This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

Java Technologies for Cyber-Physical Systems

M. Teresa Higuera-Toledano , *Fellow, IEEE*

***Abstract*—Cyber-Physical Systems (CPS) provide a strong integration and coordination between computing science, network communications and the physical world. CPS involves distributed processing in which communication and synchronization integration present additional challenges. These systems are usually real-time and embedded applications with stringent requirements, including highly precise timing characteristics, small memory footprints, flexible sensor and actuator interfaces, and robust safety characteristics. This paper outlines opportunities and challenges in the development of CPS by using distributed real-time and embedded Java technologies.**

***Index Terms*—Process control systems, Real-time systems, Embedded systems, Concurrency, Distributed systems, Java language extensions, Validation and Verification.**

I. Introduction

The integration of physical processes with computation and communication has been called *Cyber-Physical Systems* (CPS) [1]. These systems provide a strong integration and coordination between computing science, network communications and the physical world. CPS present new challenges and opportunities in several industrial areas (e.g., electronics, energy, telecommunications and instrumentation), which require the recent development in the designing of embedded systems. Note that these systems are typically distributed, embedded, and need to be low power, safe, reliable, efficient and secure.

Note that a major part of the cost of creating complex systems is spent in the integration and the *Verification and Validation* (V&V) process of the resulting systems. Therefore it is essential for the production of CPS to take into account modern languages, tools and methods allowing the exploitation of beneficial effects such as good resource management and highly-robust structures, which simultaneously provide high productivity and an easier maintenance.

Many CPS can be categorized as business-critical, but also as mission or safety-critical (e.g., automotive, aerospace and industrial automation) where even human life is sometimes at stake [1]. Here, as typical critical systems, CPS require the recent advances in real-time technology [2]. Because of that high-level programming languages and middleware are needed to timely and safely develop software for these applications. Java presents features allowing code reuse, readability, reliability, platform independence, and dynamic loading. Because of this, the use of Java is extending and it is beginning to be used in many new environments. However, Java presents some problems regarding its use in distributed, real-time and embedded systems.

Since 1997 several research works have been focused on the limits of the Java language and its execution environment to seek the possibility for real-time technology using Java [3]. The *Real-Time Java Working Group* (RTJWG), created at the end of 1999 and working under *The National Institute of Standards and Technology* (NIST), produced a basic requirements document[1], which resulted in the *Real-Time Specification for Java* (RTSJ) which is confined into the `javax.realtime` package.

The *Java Community Process* (JCP) extends the Java platform adding to it new functionalities. In order to extend the Java specification to be used in real-time systems, in November 2001, the JCP created the first *Java Specification Request* (JSR), which is called JSR-001[2] and supports the RTSJ. As result of this, the RTSJ is the first Java extension within the JCP, which was approved and closed in 2002.

Several commercial and research implementations of RTSJ are available today. However, some features are still an open research subject, among them asynchronous transfer of control and memory allocation. For this reason, a new JSR was created in 2005 (i.e., the JSR-282[3], which includes enhancements that were requested in the first RTSJ release [4] and new classes dealing with multiprocessor issues (e.g., new dispatching and allocation models, cost enforcement, interrupts affinity and processor failure). A RTSJ 2.0 draft[4], still under JSR-282, has been recently submitted to the JCP. This new version is divided into three modules: *Base Module*, *Device Module* and *Alternative Memory Areas Module* (see Table I).

The RTJWG NIST document also gives some examples of possible profiles to augment or restrict the core of real-time functionality; among them the distributed real-time profile. Because of this, in order to add distributed real-time capabilities to Java, the JCP defined in 2002 the *Distributed Real-Time Specification for Java* (DRSTJ) as the JSR-050[5]. Finally, the JCP defined in 2006 the *Java Safety Critical* (JSC) specification as the JSR-302[6]). This JSR simplifies and extends RTSJ to be used in critical and high integrity real-time systems.

In June 2014, has been created the *Cyber-Physical Systems Public Working Group* (CPS PWG), which works under the supervision of NIST, and is examining the requirements and the key aspects of CPS. In order to facilitate interoperability between elements and systems, and pro-

Dept. of Computer Architecture and Automation
Complutense University of Madrid, 28040, Spain
mthiguer@dacya.ucm.es

[1] http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf
[2] http://www.jcp.org/en/jsr/detail?id=1
[3] http://www.jcp.org/en/jsr/detail?id=282
[4] https://www.aicas.com/cms/sites/default/files/rtsj_17.pdf
[5] http://www.jcp.org/en/jsr/detail?id=50
[6] http://www.jcp.org/en/jsr/detail?id=302

