REAL-TIME JAVA FOR ON-BOARD SYSTEMS

V. Cechticky, A. Pasetti

ETH-Zentrum, Institut für Automatik, Physikstr. 3, 8092 Zürich, Switzerland
cechti@aut.ee.ethz.ch, pasetti@pnp-software.com

Abstract

The Java language has several attractive features but cannot at present be used in on-board systems primarily because it lacks support for hard real-time operation. This shortcoming is in being addressed: some suppliers are already providing implementations of Java that are RT-compliant; Sun Microsystem has approved a formal specification for a real-time extension of the language; and an independent consortium is working on an alternative specification for real-time Java. It is therefore expected that, within a year or so, standardized commercial implementations of real-time Java will be on the market. Availability of real-time implementations now opens the way to its use on-board. Within this context, this paper has two objectives. Firstly, it discusses the suitability of Java for on-board applications. Secondly, it reports the results of an ESA study to port a software framework for on-board control systems to a commercial real-time version of Java.

1 Standard Java and On-Board Applications

The attractive features of the Java language from the point of view of on-board applications include: greater safety; higher productivity; built-in support for multi-tasking; built-in support for networking operations; support for code documentation to be performed within the language; a wide user base and hence lower costs; and facilities for dynamical loading of code.

There are three major obstacles to the use of Java in on-board applications. The first is the memory and timing overheads introduced by the Java Virtual Machine or JVM. This is a problem because on-board systems are often memory- and CPU-constrained. The second obstacle is the lack of certifiable tools for developing Java applications. This is a problem because many on-board applications – typically the platform software – are mission-critical and therefore need to have very high quality levels. The final obstacle was already mentioned in the abstract and is the poor support offered by Java for hard real-time applications. This is a problem because on-board software often has hard real-time constraints (this is for instance the case of the data handling and attitude control software). This paper begins with a brief overview of the first two problems and will the concentrate on the third one.

There is general agreement that Java programs run more slowly than functionally equivalent programs written in languages like Fortran or C/C++ but there is much disagreement over the extent to which Java programs are slower. Results reported in published studies have slow-down factors in a range that goes from nearly acceptable values of 2-3 to clearly unacceptable values of 100 or more [Sch01, Mor00].

Java is a mixed compiled and interpreted language. Java source code is compiled to an intermediate form, the so-called bytecodes, and the bytecodes are interpreted by the JVM. Interpretation of high-level code takes more time than execution of compiled machine language and this was the primary factor influencing the performance of Java applications soon after the language was introduced.

In recent years, Just-In-Time or JIT compilation was introduced. In a JIT JVM, bytecodes are not directly interpreted but are compiled to machine code and then executed. Compilation is therefore done on-the-fly. Normally, the bytecodes associated to a method are compiled when the method is first executed [Kaz00]. The JIT approach has two drawbacks. Firstly, there is an overhead due to the compilation itself which must be performed in parallel to the normal application execution. This overhead is however incurred only the first time a bytecode or set of related bytecodes is encountered. In applications running over extended periods of time, it is often negligible. The second drawback is due to the fact that compilation is performed locally on a bytecode or method basis. Global optimization of the kind normally implemented by advanced off-line compilers are not possible.

Despite these drawbacks, the latest JIT compilers deliver performances that are only 20-30% inferior to those of conventionally compiled languages [Boi02]. It should also be stressed that JIT compiler technology is comparatively new and therefore further progress is still likely. One of the reasons why different studies report different Java slow-down factors is precisely due to the variation in JIT compiler performance. As JIT compiler technology matures, the performance differences between C/C++ and Java due to the different execution environment are likely to narrow. In fact, there is at least one author [Rei00] who argues that JIT compilation is intrinsically more efficient than off-line compilation and that believes that Java might eventually overtake C/C++ because of JIT. The superiority of JIT compilation lies in the greater information that the JIT compiler has about the current execution profile of an application. This information should allow it to generate machine code that is optimized for the specific execution profile of the currently instantiated application.

In a time-critical contexts, JIT compilation cannot be used because it introduces timing unpredictability – the time required to compile bytecodes the first time they are encountered. For this reason, embedded versions of Java often rely on ahead-of-time compilation where the Java bytecodes and the JVM are compiled to form one single executable that is then downloaded to the target processor. In some implementations, this type of technique can be reconciled with dynamic loading. In this case, the JVM becomes very similar to the run-time system of languages like Ada that offer built-in support for some operating system functions.

