

Programming with Non-Heap Memory in the Real Time Specification for Java

Greg Bollella¹, Tim Canham², Vanessa Carson², Virgil Champlin³, Daniel Dvorak²,
Brian Giovannoni³, Mark Indictor², Kenny Meyer², Alex Murray², Kirk Reinholtz²

¹Sun Microsystems Laboratories
2600 Casey Avenue
MS UMTV29-236
Palo Alto, CA 94043
650-336-1693
greg.bollella@sun.com

²Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-393-1986
daniel.dvorak@jpl.nasa.gov

³School of Computer Science
Carnegie Mellon University
Building 17, First Floor
Moffett Field, CA 94035
650-603-7005
champlin@cs.cmu.edu

ABSTRACT

The Real-Time Specification for Java (RTSJ) provides facilities for deterministic, real-time execution in a language that is otherwise subject to variable latencies in memory allocation and garbage collection. A major consequence of these facilities is that the normal Java practice of passing around references to objects in heap memory cannot be used in hard real-time activities. Instead, designers must think carefully about what type of non-heap memory to use and how to transfer data between components without violating RTSJ's memory-area assignment rules. This report explores the issues of programming with non-heap memory from a practitioner's view in designing and programming real-time control loops using a commercially available implementation of the RTSJ.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *classes and objects, control structures, dynamic storage management, frameworks*

General Terms

Design, Experimentation, Languages.

Keywords

Programming model, scoped memory, architecture.

1. INTRODUCTION

Automatic memory management is one of the biggest benefits of the Java programming language relative to C++. This capability, achieved through automatic garbage collection, eliminates a significant source of programmer error, enabling larger applications to be developed with fewer defects. A price for this benefit is that a thread's execution time and response latency is

non-deterministic because the garbage collector can preempt application execution at any time. This fact precludes highly predictable real-time execution in ordinary Java.

The Real-Time Specification for Java (RTSJ) [1] addresses this limitation through facilities that enable application logic to execute without interference from the garbage collector. The key idea is to provide new kinds of *Runnable* that are guaranteed not to access heap memory. Such *Runnables* can preempt the garbage collector at any time and thus run with high temporal determinism. Of course, these *Runnables* need *some* kind of working memory, so the RTSJ provides two kinds of non-heap memory. However, these new memory areas come with some VM-enforced "assignment rules" to ensure that the garbage collector's business is separated from hard real-time activities. The net result is that RTSJ programmers must confront some new design issues that go beyond issues of real-time scheduling.

This report focuses on the practical design issue of exchanging data between hard real-time components. Users of any RTSJ-compliant virtual machine will confront the same issue and will have to consider how best to use the RTSJ's non-heap memory areas. This report is about application programming, *not* about JVM design or about potential changes to the RTSJ. Since knowledge of the RTSJ is probably not extensive among OOPSLA attendees, this report includes some background to help readers understand the nature of non-heap memory areas and their consequences.

The evolution of real-time garbage collection technology for Java virtual machines will change the picture for developers of real-time systems, but that topic is beyond the scope of this report. As yet there is no commercial product that combines real-time garbage collection with the RTSJ enhancements for threads, scheduling, synchronization, asynchrony, and physical memory access.

2. PROJECT GOLDEN GATE

This report presents some early results from Project Golden Gate [2,3], a collaboration among Caltech's Jet Propulsion Laboratory, Sun Microsystems Laboratory, and the High Dependability

Copyright is held by the author/owner(s).

OOPSLA '03, October 26–30, 2003, Anaheim, California, USA.

ACM 1-58113-751-6/03/0010.

Computing Program [8] led by Carnegie Mellon University. The project is implementing JPL's state- and model-based control architecture—named Mission Data System (MDS) [4]—in the RTSJ using the first commercial implementation of the RTSJ: the TimeSys JTime virtual machine [9]. The JTime VM runs on TimeSys Linux RTOS, a low-latency version of the Linux operating system.

The work reported herein occurred as the team designed control loops for driving and steering a 6-wheel experimental Mars rover named “Rocky 7”. The rover's processor is a 300 MHz PPC 750 with 256MB RAM. The rover hardware includes 6 driving motors, 2 steering motors, 3 stereo camera pairs, a 3-axis accelerometer, a 1-axis gyroscope, a camera frame grabber, and five other motors for controlling a camera mast and an arm.



Figure 1. The Rocky 7 research rover in the Mars Yard at JPL, with camera mast raised.

