Project Golden Gate: Towards Real-Time Java in Space Missions

Daniel Dvorak[1], Greg Bollella[2], Tim Canham[1], Vanessa Carson[1], Virgil Champlin[3], Brian Giovannoni[3], Mark Indictor[1], Kenny Meyer[1], Alex Murray[1], Kirk Reinholtz[1]

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

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

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

Abstract

Planetary science missions, such as those that explore Mars and Saturn, employ a variety of spacecraft such as orbiters, landers, probes, and rovers. Each of these kinds of spacecraft depend on embedded real-time control systems—systems that are increasingly being asked to do more as challenging new mission concepts are proposed. For both systems engineers and software engineers the large challenges are in analysis, design and verification of complex control systems that run on relatively limited processors. Project Golden Gate—a collaboration among NASA's Jet Propulsion Laboratory, Sun Microsystems Laboratory, and Carnegie Mellon University—is exploring those challenges in the context of real-time Java applied to space mission software. This paper describes the problem domain and our experimentation with the first commercial implementation of the Real Time Specification for Java. The two main issues explored in this report are: (1) the effect of RTSJ's non-heap memory on the programming model, and (2) performance benchmarking of RTSJ/Linux relative to C++/VxWorks.

1. Introduction

The Jet Propulsion Laboratory (JPL)—part of the California Institute of Technology—serves as NASA's lead center for the robotic exploration of space. Many of JPL's past missions have been orbiters and fly-by spacecraft controlled in an open-loop manner using time-based sequencers, with Earth-in-the-loop decision-making expressed in the form of new sequences transmitted to the spacecraft. Increasingly, though, NASA's missions require *in situ* explorers (such as Mars rovers), where many decisions cannot await a 20–40 minute round-trip light-time delay between Earth and Mars. These in situ explorers will typically contain many closed-loop control systems, performing and coordinating multiple concurrent science and engineering

activities. Furthermore, the larger amount of software-based capabilities intended for these robots has raised a higher level of concern about system design and software reliability.

As a federally-funded research and development center, one of JPL's roles is to actively explore new technologies that help NASA accomplish its challenging space missions and help advance practices in the aerospace industry. The Java programming language was one such technology, of interest because of its simpler object model and automatic memory management, though clearly not suited for real-time applications. However, the formation of the Real-Time Expert Group in 1999 and its development of the Real-Time Specification for Java (RTSJ) marked the beginning of particular interest at JPL, led by Kirk Reinholtz. Even prior to the emergence of RTSJ v1.0, JPL and Sun conducted a study of Java for flight software, and that study identified several areas of concern where additional data was needed. As RTSJ matured and its reference implementation became available, the time came for a more in-depth evaluation.

In 2002 Project Golden Gate was formed as a collaboration among JPL, Sun Microsystems Laboratory, and Carnegie Mellon University (CMU). All three partners shared an interest in RTSJ and each brought different interests and skills to the collaboration. JPL brought knowledge of a challenging problem domain and expertise in information and control architecture for autonomous physical systems. Sun brought deep knowledge of Java, RTSJ, and real-time scheduling theory. CMU brought expertise in Java and in high-dependability computing. This active collaboration is now in its second year, as described in this paper in terms of significant issues and current results.

2. Problem Domain

2.1 Robotic Space Vehicles

Compared to many systems built for use on Earth, planetary space missions have several unusual properties that affect system design. The first two properties—long time delays and low data transmission rates—are unavoidable consequences of physics and the vastness of outer space. The round-trip light-time delay between Earth and Mars, for example, varies between 20 and 40 minutes depending on the relative positions of the planets; the delay with Pluto is over 9 hours. This means that space vehicles operating far beyond Earth orbit must be somewhat autonomous, able to maintain spacecraft health and accomplish science goals without human intervention for long periods of time. Another factor that adds to the need for autonomy is that the Earth-based facilities for communicating with such spacecraft, known as the Deep Space Network [10], are oversubscribed resources; there are many more active spacecraft than deep space antennas, so communication is scheduled

only for specific time windows. As a result, data destined for Earth must be stored on the spacecraft until the next communication opportunity.

