About 15 years of real-time Java

Conference Paper · October 2012

DOI: 10.1145/2388936.2388943

1 author:

M. Teresa Higuera-Toledano
Complutense University of Madrid
60 PUBLICATIONS **293** CITATIONS

SEE PROFILE

About 15 years of Real-Time Java

M. Teresa Higuera-Toledano
Universidad Complutense de Madrid
Ciudad Universitaria
Madrid 28040, Spain
mthiguer@dacya.ucm.es

ABSTRACT

Java is an object oriented programming language introduced by Sun Microsystems in 1995. From this date, Java has received a high interest from both industry actors and researchers. Java presents many benefits such as reliability of code, portability because its neutral architecture, and security in distributed environments, which leads it a privileged technology to develop embedded and distributed applications. Even though Java has issues as threads and automatic garbage collection, it has problems regarding its use in real-time systems, which must be solved. Since 1997 several research works has been focused on the limits of the Java language and its execution environment to seek the possibility for real-time technology using Java. This paper outlines and discusses the investigated issues and the developed supporting technology that allows the construction of real-time systems using Java.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Real-time**]: Java language—*complexity measures, performance measures*

General Terms

Standarization, Real-time Systems, Java environment, Survey

Keywords

Real-time Java, Distributed systems, Embedded systems, Memory management, Time-critical systems, Complex systems, High-safety systems

1. INTRODUCTION

This paper presents some significant research work on the use of Java in embedded real-time systems, and discusses them based on their innovative ideas and novel contributions on this topic. Real-time and embedded applications cover an extremely wide variety of domains, each within the deep diversity of stringent requirements, including highly precise timing characteristics, small memory footprints, flexible sensor and actuator interfaces, and robust safety characteristics. The interdependence between functional and real-time semantics of real-time software makes its maintenance especially difficult. In addition, embedded software systems are not portable as they depend on the particular underlying operating system and the hardware architecture. Moreover, this type of systems often involves distributed processing (e.g., missile defense, nuclear power plant, automobile braking or aircraft flight control) in which communication and synchronization and their resulting integration present additional challenges.

Since many embedded and real-time applications can be categorized as mission-critical or safety-critical (e.g., which critical human infrastructures as automobile braking or aircraft flight control) where even human life is sometimes at stake, a major part of the cost of creating these kind of embedded real-time applications is spent in the complexity of software development, its integration, verification, and validation. Therefore it is essential for the production of distributed real-time embedded systems to take into account modern languages and methods that enable higher software productivity and easier maintenance.

In general-proposal applications, the Java environment has become an attractive choice because of its safety, productivity, code reutilization, and low maintenance costs. Since embedded systems normally have limited memory, an advantage that some Java platforms present is the small size of both the Java runtime environment and the Java application programs (e.g., for embedded systems) And the dynamic loading of classes facilitates the dynamic evolution of the applications embedded in the system. Additionally, Java provides threads and automatic garbage collection, which make it a good environment to develop real-time applications.

Although Java has good software engineering characteristics, it presents important lacks which makes it unsuitable for developing distributed real-time embedded systems, mainly due to under-specification of thread scheduling and the time indeterminism introduced by the *Garbage Collector* (GC). To address these problems, some significant extensions to Java have been introduced by such efforts as the *Java Community Process* (JCP) *Real-time for Java Expert Group* created in December 1998 [63].

2. REAL-TIME JAVA SOLUTIONS

The Java Standard Edition presents some critical problems preventing it from being directly suitable for embedded and real-time systems: *(i)* it does not support a strict priority-based threading model, *(ii)* its locking mechanisms (e.g., synchronized methods) do not support priority inversion management techniques such as the use of priority inheritance or priority ceiling, and *(iii)* its garbage collection behavior often introduces unbounded delays that are incompatible with meeting real-time constraints.

In order to solve these problems, the *Requirements Group for Real-time Extensions to the Java Platform* including experts from both industry and academia worlds was formed. This group worked under supervision of the *National Institute of Standards and Technology* (NIST) to produce a basic requirement document [22].

Solutions that comply with the NIST requirements [22] are API-based extensions and resulted in two initiatives: *(i)* The *Real-time Specification of Java* (RTSJ) [64] backed by IBM and *Sun Microsystems*[1], created in 1998 under the JCP supervision is the first *Java Specification Request* (JSR) as JSR-1; and *(ii)* the *Real-Time Core Extension Specification for the Java Platform* (RT Core) [24] from the *J Consortium*[2], which consists of two separate APIs: the *Baseline Java API* for non real-time Java threads, and the *Real-Time Core API* for real-time tasks. However, since RT Core proposed modifications to the Java language, it was not well accepted by the Java community.

2.1 Real-time Java Preliminary Solutions

There are some API-based solutions that were introduced before the NIST document. One of the first and the simplest one is the *Real-Time Java Threads* (RTJT) [55], introduced by Kakuda in 1997. This solution is a prototype that introduces tasks support providing real-time Java threads and real-time synchronization mechanisms. It was implemented as an extension package to the *Java Virtual Machine* (JVM).

However, the first ideas about the use of Java in real-time systems was introduced just one year before (i.e., in 1996 by Nilsen [58]). The same author introduces two years later another proposal, the *Portable Executive for Reliable Control* (PERC) [59] [60], providing two packages: the `Real-Time` package provides abstractions for real-time systems, whereas the `Embedded` package provides low level abstractions to access the hardware.

Another real-time extension of the Java API previous to the NIST document, is based on the CSP Algebra, Occam2 and the *Transputer* microprocessor [40], which deals with single- and multi-processor environments. This solution implements the CSP channel concept in Java by attaching the priority scheduling to the communicating channels rather than to the processes, and scheduling is not part of the operating system becoming part of the application.