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

Module	Class or Interface	Feature or issue addressed
Basic	`AbstractAsyncEvent`	represents a signal, an interrupt, or an exception
	`AbstractAsyncEventHandler`	represents the handler to one or more events.
	`POSIXSignalHandler`	allows us to handle POSIX signals
	`Timer`	supports events whose occurrences are driven by time
	`Clock`	supports clocks (e.g., areal-time clock)
	`RealtimeThread`	represents real-time tasks using heap or scoped memory regions
	`NoHeapRealtimeThread`	represents real-time tasks which do not access nor reference the heap
	`PriorityScheduler`	represents the RTSJ required priority-based scheduler
	`PriorityParameters`	gives to schedulable objects a single integer to determine execution order
	`PriorityInheritance`	is a singleton class specifying use of the priority inheritance protocol
	`PriorityyCeilingEmulation`	allows the use of the priority ceiling emulation protocol also known as *highest lockers*
	`Affinity`	supports the processor-affinity concept
	`GarbageCollector`	supports the garbage collection within the Java heap
Device	`RawMemoryAccess`	models a range of physical memory as a fixed sequence of bytes
	`RawMemoryType`	interface that allows the implementation device drivers and deal with low-level hardware
	`PhysicalMemoryManager`	makes the memory manager aware of the memory types on their platform
Alternative Memory Areas	`ImmortalMemory`	is a memory resource available to all schedulable objects and Java threads for allocation
	`HeapMemory`	is a singleton object allowing non-heap real-time tasks to allocate objects in the heap
	`ScopedMemory`	represents memory spaces which have a limited lifetime
	`LTMemory`	represents a scoped memory area guaranteed by the system to have linear time allocation
	`LTPhysicalMemory`	allows objects to be allocated from a range of physical memory with particular attributes

TABLE I
JSR-282 MODULES AND CLASSES (HTTP://WWW.RTSJ.ORG/SPECJAVADOC/JAVAX/REALTIME/PACKAGE-SUMMARY.HTML).

mote their communication, this group has identified the need to develop a consensus definition, a reference architecture, and a common lexicon and taxonomy. The group recognizes three critical and first order design principles: timing, dependability, and security. To address these challenges, the CPS PWG has been divided into four initial sub-working groups[7]: *(i)* definitions, taxonomy and reference architecture, *(ii)* cases of use, *(iii)* timing and synchronization, *(iv)* cyber-security and privacy, and *(iv)* data interoperability.

Java is much more than a programming language; it is also a platform that includes a set of open source and commercial tools. These advantages make Java a good candidate for CPS. Because of this, a new JSR addressing CPS requirements is an interesting proposal. The CPS PWG efforts can be a good point to establish the basic guidelines for this new JSR proposal. Also, again as occurs with the NIST RTJWG, the resulting document can support the guidelines of some examples of possible profiles to augment or restrict the core of the Java CPS basic functionality. As example some profiles can be related with: *(i)* personalized health care, *(ii)* system of systems, *(iii)* internet of things, system of systems, or *(iv)* self-organized and autonomous.

The CPS technology is the heart of a critical infrastructure, and forms the basis of future smart services. In this paper, we will discuss how challenges in developing CPS can be addressed by using Java technologies. In order to do so, we analyze several open research and development issues by presenting some existing works. The works presented here have been selected because they have already emerged as current topics in others areas such as software engineering [17] or microprocessor architecture (e.g., components [24], multiprocessors and multi-cores[16]) or because they could serve as research topics in the next few years (e.g., safety critical and distributed Java systems).

[7] http://www.cpspwg.org/Portals/3/docs/Resources

A. Paper Organization

The rest of the paper is set out as follows: Section 2 provides the background information about the standards of Java regarding their use for real-time CPS (i.e., asynchronous programming, scheduling, synchronization, device accessing). Section 3 presents research works in order to have a reference-architecture for distributed real-time Java systems. Section 4 includes system safety and reliability engineering. Section 5 describes a practical case of use. Section 6 analyzes data interoperability. Finally, Section 7 concludes this paper.

II. Asynchronous Programming

Since a CPS must be able to react to the outside world, it needs to be able to change its execution flow in an asynchronous way for the ongoing computation. Asynchronous programming is particularly interesting in CPS because of the need to make the best use of the hardware. Asynchronous programming refers to processes that do not depend on each other's outcome, and can therefore occur on different threads simultaneously. The opposite is synchronous programming, where processes wait for one to complete before the next begins. In general, asynchronous programming is faster, while synchronous programming is safer and more predictable.

Asynchronous systems do not depend on strict arrival times of signals or messages for reliable operation. Coordination is achieved via events. Asynchronous events allow to change the program execution flow based on external stimuli or signals (e.g., interrupts, messages, or timed events) that are asynchronous with respect to the current program execution.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

Since efficiently dealing with both asynchronous events and asynchronous thread termination is essential to real-time systems, the JSR-282 addresses these features and supports it within the Basic Module. While the Basic Module introduces several mechanisms allowing asynchronous programming, timers and clocks, the Device Module provides a low level interface to deal with embedded systems. These features allow the development of asynchronous systems interacting with the real world as CPS requires.