The general consensus [Sch01, Mor00, Boi02, Kaz00, Mor00b] is that the above techniques together with careful programming and use of dedicated libraries can bring the performance of Java to within about 50%-90% of that of C/C++ and FORTRAN. This, incidentally, is probably not very far from the level at which Ada95 applications are today.

The greater memory requirements of Java applications are due to the need to accommodate the JVM code as well as the application code. Note that many JVMs are built on top of an OS and hence a full Java-based system may require both an OS and a JVM. Java bytecodes however are more compact than most assembler languages. Hence, the application part of a Java-based system will often take less space than the application code of an equivalent compiled application. In the case of applications that have memory requirements that are large with respect to those of the JVM (which typically lie between 200 and 500 kbytes), a Java system may actually have a *smaller* footprint than a conventional system. Finally, in situations where ahead-of-time compilation is used, suppliers usually give the user the option to exclude from the JVM modules that are not needed which reduces memory occupation.

The above considerations should also be made in the context of the expanding resources available to on-board systems. The transition to processors like the ERC32 has increased resources by factors of at least one order of magnitude and hence memory and CPU concerns may not be very significant.

The issue of the certifiability of Java applications is more serious. There does not seem to be any tradition of using Java in safety-critical applications although suppliers of JVMs for real-time systems report that their products have already been used in mission-critical settings in networking and industrial automation applications. Achieving certifiability for a Java application may be harder to do than for comparable applications written in conventional languages because of the need to certify the JVM as well as the application itself. This is likely to be a difficult task because the complexity of a full JVM exceeds that of an ordinary run-time system. As already mentioned above, however, implementations of the JVM aimed at the embedded and real-time market usually allow JVM modules to be selectively compiled and bundled with the Java application code. This results in a system with a degree of complexity similar to that of an application written in a conventional languages. Even in such favourable cases, however, no efforts to achieve certifiability appear to have been undertaken.

The final obstacle on the way to Java in on-board system is the need to make Java compatible with real-time requirements. This problem deserves a more articulate discussion which is presented in the next section

2 Standard Java for Real-Time On-Board Applications

Unlike most mainstream languages Java includes a concurrency model. This model, however, stops rather short of supporting hard real-time applications. The Java language was designed to enforce the "write-once-run-anywhere" paradigm or WORA. The need to preserve compatibility with a wide variety of platforms and operating systems severely constrained the specification of the concurrency constructs. Their implementation within the JVM has to rely on services offered by the underlying operating system and the diversity of commercial operating systems is such that ensuring compatibility with all of them inevitably resulted in "weak" specification at Java language level. In fact, many operating systems in common usage – most notably the older versions of Unix – simply do not support real-time operation and would have been a very poor basis upon which to build RT-enabled Java virtual machine. Java applications were furthermore intended to run in a desktop/workstation environment shared with other (non-real-time) applications. Coexistence of real-time and non-real-time applications can be problematic and this too militated against giving Java real-time capabilities. These constraints resulted in a language that, when seen from the point of view of a real-time programmer, looks under-specified. Most of the basic syntactical constructs that one normally associates with real-time programming – in particular tasking, synchronization and event handling constructs – are present but their semantics is not strong enough to build programs with the level of determinism and timing predictability that is essential in the hard real-time world.

Additionally, real-time applications need a close interaction with the hardware than is common in desktop applications – arguably the natural target for Java. Java is deficient in this respect too and its deficiency is again a consequence of its virtual machine model. In order to preserve WORA, Java applications are supposed to interact only with the JVM and the language does not offer facilities to directly manipulate memory, hardware registers or interrupt vectors. As a consequence, interrupt handlers and device drivers – basic elements of most real-time applications – are simply impossible be develop in Java.

Finally, the presence of the garbage collector and the dynamic nature of the language make static analysis difficult or impossible.

An assessment of the impact of the above shortcomings on on-board systems must start from the consideration that such systems usually have a simplified real-time architecture. Typically, on-board applications are based on cyclical non-preemptive scheduling. In some cases – normally on the payload side or on some semi-experimental frameworks like ESA's OBOSS [Obo02] – fixed priority scheduling with preemption is used but preemption is the exception rather than the rule. Thus, the distance between standard Java and real-time *as it is typically used in on-board applications* is rather smaller than general discussions of real-time Java would suggest. Table 1 summarizes the main shortcomings of Java for real-time applications and, in the last column, it assess their impact on on-board systems.