The second property—low data transmission rates—arises from the inverse square law for power radiation, which says that received signal power decreases as the square of the distance between transmitter and receiver. Information theory also tells us that channel capacity depends on signal-to-noise ratio (SNR), and since SNR decreases with distance, low data rates are the result. One way is to improve SNR is to increase the power of the radiated signal, but that's not an option for most spacecraft since they are power-limited. As a result, data transmission rates decrease rapidly with increasing distance from Earth. For example, one design for a mission to Pluto yielded a 300 bps "downlink" from the orbit of Pluto to Earth. Such low data rates mean that communication bits must be used wisely. Since instruments on modern spacecraft are capable of acquiring far more data than can be sent back to Earth, hard choices must be made about what data to send to Earth.

A third property of planetary space missions is that flight processors are generally *years* behind their commercial counterparts because they must be radiation-hardened. The economic reality is that "rad-hard" versions of commercial processors come into being only every few years. By the time they are produced they are already lagging, and by the time they are flown, they are even further behind. As a result, CPU cycles are often a limiting resource in space missions. Flight software is usually custom-made to be frugal in CPU cycles and memory.

Finally, a fourth property arises from the economics of space missions. One of the biggest cost factors in deep space missions is the launch vehicle, which is sized to the amount of mass being launched and the required trajectory. This cost pushes designers to reduce mass as much as possible. That means reducing the size of batteries and solar panels, reducing the amount of propellant, eliminating articulation mechanisms (such as camera and antenna gimbals), reducing electromagnetic shielding, etc. To exaggerate just a bit, the net result is that "everything affects everything"; resource margins are small, making spacecraft control a delicate balancing act, trying to accomplish mission objectives without violating any constraints and without oversubscribing any resources. Spacecraft software is complex because it has to monitor and manage all these interactions.

2.2 Real-Time Control Loops

Project Golden Gate has focused much of its attention on real-time closed-loop control of physical systems. Such control systems are designed for continuous operation, interacting with the real world through sensors and actuators. In our case, these are embedded control systems that live within the resource-limited world of planetary rovers and spacecraft. As Figure 1

shows, in the problem domains of interest to JPL's missions, control loop frequencies vary considerably from a few Hertz up into the kilohertz range. Design difficulty grows with the frequency and number of control loops that run on a single processor.

Because of comparatively slow flight processors, current engineering practice at JPL usually involves of lot of performance