Other preliminary approaches attempt to integrate real-time capabilities in Java-based operating systems (e.g., *JavaOS* [48], the *Java Software Co-Processor* (JSCP) [61], and the University of Utah prototypes *GVM* and *Alta* [93]).

We can find also a real-time Java solution addressing the heterogeneity of distributed environments [47]. This solution is based on a custom version of the *Chorus* distributed real-time operating system that supports a JVM over an ATM network. This extension provides real-time IPC services and includes *Earliest Deadline* scheduling.

A different solution lies in the integration of the JVM on a chip, by developing a microprocessor that runs the Java bytecodes as its native instruction set [52]. The memory design in this microprocessor, named *picoJava*, is based on a stack, and includes a RISC-style pipeline having support for thread synchronization and garbage collection. This core design never was implemented, although several semiconductor manufacturers have had the license to do it (e.g., the *JEM-1* microprocessor [7]).

In [34] we overview and examine these proposed solutions, which in general unfortunately, the groups supporting them were fragmented and did not have a continuation in a clear direction. However, the more sophisticated and complete solution (i.e., PERC [59]) is still active, presenting four platforms: *(i)* PERC *Ultra* for large applications, *(ii)* PERC *Pico* for small footprint applications, *(iii)* PERC *SMP* for multi-core and multi-processor systems, and *(iv)* PERC *Raven* for safety-critical applications.

2.2 Embedded Systems and Java

Before the JCP creation there were also some Java platforms defined to support real-time and embedded systems. In particular *Java Card*, *Embedded Java* and *Personal Java* are three different platforms from *Oracle*[3]. But none of these Java adaptations integrate the techniques and methods that real-time systems require.

Java Card [62], was presented in 1.997 to run applications on smart cards and devices having very limited memory and processing capabilities. It is too limited for most applications; it does not support threads and garbage collection.

The *Personal Java* platform, has been superseded by the *Connected Device Configuration* (CDC) developed under the JCP as JSR-36 [70] (CDC 1.0.2) and JSR-218 [66] (CDC 1.1.2). The CDC addresses the software that runs on devices like smart phones, does not have any important real-time features such as real-time memory management, real-time exception handling and real-time threat management.

The *Embedded Java* platform, has been superseded by the *Connected Limited Device Configuration* (CLDC) developed under the JCP as JSR-30 [68] (CLDC 1.0) and JSR-39 [71] (CLDC 1.1). The CLDC is basically the same as CDC except that it works without standard screens, and also has a more modular and scalable implementation.

3. THE REAL-TIME JAVA SPECIFICATION

On July 2006 the final release of the RTSJ version 1.0 was approved. However, the optimal implementation of some features of the RTSJ is still an open research subject. The RTSJ is the main standard for real-time systems implemented in Java, and several commercial and research implementations of the RTSJ are available today. The NIST document also gives some examples of possible profiles to augment or restrict the core of real-time functionality; among them are the distributed real-time and safety critical cases.

RTSJ is now under development within JCP by the membership of the JSR-282 [67]. This proposal addresses some of the simpler enhancements that have been requested in

[1]Oracle acquired Sun Microsystems in 2009

[2]This subgroup is formed by 23 companies, some of them are: *Aonix*, *Ericsson*, *Hewlett-Packard*, *Lynx*, *Microsoft*, *NewMonics*, *Siemens*, *SoftPLC*, and *Yokogawa*.

[3]These three platforms were created by *Sun Microsystems*.

Implementation	Developer	RTSJ Compliant	Type	Java Compatibility
RTSJ-RI	Timesys	YES	Refernce Implemtation	JRE 6.0
Mackinac	Sun Microsystems/ Oracle	YES	Commercial	JRE 5.0
JRockit	Oracle	NO	Commercial	JRE 6.0
IBM Websphere	IBM	YES	Commercial	JRE 6.0
Real-Time JRE	Apogee	YES	Commercial	JRE 5.0/J2ME
Aonix PERC	Atego	NO	Commercial	NO (property keywords)
Jamaica	Aicas	NO	Commercial	JRE 5.0
aJ200	aJile	NO	Commercial	NO (property bytecodes)
J-Rate	California University	NO	Open Source	JRE 1.4-5.0
OVM	Purdue University	NO	Open Source	JRE 1.4-5.0
LJRT	Lund University	NO	Open Source	JRE 1.4-5.0
JOP	Viena University	NO	Open Source	NO (property bytecodes)
Fiji	Purdue University	NO	Open Source	JRE 6.0

Table 1: Some Real-time Java implemented solutions

the first release of RTSJ (i.e., the JSR-1) [64]. The JSR-1 explicitly targets all Java platforms, but the primary target is embedded platforms (Java ME/CDC). The proposed revision continues this policy, and promotes its implementation of Java SE.

3.1 Real-time Java Implementations

Since the publishing of the RTSJ official release (e.g., the JSR-1) on January 2000, several implementations of the specification are developed (see Table 2.1). Among them, the reference implementation (e.g., JTIME from TimeSys [91], commercial implementations (i.e., the Java RTS of Oracle (i.e., *Mackinac*) [54], *JRockit* Real-Time and Mission Control also from Oracle [73], *WebSphere RT* of IBM [42], *Jamaica* of Aicas [84], Aonix PERC [6], *aJ200* of aJile [3] or *Aphelion* of *Apogee* [5]), and experimental platforms (e.g., *JRate* [1], OVM [30], LJRT [94], JOP [81], and *Fiji* [75]). Each implementation keeps and extends RTSJ in different ways (e.g., Jamaica does not support RTSJ memory areas).