Table 1: *Summary of shortcomings of standard Java for RT and their impact on on-board systems (OBS)*

Issue	Problem	Impact for OBS
Threading Model	Under-Specification: no guarantees about dispatching policy of ready threads	High: OBS need a threading model that allows timing predictability analysis
Intra-Thread Synchronization	Missing Feature: no safe means to suspend a thread until a well-defined time in the future	High: cyclical threads require a `suspend_until` facility
Inter-Thread Synchronization	Under-Specification: `wait` and `notify` facilities are present but order of release of notified threads is not specified	Low: this type of synchronization is not used in on-board applications
Access to Shared Resources	Under-Specification: no priority inversion policy is specified for `synchronized` statement	Medium: OBS normally do not need pre-emption and hence shared resources
Memory Allocation	Under-Specification: memory is allocated from the heap but allocation policy is not specified and could be non-predictable and non-deterministic	Low: in an OBS, memory is allocated only during initialization and never during RT operation
Garbage Collection	Under-Specification: gc collection algorithm and its triggering conditions are not specified and could be non-predictable and non-deterministic	High: even if memory is not allocated dynamically, gc might be triggered at unpredictable times and with unpredictable execution durations
HW Interrupt Handling	Missing Feature: no facilities for linking threads or event handlers to hw interrupts	High: handling of low-level hw interrupts important in OBS
Asynchronous Event Handling	Under-Specified: standard Java event model adequate if threading model were adequate	Low: asynchronous event handling is not used in OBS
Dynamic Class Loading	Incompatibility: dynamic class loading disrupts timing behaviour	Low: dynamic class loading is not required in OBS
Class Initialization	Under-Specification: no guarantee about the when classes are initialized	High: untimely initialization can disrupt RT operation and undermining determinism

3 Approaches to Real-Time Java

Inspection of the table above indicates that most of the inadequacies of Java for hard real-time stem from the weakness of its specifications, itself a consequence of the desire to keep the language compatible with as many operational environments as possible. Thus, one solution to the problem of making Java real-time is simply to develop a Java virtual machine that implements the real-time constructs of the Java language in a manner that allows timing predictability analyses to be performed and hence hard real-time applications to be developed. Consider for instance the problem of thread scheduling. The Java language does not give any guarantee about how threads of different priorities are scheduled. However, the language does not *forbid* any particular scheduling policy. A JVM developer is therefore free to provide an implementation that enforces a scheduling policy that is suitable for hard real-time applications. The same approach can be followed with all other shortcomings identified in the previous section that are due to under-specification. The result would be a JVM that is compliant with the language specification but that offers some additional guarantees – possibly sufficient to support some of the real-time architectures identified in the previous section.

Three commercial suppliers are offering RT-compliant implementations of the JVM. NewMonics offers PERC which is arguably the best established solution [Per02]. Esmertec offers a rival product called JBed with reduced functionalities and shorter heritage [Esm02]. Finally, aJile took a different route and developed chip that implements a RT-compatible JVM in hardware [Aji02]. This means that with the aJile concept, Java bytecodes are treated as a kind of assembler language and are directly interpreted at hardware level. Of these solutions, the last one is probably not relevant to on-board applications because of the difficulty of qualifying the aJile chip for use in space. Both the PERC and Esmertec solutions are potentially interesting and both have been used to develop industrial control systems with hard real-time constraints.

These solutions to the real-time Java problem arise from individual suppliers deciding to provide more or less idiosyncratic implementations of the Java language – possibly with additional support packages – designed to make development of real-time programs possible. An alternative route to the same goal is based on a formal definition of an extension of the language to be agreed by a community of users [Nis99]. The first milestone on this road was laid in 1999 by the National Institute of Standards and Technology (NIST) that prepared a report on the "Requirements for Real-Time Extension for the Java Platform" [Nis99]. This document became the basis for two parallel efforts to produce a specification for real-time extension of Java. One was carried out under the aegis of Sun's Java Community Process. Its output is the so-called RTSJ (standing for "Real Time Specifications for Java") which were approved as a formal extension to the Java language in November 2001 [Rtj02]. The second set of specifications are being elaborated by the J Consortium that brings together a group industry stakeholders that are independent of Sun Microsystems [Jco00]. Its output is the "Core Java" specifications.

