalized, by which threads, running at which priorities.

In some profiles, it might be most appropriate to forbid programmers from writing finalize() methods. In this environment, the timing and reclamation of memory is much easier to understand and analyze.

In other profiles, it might be most appropriate to forbid all use of ScopedMemory regions. This also provides a much simpler environment that is easier to understand and analyze.

3. Failure to Support Scalable Development

Scalability represents the ideals that small systems easily evolve into larger systems, and that complex

systems can be easily composed of many small independently developed components. Second to portability, high-level support for scalable development is probably the second most important reason that the J2SE platform offers a five- to ten-fold productivity improvement over C and C++ during the maintenance phases of the software life cycle. With J2SE, scalability is enabled through strong portability, strong type checking, object-oriented encapsulation and information hiding, and automatic garbage collection. These platform properties are enforced with the Java compiler, byte-code verifier, security manager, and run-time garbage collector.

In the development and maintenance of real-time software, scalability is especially challenging. This is because the composition of components almost always increases contention for certain shared resources (memory, CPU time, synchronization locks for shared data bases, etc). With increased contention comes the risk that certain real-time components will not get sufficient timely access to contended resources, and will consequently fail to satisfy their real-time constraints.

Strong standards for the high-level development of real-time software must establish conventions and provide automation (development and run-time tools) to help developers achieve the objective of scalability.

This section identifies several of the areas in which the RTSJ, as currently drafted, fails to enable scalability. Several recommendations for how scalability can be improved in particular profiles are also presented.

3.1. Portability

As discussed in great depth in Section 2, the RTSJ as specified does not represent a “portable platform”. Achieving portability is a prerequisite to supporting full scalability.

3.2. Separation of Concerns Between Abstraction Layers

One of the ideals of real-time Java is to provide the real-time programmer with access to the full generality and capabilities of the traditional non-real-

time Java platform. This requires the ability to efficiently and reliably integrate hard real-time, soft real-time, and non-real-time Java components. This integration should be scalable, in the sense that the behavior of traditional components does not interfere with the reliability of real-time components. Likewise, real-time components should not interfere with reliable operation of non-real-time functionality.

The RTSJ’s wait-free queue is the only way to communicate and share information between real-time components and non-real-time components. Note that `NoHeapRealtimeThread` is not allowed to access traditional Java’s heap objects. If the real-time programmer desires to pass information to the traditional Java environment, he must copy the data into a real-time object (e.g. `ImmortalMemory`), insert this object into a wait-free queue, trust the traditional Java programmer to extract the desired information from this object, trust the traditional Java programmer to send this “communication-buffer” object back to the real-time component by inserting it into a different wait-free queue, and trust the traditional Java programmer to refrain from attempting to lock the shared `ImmortalMemory` object. The level of trust required between the hard real-time developer and the non-real-time developer represents a scalability vulnerability that would be best to avoid. All of these same issues apply also to the use of wait-free queues for sharing and coordination of information between hard real-time (`NoHeapRealtimeThread`) and soft real-time (`RealtimeThread`) threads.

The RTSJ wait-free-queue protocols were designed to eliminate the need for hard real-time components to synchronize with soft real-time components. The reason it was necessary to eliminate this need was because if a hard real-time thread attempts to synchronize on an object that is already locked by a non-real-time thread, and the non-real-time thread is blocked waiting for garbage collection to complete, the RTSJ’s priority inheritance mechanism will elevate the priority of garbage collection to the level of the waiting hard real-time thread. This in turn causes garbage collection activities to interfere with execution of all other hard real-time threads running at priorities equal or

below the priority of the waiting hard real-time thread.

While the introduction of wait-free queues eliminates the need for synchronization between real-time and non-real-time threads, it certainly does not guarantee the absence of all such synchronization. Both the real-time and the non-real-time programmers are told they should not synchronize on shared objects, but they are never prohibited from doing so. And there are many ways, including the use of wait-free queues, that shared objects can become visible both to real-time and non-real-time threads. In fact, the Java memory model requires that a certain amount of synchronization take place in order to assure that data written by, for example, a hard real-time thread can be extracted from a shared object when read by a non-real-time thread.

Give strong consideration to adopting profiles that do not mix garbage-collected heap memory with `ScopedMemory` and `ImmortalMemory` within the same virtual machine address space. A hard real-time profile of the RTSJ that supports only the `NoHeapRealtimeThreads` (without heap memory, and without `java.lang.Thread` or `javax.realtime.RealtimeThread`) would avoid many of these scalability problems.