3. INTRODUCTION TO RTSJ

Ordinary Java technology is not suitable for real-time systems for several reasons: no scheduling control over threads, unpredictable synchronization delays, run-anytime garbage collection, coarse timer support, no event processing, and no safe asynchronous transfer of control. The real-time specification for Java, known as “RTSJ”, addresses these limitations through several areas of enhanced semantics.

The RTSJ was shaped by several guiding principles. Foremost among these is the principle to “hold predictable execution as first priority in all tradeoffs”. Another principle is that the RTSJ introduces no new keywords of other language extensions. Also, the RTSJ provides backward compatibility, meaning that existing Java programs run on RTSJ implementations. Importantly, the RTSJ supports leading-edge scheduling, going beyond simple priority-based scheduling.

It's important to understand that “real time” doesn't mean “real fast”. The guiding principle of predictable execution places more importance on specifying and meeting timeliness constraints than on raw throughput. Real-time applications must respond to periodic, aperiodic, and sporadic events, and the RTSJ provides facilities for informing a scheduler of such constraints and

determining if a set of constraints admits a feasible schedule. The net result in RTSJ, in contrast to purely priority-based systems, is that scheduling and dispatching can be based on explicit timeliness information.

Most real-time applications are a mixture of “hard real-time”, “soft real-time”, and non real-time parts, as shown in Figure 2. In this report we use the term “hard real-time” to mean that temporal correctness criteria must always be met. For example, if a hard real-time computation misses a deadline, the system goes into an abnormal state. By “soft real-time” we mean that temporal correctness criteria are almost always met, so an occasional missed deadline (for example) is tolerated. By “non real-time” we mean that there are no temporal correctness criteria. A key point to understand here is that a single RTSJ-compliant VM can support systems that mix hard, soft, and non real-time parts.

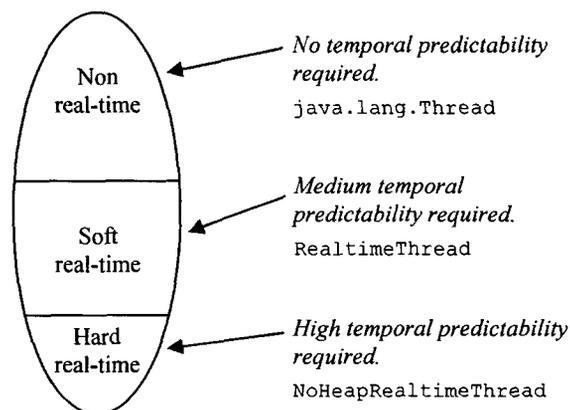


Figure 2. Most real-time systems are a mixture of hard real-time, soft real-time, and non-real-time, all of which can be supported by a single RTSJ-compliant VM.

The RTSJ extends Java semantics in several areas, as summarized below. This background information is intended to provide readers with a broad understanding of how the RTSJ supports various aspects of real-time programming. Some features of the RTSJ have been omitted for brevity.

3.1 Threads

The RTSJ introduces two new types of thread that have more precise scheduling semantics than `java.lang.Thread`. Parameters provided to the constructor of `RealtimeThread` allow the temporal and processor demands of the thread to be communicated to the system. `NoHeapRealtimeThread` (“NHRT”) extends `RealtimeThread` with the restriction that it is not allowed to allocate or even reference objects from the Java heap, and can thus safely execute in preference to the garbage collector. Such threads are the key to supporting hard real-time execution because they have implicit execution eligibility logically higher than any garbage collector.

3.2 Scheduling

The scheduling area in RTSJ provides classes that allow the definition of schedulable objects, manage the assignment of execution eligibility of schedulable objects, assign “release

characteristics” to schedulable objects, and perform “feasibility analysis” for sets of schedulable objects.

As seen in Figure 3, schedulable objects are instances of `RealtimeThread`, `NoHeapRealtimeThread`, and `AsyncEventHandler`. Each of these is assigned processor resources according to its release characteristics and execution eligibility. As shown in Figure 4, there are three types of `ReleaseParameters` to support periodic, aperiodic, and sporadic execution. Each of these subclasses contains parameters needed to determine whether a feasible schedule can be found for a set of schedulable objects.