Despite being inspired by the same set of high-level requirements, Core Java and the RTSJ have very different characteristics. The design of the former was driven by the desire to allow the development of real-time applications with speed of execution, latency and memory footprint comparable to those achievable in current C and C++ RTOS-based programs. The pursuit of run-time efficiency and small memory footprint led the J Consortium to develop a model of Java that is at odds with that used in the rest of the Java community. Their Core Java is intended to be a compiled language and compilation is not to bytecodes but directly to native code. Communications with Java applications running on conventional JVMs is possible through dedicated gateways. The Core Java execution environment is therefore best seen as a plug-in module to a standard JVM but Core Java also allows the development of stand-alone and highly optimized applications.

The intention behind Core Java seems to be to allow programmers of hard real-time systems operating in memory- and CPU-constrained environment to develop their software using a Java-like syntax. As a result, the general language model behind Core Java is rather different from that of standard Java. The most notable difference is the lack of garbage collection which again shifts responsibility for managing memory to the programmer. For this an other reasons, Core Java cannot guarantee backward compatibility (i.e. there is no guarantee that a standard Java program will execute correctly in a Core Java environment).

The RTSJ designers instead explicitly aimed at compatibility with the standard Java model. Their specifications center around the definition of APIs to allow real-time programming and their implementation implies a JVM with special capabilities to support them but otherwise identical to a conventional JVM. Both RT and non-RT threads are assumed to run within the same virtual machine.

The most obvious price that the RTSJ pay for their adherence to the JVM model is run-time and memory penalties. Their version of RT Java – unlike that of the J Consortium – will not be able to compete with conventional real-time applications built around RTOSs. They will however be ideally suited for large-scale soft and hard real-time applications where there is a concern for consistency with standard Java and a desire to use standard tools and development environments.

In another difference, the RTSJ are designed to be "open" with respect to implementation. They define interfaces and their semantics but do not mandate specific algorithms to implement them. Rather, they define a framework within which such algorithms can be defined by the developers. Where appropriate, sensible default choices are predefined. For instance, the specifications do not prescribe any scheduling algorithm but instead define a mechanism through which users can load components implementing their own scheduling algorithms. In view of its popularity, fixed priority scheduling is however provided as default scheduling algorithm. In the interest of simplicity and compactness, Core Java tends to be more specific and gives less leeway to users and future implementers.

A further crucial difference concerns the licensing model. The RTSJ were developed by an industrial consortium that had the backing of Sun Microsystems and that operated under the auspices of the Java Community Process [Sun98]. Their specifications are subject to the same licensing agreement as standard Java.

At the time of writing, the RTSJ are available in final form whereas the Core Java specifications are still in draft form. Additionally, there is one reference implementation for the RTSJ specification (from TimeSys) but none yet for Core Java.

Most observers believe that the RTSJ version of RT Java will prevail and will secure the largest presence in the RT Java market. There are at least three factors that point towards this conclusion. First and foremost, the RTSJ are better suited for soft real-time applications that will probably form the largest segment in the RT Java market. Secondly, they have the backing of Sun Microsystems and are fully integrated in the standard Java model. Finally, they are at a more advanced stage of development: more suppliers have announced support for the RTSJ than for Core Java.

From the point of view of on-board systems, the Core Java specifications seem preferable because they result in more compact run-time executives which could lead to more reliable and more easily certifiable systems. It should, however, be stressed that both versions of RT Java would be capable of implementing the typical real-time architecture in use in on-board applications: the cyclical non-preemptive architecture and the fixed priority with preemption architecture. It is also noteworthy that both versions of RT Java could support the Ravenscar profile [Bur98, Dob01, Pus01] which is the reference architecture for high-integrity applications in space.

4 An Empirical Assessment: Porting the AOCS Framework to RT Java

A full assessment of the suitability of Java for on-board applications will have to wait for the RT Java specifications to stabilize and for reliable compliant implementations to emerge. In the meantime, as a way of performing a preliminary assessment, an effort was made to port the AOCS Framework to a version of real-time Java.