3.2 Certifiable Safety-Critical Java

Since hard real-time and certifiable safety-critical systems requires specific characteristics such as: *i)* static resource allocation and usage, *ii)* minimal temporal conflicts, and *iii)* validation standard process (e.g., DO-178B / ED-12B). The real-time community has initiated, on July 2006, another JCP standardization process (i.e., the JSR-302 [69], which defines those capabilities needed to use Java technology to create safety critical applications. This specification, which have its first draft on April 2011, is based on RTSJ, and contains minimal features allowing support the specific characteristics to construct this type of systems (e.g., without dynamic loading and garbage collector), and leading the ability to validate implementations using a variety of standards, formal models, scheduling analysis, and modified condition/decision coverage analysis.

3.3 Distributed Real-time Java

In order to allow the RTSJ execution on distributed platforms, the JCP defined later, in 2002, the *Distributed Real-Time Specification for Java* (DRSTJ), as the JSR-050 [72], which has been cataloged as *inactive* for several years (i.e., from 2002 until the end of 2011). A new expert group has picked up this JSR on February 2011.

The main goal of this specification is to provide a programming model for constructing distributed systems allowing sequential control flow applications with end-to-end timeliness properties. The main idea of the DRTSJ profile is to extend the RTSJ with a real-time implementation of the *Java's Remote Method Invocation* (RMI).

4. RTSJ-BASED SOLUTIONS

Much of the work in real-time distributed, embedded and real-time Java has focused on RTSJ as the underlying base technology, and consequently we mainly address issues with, or solve problems using, this framework. We cam find a study about the current state of the art in using Java in in real-time embedded systems [39]

RTSJ identifies seven areas requiring enhancements to enable the creation, analysis, execution and management of real-time threads:

(i) Thread Scheduling and Dispatching.

(ii) Memory Management.

(iii) Synchronization and Resource Sharing.

(iv) Asynchronous Event Handling.

(v Asynchronous Transfer of Control.

(vi) Asynchronous Real-time Thread Termination.

(vii) Physical Memory Access.

Besides the requirements defined by the RTSJ itself, additional requirements were also included along with asynchronous transfer of control and memory allocation. Then, a new JSR addresses the RTSJ version 1.1 (i.e., the JSR 282 [67]) was created on September 2005 (i.e., just before to approve the JSR-1).

This section deals with several studies and solutions related to real-time analysis and mechanisms to ensure that the Java run-time environment is able to meet real-time constraints.

4.1 Thread Scheduling and Dispatching

The NIST document establishes to support priority based scheduling first, and later to improve it. The advantage of a priority-based assignment scheme is that the scheduling overhead is low, because it is designed to prioritize the interrupt handling system, and the priority mechanism is often supported by the hardware.

RTSJ introduces the concept of schedulable objects (i.e., real-time threads and asynchronous event handlers), which have parameter classes representing resource demand (i.e., scheduling, memory or release) characteristics, and bounds. The base scheduler is priority-based, preemptive, with at least 28 unique priorities, run-to-block, and can perform feasibility analysis for a schedule.

The framework for scheduling presented in [97] enhances the RTSJ environment allowing the simultaneous execution of several applications, some of them with different real-time requirements (i.e., soft, firm, and hard); allowing its cooperation. All threads, regardless of the policy under which they are scheduled, can share common resources. This approach introduces a hierarchical two-level schedule, where the first level is the RTSJ priority scheduler and the second level is under the application control specified by the programmer.

Reflexes [87] presents a different real-time Java programing model, which motivates the need for revisiting the concurrency features of Java to reduce latency. In this model, a program consists of a graph of tasks connected through unidirectional communication channels as in graph-based modeling systems typically used to design real-time control systems (i.e., *Simulink*).

A general RTSJ scheduling implementation consists to map each Java priority to an associated operating system base priority, where each operating system priority is associated with a scheduling policy. The WebSphere [42] JVM suports only two of the three linux scheduling polices[4] (i.e., SCHED_FIFO for real-time tasks and SCHED_OTHER for normal Java threads).

4.2 Dynamic Memory Management

The memory management is one of the major issues that still needs research in RTSJ [90]. Although there has been extensive research work in the area of making garbage collection compliant with real-time requirements (e.g., [77], [43], [8], [85], [29]), there is still problems to use this technique in time-critical systems.

[4]The Linux operating system supports three scheduling policies, one for standard threads (SCHED_OTHER), which is he default universal time-sharing scheduler policy that is used by most threads (i.e., non-real-time threads with priority of zero), and two for real-time applications: SCHED_FIFO and SCHED_RR (i.e., with priority higher than zero).

An appealing solution to overcome the drawbacks of GC algorithms is to allocate objects in regions associated with their lifetime. Besides the traditional heap memory, and the JVM stack, RTSJ introduces three memory region types: scoped memory, which manages short-lived objects whose lifetime is defined by a scope; physical memory, allowing objects to be allocated in a specific physical memory region; and immortal memory, an area containing persistent objects. Some tasks can allocate and reference objects within the heap, whereas others (critical) are not allowed to allocate nor reference objects within the heap.

4.2.1 Memory Regions

Since *scoped memory regions* can be nested, the full implication of the memory assignment rules must be considered and studied in detail. The *Scoped Types and Aspects for Real-Time Systems* (STARS) [4] presents a novel programming model, which can be checked statically and prevents the run-time memory errors suffered by RTSJ (e.g., assignment rules). However, given the dynamic character of the Java language, some illegal references must be checked at run-time. A solution is to use write barrier code [37].

Given that the success of RTSJ depends on the possibility of offering an efficient implementation of the assignment restrictions, the research in this area is an interesting open field. Some researchers have produced new programming paradigms and patterns in order to reduce the complexity of programming using scoped memory regions (e.g., [25], [15], [19], and [14]).

In [76], the authors argument that while programming with scoped memory regions is error prone, it provides substantially better throughput that the GC (i.e., a throughput reduction of up to 37% and, in the worst-case a 80% of latency).