When combining hard real-time components with soft real-time or non-real-time components, consider alternative partitioning mechanisms. For example, one profile might provide strong memory partitioning between hard real-time components implemented according to the constraints of a hard real-time RTSJ profile, and non-real-time components running as traditional J2SE components residing in a different “virtual machine address space”. The profile would allow hard real-time components to expose selected “capabilities” to the J2SE environment. The J2SE environment could exercise these object-oriented capabilities by invoking the “methods” that implement those services. Allow for very efficient implementations of the capability methods

by supporting integration with the J2SE JIT compiler.

3.3. Composition of Interruptibility

According to the conventions of the RTSJ, any method that declares itself to throw `AsynchronouslyInterruptedException` is considered to be interruptible by asynchronous transfer of control requests. The rules of scalable composition would suggest that a method whose signature throws this exception would behave in a certain predictable way according to the method’s signature. Unfortunately, this is only an illusion. A method’s declaration that it is “interruptible” offers little assurance regarding its true behavior. It may:

1. Be overridden in a subclass by a method that does not throw this same exception.
2. Enter into synchronized code, which is always considered to be interruption-deferred, and while there, run an infinite loop.
3. Invoke another method which does not declare itself to be interruptible, and while there, run an infinite loop.
4. Invoke some method which itself catches and masks the `AsynchronouslyInterruptedException` object that might be thrown from an inner-nested context.

A common approach toward enforcing scalable composition of real-time components is to allow a trusted supervisor activity to oversee the coordination and sharing of resources between those components. Whenever particular components misbehave, the supervisor must abort or redirect those misbehaving components. Unfortunately, the RTSJ does not provide any guarantees that the supervisor is really in control. The component software may accidentally or maliciously prevent the supervisor from being able to interrupt the running application.

Consider defining profiles that assure more reliable top-down enforcement of interruptibility. For example, a profile might provide development tools to enforce that:

- *If a method is declared to throw `AsynchronouslyInterruptedException`, that method only*

*calls other methods that also throw `Asyn-
chronouslyInterruptedException`.*

- *If a method is declared to throw `Asyn-
chronouslyInterruptedException`, any overriding
method also throws `Asyn-
chronouslyInterruptedException`.*
- *If a method is declared to throw `Asyn-
chronouslyInterruptedException`, any synchro-
nized code within the method must adhere
to a restrictive set of guidelines that
ensures the code will “terminate quickly”.*
- *If a method is declared to throw `Asyn-
chronouslyInterruptedException`, none of the catch
or finally clauses “terminate abruptly”.*
*(Abrupt termination means, for example,
jumping out of normal control flow with a
continue or break statement.) The code con-
tained within all catch and finally clauses
must “terminate quickly” (no infinite
loops, no blocking). And any catch clause
that might have caught an `Asyn-
chronouslyInterruptedException` thrown from an inner
context must rethrow that same exception.*

3.4. Composition of CPU-Time and Deadline Enforcement

The RTSJ platform itself offers no direct support to prevent cost overruns. Developers must figure out some way to determine worst-case execution times on their own. It does provide some support to prevent deadline overruns, in the form of workload feasibility analysis, but the precise semantics of feasibility analysis is not clear (See Section 2.7). RTSJ systems that are successfully satisfying all of their CPU-time and deadline constraints are operating in a state of fragile unstable equilibrium.

One of the scalability challenges faced by real-time developers is to assure that an error in one component does not compromise or interfere with other components. Unfortunately, if a given thread overruns its CPU-time or deadline, this results in the triggering of extra asynchronous event handlers to deal with the overrun situation. This only exacerbates the overload situation, increasing the likelihood that even more threads will exceed their deadlines. And since the event handlers cannot reliably abort the thread that has overrun its dead-

line or CPU-time allotment (see Section 3.3), it is very difficult to restabilize the real-time system.

Consider the creation of profiles in which style guidelines define a subset of Java for which it is possible to provide tools that automate the analysis of worst-case execution time. Require compliant implementations of this profile on a given hardware platform to provide the execution-time analyzers.

Combine the profile restrictions described in this section with the ones described in Section 3.3. Require that catch and finally clauses be execution-time analyzable as well. Require that workload feasibility analysis reserve sufficient CPU resources to reliably run the asynchronous event handlers that deal with deadline and CPU-time overruns, and also reserve sufficient CPU time to allow reliable abortion and cleanup of a thread that has exceeded its CPU-time budget or real-time deadline.