The AOCS Framework [Pas02a, Pas02b, Pas01] was developed as an object-oriented reusable architecture for on-board control systems and in particular for the Attitude and Orbit Control System or AOCS. The AOCS Framework facilitates the development of on-board control applications. It is constructed as a set of plug-compatible components. The framework components are adaptable to allow easy matching of application requirements. The adaptation mechanisms are object-composition and inheritance.

The structure of an application instantiated from the AOCS Framework is shown in figure 1. The framework provides a set of "functionality managers" that are best seen as domain-specific extensions of the operating systems. The functionality managers are invariant within the target domain of the framework and are fully reusable. The functionality managers are customized to match the requirements of an individual application by combining them with application-specific components that form the top layer of the application. Because of its heavy use of object-oriented constructs, the AOCS Framework is regarded as typical of the kind of on-board applications that would be built in Java.

The prototype AOCS Framework was developed under an ESA contract in C++. In 2001, ESA agreed to an extension of the contract to port the framework to real-time Java. The porting effort was partly funded under ESA contract 13776/99/NL/MV. It was started in the dept. of Computer Science of the University of Konstanz and was terminated in the dept. of Automatic Control of ETH-Zurich. The Java version of the Framework – like its C++ predecessor – is available together with its design documentation as public-domain software [Aoc02].

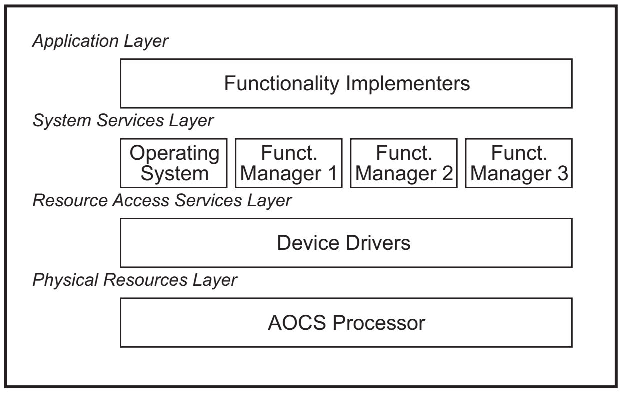

Figure 1: Structure of Applications Instantiated from the AOCS Framework

The following subsections discuss the main issues that arose during the framework porting exercise.

4.1 RT Java Platform

The porting exercise used as a target Esmertec's JBed running on a RPX Lite board mounting a PowerPC 823 microprocessor. The Esmertec JVM was selected on the basis of a preliminary analysis of its suitability and because it is supplied by a Zurich-based company (the authors are based nearby). However, in recognition of the variety of RT Java solutions currently on the market and of the uncertainty about the one that will eventually prevail, the decision was taken to make the Java version of the framework independent of the RT aspects of the language. This was achieved by writing the code for all framework components in standard Java (i.e. no use was made of constructs or services that are specific to the real-time extension of the language or to the Jbed platform). Hence, the framework, although tested on JBed, is compatible with other real-time version of Java.

The Jbed platform offers a nearly complete support of the JDK 1.1.8 APIs (personal edition) and adds to it some guarantees about real-time behaviour. In particular, it strengthens the Java threading model. Jbed schedules threads using a priority-based scheduling policy with a priority inheritance mechanism to avoid unbounded priority inversion. The more robust ceiling priority protocol is not provided. This makes it implementation of profiles like Ravenscar's impossible and generally leaves the door open to the danger of deadlocks. In addition to standard threads, Jbed provides a custom class `Task` to encapsulate threads with more flexible scheduling and task synchronization policies. In its current implementation, `Task` supports Earliest Deadline First scheduling but also allows the definition of periodic tasks. Class `Task` is integrated in the Java thread model because it is obtained by subclassing `Thread`.

Jbed provides an incremental automatic garbage collector that is scheduled as a low-priority background task. If a real-time task makes a request for memory that cannot be satisfied, the garbage collector is activated with the priority of the blocked task.

Additionally, Jbed provides a very good framework for developing device drivers in Java and for interfacing to the hardware. In particular, it is possible to link release of a task to reception of a specific interrupt.

Experience with the Jbed platform in the AOCS Framework porting project has been mixed. Two major problems stand out. Firstly, the AOCS Framework adheres to a policy of never using dynamic memory allocation in its RT part. This policy is in place to avoid dependencies on garbage collector implementations that not all RT java implementations provide. The Jbed documentation however does not document usage of dynamic memory allocation in the Java libraries and run-time services. Strict adherence to the policy of no dynamic memory allocation is therefore impossible.