A. Asynchronous Transfer of Control

The most common event handling architecture is single threaded (i.e., it only has one thread of execution). When an event occurs the event handler (i.e., the associated code with the event) is run. Event driven systems are cooperative in the sense that an event handler, once started runs to completion. The state of the event handler is unchanged at it continues on as if nothing had happened. However, an event handler can be interrupted at some point in its code and other event handlers could change data that it is using. Worse, it could even be called again. Most event handlers are not-reenterable and can leave shared data in an inconsistent state (e.g., causing data corruption). Because of that, asynchronous exceptions was deprecated in regular Java. Similarly, `stop()` and `destroy()` methods have been also deprecated in regular Java because their use could lead to a deadlock or leave shared objects in an inconsistent state.

In RTSJ, the asynchronous programming model is based on two classes: the `AbstractAsyncEvent` and the `AbstractAsyncEventHandler`, which represents a signal, an interrupt, or an exception. This model allows both to associate one or more handlers to a single event, and to associate a single handler to one or more events. An implementation of RTSJ asynchronous events [5] adapts existing algorithms (e.g., *Polling Server*, *Deferrable Server*, or *Slack-based*) to deal with soft real-time asynchronous events.

B. Asynchronous Thread Termination

Many CPS are event-driven computer systems tightly interacting with the physical world and requiring *mode changes* whether external changes occur or not (e.g., switching to an emergency state or to a save-energy state). Because of this, the asynchronous thread termination is a crucial mechanism for CPS because it can be used to adapt their behavior at run-time by changing their operating mode (e.g., adaptive control systems).

The computation in CPS requires also asynchronous thread termination to both start and stop tasks depending on external stimuli. As an example, when an external physical system change causes a computation task to be superfluous, the easiest solution is to terminate this task. Another case to stop a task is whether a potentially unbounded computation periodic task needs to be executed within a deadline[8].

[8] In some cases, the task is aborted and the system would still obtain usable results (i.e., intermediate results are valid).

In order to support asynchronous thread termination, RTSJ use exception handling and the `interrupt()` method of the `Thread` class that has been overridden to deal with timeouts and thread termination without the latency of polling.

C. Timers and Clocks

A precise timing in a real-time CPS enables better implementation of control systems and allows a more robust correlation of the acquired data for measurement applications. Time-awareness in real-time CPS is a fundamental challenge because it is still not well-understood. The important aspects of timing in real-time systems are timing predictability and temporal determinism, which are based on the concept of *time-interval* characterized by *(i)* a period that marks the origin, *(ii)* a rate at which time advances and *(iii)* the ability to correlate to an internationally defined time-scale (e.g., UTC).

The RTSJ Basic Module includes the `Timer` class, a specialized form of asynchronous events supporting events whose occurrences are driven by time. There are two forms of timers: `OneShotTimer` instances fire once, at the specified time, while `PeriodicTimer` instances fire periodically and according to a specified interval initialized at a specified time. Timers are driven by `Clock` instances. The `Clock.getRealtimeClock()` represents the real-time clock and we may represent other time clocks as well. JSC does not support the time-triggered programming model that is interesting for CPS. But this matter will be addressed in a new release.

As an alternative, we can use the *Globally Asynchronous Locally Synchronous* (GALS) SystemJ[9], a Java extension for reactive and concurrent systems. SystemJ includes both synchronous *Esterel*-like features and asynchronous *Communicating Sequential Processes* (CSP) constructs. A SystemJ program may consist of one or more asynchronous concurrent behaviors (i.e., processes) called clock-domains. Each clock-domain executes asynchronously and has its logical clock, and can have one or more synchronous concurrent sub-behaviors (i.e., reactions), which execute concurrently by advancing their states in lockstep. A reaction consists of the synchronous composition of a set of threads that communicate and synchronize via signals. The output signals must be resolved in each tick.

D. Device Access

Interconnectivity is only a portion of the CPS. The real protagonist is data used to make decisions about control and send commands back to the appropriate devices. The JVM executes on top of an OS implemented for a particular hardware environment, abstracting the details of both the OS and hardware from the developer. This is the foundation of the *Write Once, Run Anywhere* (WORA) portability, which greatly simplifies application development. The Java WORA principle helps distributed applications work

[9] http://systemjtechnology.com

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

together using a common language and toolset end to end, at different levels in the network hierarchy.

Real-time applications often run in an embedded environment, that is the case of CPS. Java Embedded solutions[11] target smaller embedded devices and runs on resource-constrained devices. Java SE Embedded can run on devices with as little as 16 MB of memory headless, while Java ME can run on micro-controllers with as little as 128 KB of memory.