3.5. Composition of Timeouts

The RTSJ provides mechanisms that can be used to set timeouts on the behavior of particular code segments. When I speak of composition of timeouts, I am describing the notion that timeout requests may be nested within each other. For example, at a very low-level, a device driver for a network interface card (NIC) may timeout if it does not receive a successful acknowledgement from the NIC hardware following a given command request. At some intermediate level, a network communication protocol stack may timeout and retransmit an information packet if it does not receive an acknowledgement from some remote computer. And at a higher application level, a data-base transaction manager might choose to abort a transaction if it fails to receive commit confirmations from all parties within a certain timeframe.

A careful reading of the RTSJ specification makes clear that it is possible to carefully write timeout expressions so that they can be successfully nested within each other. If the inner-most timeout request expires, only the inner-most context is aborted. If

an intermediate-level timeout is signaled, the innermost timeout context allows the timeout exception to propagate to the intermediate-level context, where it is processed.

Unfortunately, the protocols for nesting of RTSJ timeouts require trusted cooperation of all the components that might be on the run-time stack when the timeout alarm is signaled. Malicious or erroneous behavior by particular components will cause the timeout signal to be processed by the wrong context, or to be ignored entirely.

Consider defining a profile in which development tools enforce style guidelines that guarantee that nested timeout requests are properly handled within the intended contexts.

3.6. Composition of Memory Abstractions

The RTSJ defines several different dynamic memory allocation abstractions. In addition to traditional garbage-collected heap abstractions, the RTSJ supports `ImmortalMemory` and `ScopedMemory` data types. Furthermore, the RTSJ supports two new thread types, known as `RealtimeThread` and `NoHeapRealtimeThread`. A `NoHeapRealtimeThread` is only allowed to access `ImmortalMemory` and `ScopedMemory`. A `RealtimeThread` is allowed to access all three kinds of memory. There are special restrictions on which objects are allowed to refer to which other objects. These restrictions are enforced by run-time checks. A write barrier implemented for each `RealtimeThread` prevents the creation of links from heap objects to `ScopedMemory`, from `ImmortalMemory` to `ScopedMemory`, and from a deeply nested `ScopedMemory` object to a shallow `ScopedMemory` object. The write barrier implemented for each `NoHeapRealtimeThread` prevents links from `ImmortalMemory` to `ScopedMemory`, and from deeply nested `ScopedMemory` to more shallow `ScopedMemory`. Additionally, a read barrier implemented for each `NoHeapRealtimeThread` prevents it from fetching a reference to a heap object from within an existing `ImmortalMemory` or `ScopedMemory` object.

The scalability problems are as follows:

- Code that runs fine in a traditional Java environment may violate the special restrictions imposed on execution of `RealtimeThread` and/or `NoHeapRealtimeThread`.
- There are no syntactic markers and no established style conventions to allow programmers to distinguish between code that is designed to run according to the special rules of each distinct thread model.
- Even with syntactic markers, it is computationally intractable to prove at compile or link time that program components will successfully compose. The full generality of the RTSJ requires run-time checks and the concomitant likelihood that certain programming errors will not be detected until run time.

Consider defining a profile that provides development tools to statically enforce style guidelines that will assure composability with respect to interaction between memory allocation models. One such model, which is based on the use of meta-data annotations, has been proposed for the safety-critical Java standard.

3.7. Resources for Thrown Exceptions

An important responsibility of a real-time developer is to assure that all resources required for reliable operation of software are always available. Thrown exceptions represent a special challenge because exception handling represents unexpected control flow which consumes memory and CPU-time resources.

According to traditional Java conventions, when an exception is thrown, a new data structure is allocated and initialized to represent the current state of the stack backtrace. One difficulty imposed on the RTSJ programmer is anticipating the size of this data structure and assuring that the current allocation context has sufficient allocatable memory to represent it.

But there is another, more subtle composability problem with throwing exceptions from within RTSJ program components. In particular, when the exception data structure is constructed, it is typically allocated from within a `ScopedMemory` alloca-

tion context. A composability problem arises if the catch clause that is responsible for dealing with this exception is nested outside the entry into the `ScopedMemory` context that represents the thrown exception. At the point where the exception attempts to propagate beyond entry into the `ScopedMemory` context, the originally thrown exception will be replaced with an uninformative `ThrowBoundaryError` exception, which will be allocated in the surrounding allocation context.