Secondly, the AOCS is an inherently cyclical system and most of its functionalities are usually implemented as periodic tasks. Additionally, in an AOCS there is a requirement that sensors and actuators be sampled at specific times within a cycle and hence the periodic tasks that control the sampling must be exactly phased Jbed supports periodic tasks[1] but does not have a `sleep_until` facilities. This makes it impossible to control accurately the phase of tasks.

4.2 Scheduling and Synchronization

The framework components are designed to be used in a multi-threading environment. The framework components can be divided into two categories with respect to scheduling and synchronization issues:

- active components, namely components that can have their own thread of execution
- passive components, namely components that are not active

In a typical instantiation, to each active component, a dedicated thread is associated. How the threads are scheduled is an instantiation issue. The active components defined by the framework are written to be independent of how often they are activated and of the order in which they are activated. This makes the framework independent of the scheduling policy.

Operation in a multithreading environment requires the implementation of mechanisms to preserve data integrity. In the C++ version of the framework, this was not done because of the lack of support for synchronization mechanisms in C++. In the Java version of the framework, this is done at three levels. At the *first integrity level*, it is necessary to ensure that objects are accessed in mutual exclusion. Consider the following method call:

```
a.method();
```

execution of the method might change the internal state of object a and therefore this method call can only be guaranteed to be thread-safe if its execution is not interrupted by other threads that perform operations on object a. In the AOCS Framework, this objective is achieved by marking all methods that may be called during the operational phase `synchronized`.

At the *second integrity level*, it is necessary to ensure that data that are acquired from the same objects through separate operations are consistent. Consider the following situation:

```
v1 = a.method1(...);
v2 = a.method2(...);
. . .    // process v1 and v2
```

Suppose that there is a requirement that the two return values `v1` and `v2` are mutually consistent in the sense that, while they are acquired, the state of object a is not altered. In a non-real-time environment, the simplest way to cope with the above problem would be for the processing code to clone object a and thus create an internal copy that can be processed in the absence of interference from other threads (NB: this approach only works if the `clone` method is implemented as a `synchronized` method). However, creating internal clones requires performing a dynamic memory allocation which is not compatible with hard real-time constraints. Hence, the solution adopted in the framework is as follows:

```
synchronized (a) {
    v1 = a.method1(...);
   v2 = a.method2(...);
```

[1] But we found one bug in the Jbed kernel (version 1.2) which makes exact definition of the period of a task impossible.

```
   }
   . . .    // process v1 and v2
```
The acquisition operations are enclosed in a synchronized block that ensures that the state of the object from which the data are acquired is not altered by other threads.

At the *third integrity level*, it is necessary to ensure that data acquired from several sources are consistent. A typical example is the processing of sensor data by the controller. Consider for instance a situation where a controller processes data from two different sensors and produces data for two different actuators. Clearly, if the sensor acquisition task is interleaved with the control processing task and the latter is in turn interleaved with the actuator commanding task, a situation may arise where the controller takes as input data that have been acquired from the two sensors at different times or, similarly, a situation may arise where the actuators receive commands that have been computed in two different cycles.

The framework takes the view that this latter kind of integrity issues must be dealt with at application design level. Normally, this would be done by defining a scheduling policy that ensures that the tasks that produce certain data have terminated their execution before the task that processes those data can begin. Thus, in the example situation outlined above, one would assign a release time for the controller task that is later than the deadline for the sensor acquisition task, and a release time for the actuator commanding task that is later than the deadline for the controller task.

4.3 Framework Memory Model

Automatic garbage collection is one of the strengths of Java. Garbage collection is notoriously difficult to reconcile with real-time behaviour. Hence, although some versions of RT Java (notably the RTSJ) have incorporated garbage collection service, the AOCS Framework assumes that no such services are available. All memory required for the framework is allocated during initialization and never released during real-time operation. This policy had already been followed in the C++ version of the framework but was harder to implement in the Java version for three reasons: unlike C++, Java does not allow the creation of objects on the stack; the standard implementation of the Java event model relies on dynamic memory allocation; most JVM will use dynamic memory allocation implicitly in the run-time system or in Java libraries. The first problem was solved by avoiding all use of object variables that are local to methods. The second problem was addressed by modifying the Java event model to avoid use of the `new` operator. The third problem requires use of a JVM implementation that documents where dynamic memory allocation is used. This was not possible in the AOCS Framework porting project due to limitations of the Jbed platform (see section 4.1).

