On the Design of a Java Virtual Machine for Mixed-criticality Systems

Hamza Hamza  
School of Computer Science  
University of Hertfordshire  
Hatfield, UK  
h.hamza2@herts.ac.uk

Alan Hughes  
e2E Services Ltd.  
Welwyn Garden City, UK  
alanhughes@e2eservices.co.uk

Raimund Kirner  
School of Computer Science  
University of Hertfordshire  
Hatfield, UK  
r.kirner@herts.ac.uk

ABSTRACT
Java has been developed with the particular suitability for embedded computing in mind due to its high portability. However, when it comes to safety-critical systems, some beneficial features of Java, like Garbage Collection, are less suitable. Over the years communities have been working on suitable compromises, leading to the work on domain-specific Java standards like the Real-Time Specification for Java (RTSJ) and more recently the Safety-Critical Java (SCJ).

In this paper we present the agenda and outline the design for a new Java Virtual Machine (JVM) for mixed-criticality systems, with the particular emphasis on not forbidding the use of standard Java libraries within the non-critical tasks. We propose a high-level design of a JVM featuring design ideas to allow support for mixed-criticality systems.

Keywords  
Java, Safety-critical Java, real-time systems, mixed-criticality systems

1. INTRODUCTION
Java, as an object oriented programming language introduced by Sun Microsystems in 1995, is widely adopted in many sectors due to its code reliability, portability, maintainability and automatic memory management. Although Java embraces a multi-threading environment, it lacks some of the important characteristics that make it suitable for real-time systems or safety-critical systems, because of non-deterministic timing behaviour. The automatic memory management and an unpredictable threads scheduling order cause non-deterministic timing behaviour.

This has motivated the research community since 1996 for-wards making Java suitable for real-time systems [3]. The Java community Process (JCP), founded in 1998 and supported by IBM and Sun Microsystems, proposed the Real Time Specification of Java (RTSJ), also known as JSR 282, that outlines seven areas of enhancements for real-time applications. These are: thread scheduling with priority based techniques, new memory management based on scope techniques where garbage collection does not interfere, resource sharing management, asynchronous event handling, and asynchronous transfer of control, asynchronous thread termination and physical memory access [1].

However, RTSJ is not fully suitable for safety-critical systems, e.g., it has some technical issues related to the scheduling, raw memory access, and managing responses to external events [4]. As a consequence, Safety-Critical Java (SCJ), also known as JSR-302, has been introduced to enable the creation of safety-critical applications using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B [6]. SCJ defines a subset of RTSJ, intended for safety-critical applications. For example, the scoped memory model of RTSJ has been restricted in order to allow static analysis of the memory usage. The thread model is limited to periodic and asynchronous event handlers in order to simplify the schedule-ability analysis. The concept of missions and sub-missions at higher levels reintroduces dynamic features of RTSJ in a safer form. SCJ also puts emphasis on using periodic event handlers instead of threads.

In this paper we outline our work towards the development of a JVM for SCJ that supports mixed-criticality applications, without prohibiting the use of standard Java libraries in tasks of low criticality. The implementation of this JVM will be done in Ada.

2. RELATED WORK
A recent open source implementation of SCJ is HVM, which is an open source Java execution environment under the license of GNU Lesser General Public License (LGPL). Being licensed under LGPL basically means that HVM includes an open source Java-to-C compiler with an embedded interpreter. It can translate a Java program into C code. The input to the translation process is a set of Java source files and the output is a set of C source files. HVM produces self-contained ANSI C code that can be compiled using a cross compiler for the desired target. To support the task isolation concept of SCJ, the HVM uses a small hardware abstraction.
layer. It introduces primitives for pre-emptive task scheduling, memory management, device access through hardware objects; first level interrupt handling, a monitor, and a real-time clock [5].

Besides SCJ, another attempt for using Java in safety-critical application domains is Ravenscar-Java, which is a subset of both standard Java and RTSJ [2]. Ravenscar-Java also supports Isolates, which is necessary to develop mixed-criticality applications.

Pure implementations of either SCJ or Ravenscar-Java forbid the use of many standard Java libraries, regardless of whether a task implements a service of low or high criticality. Our focus is to provide a virtual machine architecture that overcomes this limitation.

3. INDUSTRIAL MOTIVATION
In this section the industrial motivation to develop a new JVM for mixed-criticality systems is presented.