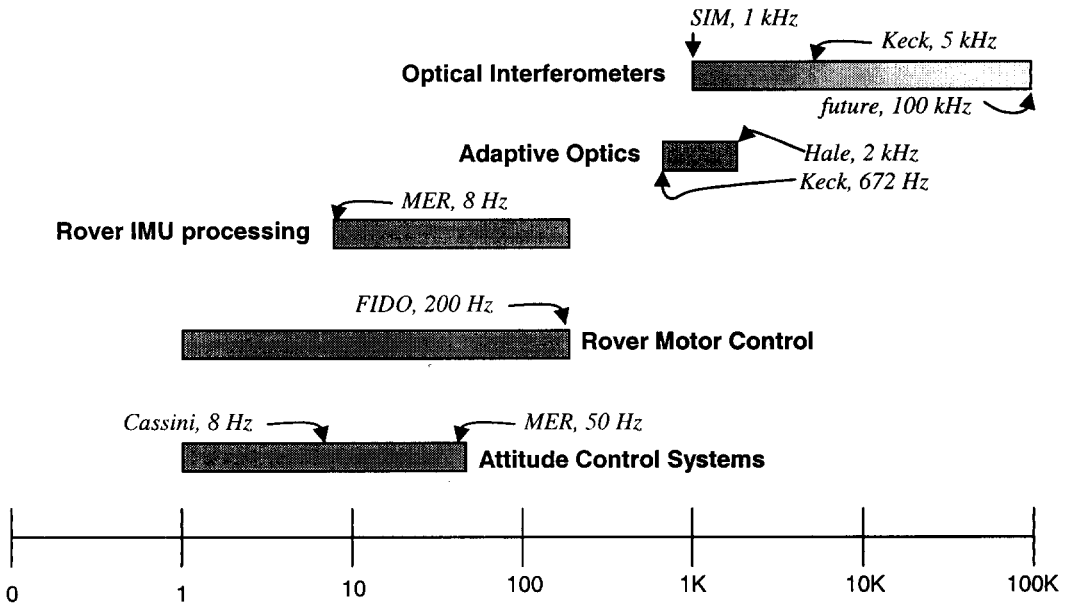

Figure 1. *Software control loop frequency in Hz*

evaluation and hand-tuning of software. For example, the Cassini spacecraft's processor for the attitude control system normally runs at 60–70% utilization, with expected periods of 80–90% load. This stands in sharp contrast to some automotive control systems where processors are sized to keep utilization below 20%.

3. Rocky 7 and MDS: A Research Platform

3.1 The Rocky 7 Rover

Project Golden Gate's work has focused in part on designing and developing software for driving and steering "Rocky 7", a research rover at JPL. Although Rocky 7 will never go to Mars, it is outfitted with the kinds of sensors and actuators that exist on real Mars rovers. In fact, the design of Rocky 7 helped shape the design of Sojourner, the rover that landed on Mars in 1997 as part of the Mars Pathfinder mission. The rover hardware includes six driving motors, two steering motors, three joint motors for the camera mast, two joint motors for the instrument arm, three stereo camera pairs, a camera frame grabber, a 3-axis accelerometer, a 1-axis gyroscope and a wireless data link. As such, Rocky 7 serves as a realistic test-bed for experimentation.

The rover's processor is a 300 MHz PPC 750 with 256MB RAM (upgraded from the original RAD6000 processor used on Mars Pathfinder). The operating system is TimeSys Linux RTOS [9], a low-latency version of Linux, and the virtual machine running on top of that is TimeSys JTime[1], the first commercial implementation of RTSJ. The work reported herein occurred as the team designed control loops for driving the Rocky 7 rover, taking pictures periodically with the hazard cameras, and transmitting telemetry to a base station.

Figure 2. Rocky 7 rover at work in the JPL Mars Yard.

3.2 MDS Architecture

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

As shown in Figure 3, 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 variable 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.

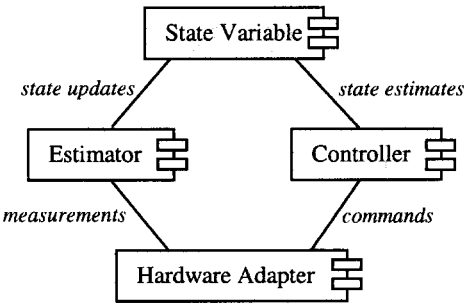

Figure 3. A simple hard real-time control loop in MDS involves data flows among four components.

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, as described in section 4. In our case, coordinated control of

[1] JTime is a registered trademark of TimeSys Corporation.

the 6 driving motors and 2 steering motors on the Rocky 7 rover involved nine control loops. A more complete rover software implementation would include additional control loops for the camera mast and pan/tilt mechanism, robot arm and instruments, uplink and downlink communications, and picture-taking with stereo cameras for both navigation and hazard avoidance.