Consider defining a profile in which RTSJ components refrain from accessing all of the information associated with particular exception objects. Within this profile, it is not necessary to represent the entire stack backtrace. The limited stack backtrace information that is required can be represented in a fixed-size thread-local buffer. Using this approach, it is possible to avoid the need for memory allocation at the time an exception is thrown. Simply preallocate one shared `ImmortalMemory` instance of each of the standard exceptions that might be thrown from the standard libraries associated with this profile.

3.8. Composition with Kernel Services

Since a real-time programmer is responsible for understanding resource requirements and guaranteeing availability of all required resources, he must understand the resources consumed by the operating system kernel. To support scalable composition of software components, the resources required by the “kernel” services must be consistent across all compliant implementations of a particular profile. Unfortunately, the RTSJ fails to characterize the resource requirements for particular services and does not even require implementers to provide answers to questions such as the following:

- What is the time complexity of each byte-code instruction? Which instructions run in constant time and which require variable time? Which ones allocate memory? For those that run in variable time, what are the parameters and formulas with which computation time can be calculated?

- How much memory and CPU time is required to start up a new thread?
- How much memory and CPU time is required to introduce a new asynchronous event handler?
- Which of the standard RTSJ libraries execute in constant time? What are the parameters and formulas with which computation time can be calculated for the others?
- Which of the standard RTSJ libraries allocate memory? How much memory? Which flavors of memory are allocated?
- What is the overhead of scheduling? Does the run-time thread scheduler run in constant time? Does its execution time depend on how many threads are ready to run? How does this scale?
- What is the overhead of performing time-related services (setting alarm timeouts, requesting to sleep, processing each timer tick)? Does this depend on how many other threads are using services associated with the same timer? How does this scale?
- What is the overhead of acquiring and releasing synchronization locks? Does this allocate memory? Where is the memory allocated and when is it reclaimed? Is the time complexity constant? Does execution time depend on how many other threads are contending for the same lock? How does this scale?

There are huge numbers of uncertainties that make it very difficult for real-time programmers to proactively manage the resources required for reliable operation of their real-time applications. Programmers may attempt to satisfy these uncertainties with extensive testing, but that testing will not necessarily identify the worst-case resource requirements, and certainly does not represent the breadth of alternative RTSJ implementations on which they might one day desire to run their code.

Consider creation of profiles in which certain guarantees regarding the resource requirements of the built-in kernel services are carefully constrained.

3.9. Scalability of Scheduler Abstractions

One of the dimensions in which the RTSJ designers envisioned future scalability is in the realm of

alternative real-time schedulers. The RTSJ requires that all implementations support fixed-priority preemptive scheduling. Additionally, it provides placeholders to allow insertion of supplementary real-time schedulers. Unfortunately, the RTSJ definition requires all supplementary schedulers to coexist with the default or base scheduler. This adds considerable complexity to the programming model and to the run-time implementation. There is no intuitively obvious answer to question such as:

- If I have a fixed priority thread at priority 987, and an earliest-deadline-first thread with a relative deadline of 1 ms, which one should be scheduled next?
- If those same two threads are waiting to access a lock that uses the priority inheritance monitor control policy, what priority should be “endowed” to the thread that owns the lock? And which of these two waiting threads should be granted next access to the lock?

Consider creation of profiles in which all program components use only one scheduler, which need not be the “base scheduler”. Carefully define (or redefine) the meaning of priority inheritance and/or priority ceiling monitor control policies as they pertain to this alternative scheduling technology.

3.10. Mutable Kernel Data Structures

The following is an oft-repeated statement throughout the RTSJ specification:

“This class is explicitly unsafe in multi-threaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.”

Unfortunately, this represents a serious flaw in the design of the RTSJ. The rules of the Java memory model are such that in the absence of synchronization, individual threads are free to cache the contents of objects within local temporary variables and/or machine registers. This means the RTSJ “kernel”, which is outside the realm of user-modi-

fiable code, may never see changes to these shared data structures, because the kernel code was written with the assumption that “no synchronization is done”.

In practice, this is probably not a problem until we begin to see optimizing JIT compilers and/or SMP implementations of the RTSJ. At that time, it may become necessary to rewrite all of these APIs.