4.2.2 Garbage Collector

When using incremental GCs, two major sources of blocking are exhibited: root scanning and heap compaction. Finding root nodes of an object graph is an integral part of GC, while heap compaction is necessary to avoid unbounded heap fragmentation.

Objects allocated within memory regions may contain references to objects within the heap, the collector must take into account the external references (e.g., as root nodes [36]), adding them to its reachable graph. Whenever an external reference is not used anymore (e.g., when a scoped memory regions is collected), it must be removed, ensuring its reclamation by the GC when it becomes garbage.

A GC solution avoiding fragmentation is in the *Jamaica* VM [85], which divides objects and arrays into fixed-size blocks that are arbitrarily allocated in memory. While the blocks composing an object are linked in a list structure, the blocks composing an array are linked in a tree structure. A typical access to an object field may require only one access, but the worse case depends on the object size.

An interesting solution to the unbound size of arrays was given by the *Metronome* GC [8], in which an array consists of a *spine*, which is an array of pointers to smaller array parts called *arraylets*. This research has continued by two different lines: *(i)* the *Schism* solution [77] focus on improving the GC algorithm where arraylets are used, in order to make it tolerant with fragmentation, and *(ii)* the work presented in [80] which discusses a set of enhancements to the basic arraylet concept (e.g., in-lining the first n array bytes into the spine, lazy allocation and zero compression, fast array copy, and arraylet copy-on-write).

A real-time GC must be triggered like a task, and when another task with higher priority arrives, the GC is stopped. This issue has been investigated in the context of real-time Java for many years (e.g., [9], [57], and [79]). The *Metronome* GC [8] uses two different scheduling policies: one based on time, the other based on work. Both schedulers are good for high priority tasks, but for critical tasks they present latency problems.

Although during last years the real-time garbage collection for uniprocessor systems has become feasible, there is not so far a solution for a multiprocessor GC which meet the requirements of hard real-time systems. In [86] the Jamaica GC has been extended to deal with multiprocessor systems, while in [78], authors present a hardware-based real-time GC for a Java multi-processor providing non-disruptive and analyzable behavior.

Another open question lies in the definition of multiple heaps or a single one. In the case of multiple heaps (e.g., *Reflexes* [87]), it is interesting to consider whether a heap may be shared by a set of real-time threads.

In addition, a distributed GC is needed to collect data structures that are shared across heaps. The solution presented in [12] provides a distributed time-predictable GC. Additionally, to avoid garbage collection delays, a new construct for remote objects is introduced which guarantees that the memory required to perform a remote invocation (i.e., at the server side) does not use heap-memory.

4.2.3 Read and Write Barriers

The execution of the write barriers for both scoped memory regions and the GC may lead to a quite substantial time overhead that must be characterized and bounded [35]. The most common approach to implement read/write barriers is by adding in-line code as in [13]. This solution requires compiler cooperation (e.g., JIT) and presents a serious drawback because it could double the application's size.

Alternatively, the solution proposed in [37] instruments the bytecode interpreter, avoiding space problems, but still requires a complementary solution to handle native code. The use of specialized hardware support in embedded systems allows performing write barrier checks in parallel with the store operation [38]. In such case, the detection of whether an action is required or didn't take no added time.

4.3 Synchronization and Resource Sharing

Java monitors present the priority inversion problem. Protocols, such as *priority inheritance* and *priority ceiling*, provide a solution to this synchronization problem. The semantic of the `synchronized` keyword has been extended in RTSJ to avoid priority inversion, and waiting queues are priority ordered.

The work presented in [28] proposes mechanisms for measuring and policing the potential blocking time of threads. This mechanism is studied in the context of both basic priority inheritance and priority ceiling protocols.

The *Reflexes* solution [87] prevents synchronous operations by replacing lock-based synchronization with a communication scheme based on the *Java Memory Model* (JMM) for a multithreaded program, whih is obstruction-free.

Multiprocessor resource allocation policies and protocols are rather simplistic, and nested resource accesses are sometimes prohibited. An alternative solution based on deadlock avoidance rather than deadlock prevention has been presented in [96] within the RTSJ environment.

As a new microprocessor generation, multicore systems provide an alternative to traditional scalability architectures by allowing deploy multiple application instances on many smaller systems units. The *fork/join* paradigm [46], now included in JDK 7, is designed to implement recursive algorithms where the control path branches out over a few paths (i.e., divide-and-conquer). The adaptation of this parallel synchronization mechanism to real-time and their introduction into JSR-282 is a field requiring research. We can find an example of this adaptation in [49].

In order to take performance advantage of the multiple cores, the application needs to be more concurrent, and programmers need to find more parallelism while taking into account time requirements within the application domain. An interesting parallel programming model is *Actors* [2], which is a potential alternative mechanism of communication and synchronization to the general model of communication in Java (i.e., *shared memory*).

A more fundamental change in synchronization and programming models, however, is required to fully exploit the power of multicore systems. The *Transactional Memory* (TM) paradigm [56], which provides an alternative concurrency control mechanism to lock-based synchronization.

TM is an attractive concept expressing parallelism for multi-core architectures, avoiding problems of lock-based methods and making easy programming. Currently this is one of the hottest topics research in several areas (e.g., programming languages, computer architecture, and parallel programming). This paradigm has been applied within the RTSJ context in [53] to realize atomic operations supported by the hardware, confirming that TM is an interesting alternative to traditional concurrency control mechanisms.

Nowadays, the PERC `atomic` [60] statement can be supported by the TM paradigm, which can be implemented by a specific hardware support. A TM implementation is available within the `java.util.concurrent.atomic` package (i.e., JSR-166) [65], which was included within JDK 5.

4.4 Asynchronous Event Handling

Since in a real-time context, not all events are predictable in time and frequency, the ability of real-time systems to react to sudden events correctly requires reliable asynchrony techniques. Interrupts and asynchronous exceptions are the mechanisms enabling a thread to be asynchronously interrupted by another.