4. RTSJ Programming Model

[*Note to reviewers: It wasn't clear to me how much background material about RTSJ that I should include for this audience. I can include a lot more if desired, as provided in reference 12.*]

4.1 Scheduling Interface

An overarching theme of the MDS architecture is to elevate cross-cutting concerns into "architecture space" where they can be seen and managed in a coherent way, *not* left as a later problem for system integration. The RTSJ's well-defined scheduling interface fits this theme well. First, it emphasizes analysis and design based on explicit timeliness requirements. Priority assignments may be a way to satisfy certain requirements, but priorities themselves are not the requirements. Second, the notion of release characteristics—*i.e.* cost, deadline, minimum inter-arrival time, cost overrun handler and missed deadline handler—as well as schedulable types—RealtimeThread, NoHeapRealTimeThread and AsyncEventHandler—help newcomers to real-time programming approach design in a disciplined way. Third, the concept of feasibility analysis is important not only at runtime but also at design time. Systems engineers can keep track of schedule feasibility even in early stages of design, using rough estimates of release parameters and refining those values as design and implementation proceed. In short, the scheduling interface of RTSJ is an important contribution.

4.2 Memory Management

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 RTSJ 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: scoped memory and

immortal memory. Objects allocated in immortal memory persist for the life of the application, so the only way to reuse such objects is to write new values into them. Objects allocated in scoped memory persist only until the scope is emptied, which occurs when all threads exeunt the scope. Table 1 shows what kinds of threads are allowed to allocate objects in the different kinds of memory areas. These new memory areas come with VM-enforced "assignment rules", as shown in Table 2, to ensure that the garbage collector's business is separated from hard real-time activities. The net result is that, when programming hard real-time activities, RTSJ programmers *cannot* follow the normal Java practice of allocating heap objects and passing around references, expecting automatic reclamation by a garbage collector. Further, they must consciously size these new memory areas.

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

	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

Consider again the hard real-time control loop depicted in Figure 3. For each execution of a component, it may generate internal transient data that never leaves the component, it may need to save some data that persists across executions, and it may also read and/or write data to other components. Since heap memory cannot be accessed by a NoHeapRealtimeThread, the solution must involve scoped memory and/or immortal memory.

In RTSJ there is a single instance of *ImmortalMemory* and all threads can gain access to it by calling a public static `instance()` method. In contrast, there can be many instances of *ScopedMemory*. A *NoHeapRealtimeThread* is a kind of java.lang.*Runnable*, and it gains access to a specific *ScopedMemory* either via constructor argument or via a call to the memory area's `enter()` method. As a *Runnable*, when its run method is called, all allocations come out of that scoped memory area until it completes the *run* method or until it enters another scoped memory area. When all *Runnables* exeunt a scoped memory area, the scope will be emptied before being entered again (all objects finalized and discarded).

In our design each component has its own memory area, which we call a *scoped memory scratchpad.* The component's *run* method is called in the scope of the scratchpad memory area. When that component calls another component to get data, the called component can do a *new* to allocate an object to return, and that object is placed in the caller's scratchpad since the scratchpad was the last entered memory area. When the call returns, the caller must either finish with the returned object or copy it into a more persistent area before finishing the *run* method. In the latter case, we give the component access to a memory pool allocated in immortal memory. The component must obtain an unused object from the pool and copy the data into the pool object. At some later time, when the pool object is no longer needed, it must be released back to the pool.

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.

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 framework code, rather than application-specific 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. For more details, see [12].

4.3 Thoughts on Real-Time Garbage Collection

One objective of the JSR-1 expert group that shaped the RTSJ was to bring real-time programming to Java programmers, but the programming model raises a hurdle. The RTSJ's ability to satisfy demanding timeliness requirements requires a significant change in the programming model enjoyed by Java programmers everywhere. However, for suitable applications, real-time garbage collection would enable programmers to apply the normal Java programming model, and thus avoid all the issues and complications discussed above. The key, of course, is whether an application's timeliness requirements and rate of

garbage production is compatible with a collector's execution overhead and worst-case latency. For example, the Metronome real-time garbage collector can achieve 6 ms pause times with 50% CPU utilization [13]. Clearly, such technology will be a better answer for some applications than trying to deal with scoped memory and immortal memory. Unfortunately, given that CPU cycles are a limiting resource in most planetary space missions, this technology is not an obvious answer for JPL.