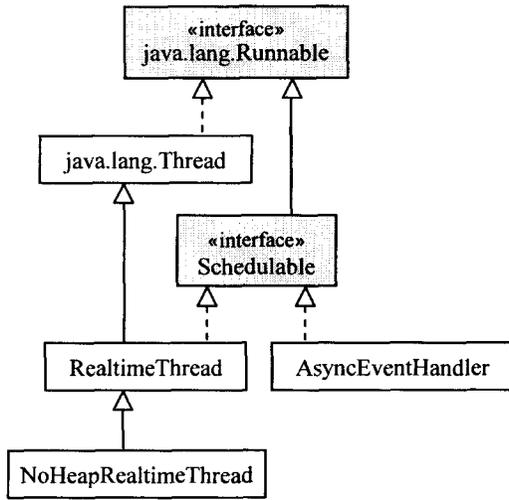


Figure 3. The RTSJ introduces `RealtimeThread`, `NoHeapRealtimeThread`, and `AsyncEventHandler` as new types of `Runnable`.

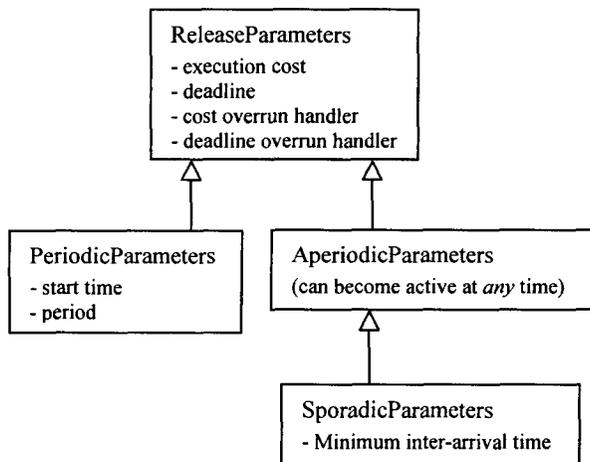


Figure 4. Release parameters supply processor and temporal demands needed to determine schedule feasibility.

3.3 Memory Management

The RTSJ contains classes that allow the definition of regions of memory outside the traditional Java heap. These new memory areas—called `ImmortalMemory` and `ScopedMemory`—are not managed by a garbage collector. This means that instances of `NoHeapRealtimeThread` can use such memory to communicate results within hard real-time areas as well as between hard real-time areas and soft- or non real-time areas.

`ImmortalMemory` is a single memory area that is shared among all threads. Objects allocated in the immortal memory live until the end of the application. In fact, unlike standard Java heap objects, immortal objects continue to exist even after there are no other references to them. Importantly, objects in immortal memory are never subject to garbage collection.

`ScopedMemory` is an abstract base class for memory areas having limited lifetimes. A scoped memory area is valid as long as there are real-time threads with access to it. A reference is created for each accessing thread when either a real-time thread is created with a `ScopedMemory` object as its memory area, or when a real-time thread runs the `enter()` method for the memory area. When the last reference to the object is removed, by exiting the thread or exiting the `enter()` method, finalizers are run for all objects in the memory area, and the area is emptied. Objects in scoped memory are never subject to garbage collection.

The memory management enhancements in RTSJ also include facilities for access to physical memory, facilities for non-heap memory allocation in linear time, and facilities for obtaining information about the temporal behavior of the garbage collector, such as its preemption latency.

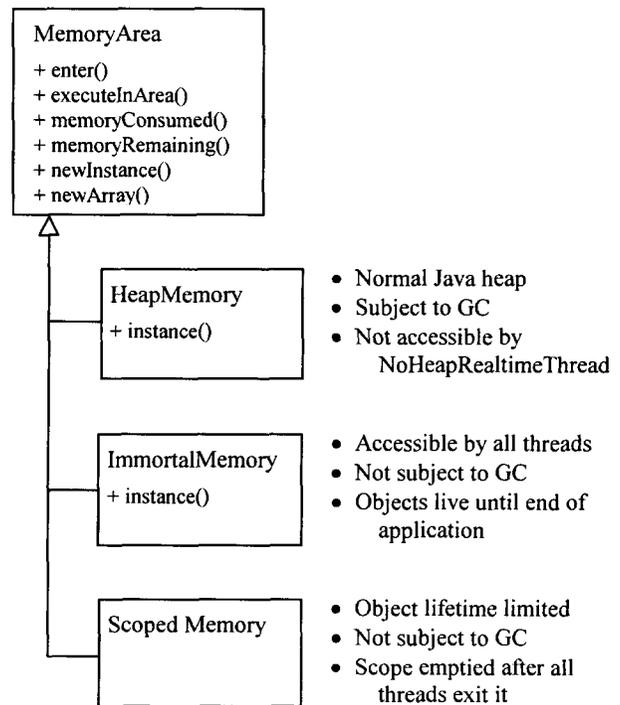


Figure 5. The RTSJ introduces two kinds of non-heap memory that are not subject to garbage collection.