Based on a comparison and analysis of different run-time implementations of *Asynchronous Event Handlers* (AEHs) in an RTSJ compliant system, the approach presented in [44] presents a new method to implement AEHs more efficiently by separating the service from the handler.

In order to have a lightweight concurrency mechanism, a scheduling scheme for AEH that minimize the number of server threads has been presented in [95]. This scheme defines the worst-case scenario that demands the least upper bound of servers for both blocking and non-blocking handlers.

Another experience on the implementation of AE in RTSJ is then presented in [50], which adapts existing algorithms (e.g., *Polling Server*, *Deferrable Server*, *Priority Exchange*, *Sporadic Server*, or *Slack-based*). Since RTSJ proposes a many-to-many relation (i.e., $n - n$) between the events and handlers, the problem here is how to map handlers to servers, since to find an efficient map is generally difficult to determine.

4.5 Asynchronous Transfer of Control

When some change in the system environment needs immediate attention, the real-time application must abandon its current execution or take appropriate action (e.g., switching to an emergency state). RTSJ provides a safe mechanism for abnormal stopping threads and transferring control based on exception handling and a generalization of the `interrupt()` method of the `Thread` class, which has been overridden to allow us to deal with timeouts and thread termination without the latency of polling. The asynchronous transfer of control is a crucial mechanism for real-time applications, however it is difficult to find a specific work about it in the context of the RTSJ.

The RTSJ extension presented in [10] generalizes the concurrency model of RTSJ proposing a unique type of thread that may be used by all applications, making it more flexible. This solution has been presented in a distributed real-time Java environment, in order to provide an efficient solution to a time predictable version of the Java-RMI. This solution introduces a change of mode in the RTSJ thread model, allowing a real-time thread to switch between the heap and non-heap mode[5].

4.6 Physical Memory Access

Java was designed for embedded systems, as a platform independent language and runtime system. But this does not support direct access to low-level I/O devices. Since real-time systems often run in an embedded environment, RTSJ provides classes to access physical memory.

4.7 Multiprocessor/Multicore Support

Java processors are faster than an interpreting JVM, and also use less resources than a JIT compiler. An example of implementation of the JVM in hardware is JOP [81], which main focus is on time-predictable bytecode execution, being carefully designed to avoid time dependencies between bytecodes. Furthermore, eight JOP cores together with an arbitration control device and a shared memory has been interconnected via a system on chip bus in JopCMP [74]; a symmetric shared-memory multiprocessor [83].

Another example is *Komodo*[6] [20]; a real-time Java multithreaded micro-controller. Its real-time scheduling is priority-based and can select a new thread after each bytecode instruction, which results in a very pessimistic WCET. As a commercial example, the *aJile* Java microprocessor is based on the 32-bit JEM2 Java chip [33] developed by *Rockwell-Collins*, which is an enhanced version of JEM1 [31], created in 1997.

By integrating in hardware, the kernel and JVM components, the flexibility and scalability are highly increased. The most representative example is the Azul JVM [88], which allows an individual application instance to scale to 670 GBytes of memory and hundreds of processor cores. This commercial multicore JVM supports a hardware-assisted incremental GC.

[5]The RTSJ supports two kinds of real-time threads (i.e., Operating on the heap and avoiding the influence of the GC).

[6]The commercial version of *Komodo* is *jamuth*.

5. DISTRIBUTED REAL-TIME JAVA

The distributed profile of RTSJ (i.e., DRTSJ under the JSR-50) allows us to enforce both real-time constraints and conventional functional requirements in a distributed Java environment. This specification defines three integration levels, each one requiring some support from the underlying system and offering some benefit to the programmer:

- *L0* is the minimal integration level, and considers the use of RMI without changes,
- *L1* is the first real-time level, and considers real-time properties in the RMI, and
- *L2* offers a transactional model for RMI, and introduces the concept of distributable real-time threads, which transfer information across the nodes.

The status of DRTSJ was given as inactive in 2006 (see `http://drtsj.org`). However there is some work related with DRTSJ that must be taken into account (e.g., [26] treats the problem of recovering from failures of distributable threads). In the following we sketch the most representative studies related to this specification profile.

The design of the DREQUIEMI platform [11] allows to support the corresponding level L1 of DRTSJ, and also it incorporates some elements taken from L2. This distributed Java architecture is based on RTSJ and RMI, having influences from RT-CORBA, and some time-triggered principles. It is structured in five layers and offers four services:

(i) a stub/skeleton service, allowing a remote invocation with a certain degree of predictability;

(ii) a *Distributed Garbage Collection* (DGC) service, eliminating unreferenced remote objects in a predictable way;

(iii) a naming service, offering a local white page service to the programmers;

(iv) and a synch/event service for data-flow communications, which is not included currently in RMI.

Another work in profile definition for time-predictable RMI is RT-RMI [89], which addresses three different environments (i.e., safety critical, business-critical, and flexible business-critical systems) considering a different profile for each environment (i.e., RMI-HRT, RMI-QoS, and RT-Components):

(i) The RMI-HRT profile [89] defines the requirements of safety critical systems (see Section 6). This profile presents a correct and highly deterministic behavior; it must support systems where deadline misses can cost human lives or cause fatal errors (i.e., hard real-time systems), and requires *Worst-Case Response Time* (WCRT) analysis.

(ii) The RMI-QoS profile [21] defines the requirements of business-critical systems (i.e., for soft real-time systems). The corresponding DRTSJ level is L2.

(iii) The RT-Components profile [92] defines the requirements for flexible business-critical systems (e.g., for multimedia, ambient intelligence, and mission critical systems with survivability). The corresponding profile considers a support based on OSGi. The corresponding DRTSJ level is L1.