Java was designed for embedded systems, but it does not support direct access to low-level I/O devices. The RTSJ Device Module provides classes to access physical memory and classes to create objects with specific physical memory characteristics. The `RawMemoryType` interface allows us to implement device drivers and deal with low-level hardware (e.g., I/O space, memory-mapped registers, flash memory, battery-backed RAM). This module also provides, as an option, the POSIX signal interface for interacting with the system.

III. Scheduling and Synchronization

The CPS configuration both from timing and latency requires the study and research of new methods in time-based scheduling, synchronization, and resource sharing. In addition to events, timers and clocks support, the Base Module includes the classical concepts to deal with real-time systems (i.e., real-time scheduling). In order to avoid the synchronization problems of critical tasks with the *Garbage Collector* (GC), the Alternative Memory Areas Module provides a stack-based memory management based on memory regions.

A. Real-time Threads and Scheduling

The main issue of real-time programming is the time-predictable execution of tasks. Here, RTSJ introduces the concept of *schedulable object* (i.e., the `RealtimeThread`[12] and its subclasses and `AbstractAsyncEventHandler` and its subclasses), which are the objects managed by the base scheduler. The general opinion of the NIST group is to support static priority based scheduling (e.g., RMS) first and later to improve it to support dynamic priorities (e.g., EDF). However, RTSJ allows us to install arbitrary scheduling algorithms (e.g., a platform-specific schedule). The approach presented in [6] is based on 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.

As many recent processor architectures provide the *Dynamic Voltage and Frequency Scaling* (DVFS) capability[13] , power-aware cluster systems have been built using such processors. In general, these techniques take advantage of the fact that application performance can be adjusted to utilize idle time on the processor for energy savings. The work presented in [7] is based on the observation that a significant percentage of time spent in idle mode is due to the accumulation of small gaps between tasks.

B. Multicore Systems

As the complexity of CPS rapidly increases, multi-core processors and parallel programming models become essential. Multicore systems provide an alternative to traditional scalability architectures by allowing to deploy multiple application instances on many smaller systems units. The *fork/join* paradigm, 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 `Affinity` class of the RTSJ Basic Module adds the processor-affinity concept as a set of processors that can be associated with certain types of tasks. The timing challenge can be addressed because most algorithms for CPS are parallelizable. In [8] the authors extend the fork/join paradigm to be scheduled in real-time, where the environment is the RTSJ. However, the adaptation of this parallel synchronization mechanism to real-time still requires research. Here, it is interesting that the number of parallel threads can vary depending on the physical attributes of the system, which makes it useful in CPS. Since *Scala*[14] executes on a Java virtual machine, some authors propose its use in real-time systems [9].

C. Synchronization and Resource Sharing

Java monitors present the typical priority inversion problem and do not guarantee a real-time policy in queuing. Because of that, the semantics of the `synchronized` keyword has been extended in RTSJ to avoid priority inversion by including *Priority Inheritance Protocol* (PIP) and *Priority Ceiling Protocol* (PCP), and waiting queues are ordered based on priority. Non-blocking synchronization is free of the priority inversion problem, making unnecessary PIP and PCP. Moreover, this type of synchronization improves time predictability, which is the best advantage in real-time CPS. A wait-free synchronization scheme has ben presented in [10] introducing the paradigm of *guarded section*. Only one process at a time is allowed to pass through a guarded section, which avoids processes blocked at its entrance, at difference than critical sections.

Another non-blocking alternative to lock-based synchronization is *Transactional Memory* (TM) paradigm[15].This is an attractive concept expressing parallelism for multicore architectures, which is currently one of the most interesting topics for research in several areas (e.g., programming languages, computer architecture, and parallel programming). This paradigm is available within the `java.util.concurrent.atomic` package (i.e., JSR-166[16]), which was included within JDK 5. The absence of multilateral blocking synchronization is particularly interesting for event-triggered systems that follows an asyn-

[11] Java SE Embedded, Java ME Embedded, and Java Card

[12] The `RealtimeThread` specializes the Java `Thread` class for real-time.

[13] http://www.ti.com/lit/an/slva646/slva646.pdf

[14] This is an object-oriented language with functional aspects allowing to easily execute parallel programs in multicore systems.

[15] http://cs.brown.edu/ mph/HerlihyM93/herlihy93transactional.pdf

[16] http://www.jcp.org/en/jsr/detail?id=166

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

chronous programming mode where competing processes can synchronize on concurrent state changes, which is the general case of CPS. Note that when using a real-time scheduling (e.g., RMS), the non-blocking synchronization avoids blocking high-priority task, while low-priority tasks can be subject to starvation.