There’s another problem with mutable kernel data structures, such as `ReleaseParameters` and `SchedulingParameters`. The RTSJ allows the fields of these parameter objects to be changed, but tells us that the scheduling behavior will not be affected until the corresponding thread’s `setReleaseParameters()` or `setSchedulingParameters()` method is called. This might result in a situation in which a thread’s scheduling behavior, as specified by the parameter values the last time the `setReleaseParameters()` or `setSchedulingParameters()` methods were called, differs from the current values of the `ReleaseParameters` and `SchedulingParameters` objects that are associated with the thread. All of this violates the fundamental programming language design rule of “least astonishing results”.

Consider creation of a profile in which all kernel data structures are immutable. Enforce style guidelines that prohibit the invocation of any methods that would change the values of shared kernel data structures.

4. Performance and Footprint Issues

During the 8 years that Aonix/NewMonics has been providing real-time Java technologies to a breadth of industries, we have encountered a huge variety of contradictory requirements and expectations. We have come to understand that there is a broad class of soft real-time developers who desire to leverage the full benefits and strengths of the traditional J2SE platform, including automatic garbage collection, standard libraries, symbolic debuggers and performance profilers. And then there is a very different class of developer who wants to bring some of the benefits of Java into very different environments than would be feasible using traditional Java technologies. These folks

often have requirements for compliance with hard real-time deadlines, high performance (comparable to C), and very tight memory constraints (of less than 100 KBytes). This second broad class of developers recognizes that they will have to work harder to write their hard real-time, high performance, small footprint Java code than developers of traditional enterprise Java applications. However, their hope is that it will be easier for them to write and maintain this hard real-time, high performance, small footprint code in Java than it is in C or C++.

We at Aonix fully endorse this vision. Unfortunately, the RTSJ designers tell us that “real-time is not real fast”. Today’s best available compiled implementations of RTSJ run slower than mainstream Java, and that typically runs about a third the speed of optimized C code.

By subsetting the set of RTSJ features in appropriate profiles, we believe it is possible to achieve the desired performance, footprint, and determinism tradeoffs.

4.1. Extensions vs. Platform

One memory footprint challenge results because the RTSJ is packaged as extensions rather than as a stand-alone platform. This means you must begin with one of the existing Java platforms (J2ME CLDC or CDC, J2SE, J2EE) and then enhance or extend that platform by adding the RTSJ libraries and semantics. In terms of memory footprint, this means a compliant RTSJ implementation can be no smaller than the Java platform which it is extending.

Consider defining profiles in which the programmer is restricted from using the full generality of the accompanying Java platform. In this way, the unused libraries and run-time support capabilities can be eliminated from the run-time footprint.

4.2. Three Flavors of Threads

The RTSJ’s support for three different kinds of threads leads to both footprint and performance challenges. In the simplest of compiled implementations, the code generator must insert both write

barriers and read barriers for all access to object instance variables. Execution of this barrier code is conditioned upon the type of thread that is executing the code. This approach results in slightly increased code size and much slower execution than traditional Java.

An alternative code generation model produces three different translations of every method, one for each of the different thread execution models. To support this model, you maintain three virtual method tables for each class. Under this model, the run-time penalties associated with the real-time threads are isolated to those threads, and the code for individual methods generally runs faster because it doesn’t have to check which kind of thread it is. But the code still runs much slower than C, and the code size is three times larger than the traditional Java code footprint.

Consider defining profiles in which programmers are restricted to use only one of the three thread types, as has been suggested in the safety-critical Java profile.

4.3. Wait-Free Queues

The only available interface between hard real-time and non-real-time components is known as a wait-free queue. There are several efficiency issues regarding the use of this protocol:

- First, the process of copying data and performing the synchronization necessary to commit changes to and fetch changes from shared data structures consumes CPU time. There is no opportunity to exploit zero-copy algorithms and data structures.
- Second, because the hard real-time threads are not able to block, they are required to periodically poll for available buffer space. Frequent unproductive polling wastes CPU cycles. Sleeping between consecutive poll requests introduces undesirable latency.

Consider introducing profiles that use alternative mechanisms for communicating between hard real-time and non-real-time threads. Blocking synchronization can be implemented safely if all blocking is performed by code written by the hard

real-time developer. Efficient zero-copy data structures can be implemented using a JIT or AOT compiler that understands the special protocols required to access hard real-time services from the traditional Java domain.

4.4. Enforcement of Deadline and Cost Overruns

Deadline enforcement is required in all compliant RTSJ implementations, but the RTSJ does not specify the accuracy with which deadline overruns must be detected. If a thread overruns its deadline by one microsecond, is a compliant RTSJ implementation required to raise an overrun event? What if the thread overruns its deadline by one nanosecond? Another uncertainty from the RTSJ implementation is exactly how quickly the deadline overrun event must be triggered. Must I raise the event at the exact moment the thread overruns its deadline, or can I wait until the next timer tick before I do so? Can I wait even longer? The cost of implementing deadline overrun detection is very high if accurate and timely overrun detection is required.