3.4 Synchronization

The RTSJ contains classes that allow application of the priority ceiling emulation algorithm to individual objects; allow the setting of the system default priority inversion algorithm; and allow wait-free communication between real-time threads and regular Java threads. This strengthens the semantics of Java synchronization for use in real-time systems by mandating priority inversion control. The wait-free queue classes provide protected, concurrent access to data shared between instances of `java.lang.Thread` and `NoHeapRealtimeThread`.

3.5 Time

The RTSJ contains classes that allow description of a point in time with up to nanosecond accuracy and precision (dependent on the precision of the underlying system), and allow distinctions between absolute points in time, times relative to some starting point, and rational time, which allows the efficient expression of number of occurrences per some interval of relative time.

The time class relationships are depicted in Figure 6. Instances of `AbsoluteTime` represent absolute time expressed relative to midnight January 1, 1970 GMT. Instances of `RelativeTime` encapsulates a point in time that is relative to some other time value. Instances of `RationalTime` express a frequency as an integral number of cycles per an amount of relative time.

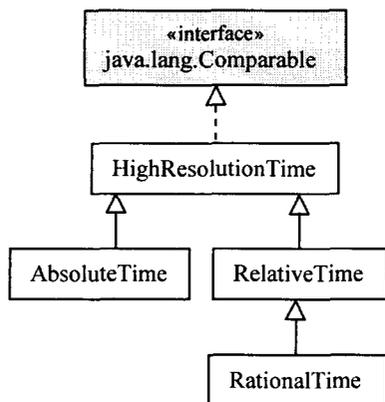


Figure 6. High resolution time supports timing with nanosecond accuracy and precision, subject to the underlying system's accuracy and precision.

3.6 Timers

The RTSJ contains classes that allow creation of timer whose expiration is either periodic (`PeriodicTimer`) or set to occur at a particular time (`OneShotTimer`). RTSJ also defines an abstract base class for clocks, recognizing that real systems often have other kinds of clocks (e.g. simulation clocks, user time clocks), and allows timers to specify such a clock in place of the default system clock.

3.7 Asynchrony

The RTSJ contains classes for binding the execution of program logic to the occurrence of internal and external events. Specifically, an asynchronous event is represented as an instance of class `AsyncEvent` or a subclass. An event occurrence may be

initiated by application logic (by invoking the event instance's `fire()` method) or by the occurrence of a "happening" that is external to the JVM, such as a hardware interrupt.

Each instance of `AsyncEvent` may have one or more instances of `AsyncEventHandler` associated, as shown in Figure 7. The converse also holds: every instance of `AsyncEventHandler` may have one or more instances of `AsyncEvent` associated. Every time an event occurs, the associated handlers are made eligible to run; dispatching of the handler is subject to its release parameters.

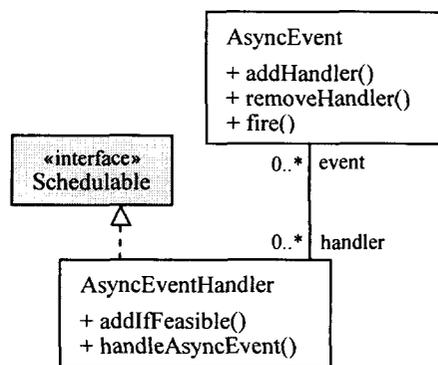


Figure 7. Asynchronous events and their handlers can have a many-to-many relationship in the RTSJ.

4. SOME IMPLICATIONS OF RTSJ

The central problem associated with using Java for systems with hard real-time requirements is that garbage collection always has non-interruptible sequences of instructions, the execution duration of which cannot be predicted. Furthermore, garbage collection must always be able to enter a critical section, and so any activity in the system is subject to unpredictable delays waiting for garbage collection. Memory allocation or de-allocation can cause a GC critical section to run, but these may run at anytime, not directly caused by a call to `new`.

The RTSJ solves this problem with two constructs: the *no-heap realtime thread* and *scoped memory*. The no-heap real-time thread (NHRT) runs at a higher priority than the garbage collector, and so is not subject to delays caused by it in normal threads. Garbage collection is locked out while a no-heap real-time thread is executing. In fact, an NHRT can preempt the garbage collector at any time, and so it must not be allowed to change anything that could affect the state of garbage collection, including heap memory or any objects allocated from it.

If code running under such a thread needs to allocate memory, this could lead to a conflict: garbage collection needs to run to satisfy the allocation request, but garbage collection can't run because the no-heap real-time thread cannot tolerate unpredictable delays. Therefore, the code in the no-heap real-time thread cannot be allowed to make allocations from memory that is managed by the garbage collector (i.e. the heap), nor make changes to heap, because these could affect the garbage collector. For each thread type, Table 1 shows what from kinds of memory it can allocate objects using 'new'.