Still, there are other approaches. It's important to note that flight software is highly engineered, not off-the-shelf, so it's reasonable to make memory management a more conscious part of design. Just as Lisp programmers learned long ago, Java programmers can learn to write code that is either garbage-free or very low and predictable in garbage production. Also, for some applications, there are perfect times for garbage collection. For example, a Mars rover can periodically "park", and if it can force garbage collection at those times, then it can avoid unexpected latencies while driving and taking science observations and communicating with Earth. The trade space here runs from fully automatic memory management (a la Java) to mixed automatic/manual memory management to completely manual memory management (a la C/C++), with performance and verifiability as top concerns, often at odds with each other. Better analysis tools may help us achieve both.

5. Performance Benchmarking

Given the comparatively lower performance of radiation-hardened flight processors, any new software technology such as RTSJ and Linux will be scrutinized with respect to its effects on CPU and memory usage. In the case of flight projects at JPL, this means comparing it to the current dominant software platform, namely, the C/C++ language running on the VxWorks real-time OS. As a result, we have focused our early benchmarking efforts on measurements that are comparable between the two platforms. These measurements include timing jitter, application throughput, memory usage, and overhead related to I/O. Preliminary results are summarized below. All tests were conducted on a PowerPC 750 running at 300 MHz with 256MB RAM and 1MB L2 cache. The Java data collection programs were run on the TimeSys JTIME Platform *without* using the ahead-of-time (AoT) compiler since it wasn't available at the time. The C++ data collection programs were run on the WindRiver's VxWorks (5.2) platform with the ACE operating system wrapper.

[*Note to reviewers: We expect to have updated results with AoT and with OVM by the time of the conference.*]

5.1 Timing Jitter

The jitter experiment was designed to determine if there was any significant difference in jitter of thread dispatch between the TimeSys JTIME (RTSJ) platform and the WindRiver VxWorks (C++) platform. Each data collection run consisted of a single, high precision, periodic test thread with zero or more non-precision background threads performing random allocation

of memory blocks ranging from one byte to one kilobyte in size. Results showed that the two platforms were comparable. As expected, RTSJ's RealtimeThreads exhibited much more jitter than NoHeapRealtimeThreads due to latency introduced by the garbage collector.

5.2 Throughput

The throughput experiment separately measured floating-point and integer operations throughput. As expected, given the lack of an ahead-of-time compiler for RTSJ, the C++ performance was much better. Specifically, the ratio of RTSJ to C++ execution time was 4.73 for logical shifting operation, 4.28 for integer arithmetic operations, and 2.24 for floating point operations. We will rerun these tests later with the AoT compiler.

5.3 Startup Delay

High-energy particles in outer space can cause "single event upsets" in a processor, causing the processor to raise a machine exception or freeze. Recovery involves rebooting, and the length of time that that takes can make the difference between a temporary glitch and a mission-ending failure. The quicker the processor can start running application code and recovering its state, the better. To our surprise, C++/VxWorks exhibited a 50% longer startup delay than RTSJ/Linux. We attribute the difference to the time VxWorks spent loading symbols and configuring the network adapter.

5.4 Disk and Memory Footprint

The following table shows the footprint needed to run a small application on each platform. The respective sizes are characteristic, but in actual use, these figures could vary widely. It should be noted that the Linux kernel requires less disk space than the VxWorks kernel, and that while the Java application code is extremely compact, the Java Virtual machine is large. Also, it should be noted that the Java application compiles to a larger image at run time than the C++ code.