Other relevant work on how to develop a framework for DRTSJ L1 is presented in [18]. In this solution, the client-side remains almost unchanged while the server suffers several modifications, such as the addition of extra supporting scheduling parameters and a thread-pool attached to each remote object. This framework defines three different policies to generate client-side parameters, and supports their propagation to the server-side and new classes.

5.1 Components-based Solutions

Compositional techniques are the result of years of research in software engineering. Structuring systems as interacting components facilitates their design, analysis and construction process, and deals with scalability, evolution and complexity. Regarding component-based real-time systems, the most important principle to be considered is that non-functional properties such as timeliness and testability must not be affected by the system integration.

The Compadres [41] solution presents a component framework for distributed real-time and embedded systems, offering several advantages, among them:

(i) Abstraction of the complexity of memory management since the compiler automatically generates the scoped memory architecture for components.

(ii) Communication among components through strongly-typed objects.

(iii) Communication between the components assembled by connecting ports.

An event manager component design has been presented in [51]. This model uses the configurable CORBA-idl interfaces; it has a global view of the system from the event happening with the release on a processor of its handlers within a server. This component treats the event handlers as servers and the processors as resource allocations.

6. HIGH-INTEGRITY SYSTEMS

The *Safety Critical Java Specification* (SCJS) profile reduces, restricts and simplifies the RTSJ programming model to be suitable of verification and validation of safety-critical codes (e.g., for aerospace vehicles or air-traffic control systems). Safety-critical systems are distinguished by stringent requirements (e.g., space, time predictability, dependability, and safety) at both process and product levels. Such systems must pass a verification and validation process. This profile presents the following characteristics: *i)* no garbage collection, *ii)* safe stack allocation, *iii)* very small memory footprint *iv)* simple run-time environment, and *v)* efficient throughput (i.e., nearest to optimized C).

Additionally, SCJS will identify all classes and methods not used by the application at runtime, because a safety critical must run without classes and methods not used. So that the DO-178B dead code elimination requirements must be supported.

In order to provide static correctness guarantees to the RTSJ memory model, a number of research works restrict real-time *profiles* (e.g., the version *Raven* of PERC, *Ravenscar Java* [45] and *Predictable Java* [16]) in which the complexity of scoped regions is highly reduced, and their flexibility limited. Considering the cost of failure in safety-critical systems, the effort of adopting static memory management in such systems can be justified.

The restricted memory region model that SCJ presents introduces a significant simplification of the RTSJ assignment rules (e.g., scoped memories are thread private). In SCJ L0 and L1, where there are not nested mission, the hierarchy of nested scoped can grow only in a linear shape. Moreover, the shared memory regions (i.e., immortal and mission) are at a fixed nested level within the memory region stack, making it impossible that two schedulable objects use the same scope at different nested levels. A study about the implications of this new memory model is given in [82]. The author proposes to unify the three memory types (i.e., immortal, mission and scoped) providing a single class to represent all three SCJ memory regions.

An important part of critical system development is scheduling analysis (e.g., [17]) which allows to ensure that all tasks in the system meet their deadlines and can run without overruns on a given hardware platform. The use of CSP and others constructs based on *Circus* to the development of SCJ programs has been study in [23]. Scheduling analysis and WCET analysis have been developed in recent years as a viable approach to automate the application development process. Some of these approaches are *Model Checking*-based (e.g., [27]). In [32] we find how to specify timing constraints for subtypes allowing the use of polymorphism and object-oriented design patterns.

Within the context of the DRTSJ, the *Profiles for RT-RMI* solution [89] presents a profile for safety critical systems (see Section 5), which is based on preemptive priority-driven scheduling and requires the use of priority ceiling protocols. It also considers two execution phases: *initialization* and *mission*.

7. CONCLUSIONS

RTSJ is the standard Java extension adding real-time programming constructs and constraints to the Java environment. This specification introduces the scoped memory concept, which is an original solution that combines pre-allocated space with a real-time GC. However, scoped memory presents some difficulties regarding their use, which are not yet totally resolved, although RTSJ is becoming mature. Moreover, regarding its use in multicore systems, some strengths and weaknesses are emerging, among them dynamic memory management and synchronization. Since in the past there were several and diverse works addressing issues to make Java real-time, some ideas can be revisited and picked up again (e.g., the algebraic model or the `atomic` keyword of PERC)

The DRTJ profile supports the development of distributed Java programs with real-time restrictions. The original JSP group has been inactive for several years. However important research work about the support of distributed real-time Java applications has been produced during these years. Recently has been created a new group, and new distributed Java paradigms can be taken into account in order to extend their use in real-time domains (e.g., object space, mobile agents, network services, or collaborative applications). It is also mandatory to consider adaptive component paradigms (e.g., OSGi), model driven architectures and existing standards for distributed control and automation (e.g., IEC 61499).

The SCJS profile supports the development of programs that must be certified. This specification includes a number of annotations used to constrain the behavior of programs, and the rules used to check statically the semantics programs. Programming with RTSJ scoped memory is difficult and results error-prone. However, the alternative of a real-time GC is not suited for all real-time applications (e.g., safety-critical). Because of this, SCJS particularly considers a simplification of both memory model and thread model.

8. ACKNOWLEDGMENTS

This research was supported by Consejería de Educación de la Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through Research Program S2009/TIC-1468, and by Ministerio de Educación y Ciencia, through the research grant TIN2009-.07146.

9. REFERENCES

[1] A.Corsaro. jrate documentation at sourceforge.

[2] G. Agha. Semantic considerations in the actor paradigm of concurrent computation. In *Seminar on Concurrency*, pages 151–179, 1984.

[3] aJile. The real-time multimedia microprocessor for java platforms aj200.

[4] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time java memory management. *Real-Time Systems*, 37(1):1–44, 2007.

[5] Apogee. Aphelion, 2004.

[6] Atego. Aonix perc.