Cost overrun detection is an optional capability of the RTSJ. When it is supported, the system is required to signal an overrun event whenever a thread consumes more than its budgeted amount of CPU time. If a thread overruns its CPU time budget by a single microsecond, is a compliant RTSJ implementation required to trigger the cost overrun event? What if it overruns by only a nanosecond? Must I trigger the cost overrun event at the exact moment the thread overruns its cost? Or can I wait until the next timer tick before I do so? Can I wait even longer? The cost of implementing cost overrun detection is very high if accurate and timely cost overrun detection is required.

Consider introducing profiles that explicitly allow certain violations of deadline overrun and cost overrun to be overlooked, without requiring the corresponding overrun event to be triggered. Also, consider defining a profile such that late triggering of the overrun event handler is explicitly allowed. Make these profile defi-

nitions clear and precise, so that portable real-time Java code targeted to this profile knows exactly what behavior to expect.

As an alternative, consider introducing a profile that performs no deadline enforcement and no cost overrun enforcement.

4.5. Synchronization Locks

Since Java is designed to support multi-threaded applications, synchronization is built in to the language. Typical Java programs perform an abundance of synchronization operations. Thus, virtual machine implementers invest considerably in optimizing the implementation of synchronization locks.

The RTSJ supports two distinct modes of synchronization, and allows individual implementations of the RTSJ to support even more. The RTSJ programmer has the opportunity to independently specify for every object in the system which mode of synchronization that object uses if the object is ever locked. Furthermore, the RTSJ programmer can change the synchronization mode for particular objects on the fly. All of this adds considerably to the implementation complexity of synchronization locks. Many of the optimizations that are commonly applied to locks of one sort will not work if multiple kinds of locks are supported, especially if the locking modes for particular objects are allowed to change on the fly. Thus, RTSJ implementations of synchronization services generally run slower than the synchronization services of traditional Java.

Another performance challenge arises because a priority ceiling lock can be implemented very efficiently if certain programming conventions are adhered to, but the RTSJ does not require adherence to these programming conventions. On a single processor, there is never a need to queue pending priority ceiling lock requests because once a first thread has entered the lock, no other thread that is eligible to enter the same lock is allowed to run (because none of their priorities exceeds the priority of the ceiling lock object). Thus, it is not possible for a second thread to request entry to a priority ceiling lock while some other thread holds

that priority ceiling lock. This implementation technique only works if the code executed while holding the priority ceiling block is prohibited from performing any action which might block. Note that software components can be proven to comply with this property statically.

Consider defining a profile that only supports priority ceiling protocol without priority inheritance. Consider defining a profile that introduces a static compile-time marker to distinguish objects that will synchronize with the priority ceiling mechanism from those that will synchronize with priority inheritance. Consider enforcing for a particular profile that the code contained within the bodies of code associated with synchronized statements of priority ceiling lock objects does not block.

5. Missing Generality

In general, a chain is no stronger than its weakest link. A system comprised of 99% high-quality real-time Java code and 1% fragile, poorly abstracted C code is not necessarily 99% reliable. A single bug in the C code can bring down the whole system. This is especially true when the C code is integrated with the Java code through the use of the fragile JNI protocol.

To realize all of the high-level benefits of real-time Java, it is important that real-time Java be able to address the full spectrum of developer needs. Otherwise, programmers will frequently find it necessary to step outside the Java model to implement in C or some other legacy language functionality for which real-time Java was not well suited.

Here, we identify several specific real-time programming requirements that are not well served by the RTSJ definition as originally drafted. Though particular RTSJ implementations are free to add libraries and/or provide development tools to address these shortcomings, we at Aonix feel that the needs of the developer community are best served by establishing profiles that standardize the mechanisms used to address the various shortcomings.

5.1. Memory Partitioning

Within the soft real-time garbage collected domain, reliable composition of independently developed components may require the ability to impose limits on the allocation rates and maximum retention of heap memory for particular software components. Otherwise, misbehavior by one component can compromise the reliability of all other components in the system.

Consider defining a soft real-time profile that provides standardized library support for reliable memory partitioning.