Table 1. Thread types and the memory areas in which they can allocate objects using 'new'.

Can thread allocate objects using 'new'?	Heap Memory	Immortal Memory	Scoped Memory
java.lang.Thread	Yes	No	No
RealtimeThread	Yes	Yes	Yes
NoHeapRealtimeThread	No	Yes	Yes

One solution is to not allow code in no-heap real-time threads to allocate memory at all, but this is a severe limitation, very unattractive for Java programming. Another solution is to allow the NHRT to allocate memory, but only from a pool of memory that is not under the responsibility of garbage collection, and which can therefore be allocated from without having garbage collection eligible to run. This is what the RTSJ does with the introduction of the *scoped memory* construct, expressed by the *ScopedMemory* abstract class.

A scoped memory area is a block of memory of fixed size that can be associated with a running NHRT. The NHRT is said to *enter* the scoped memory area, and is said to be running in the scope of that memory area. When the NHRT is running in a scoped memory area, it can allocate memory from that area in the usual way, using the *new* statement. Objects allocated from the memory area exist as long as some NHRT (multiple NHRT's can use the same scoped memory area) stays in the scope of that memory area. An NHRT leaves the scope either by exiting the scope explicitly or by finishing execution. When all NHRT's have exited the scope, all objects allocated in the scope by any thread become no longer accessible. But those objects are not garbage collected, since there is no garbage collector that is responsible for freeing objects in the memory area. If a NHRT subsequently enters that scope again, it will be as if no objects were ever allocated in that scoped memory area.

A NHRT can never allocate more objects in a scoped memory area than the fixed size of that area. Thus, code running in a NHRT must be designed up front with known memory allocation needs that cannot be exceeded.

With this RTSJ solution, it is now possible to know the maximum amount of time that a sequence of instructions will experience. This assurance is achieved by running that sequence of instructions under a NHRT, which necessarily puts the execution of that code in a scoped memory area.

Compared to the normal Java programming model, programming with NHRTs and scoped memories is more complicated. Fortunately, as shown in Figure 2, most real-time systems contain a relatively small hard real-time part, so the added complexity is contained. The soft real-time part as well as the non real-time part can freely allocate memory from the heap, relying on garbage collection to reclaim those allocations when no longer needed.

The RTSJ solution also has implications for communication between hard real-time elements and other elements of the system, especially for information flow from hard real-time to soft- or non-real-time components. There is a distinct information boundary between these two types of system elements, which we

call the hard real-time boundary. An object allocated in the scope of a memory area cannot be referenced – is not even visible – to code running outside the scope of the memory area. An object on heap, allocated by non-real-time code, cannot be assigned to by hard real-time code. More formally, the RTSJ defines memory area assignment rules preclude certain kinds of references from one memory area to another. These rules are shown in Table 2.

Table 2. The RTSJ's memory area assignment rules restrict certain kinds of inter-memory references in order to separate garbage collection from hard real-time activities.

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same, outer or shared

So how then, can information computed by hard real-time system elements be communicated to soft- or non-real-time system elements? We present our design solutions in Section 6.

5. CONTROL LOOPS IN MDS

Our problem domain is that of real-time closed-loop control of physical systems. Such control systems are designed for continuous operation and they interact with the real world through imperfect sensors and actuators. In our case, these are embedded control systems that live within the resource-limited world of planetary rovers and spacecraft.

The design of our control loops is governed by the architecture of the Mission Data System (MDS), an information and control architecture that emphasizes explicit representation of physical states (continuous as well as discrete states), explicit models of hardware and physical effects, and goal-oriented operation that enables varying levels of onboard autonomy [4].

As shown in Figure 8, real-time control loops in MDS involve four kinds of components: hardware adapters, state variables, estimators, and controllers. There is a hardware adapter for each controllable hardware unit, and each one provides software interfaces for sending commands and obtaining measurements. A state variables is a component that holds information about a physical state (such as rover position) and whose value history is made available as telemetry. An estimator interprets measurements from potentially multiple sensors in order to generate state estimates. A controller compares current state estimates to a 'goal' (a constraint on the value of a state variable over a time interval) and issues commands to actuators, as needed, to influence a physical state.