Disk Footprint		**Memory Footprint**
VxWorks/C++	**TimeSys Linux/Java**	
• VxWorks kernel (including C++ libraries) – 1629K • C++ application (including ACE) - 2392.5K • Total application size - 4021.5K	• TimeSys Linux kernel – 639K • TimeSys JTime Virtual Machine (required for java applications) – 9420.8K • Java application byte code – 11K • Total application size – 10070.8 K (9.83M)	• C++ application memory usage: 156.3 K • Java application (using default JVM settings): 7780 K NOTE: Small Java applications have a higher memory requirement due to the Virtual Machine.

5.5 JNI Overhead

Originally we planned to use RTSJ's physical memory access in order to perform memory-mapped I/O directly from Java code. However, due to a limitation in JTime, we were not able to access the memory-mapped addresses. As a result, to keep

making forward progress, we implemented drivers in C and used JNI to cross the boundary between the two languages. We noticed that JNI introduces considerable overhead; copy semantics are expensive, and so the number of JNI crossings had a large effect on I/O performance. To get better performance we moved more functionality into the C/C++ code to reduce the number of boundary crossings. We plan to revisit this issue as soon as JTime supports access to the entire physical memory space.

6. Future Work

Our ongoing and future work divides into two main categories: benchmarking and programming model. In terms of benchmarking, we are developing more extensive benchmarks and will run all of them with the TimeSys AoT compiler. We also plan to run our RTSJ benchmarks on OVM [11] as soon as the OVM project completes implementation of the RTSJ memory areas (expected in February 2004). The comparison between JTime and OVM may reveal areas where performance is influenced more by the VM design than by complexities of the RTSJ.

In terms of a programming model for RTSJ, we are exploring an architecture that presents a more object-oriented face to the programmer. Estimation and control functionality that has to be run in a particular sequence at a particular period is grouped together in the same "sphere". Each sphere can expose certain kinds of data on its surface for other spheres to see and use, with the guarantees that the data's age is no greater than the period of its sphere and that clients will see the data as an immutable object. This approach emphasizes design based on timeliness constraints, rather than priorities and strict ordering, enabling potentially better CPU utilization through modern scheduling algorithms.

7. Summary

Project Golden Gate is a collaboration among JPL, Sun, and CMU that formed around the goal of experimenting with the Real-Time Specification for Java (RTSJ) and assessing its suitability as an implementation language for space missions, particularly in situ robots such as Mars rovers. The RTSJ is interesting in this context partly because it elevates timeliness characteristics and feasibility analysis to a first-class design aspect, partly because of the potential for improved programmer productivity, and partly because of the prospect for building complex, dependable control systems in a more repeatable way.

To date, the team has implemented and demonstrated mobility control of a 6-wheel experimental Mars rover using the TimeSys JTime virtual machine running on TimeSys Linux RTOS, and has conducted several performance comparisons to C++ running on VxWorks. Outer space is an unforgiving environment for software failures, so dependability is an ever-

present concern that is being examined in terms of programming model and verifiability. Intelligent use of the RTSJ's scoped memory and immortal memory are particularly important.

8. 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 Technologist for funding under the Research & Technology Development (R&TD) Program, and the strong support of the R&TD committee on Advanced Software Techniques & Methods Initiative. The authors also wish to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

9. REFERENCES

[1] Greg Bollella et al, The Real-Time Specification for Java, Addison-Wesley, 2001. http://rtj.org

[2] Project Golden Gate. http://research.sun.com/projects/goldengate/

[3] http://www.cs.unc.edu/rtss2002/absBollela.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://hdcp.org

[9] TimeSys Corporation, http://www.timesys.com/

[10] Deep Space Network, http://deepspace.jpl.nasa.gov/dsn/

[11] The OVM Project, http://www.ovmj.org/

[12] Dvorak et al, Programming with Non-Heap Memory in the Real Time Specification for Java, Proceedings of 2003 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003), Anaheim, California, October, 2003.

[13] David Bacon, Perry Cheng, and V. T. Rajan, The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. Proceedings of the workshop on Java Technologies for Real-Time and Embedded Systems, Catania, Sicily, Italy, November, 2003.