4.4 Framework Components as JavaBeans

The AOCS Framework is a component-based framework. In the C++ version, the term "component" designates an object that exposes one or more interfaces and that interacts with its clients only through these interfaces. The framework is organized as a set of such components. In the Java world, component-based applications are built on the JavaBeans standard and the framework was translated to be organized as a set of bean components. In practice, this means that all framework components are transformed into Java beans and that as many as possible of the operations involved in instantiating a framework are represented as bean operations (linking event sources with event listeners and setting bean properties).

4.5 Language Expressiveness

One reason why Java is safer than C++ is that it avoids dangerous constructs. One consequence is that Java is less expressive than C++. During the porting exercise, this created a problem in only one case. The C++ version of the framework uses templates to facilitate the construction of container classes. In Java, templates do not exist and it was necessary to create a dedicated class for each type of container with a significant loss of efficiency.

4.6 Documentation

Java includes facilities for automatically generating architectural-level documentation from the code (the `javadoc` tool). This feature was used extensively in the Java framework project to remove one level of documentation in the framework.

5 RT Java Framework Test

A framework is tested by using it to instantiate a prototype application within its domain. The RT Java Framework was tested by using it to instantiate a simple control system for a Swing Mass Model (SMM). The SMM is a standard laboratory equipment for testing control algorithms with hardware-in-the-loop. It consists of two rotating disks connected with a torsional spring. The objective of the control action is to control the speed and angular position of the right-hand disk by acting on the left-hand disk. The system actuator is the electric motor that drives the left-hand disk and that can control both its position and its angular speed. The system sensors are position and angular speed sensors on the right-hand disk. A second motor is available that can apply a torque to the right-hand disk to simulate the presence of a disturbance torque. The SMM apparatus is shown in figure 2.

In the configuration used for the instantiation of the RT Java Framework, the SMM is used as a single degree of freedom system where the angular speed of the right-hand motor is used to control the speed of the left-hand disk. The left-hand motor is not used.

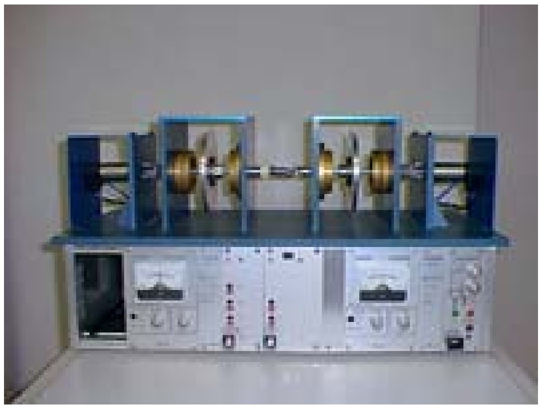
Figure 2: Swing Mass Model Apparatus

The application instantiated from the AOCS Framework to control the SMM has the following features:

- Two operational modes
- Failure detection checks on the main system variables
- Failure recovery actions autonomously executed upon detection of failures
- Provision of telemetry data in two different and alternating formats
- Processing of four user telecommands
- Capability to perform manoeuvres (activated by telecommand) to force a profile on the control set-point

The total number of classes in the prototype application is 421. This includes the classes making up the JVM run-time system. The application-specific classes are just over 200. The prototype instantiation was based on four RT threads with the following functions:

- Thread 1: periodic thread to collect and process sensor measurements (including implementation of failure detection and recovery algorithms and of control laws)
- Thread 2: periodic thread to send commands to the actuators
- Thread 3: periodic thread to send TM data to a ground station simulator

- Thread 4: periodic thread that checks whether any telecommands have been received and, if so, loads them into the telecommand manager

The period was in all cases 1 second corresponding to the control cycle of the target application. The first two threads run at very high priority and cannot be pre-empted. They only perform RT-safe operations. The bottom two threads perform RT-unsafe operations (in particular, they use dynamic memory allocation). For this reason, they run at lower priority and can be pre-empted by the first two threads.