The dominant data flow around a control loop involves four flows: controllers query state variables for state estimates; controllers submit commands to hardware adapters; estimators query hardware adapters for measurements; and estimators update state variables. The main challenges in software design for hard-real-time control loops using the RTSJ involve appropriate use of non-heap memory for these four data flows. In our case,

coordinated control of the 6 driving motors and 2 steering motors on the Rocky 7 rover involved 9 control loops, so we were motivated to find a good general solution.

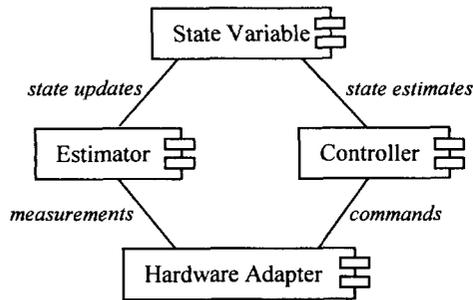


Figure 8. A simple hard real-time control loop in MDS involves data flows among four components.

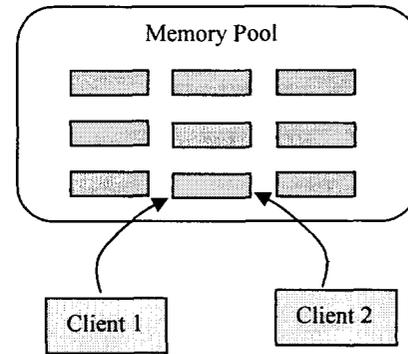


Figure 9. In a shared memory pool multiple clients share a single pool of objects of a given type, obtaining objects as needed and releasing them back to the pool when no longer needed.

6. SCOPED MEMORY SCRATCHPADS

This section describes some design considerations in how to handle information transfer between hard real-time components, and then describes the design we adopted, which we termed “scoped memory scratchpads”.

6.1 Complexities of Memory Management

The most radical change to the Java programming model when using *NoHeapRealtimeThreads* is that programmers are now required to manage their own memory. Being dissociated from general garbage collection requires the use of other means to recycle object references. One way to do that is to use a “memory pool” (sometimes called a “buffer pool”): a block of memory populated at initialization time with pre-allocated objects, often of the same type. A client that needs a particular kind of object obtains one from a pool, assigns it a value, and uses it. Eventually the object must be released back to the pool, thereby making it available for reuse.

6.1.1 Shared Pools

In our component architecture data needs to flow among separate component objects through well-defined interfaces. In ordinary Java it is natural and efficient to pass values by reference. Thus, when using memory pools, a natural approach is to have a single shared memory pool for each type of object that participates in inter-component communication. A component that produces information obtains a free object from the pool, assigns a new value to the object, and passes a reference to the object across an interface. The receiving component uses the reference to access the object and, eventually, releases the object back to the pool, as depicted in Figure 9. Since a memory pool can exist in scoped memory, a further obvious requirement is that the receiving components have access to the memory area in which the object was allocated. Also, since components may run on separate threads, pool operations must be multithread-safe.

For object references that cross component boundaries we considered a scheme in which objects in the pool would contain a reference count, which the application code would be partially responsible for maintaining. The counter would be updated through synchronized methods. Whenever a reference to an object was obtained, either from the pool or from another component, the counter would have to be incremented. The objects would have a method that atomically decremented the counter and released the object back to the pool if the counter went to zero. It would be the user’s responsibility to call that method. Although we realize that using a simple reference count to solve this problem might not be theoretically possible in the general case, we felt that due to the structure imposed by the MDS architecture and our component model we could reasonably restrict the movement of object references across component boundaries and provide some automation in controlling the reference count such that simple reference counting would prove sufficient.

6.1.2 Restricted Pools

A simpler and safer pattern involving memory pools is to restrict the visibility of a pool to within one component. Each component that needs to use objects of a given type has its own pool of objects of that type. To simplify object reference reuse we constrained the architecture so that references to an object would never be allowed to pass across component boundaries. This requires a mechanism for copying data across an interface instead of passing a reference.

Copying eliminates the need for general reference counting since there is never more than one user of a given object, but it still requires a strict discipline within the component to recycle object references in an organized manner.

6.1.3 Complexities of Memory Pools

Memory pools bring back vulnerabilities to the same kind of programmer error that is prevalent in any language requiring manual memory management. Pools require programmer discipline to return every object back to the pool when it is no

longer being used. Two kinds of mistakes can occur. First, any failure to return an object to its pool results in a memory leak, and the hardest memory leaks to find are the slow ones. Second, and worse, an application may give an object back to the pool but continue to use the reference (accidentally, of course). That kind of bug can be extraordinarily hard to find because the effects may be non-local.