D. Dynamic Memory Management

Memory management is one of the major issues still needing research in RTSJ, because of synchronization problems relative to the GC within the Java heap. RTSJ supports two kinds of real-time threads (i.e., operating on heap and avoiding the influence of the GC[17]). The Alternative Memory Areas Module includes the *Scoped Memory Regions* (SMR) classes, which support an alternative dynamic-memory management scheme to the garbage collected heap, with both time- and space-predictable behavior. But, this makes programming complex and error prone.

In addition to SMR, the Alternative Memory Areas Module includes facilities for providing what memory to use for Java objects, and introduces wait-free queues to allow non-blocking communication among critical tasks, which do not access the heap, and regular Java threads (i.e., non real-time) using the heap. Non-blocking synchronization is also an interesting model to synchronize the GC and the application that still requires research effort. Here the GC must have a priority lower than that of real-time threads and higher than that of normal Java threads.

IV. Reference Architecture

The main goal of the CPS *Reference Architecture* (RA) working group is to provide a guidance allowing smart devices work together to improve our *quality of life* (e.g., personalized health care, traffic flow management, or smart manufacturing). In CPS, timing behavior of communication is also important for correctness of results. We must consider not only the measurements of sensors and their data processing, but also the transmissions of these data have to be finished before a pre-defined deadline. The CPS configuration not only requires to meet timing and latency requirements in a local way, but also time properties must be propagated in a distributed way to enable large converged networks. We also need alternative sources to trace a standard reference time and create timing network topologies to support diverse and redundant paths. In other words, the CPS RA must be addressed in two different directions:

1. Application design, implementation and evaluation (e.g., middleware and multicore synchronization).
2. Models for networking and communication (e.g., sensor and actuator network protocols).

[17] In addition to `RealtimeThread` objects, RTSJ introduces the `NoHeapRealtimeThread` class supporting schedulable objects which do not access nor reference the heap.

A. Distributed Middleware

For a real-time extension of a distributed system, the most adequate abstraction can be categorized as: control-flow, data-flow, and networked (see Table II). The distributed profile of RTSJ (i.e., DRTSJ) allows 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 benefits to the programmer:

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

Today the status of this JSR is dormant. However, there is some work related with DRTSJ that must be taken into account (e.g., [11],[12]). These solutions assume a TCP/IP network with a RSVP management infrastructure and are RTSJ-compliant. Regarding CPS, asynchronous programming is particularly interesting because their inherent coupling with the physical world. Then, UDP-based alternatives (e.g., the asynchronous messages-passing protocols used in Actors[18]) must be subject of study and investigation regarding their use in CPS.

B. Communication Protocols

The standard Actor semantics provides properties (e.g., encapsulation, fair scheduling, locality of reference and migration transparency) that enables compositional design and makes the architecture scalable. Moreover, this model provides failure isolation while improving performance. Because of that, the Actor approach requires attention regarding possible contributions on ideas to DRTSJ. Name transparency and scalability makes this framework interesting regarding real-time Java-based CPS running under multicore platforms. Asynchronous message passing is a key source of indeterminism. This makes the Actor model unsuitable for real-time CPS.

The Actor behavior is determined by the order in which the messages are processed. However, it is possible to restrict the order in which messages are processed by using synchronous messaging based on the *request-replay messaging* pattern. This communication model is supported as a primitive in the *Scala Actors*[19], which can be extended to support real-time characteristics through a research effort [9]. In this regard, the work presented in [13] proposes a middleware to provide a set of high level services for sensor-based networks that is based on RTSJ.

V. Safety, Reliability and Certification

The software design and implementation of CPS must lead to a correct execution and the noisy or inaccurate

[18] http://www.erights.org/history/actors.html
[19] http://www.scala-lang.org/old/node/242

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

Paradigm	Mechanism	Technology	Java Support
Control-flow	move both data and the application code	*Remote Procedure Call* (RPC)	em Remote Method Invocation (RMI)
Networked	exchange asynchronous or synchronous messages	IPC or message passing	*Java's Message System* (JMS)
Data-flow	move data among application entities	*Data Distribution Services* (DDS)	JMS

TABLE II
MAIN DISTRIBUTED PARADIGMS ALLOWING A REAL-TIME EXTENSION.

information sensed must be checked and handled to ensure an accurate understanding of the environment. Safety-critical CPS are distinguished by stringent requirements (e.g., space, time predictability and dependability) at both process and product levels. These systems must pass a V&V and a certification process.

A. Distributed Control Systems Development

The complexity of software in safety-critical CPS has reached the point where new techniques are needed to ensure system dependability while improving productivity. Hence, the IEC 61499 (i.e., the international standard for distributed control and automation)[21] introduces component-based in industrial automation and control systems. The main construct of this architecture is the *Function Block* (FB), which basically contains algorithms and communicating state machines that have both events and data as inputs and outputs and which result in a modular and hierarchical control application design [14].