3.1 Development of Satellite Terminals
While the technology underpinning satellite communications has advanced over time, the process followed by many manufacturers when developing a new end-user satellite terminal has not significantly evolved since the 1970’s. A development program will typically develop the hardware platform and software components of the terminal in parallel. Until the terminal’s hardware platform becomes available, software developers will use a surrogate system, normally referred to as the Host Development Environment (HDE), to support the development of the terminal’s software components.

HDEs are often based on standard commercial platforms, and hence have very little in common with the terminal’s hardware platform. As a direct consequence of this the software developers are forced to port the terminal’s software components from the HDE to the terminal’s hardware platform once the latter becomes available. The HDEs are normally discarded at this point since their further use would only entail a repeat of the porting exercise (something no terminal development wishes to go through more often than absolutely necessary).

However, the problems implicit in the development processes used by satellite terminal manufacturers does not stop here. While modern software development processes place considerable emphasis on the concepts of “code re-use”, satellite terminal manufacturers often start with little more than a clean sheet of paper and (sometimes) a commercially-obtained protocol stack that requires significant tailoring in order to adapt it to the hardware platform. As a consequence the development of a new satellite terminal is normally considered a very costly exercise that typically requires between 2½ and 3 years to complete (measured from commencement of the development program to the commercial release of the terminal).

3.2 The Role of Real-Time Java
Satellite terminal manufacturers have typically discounted Java as a tool for supporting the development of new terminals. The common perception is that Java is resource-hungry, carries a high processing overhead due to the garbage collection activities, and is incapable of achieving the millisecond (and sometimes sub-millisecond) scheduling accuracy required by the software components incorporated into the satellite terminal. While these criticisms are certainly true for conventional JVMs, experience in the use of real-time JVMs in other industries indicate that this latter technology could easily evolve to a point where it could be used as the primary programming tool supporting satellite terminal development.

We expect multiple benefits from using a real-time JVM to host a satellite terminal’s software:

1. Simple portability of functional behaviour to a new hardware platform (correct timing behaviour still has to be shown)
2. Significant development-cost reduction by exploiting the vast amount of freely available Java-based tools and components, many of which have been developed to industry-production quality.
3. Ample availability of software experts, as Java is currently very fashionable in computer science curricula.

3.3 Real-Time JVM Industry Requirements
In order to define the principal industry requirements for Java it is necessary to determine the principal characteristics of a satellite terminal’s software components.

![Figure 1: Conceptual Satellite Terminal Software Architecture](image-url)
regime (see Figure 1). The lowest level contains the device drivers, interrupt handlers and related elements; these typically have rigid real-time constraints that may require responses in the micro-second domain. Immediately above these lies the satellite terminal’s protocol stack(s); these often have much softer real-time constraints, with the protocols being specified with responses in the millisecond range (although some elements of a satellite protocol, for example the timing of transmission bursts, may require sub-millisecond accuracies).

Above the protocol stack are a set of high-level drivers that interface to the protocol stack, and which typically implement the satellite terminal’s core functionality. For example, in the case of a satellite terminal designed to transport Internet Protocol (IP) traffic, this layer would include bridges between the terminal’s network and satellite interfaces, packet encapsulators, firewalls, Quality of Service (QoS), Performance Enhancing Proxies (PEPs), etc. Typically, these components of the satellite terminal will not have any mandated real-time constraints; however the satellite terminal’s manufacturer would want them to achieve the best operational performance for them as possible. Hence these components can be classified as being “semi real-time”.

At the top of the software stack lies various software components (for example a terminal manager, user interface, configuration manager, et al) that have no real-time constraints, and hence can (in principal) be executed in either a conventional JVM or in a real-time JVM using conventional-scheduled threads. The latter approach is preferable since it eliminates the intrinsic overheads associated with the concurrent support two executing JVMs.

We can therefore identify a core requirement that will enable industrial satellite terminal manufacturers to exploit the full benefits of Java in their products: a JVM is required that is capable of efficiently supporting multiple concurrent threads ranging from “soft” real-time through to no real-time, with the particular demand that the implementation of the non-real-time part can deploy the libraries of standard Java.