The memory requirement for the prototype application is 522 kBytes (data+code). Of these, less than 200 kBytes are for the Java run-time system. Note that, in the Jbed execution model, the Java class files are compiled to native code and linked to the JVM. The linker only includes classes and code that are actually referred to. These memory figures are therefore highly optimised.

Typical timing requirements for the execution of one cycle of the four above threads are:

- Thread 1: 10-26 ms, depending on operational conditions
- Thread 2: 1 ms
- Thread 3: 10-17 ms, depending on amount of TM data
- Thread 4: 1 ms

The processor was as a already mentioned a PowerPC 823 and the measurements were made when it was running at a frequency of 66 MHz. Note that there are no tools for computing the worst case execution times of a Java applications and the above values were derived from observation of a large number of measurements.

6 Conclusions

There are three conclusions to this paper. The first conclusions is that standard Java is at present unsuitable for on-board applications primarily because of its lack of heritage in mission- and safety-critical applications and because it does not support real-time programming. In evaluating the latter point, however, it must be stressed that the real-time needs of on-board applications are rather limited (most on-board applications use simple cyclical non-preemptive scheduling) and that therefore the distance between standard Java and RT java is less great than general discussions of the subject implies.

The second conclusion is based on a market survey of suppliers of Java implementations that showed that RT-compliant versions of Java capable of supporting the needs of on-board applications are already available as commercial products and that they will soon be followed by versions that comply with formal extensions of the language to cover real-time needs.

The third conclusion is supported by the experience of porting the AOCS Framework to a RT version of Java. Within the limitations of the selected platform – Esmertec's Jbed – the porting exercise was very successful. As expected, the chief benefit of the switch to Java lay in the possibility of making the framework code compatible with multi-threaded operation.

On the whole, our assessment is that, once the expected implementations of RT java come on the market, implementation of real-time on-board applications in Java poses no major problem and that Java has the potential of replacing Ada as the language of choice for this class of implementations.

7 References

[Aji01] aJile web site, www.ajile.com

[Boi02] Boisverr R, *et al, Java and Numerical Computing,* downloadable from Java Grande web site at: www.javagrande.org/leapforward/cacm-ron.pdf

[Bro01] Brosgol B, Dobbing B, *Can Java Meet its Real-Time Deadlines?,* Proceedings of Ada-Europe 2001, LNCS Series Vol. N. 2043, pag. 68-87, Springer-Verlag, 2001

[Bur98] Burns A, Dobbing B, Romanski G, *The Ravenscar Tasking Profile for High-Integrity Real-Time Programs,* Ada-Europe 1998, LNCS 1411, pag. 263-275, Springer-Verlag, 1998

[Dob01] Dobbing B, *The Ravenscar Profile for High-Integrity Java Programs?*, Ada Letters, March 2001, N. 1, pag. 56-61

[Esm01] Esmertec Home Page, http://www.esmertec.com/

[Jco01] J Consortium Home Page, http://www.j-consortium.org/

[Kaz00] Kazi I, *et al, techniques for Obtaining High Performance in Java Programs,* ACM Computing Surveys, Vol. 32, N. 3, Sept. 2000, pag. 213-240

[Mor00] Moreira *et al, Java Programming for High Performance Numerical Computing,* IBM Systems Journal, Vol. 39, N. 1, 2000, pag. 21-56

[Nis99] Carnahan L, *Requirements for Real-Time Extensions for the Java Platform*, NIST Special Publication, Sept.. 1999

[Obo02] OBOSS Home Page, spd-web.terma.com/Projects/OBOSS/Home_Page/

[Pas01] A. Pasetti, *et al, An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems In Aerospace Conference, Nice, May 2001,

[Pas02a] www.aut.ee.ethz.ch/~pasetti/AocsFramework/index.html

[Pas02b] A. Pasetti, *Software Frameworks and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag, 2002

[Pas02c] www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html

[Pus01] Puschner P, Wellings A, *A Profile for High-Integrity Real-Time Java programs,* Proceedings of the 4-th International Symposium on Object-Oriented Real-Time Programming, Los Alamitos, CA, May 2001

[Per01] PERC web site, www.newmonics.com/perc/info.shtml

[Rtj01] Real Time Java Expert Group Home Page, http://www.rtj.org/

[Sch01] Schatzman J, Donehower R, *High-Performance Java Software Development,* Java Report, Feb. 2001, pag. 24-42