Regarding real-time Java the work presented in [15] provides an implementation approach that maps the elements of an IEC 61499 FB network to RTSJ. This approach derives the class `FunctionBlock` from the RTSJ `NoHeapRealTimeThread` class (i.e., allocates outside the heap) and must accomplish with the RTSJ assignment rules.

B. Safety Critical Systems Certification

The *Safety Critical Java Specification* (SCJS) profile reduces, restricts and simplifies the RTSJ programming model to be suitable for V&V of safety-critical applications (e.g., for aerospace vehicles, air-traffic control systems or some enterprise and financial applications). As it is based on the DO-178B[22], dead code elimination is supported. Additionally, it presents the following characteristics: *(i)* no garbage collection, *(ii)* safe stack-based allocation, *(iii)* very small memory footprint, *(iv)* simple run-time environment and *(v)* efficient throughput (i.e., nearest to optimized C). Considering the cost of failure in safety-critical systems, the effort of adopting static memory management in such systems can be justified. As DSRT, SCJ specification defines three compliance levels:

- L0 supports single processor platforms.
- L1 supports multiprocessor platforms, but programs must be fully partitioned.
- L2 supports multiprocessor platforms where programs can be globally scheduled.

In contrast to RTSJ, there is not jet a *Reference Implementation* (RI) for SCJ. In [16] we can find a work-in-progress implementation of the `javax.safetycritical` package. The *native thread* approach for multiprocessor support for levels L1 has been verified respectively in [17] and [18] by using a state-rich process algebra that combines Z and CSP.

Certification of safety-critical multicore systems has been recognized by the *Certification Authorities Software Team*[23] (CAST) which strongly suggests identify, analyze, and mitigate all sources of time indeterminism. CAST identifies shared hardware elements (i.e., main memory, cache, I/O bandwidth and interconnections), as possible sources of safety violations. And also, it recognizes as a major goal the development of evidence-based validation and certification procedure.

VI. USE CASES

There is a need to develop a methodology for the design and analysis of medical safety critical systems, because of this, we focus on the cardiac pacemaker case study. As a typical CPS system, the pacemaker integrates the design of control algorithms and the computational platforms on which they run. Note that the pacemaker needs to know about the intrinsic timing behaviour of the heart and uses this knowledge to monitor the time interval between natural heart beats to determine whether or not the heart should be paced. The pacemaker controls the heart rhythm through sensing and pacing operations. In order to do this, the pacemaker controls five elements: *(i)* an atrium sensor, *(ii)* an atrium pulse generator, *(iii)* a ventricle sensor, *(iv)* a ventricle pulse generator, and *(v)* a rate modulation sensor.

The solution presented in [19], has been structured in five SCJ handlers (i.e., two aperiodic handlers for the atrial

[21] http://webstore.iec.ch/preview
[22] Software Considerations in Airborne Systems and Equipment Certification
[23] An international group of avionics certification and regulatory representatives from North and South America, Europe, and Asia.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

and ventricle pulse generators, two periodic handlers for atrial and ventricle sensor and a periodic handler for the rate modulation sensor). This structure makes this solution simple, elegant and orthogonal with the underlying hardware. The preemptive scheduling is RMS where the aperiodic handlers have the highest priority. However, the control requirements of the cardiac pacemaker do not conveniently fit the classical periodic programming used in real-time systems and supported by SCJ.

As another approach, a SystemJ-based approach for the pacemaker is provided in [20]. This is a reactive event based solution that has been implemented within a clock-domain (i.e., it only uses the synchronous subset of SystemJ). The process consists of three synchronous parallel reactions. This solution allows handle multiple events occurring simultaneously, unlike SCJ, which does not specify the semantics of handling multiple events. The SystemJ solution uses formal semantics based on linear temporal logic providing verification of the time guarantees, which is normal in reactive systems. This is difficult in typical real-time systems and not possible for the SCJ model.

VII. Data Interoperability and Security

Sensors determine the state of real world objects by monitoring the environment (i.e., the physical world) and the wireless networks carry this information to a database for further processing (i.e., the cyber world). Also, actuator networks provide two-way interaction between the cyber world and physical world. This model produces large amounts of data, which requires to be filtered, analyzed and processed in a timely way. Here the real-time filter and analysis of large amount of data with different structures and nature is a critical requirement that must be addressed.

Some CPS requires the integration of physical data into the cloud to be processed in a timely way. In some cases (e.g., smart cities), there are multiple subsystems interacting with each other, some of them open (e.g., closed-loop controlled). The designer of complex CPS are faced with challenges in different maters such as real-time, fault tolerance, autonomy, mobility and intelligence. New security requirements emerge as CPS become more and more complex. Machine learning and *self-X* approaches (e.g., self-healing, self-adaption and self-reconfiguration) plays an important role in real time data analysis in open CPS.