5.2. Interrupt Handling

Soft real-time garbage collected Java is able to reliably handle timing constraints ranging from 1 to 100 ms. But device drivers and first-level interrupt handlers cannot be reliably implemented using soft real-time Java techniques because hardware-scheduled interrupt handling may preempt garbage collection activities in the middle of operations that need to be treated as atomic (indivisible). Given that developers generally find it much easier to develop soft real-time application code using standard J2SE-style libraries rather than hard real-time code using the restrictive `NoHeapRealtimeThread` protocols, device drivers and interrupt handlers represent one of the few domains for which only the very specialized capabilities of `NoHeapRealtimeThreads` are suited. Unfortunately, the RTSJ specification as originally drafted does not support the implementation of interrupt handlers. Interrupt handlers can be implemented in C, C++, and Ada, but not in the current RTSJ specification. Forcing implementers of device drivers and interrupt handlers to mix the use of C and Java adds considerably to the complexity of the software because special protocols must be developed to enable sharing of information between the C interrupt handler and the Java application software. And the use of JNI compromises the integrity checking that is built in to Java's compilers and byte-code verifiers.

Consider defining a hard real-time profile that provides standardized library support for portable implementation of first-level interrupt handlers, such as the libraries

that have been proposed for the safety-critical Java standard.

5.3. Sizing of Run-Time Stacks

In order to support reliable operation of real-time tasks, it is important to assure that the run-time stacks are large enough to handle all of the thread's run-time requirements. Unfortunately, the RTSJ does not provide any support for analyzing or measuring run-time stack requirements, and does not provide any mechanism for specifying the stack size for a new thread.

Consider defining a real-time profile that standardizes the development tools and libraries that can be used to measure and analyze run-time stack requirements, and to set the stack size for a new real-time thread.

5.4. Resumption of Asynchronously Interrupted Thread

The RTSJ specification argues that there is no need to support resumptive semantics when a thread is asynchronously interrupted because resumptive semantics can be implemented using asynchronous event handlers. The RTSJ authors, however, overlooked a common and important scenario. Suppose a particular running thread *A* is responsible for maintaining certain shared data structures. Assume that this thread carries considerable context in its thread state, so that it is very costly to shutdown this thread and then restart it. This thread *A* holds a "lock" on these shared data structures. Now suppose some situation arises under which a different thread *B* recognizes that changes to these shared data structures are required. This thread *B* cannot modify these data structures directly, because thread *A* holds a lock on the data structures. An asynchronous event handler also cannot modify these data structures for the same reason. Only thread *A* can modify the data structures. The ideal implementation is to send an asynchronous interrupt signal to thread *A*, allow thread *A* to make the requested modifications to the shared data structure, and then allow thread *A* to resume what it was already doing.

Consider defining a profile that standardizes the library capability required to resume a thread following asynchronous interruption of the thread's execution. Carefully define for this profile the intended behavior when a blocking operation is interrupted and subsequently resumed.

5.5. Monitoring of CPU Time Consumption

The RTSJ provides standard mechanisms for enforcing CPU-time budgets for particular real-time threads, but does not provide the ability to measure how much CPU time is being consumed by threads running RTSJ programs. It would be desirable to have this information available, as it assists developers in tuning system performance. This information can also be helpful in performing rate-monotonic schedulability analysis in the absence of RTSJ's optional support for feasibility analysis.

Consider defining a profile that standardizes the library services required to monitor CPU time consumption of running RTSJ threads.

5.6. Passive Real-Time Clocks

The RTSJ defines an abstract Clock class, which represents the ability to keep track of high-precision timing information. Each compliant RTSJ implementation must provide a default real-time clock and may optionally support additional real-time clocks to represent alternative representations of time. One clock might be driven by a 1 ms tick period, another by a 100 ms tick period, one could periodically synchronize with time as calibrated from Global Positioning satellite readings, and another might calibrate with atomic time as broadcast from Ft. Collins, Colorado. Regardless of how a particular implementation of Clock calibrates its time, the RTSJ API requires that any implementation of Clock can serve to drive the execution of asynchronous events when supplied as an argument to a Timer constructor. Unfortunately, this precludes as valid implementations of `javax.realtime.Clock` the use of highly accurate read-only hardware clocks that are present in many real-

time environments. A separate API is required for a read-only clock that is not able to trigger execution of asynchronous activities. Note that these hardware read-only clocks typically offer far greater timing accuracy than is feasible with tick-driven software implementations of real-time clocks.