6.1.4 Complexities of using Core / 3rd Party libraries in Scope

Java programmers are accustomed to using the wealth of the Java core libraries and even 3rd party libraries to construct applications. For the most part these libraries take for granted that they are being used in a VM that has GC. This can be a big problem when used in conjunction with *NoHeapRealtimeThread* and *ScopedMemory* since *ScopedMemory* is finite.

6.2 Scoped Memory Scratchpads

As stated earlier, the RTSJ provides a kind of non-heap memory area termed *ScopedMemory*. A *ScopedMemory* area can be entered by a thread, its critical section is bounded by a *java.lang.Runnable*, and any allocation that *Runnable* does after having entered the memory area—and until leaving it by exiting its *run* method—comes out of that memory area. When all *Runnables* execute the memory area, the scope will be emptied before being entered again.

We considered a scheme in which each component has its own memory area, which we call a *scratchpad*. The component's *run* method is called in the scope of the scratchpad memory area. When the component calls another component to get data, the callee can do a *new* to allocate an object to return, and that object is placed in the caller's scratchpad since the callee is running by call from the *run* method of the caller. When the call returns, the caller must either finish with the returned object before leaving the *run* method (i.e. exiting the scope), or make a copy of the object into a more permanent location before exiting the scope.

By using a wrapper on the component that handles the mechanics of entering the scope, the component's logic need not be aware that it is running in the scope of a scratchpad memory area, or even that it is running under a RTSJ-compliant VM instead of a regular JVM. It must however not keep references returned from an interface call and expect them to be valid the next time the component is run. Components that receive data by being called must only satisfy the requirement that they not hold onto a reference received in the call; if they need to have the object after returning, they must copy it. To avoid memory allocation a restricted pool is used per component for this copy. The design of our real-time thread wrapper is shown in Figure 10.

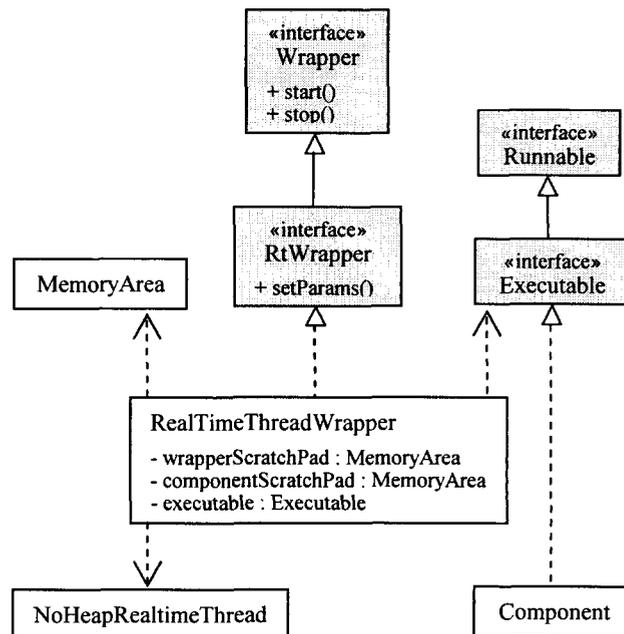


Figure 10. A real-time thread wrapper handles the mechanics of entering a scoped memory area so that application programmers can focus on functionality.

6.3 How do Scoped Memory Scratchpads simplify the RTSJ programming model?

We selected scoped memory scratchpads as the best combination of agreeable Java style, safety from programmer error, and real-time determinism. Scoped memory is “agreeable” in the sense that Java programmers can allocate and manipulate objects in a familiar manner (using ‘new’), without GC interference, but they must be cognizant of memory access restrictions and they must ensure that all threads exit the scoped memory in order to empty it. Scoped memory scratchpads *do* depend on programmer discipline to ensure that all threads exit a scope in order to empty it, but this aspect can be handled in our framework code, rather than adaptation code. Although restricted memory pools are still used in conjunction with the scratchpad approach, and thus require the discipline of releasing objects back to the pool, all of the pool management is confined to a single component and is thus much easier to design and verify.

This design also allows the use of core and 3rd party libraries without concerns of memory leaks since, on exit of the *Runnables*, transient objects are released in a more stack-based approach to garbage collection.

7. FRAMEWORK SOFTWARE BENEFITS

In the Golden Gate/Mission Data System approach to software development, we make a clear distinction between aspects of the flight system that support *particular* objectives of a mission versus services and functionality that are common across most missions. A domain expert develops the functional aspects of the flight software such as control algorithms or hardware driver implementations. Such activities are developed more effectively

when they are left unburdened by the prevalent considerations of communication between flight software subsystems, software deployments, or the complexities of underlying technologies.