A. Dealing with Big Data

Real-time big-data processing requires further research, where the system components also can provide data at different rates. Here, the big-data processing must ensure not only the correct input/output, but also their processing must be given at right times. That is, big data requires to be processed at real-time. The JDK 8 presents new features (e.g., lambdas and parallel streams) that allow to deal with big data. The Java 8 stream processing uses the fork/join paradigm, included in JDK 7, which is the main model of parallel execution in the *Open Multi-Processing* (OpenMP) framework[24].

The use of the current Java 8 stream processing by real-time threads can result in significant priority inversions. In order to avoid this problem, the approach presented in [21] extends the `ForkJoinPool` class by the `RealtimeForkJoinPool` class that supports RTSJ's real-time threads. This work has been extended in [22]. The integration big data and RTSJ has been also addressed in [23].

B. Cyber-security and Privacy

With increased connectivity of CPS, security is becoming an increasingly vital consideration for embedded solutions. CPS operates in a wide range of contexts and environments, which make relevant security and privacy properties. Providing security in real-time CPS is challenging because security measures typically degrade the performance. Particularly, physical timing signals introduce new security requirements. Security problems (e.g., integrity), privacy issues (e.g., confidentiality), and timing requirements (e.g., predictability) in this kind of systems are related with safety, reliability, and resilience.

The Java sandbox provides a secure environment to run multiple applications isolated from each other, as well as the OS and other components. The *Write Once, Run Anywhere* (WORA) development model can be applied to security by developing component-based Java applications according to security standards. Secure communication is ensured with *Java Secure Socket Extension* (JSSE) and the *Java Authentication and Authorization Service* (JAAS) for user, device, and data identity, and the *Public-Key Cryptography Standard* (PKCS) for data encryption. Also, the *Security and Trust Services* (SATSA) API provides communication capabilities and encrypted security features.

The *Common-Off-The-Shelf* (COTS) software and services is usually built and delivered by vendors, and offers a software quality based on competition (i.e., it is the market, not industry, who drives the development of the application), which generally improves the product . However, when using COTS, the security is a serious risk because this software can contain vulnerabilities. The work presented in [24] extends this new practice to distributed real-time Java applications.

C. Adaptive and Autonomous Systems

A pacemaker adapting the pacing rate to the heart behaviour provides a high quality of live to the patient. Some real-time CPS (e.g., health-care and home automation) require some forms of dynamic support in order to adjust their-self to operate without affecting timing contracts, even if there are run-time changes[25]. Adaptive systems are able to adapt their behavior and their structure because of changes in the environment or within the system itself, which allows to optimize system configurations (e.g., to add new features, or to handle failure cases).

[24] http://openmp.org

[25] Some of these system are designed to control the whole life-cycle.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2016.2630801, IEEE Transactions on Industrial Informatics

Dynamic reconfiguration (e.g., by using the OSGi framework[26]) provides a powerful mechanism to adapt CPS. The OSGi platform is implemented in Java, but it does not have support for real-time applications. The approach presented in [25] provides an admission control protocol and temporal isolation, which allows to install OSGI-based components into the system guaranteeing their timing resources and ensuring safe updating. However, this is still challenging for critical CPS because this updating must be taken into account in the *Worst-Case Execution Time* (WCET) analysis.

Regarding component-based CPS, the most important principle to be considered is that: *(i)* non-functional properties (e.g., timeliness and testability) must not be affected by the integration of the system, *(ii)* resource reservation may cause overload situations given the impossibility to predict the number of components in the system, and *(iii)* scheduling analysis allows to assign priorities within components, but not across components, which can cause problems (e.g., deadlocks, starvation and missed deadlines).

VIII. Conclusion

CPS impact our world and lead to radical changes in many application fields (e.g., healthcare, automotive industry, aircraft or energy). These systems include time-aware operations executing in resource-constrained platforms. Here safety control and cyber-security problems are challenge. Some CPS requires the integration of several domains (e.g., closed-loop control, real-time processing, big-data analysis and self-adaptive systems) that must be addressed by new techniques and engineering methodologies.

In this paper we overview the existing Java-based approach by dealing with real-time requirements, high-integrity properties and service-oriented component models. The research in CPS has very important open issues that must be solved. Real-time Java technologies can be conveniently used to develop this kind of systems. It is also mandatory to consider adaptive component paradigms, robust and flexible methodologies and existing standards for development and certification.

References

[1] E. A. Lee, "CPS Foundations," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 737–742.

[2] A. Fisher, C. A. Jacobson, E. A. Lee, R. M. Murray, A. L. Sangiovanni-Vincentelli, and E. Scholte, "Industrial Cyber-Physical Systems - ICPHYS." in *CSDM*, M. Aiguier, F. Boulanger, D. Krob, and C. Marchal, Eds. Springer, 2013, pp. 21–37.