[7] R. Atherton. Already being adopted for traditional enterprise computing tasks, java is also making its way into manufacturing operations. *IEEE Spectrum*, December 1998.

[8] D. F. Bacon, P. Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *OTM Workshops*, pages 466–478, 2003.

[9] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, 1992.

[10] P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres. Simplifying the dualized threading model of rtsj. In *ISORC*, pages 265–272, 2008.

[11] P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres. Simple asynchronous remote invocations for distributed real-time java. *IEEE Trans. Industrial Informatics*, 5(3):289–298, 2009.

[12] P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres. No-heap remote objects for distributed real-time java. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[13] W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT*, pages 289–305, 2001.

[14] E. Benowitz and A. Niessner. A patterns catalog for RTSJ software designs. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), Lecture Notes in Computer Science*, volume 2889, pages 497–507, 2003.

[15] E. G. Benowitz and A. F. Niessner. A patterns catalog for rtsj software designs. In *OTM Workshops*, pages 497–507, 2003.

[16] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A predictable java profile: rationale and implementations. In *JTRES*, pages 150–159, 2009.

[17] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES*, pages 106–114, 2008.

[18] A. Borg and A. J. Wellings. A real-time rmi framework for the rtsj. In *ECRTS*, pages 238–, 2003.

[19] A. Borg and A. J. Wellings. Reference objects for rtsj memory areas. In *OTM Workshops*, pages 397–410, 2003.

[20] U. Brinkschulte, S. Uhrig, and T. Ungerer. Der mehrfädige komodo-mikrocontroller (the multithreaded komodo microcontroller). *it - Information Technology*, 47(3):117–122, 2005.

[21] J. F. Briones, M. A. de Miguel, A. Alonso, and J. P. Silva. Quality of service composition and adaptability of software architectures. In *ISORC*, pages 169–173, 2009.

[22] L. Carnahan and M. Ruark. Requirements for real-time extensions for the Java platform. Technical report, RTJWG, September 1999. http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf.

[23] A. Cavalcanti, A. Wellings, J. Woodcock, K. Wei, and F. Zeyda. Safety-critical java in circus. In *JTRES*, JTRES '11, pages 20–29, New York, NY, USA, 2011. ACM.

[24] J. Consortium. Draf international j-consortium specification. Technical report, J Consortium, September 1999. http://www.j-consortium.org.

[25] A. Corsaro and C. Santoro. Design patterns for rtsj application development. In *OTM Workshops*, pages 394–405, 2004.

[26] E. Curley, B. Ravindran, J. S. Anderson, and E. D. Jensen. Recovering from distributable thread failures in distributed real-time java. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[27] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. Metamoc: Modular execution time analysis using model checking. In *WCET*, pages 113–123, 2010.

[28] O. M. dos Santos and A. J. Wellings. Measuring and policing blocking times in real-time systems. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[29] Eric J, Bruno, Greg Bollella. *Real-Time Java Programming with Java RTS*. PRENTICE HALL, 2009.

[30] J. V. et al. The ovm project, 2004.

[31] D. A. Greve. Symbolic simulation of the jem1 microprocessor. In *FMCAD*, pages 321–333, 1998.

[32] G. Haddad and G. T. Leavens. Specifying subtypes in scj programs. In *JTRES*, JTRES '11, pages 40–46,

New York, NY, USA, 2011. ACM.

[33] D. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the javatm virtual machine. In *ISORC*, pages 53–59, 2001.

[34] M. T. Higuera-Toledano and V. Issarny. Java embedded real-time systems: An overview of existing solutions. In *ISORC*, pages 392–391, 2000.

[35] M. T. Higuera-Toledano and V. Issarny. Analyzing the performance of memory management in rtsj. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 26–33, 2002.

[36] M. T. Higuera-Toledano and V. Issarny. Improving the memory management performance of rtsj. *Concurrency and Computation: Practice and Experience*, 17(5-6):715–737, 2005.

[37] M. T. Higuera-Toledano, V. Issarny, M. Banâtre, G. Cabillic, J.-P. Lesot, and F. Parain. Region-based memory management for real-time java. In *ISORC*, pages 387–394, 2001.

[38] M. T. Higuera-Toledano, V. Issarny, M. Banâtre, and F. Parain. Memory management for real-time java: An efficient solution using hardware support. *Real-Time Systems*, 26(1):63–87, 2004.

[39] M. T. Higuera-Toledano and A. J. Wellings. *Distributed, Embedded and Real-Time Java Systems.* Springer, 2011.

[40] G. Hilderink. A new java thread model for concurrent programming of real-time systems. *Real-Time Java*, January 1998.

[41] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad. Compadres: A lightweight component middleware framework for composing distributed real-time embedded systems with real-time java. In *Middleware*, pages 41–59, 2007.

[42] IBM. Ibm websphere real time 1.0.

[43] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst.*, 29(3):8, 2011.

[44] M. Kim and A. J. Wellings. Efficient asynchronous event handling in the real-time specification for java. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[45] J. Kwon, A. J. Wellings, and S. King. Ravenscar-java: a high integrity profile for real-time java. In *Java Grande*, pages 131–140, 2002.

[46] D. Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[47] C. Lizzi. Java real-time distributed processing over atm networks with chorus/os. In *ETFA '99*. IEEE Computer Society, 1999.

[48] P. Madany, S. Keohan, D. Kramer, and T. Saulpaugh. Javaos: a standalone java environment. Technical report, JavaSoft, http://java.sun.com/products/javaos/, 1998.

[49] C. Maia, L. Nogueira, and L. M. Pinho. Combining rtsj with fork/join: a priority-based model. In *JTRES*, JTRES '11, pages 82–86, New York, NY, USA, 2011. ACM.

[50] D. Masson and S. Midonnet. Rtsj extensions: event manager and feasibility analyzer. In *JTRES*, pages 10–18, 2008.