Unfortunately, existing real-time JVMs do not provide the use of standard Java libraries for the non-real-time part of the application. Thus, the use of existing real-time JVMs would demand that all components of an application are implemented purely with their special libraries, irrespective of whether they have real-time requirements or not, which demands higher development

4. HIGH-LEVEL DESIGN OF JVM

The design of this safety critical Java virtual machine is still at its preliminary stages, we aim to make the design robust, clear, and maintainable. The design approach is based on multiple-level architecture starting from high level abstract design to detailed class/method diagram. Figure 2 shows the first level of design which includes 5 packages, each package has a list of classes:

- Java virtual machine package: this includes the class loader, execution engine (interpreter) and the relative classes that help in linking the class files and extract-

5. A JVM FOR MIXED-CRITICALITY

In this section we outline some considerations for the inter-comm package mentioned in Section 4.

One of the aims of our design of a safety critical Java virtual machine is to support Safety-critical Java (SCJ) as well as standard Java as a form of hybrid Java virtual machine. We intend to provide this coexistence of different programming paradigms in order to support mixed-criticality applications. The central concept of a mixed criticality system is that it provides services of different criticality. Services of higher criticality are considered to be more central for the system utility than services of lower criticality. Thus, if a system

Figure 2 shows also a standard Java virtual machine that may communicate with the Safety critical Java virtual machine through the inter-comm library.
follows short of resources required to provide all services, then it would be a rational strategy to first reduce the quality of services of lower criticality.

Typically, the program behaviour of services of different criticality is different as well. The program code that implements services of high criticality has to be more predictable than the code of services of low criticality. What that means in the concrete case of course depends on the application domain, its applicable certification standard, and the concrete criticality classification.

For services of higher criticality we typically require: predictable and boundable resource demands, design for predictability, highly static behaviour, isolation of subsystems.

For services of lower criticality we typically focus on: support of dynamic behaviour, e.g., adaptation and reconfiguration, design for openness, performance-optimisation of the common case, avoidance of safety-specific development processes to safe costs.

From these quite diverse lists of demands typically arising for services of higher and lower criticality, we have chosen the quite radical approach of aiming to support SCJ as well as standard Java. For example, a team of Java developers writes the network communication of an application in standard Java, while specialised experts write the code for a safety-critical control loop in SCJ. To make such an development process feasible, requires a strict separation of subsystems dealing with different criticality levels. A key aspect for separation is the decoupling of the relevant subsystems in multiple dimensions, for example decoupling in time, space and synchronisation.

5.1 Inter-Comm Package

Our approach for decoupling of subsystems is based on a dedicated Inter-Comm Package, which provides stream-based message communication with adequate decoupling mechanisms. The principles of these communication interfaces for mixed-criticality systems has been proposed by Maurer and Kirner [7]. The semantics of these mixed-criticality interfaces depends on the progress model of sender and receiver components as well as the message type. Using the SCJ terminology, the sender and receiver components are event handlers. We basically differ between the following progress models:

1. Time-triggered Activation: The activation of an event handler is based on a static schedule driven by progress of time. This activation pattern maps to the cyclic executive required for compliance Level 0 of SCJ.

2. Event-triggered Activation: The activation of an event handler is driven by the arrival instant of input messages.

3. Bounded Event-triggered Activation: The activation of an event handler is driven by the arrival instant of input messages. However, the input rate is limited to a specified upper rate as described in [7]. This rate limitation can cause message losses if the actual rate exceeds the rate limit.

For the reasons discussed in [7] the communication of event handlers with different activation pattern can cause loss or duplication of messages. In order to reason about the compliance of message loss/duplication with the required functionality and also to provide a basis for optimisations, we also envision a labelling of the message types. Without any further specification the default type of a message is event message. Alternatively, a message can be of type semi-state message or state message [7]. The identification of the most precise message type will be important to design and reason about eligible fault-tolerance mechanisms of a system.

While going through the refinement of your design of the Java virtual machine, we will work on a more realisation of the proposed Inter-comm Package for provision of mixed-criticality interfaces.

6. CONCLUSION

In this paper we have outlined the ongoing development of a safety critical Java virtual machine (JVM) supporting the development of mixed-criticality systems, with support for standard Java libraries in non-critical tasks. We have provided a business rationale of why we see the development of such a JVM as feasible and important. The refinement of our outlined high-level JVM design and its implementation is ongoing work.

7. REFERENCES