One of the goals of Golden Gate is to relieve the application programmer of these concerns by addressing them with a set of common services that are encapsulated in a framework and programming model, within the context of a component/connector architectural style. This architecture style supports the separation of computation and communication, captured respectively in first class software entities of components and connectors. The adaptation of Golden Gate/MDS involves implementing mission-specific functionality in components and mediating communication of these functional elements to connectors. An objective of the Golden Gate/MDS model is to focus the adapter's work solely on domain-specific implementation, while delegating common framework patterns to a set of generic services accessible via supporting APIs.

For the Golden Gate project, RTSJ/Java-specific issues such as memory management and communication between hard and soft real time subsystems were primary candidates for framework encapsulation. In reasoning about the component scoped memory scratchpad methodology, we viewed the component as a state machine, whose execution is a set of state transitions, during which memory usage is finite and not persistent outside the scope of the execution. The component state undergoes changes based on its executions and/or executions associated with components that communicate with it. Hence, in the context of memory management, a component has a dual task: it must manage its mutable state in a persistent manner while managing objects of transient nature allocated during its execution. The use of the scoped memory construct of the RTSJ as a scratchpad for execution provided a solution for these short-lived component executions.

In order to manage the complexity of using the RTSJ scoped memory construct, we created a wrapper to easily specify and configure the memory area used and type of thread that would execute the component's methods.

The result is a programming model that was flexible enough to easily interchange between execution in RTSJ scoped memory scratchpad and heap memory and their respective thread types. As previously stated, with the exception for some programmer discipline related to data copying, this model allows for memory allocation techniques in the component that are equivalent to usual Java. By maintaining a clear separation between execution specifications and functional code, the potential interleaving of these distinct properties is reduced — the domain expert can leverage a reliable set of framework services to specify execution attributes while being removed from technical details of the RTSJ that implement them.

8. FUTURE WORK

Much of our current work is focused on measurements of performance and resource usage that can be compared between two software platforms: RTSJ/Linux and C++/VxWorks. These metrics include memory footprint, timing jitter, floating-point throughput, memory allocation time, and boot-up time. We are

also measuring RTSJ-specific quantities such as the overhead for entering/exiting scoped memory and for firing asynchronous events. This work leverages and organizes a substantial body of performance tests from several sources including Boeing Corporation, the Air Force Research Laboratory, Sun Microsystems Laboratories, the Open VM project, and Washington University in St. Louis.

We have just begun looking into some interesting work at Washington University that explores using AspectJ and additional prototype tools to automatically build memory area scope structure into perfectly regular Java code [5]. This holds promise for the future. We will keep on the lookout for garbage collectors made specifically for RTSJ, like perhaps limited scope collectors with predictable running times. See [6] for a discussion of RTSJ-specific GC optimizations. Also, we will explore the regulated garbage collection option further such as the work described in [7].

9. ACKNOWLEDGEMENTS

This work was performed jointly by the Jet Propulsion Laboratory of California Institute of Technology, by Sun Microsystems Laboratory, and by Carnegie Mellon University. The work at JPL was performed under contract with the National Aeronautics and Space Administration. The JPL team thanks the Office of the Chief Scientist for funding under the Research & Technology Development Program, and the strong support of the R&TD committee on Advanced Software Techniques & Methods Initiative.

10. REFERENCES

- [1] Greg Bollella et al, *The Real-Time Specification for Java*, Addison-Wesley, 2001. <http://rtj.org>
- [2] Project GoldenGate. <http://research.sun.com/projects/goldengate/>
- [3] <http://www.cs.unc.edu/rtss2002/absBollella.html>
- [4] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A. Software Architecture Themes in JPL's Mission Data System. Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, Montana, March, 2000.
- [5] Cytron, R., Deters, M., Automated Discovery of Scoped Memory Regions for Real-Time Java. http://www.cs.wustl.edu/~mdeters/doc/papers/automated_discovery_of_scoped_memory_regions_abstract.html.
- [6] Cytron, R., White Paper: RTSJ Memory Management. <http://ww.cs.wustl.edu/~cytron/WhitePaper00/wp.html>.
- [7] Kim, T., Chang N., Kim, N., Shin, H., Scheduling Garbage Collection for Embedded Real-Time Systems. <http://citeseer.nj.nec.com/523601.html>.
- [8] High Dependability Computing Program, <http://west.cmu.edu/research/hdcp.html>
- [9] TimeSys Corporation, <http://www.timesys.com/>