[51] D. Masson and S. Midonnet. The design of a real-time event manager component. In *NOTERE*, pages 291–296, 2010.

[52] H. McGhan and M. O"Connor. picojava: a direct execution engine for java bytecode. *IEEE Computer*, 31(10):22–30, 1998.

[53] F. Meawad, M. Schoeberl, K. Iyer, and J. Vitek. Real-time wait-free queues using micro-transactions. In *JTRES*, JTRES '11, pages 1–10, New York, NY, USA, 2011. ACM.

[54] S. Microsystems. Mackinac white paper, 2005.

[55] A. Miyoshi, H. Tokuda, and T. Kitayama. Implementation and evaluation of real-time java threads. In *Real-Time Systems Symposium.* IEEE Computer Society, Decembre 1997. pag. 166-174.

[56] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[57] K. D. Nilsen. High-level dynamic memory management for object-oriented real-time systems. *OOPS Messenger*, 7(1):86–93, 1996.

[58] K. D. Nilsen. Invited note: Java for real-time. *Real-Time Systems*, 11(2):197–205, 1996.

[59] K. D. Nilsen. Adding real-time capabilities to java. *Communications of the ACM*, 41(6):49–56, June 1998.

[60] K. D. Nilsen. Adding real-time capabilities to java. *Commun. ACM*, 41(6):49–56, 1998.

[61] NSIcom. Encapsulating java on embedded systems. Technical report, white-paper, http://www.nsicpm.com, 1998.

[62] Oracle. Java card technology.

[63] Oracle. The java community process program.

[64] Oracle. Jsr 1, real-time specification for java, http://www.jcp.org/en/jsr/detail?id=1.

[65] Oracle. Jsr 166, concurrency utilities, http://www.jcp.org/en/jsr/detail?id=166.

[66] Oracle. Jsr 281, cdc 1.0.2, http://www.jcp.org/en/jsr/detail?id=281.

[67] Oracle. Jsr 282, rtsj version 1.1 http://www.jcp.org/en/jsr/detail?id=282.

[68] Oracle. Jsr 30, cldc 1.0, http://www.jcp.org/en/jsr/detail?id=30.

[69] Oracle. Jsr 302, safety critical java technology, http://www.jcp.org/en/jsr/detail?id=302.

[70] Oracle. Jsr 36, cdc 1.1.2, http://www.jcp.org/en/jsr/detail?id=36.

[71] Oracle. Jsr 39, cldc 1.1, http://www.jcp.org/en/jsr/detail?id=39.

[72] Oracle. Jsr 50, distributed real-time specification, http://www.jcp.org/en/jsr/detail?id=50.

[73] Oracle. Oracle jrockit jvm.

[74] C. Pitter and M. Schoeberl. A real-time java chip-multiprocessor. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[75] F. Pizlo. Fiji vm.

[76] F. Pizlo and J. Vitek. An emprical evaluation of memory management alternatives for real-time java. In *RTSS*, pages 35–46, 2006.

[77] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *PLDI*, pages 146–159, 2010.

[78] W. Puffitsch. Hard real-time garbage collection for a java chip multi-processor. In *JTRES*, JTRES '11, pages 64–73, New York, NY, USA, 2011. ACM.

[79] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *LCTES*, pages 93–102, 2003.

[80] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: divide arrays and conquer speed and flexibility. In *PLDI*, pages 471–482, 2010.

[81] M. Schoeberl. A java processor architecture for embedded real-time systems. *Journal of Systems Architecture - Embedded Systems Design*, 54(1-2):265–286, 2008.

[82] M. Schoeberl. Memory management for safety-critical java. In *JTRES*, JTRES '11, pages 47–53, New York, NY, USA, 2011. ACM.

[83] M. Schoeberl, F. Brandner, and J. Vitek. Rttm: real-time transactional memory. In *SAC*, pages 326–333, 2010.

[84] F. Siebert. The jamaica vm.

[85] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *JTRES*, pages 94–103, 2007.

[86] F. Siebert. Concurrent, parallel, real-time garbage-collection. In *ISMM*, pages 11–20, 2010.

[87] J. H. Spring, F. Pizlo, J. Privat, R. Guerraoui, and J. Vitek. Reflexes: Abstractions for integrating highly responsive tasks into java applications. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.

[88] A. Systems. Vega 3 compute appliances, http://www.azulsystems.com/products.

[89] D. Tejera, A. Alonso, and M. A. de Miguel. Rmi-hrt: remote method invocation - hard real time. In *JTRES*, pages 113–120, 2007.

[90] The Real-Time for Java Expert Group. *Real-Time Specification for Java*. ADDISON-WESLEY, 2000. http://www.rtj.org.

[91] TimeSys. Jtime, http://www.timesys.com/java.

[92] R. Tolosa, J. P. Mayo, M. A. de Miguel, M. T. Higuera-Toledano, and A. Alonso. Container model based on rtsj services. In *OTM Workshops*, pages 385–396, 2003.

[93] P. Tullmann and J. Lepreau. Nested java processes: Os structure for mobile code. In *Proc. of the Eighth ACM SIGOPS European Workshop*. http://www.cs.utah.edu/projects/flux, September 1998.

[94] L. University. Ljrt - lund java based real-time.

[95] A. J. Wellings and M. Kim. Asynchronous event handling and safety critical java. In *JTRES*, pages 53–62, 2010.

[96] A. J. Wellings, S. Lin, and A. Burns. Resource sharing in rtsj and scj systems. In *JTRES*, JTRES '11, pages 11–19, New York, NY, USA, 2011. ACM.

[97] A. Zerzelidis and A. J. Wellings. A framework for flexible scheduling in the rtsj. *ACM Trans. Embedded Comput. Syst.*, 10(1), 2010.