[3] M. T. Higuera-Toledano, "About 15 Years of Real-Time Java," in *JTRES*, 2012, pp. 34–43.

[4] P. C. Dibble and A. J. Wellings, "JSR-282 Status Report," in *JTRES*, 2009, pp. 179–182.

[5] D. Masson and S. Midonnet, "RTSJ Extensions: Event Manager and Feasibility Analyzer," in *JTRES*, 2008, pp. 10–18.

[6] A. Zerzelidis and A. J. Wellings, "A Framework for Flexible Scheduling in the RTSJ," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 1, 2010.

[7] A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar, "Rate-Harmonized Scheduling for Saving Energy," in *RTSS*, 2008, pp. 113–122.

[8] H. T. Mei, I. Gray, and A. J. Wellings, "Integrating Java 8 Streams with the Real-Time Specification for Java," in *JTRES* 2015, pp. 10:1–10:10.

[9] M. Schoeberl, "Scala for Real-Time Systems?" in *JTRES*, 2015.

[10] G. Drescher and W. Schröder-Preikschat, "Guarded Sections: Structuring Air for Wait-Free Synchronisation," in *ISORC* 2015, pp. 280–283.

[11] D. Tejera, A. Alonso, and M. A. de Miguel, "RMI-HRT: Remote Method Invocation - Hard Real Time," in *JTRES*, 2007, pp. 113–120.

[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.*, vol. 10, no. 1, 2010.

[13] E. Cañete, J. Chen, M. Díaz, L. Llopis, and B. Rubio, "A Service-Oriented Approach to Facilitate WSAN Application Development," *Ad Hoc Networks*, vol. 9, no. 3, pp. 430–452, 2011.

[14] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review," *IEEE Trans. Industrial Informatics*, vol. 7, no. 4, pp. 768–781, 2011.

[15] M. Tangermann, C. Schwab, A. Lüder, L. Ferrarini, and C. Veber, "Encapsulation of IEC 61499 Function Blocks Using Real-Time Java According to the RTSJ," in *OTM Workshops*, 2004, pp. 346–358.

[16] S. Zhao, A. J. Wellings, and S. E. Korsholm, "Supporting Multiprocessors in the ICECAP Safety-Critical Java Run-Time Environment," in *TRES* 2015, pp. 1:1–1:10.

[17] L. Freitas, J. Baxter, A. Cavalcanti, and A. J. Wellings, "Modelling and Verifying a Priority Scheduler for an SCJ Runtime Environment," in *IFM* 2016, pp. 63–78.

[18] M. Luckcuck, A. Cavalcanti, and A. J. Wellings, "A Formal Model of the Safety-Critical Java Level 2 Paradigm," in *IFM* 2016, pp. 226–241.

[19] N. K. Singh, A. J. Wellings, and A. Cavalcanti, "The Cardiac Pacemaker Case Study and its Implementation in Safety-Critical Java and Ravenscar ADA," in *JTRES*, 2012, pp. 62–71.

[20] H. Park, A. Malik, M. Nadeem, and Z. Salcic, "The Cardiac Pacemaker: SystemJ versus Safety Critical Java," in *JTRES*, 2014, pp. 37:37–37:46.

[21] H. Mei, I. Gray, and A. J. Wellings, "Integrating Java 8 Streams with the Real-Time Specification for Java," in *JTRES*, 2015, p. 20.

[22] H. T. Mei, I. Gray, and A. J. Wellings, "Real-Time Stream Processing in Java," in *Reliable Software Technologies - ADA-Europe*, 2016, pp. 44–57.

[23] P. Basanta-Val, N. F. García, A. J. Wellings, and N. C. Audsley, "Improving the Predictability of Distributed Stream Processors," *Future Generation Comp. Syst.*, vol. 52, pp. 22–36, 2015.

[24] I. Estévez-Ayres, P. Basanta-Val, and M. García-Valls, "Composing and Scheduling Service-Oriented Applications in Time-Triggered Distributed Real-Time Java Environments," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 152–193, 2014.

[25] T. Richardson and A. Wellings, "An Admission Control Protocol for Real-Time OSGI," in *ISORC*, 2010.

[26] http://www.osgi.org/Specifications/HomePage

M. Teresa Higuera-Toledano Teresa Higuera obtained a MS degree at the Technical University of Madrid and her PhD in computer science from the University of Rennes by holding a PhD studentship for three years at INRIA in Rocquencourt. Currently, she is a Professor at the Complutense University of Madrid, and previously she worked full-time as teaching assistant at Technical University of Madrid for seven years and for two years as system engineering at IBERIA. Her principal research interest are embedded, real-time and distributed systems. In these fields she has published several articles, papers and technical reports.