Consider defining a profile that standardizes the library services required to access hardware implementations of read-only real-time clocks. Ideally, this clock service would be integrated within the existing RTSJ Clock hierarchy so that this more accurate notion of time could be used to detect deadline and CPU-time cost overruns.

5.7. Certification Requirements

Sometimes less is more. In certain highly regulated market segments, software must be certified by external auditing agencies before the product can be deployed. Safety-critical software for commercial aviation, passenger rail, nuclear power generation, and medical instrumentation is one such domain.

Intense scrutiny of every line of code is required by typical safety certification guidelines. Developers are expected to prove theorems prior to deployment that instill confidence that the software will always function correctly. The burden of certification is so high that the cost per line of code for safety-certified systems is typically 10-30 times higher than the cost for comparable non-certified functionality.

All of this argues for a very simple run-time and extensive use of static analysis tools to assist in proving theorems regarding correct operation of the software. Unfortunately, the existing RTSJ specification describes a run-time environment that is large and quite complex. Many of the program attributes that would have been declared and enforced statically in more traditional safety-critical programming languages like Ada cannot be noted in the RTSJ source code or verified by static checkers. Rather, the RTSJ expects attributes to be associated with program components dynamically,

through the use of various RTSJ services to, for example, establish scheduling and synchronization parameters. Finally, the approach of RTSJ is to detect system integrity violations using run-time checks, and respond with exception and asynchronous event handlers rather than the safety-critical developer's more traditional approach of proving system integrity prior to deployment so that no run-time checking is required and there is no need to prepare for the eventuality that run-time exceptions or asynchronous events might be triggered by a misbehaving RTSJ program.

Consider creating a safety-critical profile that establishes as a standard (1) a small subset of the full RTSJ features, (2) program style guidelines for static annotations based on standard Java meta-data and assertion capabilities, and (3) describes the special development tools that enforce static properties and provide efficient resource-constrained translations of safety-critical RTSJ source code.

6. Summary

The RTSJ, as standardized, was designed to support incredible generality. As such, it is a powerful research tool for the study of real-time software, and this is one reason we see most of the early adoption of RTSJ to be among academic audiences.

As the industry moves towards commercial and large-scale defense deployments based on RTSJ, it is important to establish certain standards for interoperability and portability that are not present in the RTSJ as specified. While it might be desirable to see changes (sometimes radical changes) to the RTSJ specification itself, we are not optimistic that such changes will be delivered in a timely manner.

Instead, our recommendation is to establish a small number of compatible and complementary profiles of the RTSJ. Each profile defines:

1. A subset of the full RTSJ and accompanying Java libraries, clarifying semantics for each supported service where the existing specifications are ambiguous or vague.

2. An optional collection of supplementary libraries that are not available in the standard RTSJ, but might be necessary or desirable to support the special needs of the market segment(s) targeted by the profile.
3. Optional standards for program annotations using Java 5.0 meta-data and assertion syntaxes.
4. Optional standards for developer tools to process the program annotations, enforcing consistency and facilitating efficient and portable deployment.

The profiles are compatible in the sense that code written for certain profiles will run reliably in certain other upwards-compatible profiles. We expect, for example, that a hard real-time mission-critical profile would be upwards compatible with the safety-critical profile. The profiles are also compatible in the sense that code written for certain profiles could efficiently coexist with and complement code written for certain other profiles. We expect, for example, that code written for a hard real-time mission-critical profile could very efficiently coordinate with code written for a soft real-time mission-critical profile. Typical mission-critical systems are comprised of multiple software layers, with each layer providing different levels of abstraction.

The profiles are complementary in the same sense as a good American football quarterback complements a good wide receiver. Together, the various profiles make it possible to efficiently and reliably address all needs of mission-critical and safety-critical development. Our vision is that the safety-critical profile complements the hard real-time mission-critical profile in that both use the same static analysis tools, compilers, debuggers, performance profilers, and code coverage analyzers. Further synergy exists because the standard libraries for the safety-critical profile would be a proper subset of the libraries for the hard real-time mission-critical profile. In our vision, the hard real-time mission-critical profile complements the soft real-time mission-critical profile in the sense that it would provide reliable, safe, and efficient integration of components running in each of the two respective profiles.

Leverage between profiles is very important. If independently developed profiles are not complementary and compatible, they are no better than introduction of a completely new and distinct programming language. Picture the greatest wide receiver in the world, without a quarterback to throw him a pass, or without a strong offensive line to protect that quarterback.